

Chapter 3. Oscillation

“Trigonometry is a sine of the times.”
-- Anonymous

In this Chapter:

- Angles: Degrees vs. Radians
- Angular velocity and acceleration
- Basic Trigonometry
- Pointing in the direction of motion
- Cartesian and Polar Coordinates
- Oscillating Motion
- Drawing a Wave pattern
- Oscillating Pendulums
- Oscillating Springs

In Chapters One and Two, 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: the mathematics of triangles, specifically right triangles.

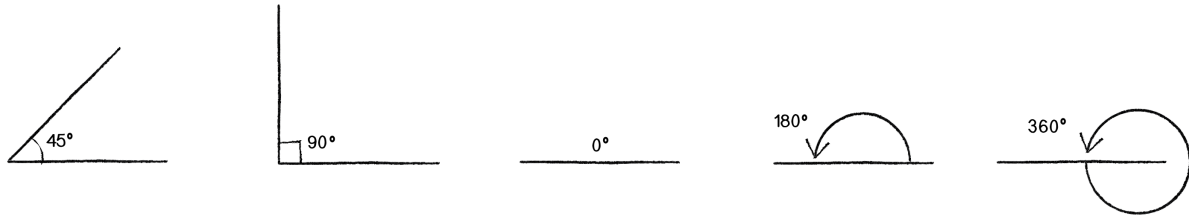
Trigonometry is going to give us a lot of tools. We’ll get to think about angles and angular velocity and acceleration. It’s going to teach us about the sine and cosine functions, which when used properly can yield an nice ease-in, ease-out wave pattern. It’s going to allow us to calculate more complex 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.

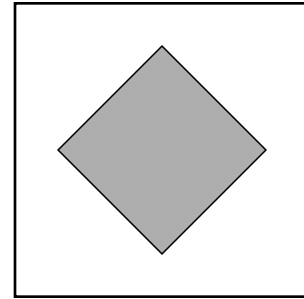
3.1 Angles

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 question while using the *rotate()* function to rotate and spin objects.

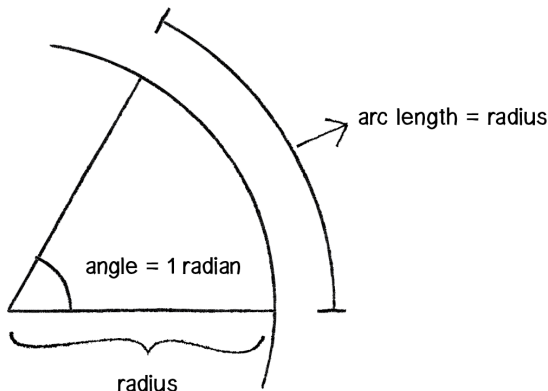
The first order of business is **radians** and **degrees**. You’re probably familiar with the concept of an angle in degrees. A full rotation goes from zero to 360 degrees. 90 degrees (a right angle) is 1/4th of 360, shown below as two perpendicular lines.



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



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 x.x). 180 degrees = π radians, 360 degrees = 2π radians, 90 degrees = $\pi/2$ radians, etc.



[DRAW THIS MORE TO SCALE]

The formula to convert from degrees to radians is:

$$\text{radians} = 2 * \pi * (\text{degrees} / 360)$$

Fortunately for us, 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. In addition, 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: <http://www.processing.org/learning/transform2d/>).

PI, what is it?

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 center). It is equal to approximately 3.14159.

Exercise: Rotate a baton-like object (see screenshot 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 X.X, let's say we were going 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, we'd have something like:

Example 3.x: Angular Motion using rotate()

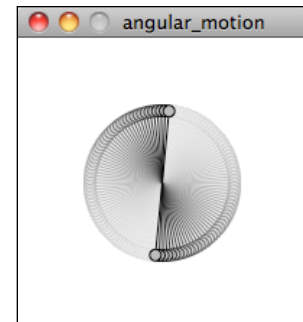
```
float angle = 0;                $$ location
float aVelocity = 0;            $$ velocity
float aAcceleration = 0.001;    $$ acceleration

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

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 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.

Example 3.x: Angular motion added to Mover object

```
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-fashion motion of cartesian location
  location.add(velocity);

  aVelocity += aAcceleration;    $$ New-fangled 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();

    translate(location.x,location.y);
    rotate(angle);
    rect(0,0,mass*16,mass*16);
    popMatrix();
}
```

\$\$ pushMatrix() and popMatrix() are necessary so that the rotation of this shape doesn't affect the rest of the world

\$\$ Set the origin at the shape's location
\$\$ Rotate by the angle

Now, if we were to actually go ahead and run the above code, we wouldn't see anything new. This is because 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.

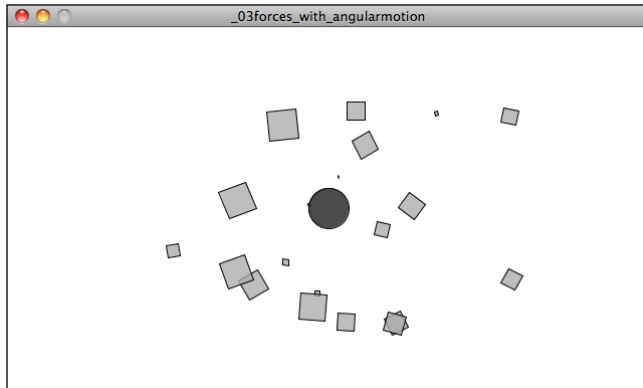
```
float aAcceleration = 0.01;
```

However, a more interesting result can be produced by dynamically assigning an angular acceleration according to forces in the environment. Now, we could head far down this road, trying to model realistically the physics of angular acceleration using the concept 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 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 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:



```
void update() {

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

    aAcceleration = acceleration.x / 10.0;    $$ calculate angular acceleration according to
                                              acceleration's horizontal direction and
                                              magnitude

    aVelocity += aAcceleration;
    aVelocity = constrain(aVelocity,-0.1,0.1);  $$ Use constrain() to ensure that angular
                                              velocity doesn't spin out of control

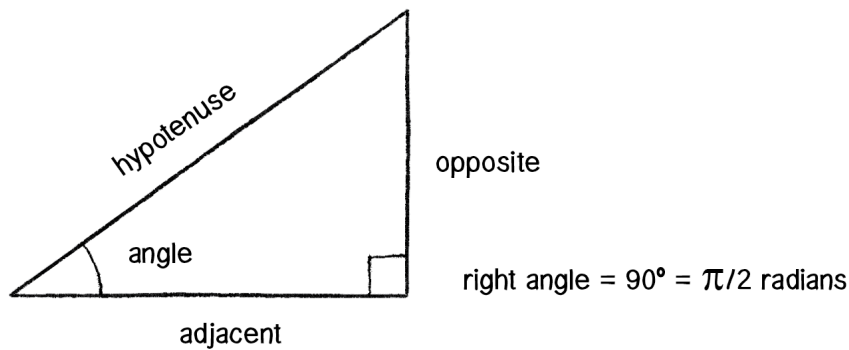
    angle += aVelocity;

    acceleration.mult(0);
}
```

Exercise: 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 its shot from the cannon. How realistic can you make it look?

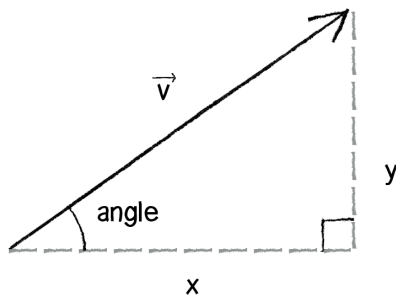
3.3 Trigonometry

I think it may be time. We've looked at angles, we've spun an object. It's time for: *sohcahtoa*. Yes, *sohcahtoa*. Strangely enough, this seemingly nonsensical word is the foundation for a lot of computer graphics work. Anytime you need to calculate an angle, determine the distance between points, deal with circles, arcs, lines, etc., you will find that a basic understanding of trigonometry is essential. And *sohcahtoa* is a (albeit somewhat absurd) mnemonic device for remembering the definitions of the trigonometric functions, sine, cosine, and tangent.



- *soh*: $\sin = \text{opposite} / \text{hypotenuse}$
- *cah*: $\cos = \text{adjacent} / \text{hypotenuse}$
- *toa*: $\tan = \text{opposite} / \text{adjacent}$

Take a look at the above diagram 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:

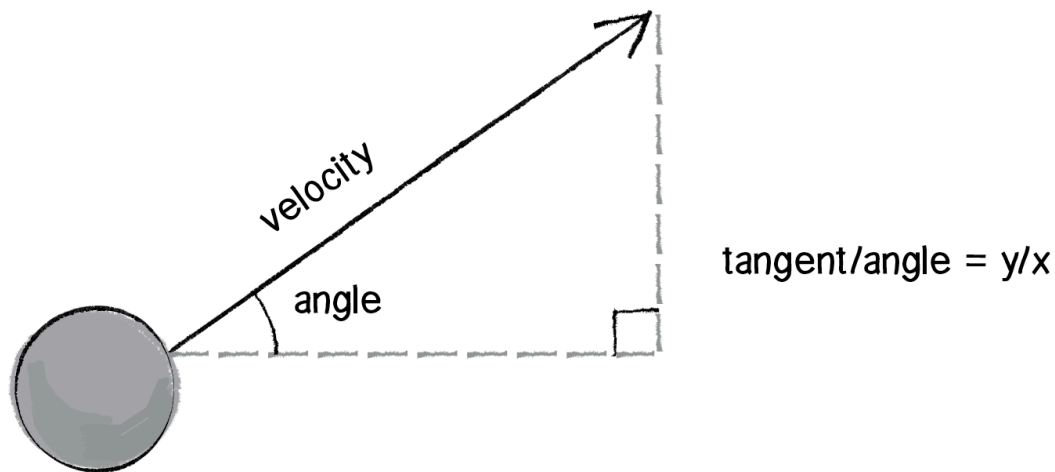
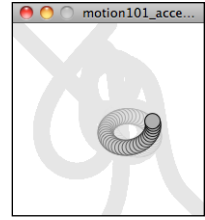


See how we create a right triangle out of a vector? The vector arrow itself is the hypotenuse and the components of the vector ('x' and 'y') are the sides of the triangle. The angle is an additional means for specifying the vector's direction (or "heading").

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. Let's 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.x, 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 you 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 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 SHOULD NOTE VELOCITY.X AND Y COMPONENTS FOR VECTOR]

OK. We know that the definition of tangent is:

$$\text{tangent}(\text{angle}) = \text{velocity}_y / \text{velocity}_x$$

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." (There is also an "inverse sine" and an "inverse cosine".)

If the tangent of some value 'a' equals some value 'b', then the inverse tangent of 'b' equals 'a', i.e.

$$\begin{aligned} \text{if:} & \quad \text{tangent}(a) = b \\ \text{then:} & \quad a = \text{arctangent}(b) \end{aligned}$$

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

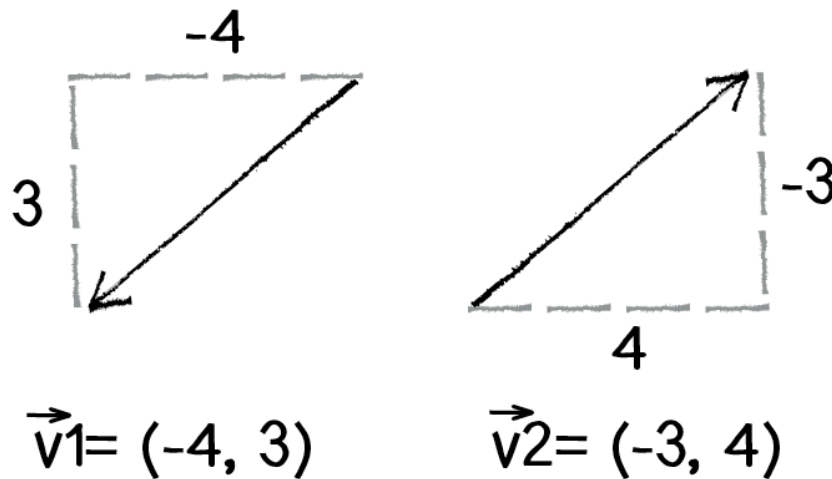
if: $\text{tangent}(\text{angle}) = \text{velocity}_y / \text{velocity}_x$
 then: $\text{angle} = \text{arctangent}(\text{velocity}_y / \text{velocity}_x)$

Now that we have the formula, let's see where it should go in our Mover's *display()* function. Notice how 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. There's a pretty big problem, though. Let's consider the following two velocity vectors:



[DIAGRAM IS WRONG, V2 should equal (4,-3)]

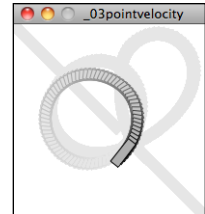
Though superficially similar, the above two vectors point in quite different directions, opposite in fact! However, if we were to apply our formula to solve for the angle to each vector...

$V1 \Rightarrow \text{angle} = \text{atan}(-4/3) = \text{atan}(-1.25) = -0.9272952 \text{ radians} = -53 \text{ degrees}$
 $V2 \Rightarrow \text{angle} = \text{atan}(4/-3) = \text{atan}(-1.25) = -0.9272952 \text{ radians} = -53 \text{ degrees}$

We get the same angle for each vector. This can't be right for both since 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 (and pretty much all programming environments) has a nice function called *atan2()* which does it for you.

```
void display() {
  float angle = atan2(velocity.y,velocity.x);  $$ Using atan2() to account
                                              all possible directions

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



To simplify this even further, the *PVector* class itself provides a function *heading2D()*, which takes care of calling *atan2()* for you so you can get the direction angle, in radians, for any Processing *PVector*.

```
float angle = velocity.heading2D();  $$ The easiest way to do this!
```

Exercise: 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 it is currently moving.

3.5 Polar vs Cartesian coordinates

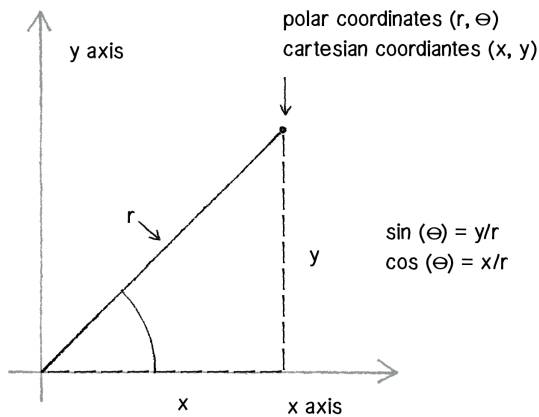
Any time we display a shape in Processing we have to specify a pixel location, a set of x and y coordinates. These coordinates are known as Cartesian coordinates, named for the French mathematician René Descartes who developed the ideas behind Cartesian space.

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.



The Greek letter θ (“theta”) is often used to denote an angle. Since it’s the convention for a polar coordinate to be referred to as (r, θ) we’ll use “theta” as a variable name when referring to angle.

$$\begin{aligned} \text{sine}(\text{theta}) &= y / r & \rightarrow & y = r * \text{sine}(\text{theta}) \\ \text{cosine}(\text{theta}) &= x / r & \rightarrow & x = r * \text{cosine}(\text{theta}) \end{aligned}$$

For example, if r is 75 and θ is 45 degrees (or $\pi/4$ radians), we can calculate x and y as follows. The functions for sine and cosine in Processing are *sin()* and *cos()* respectively. They each take one argument, a floating point 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 “ θ .”

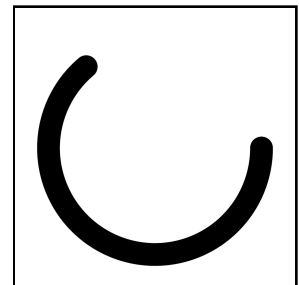
Example 3.x: Polar to Cartesian

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

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

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);

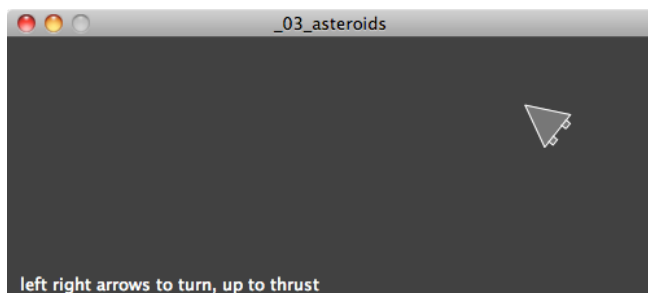
theta += 0.01;
}

```

Exercise: Using the above example, 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!



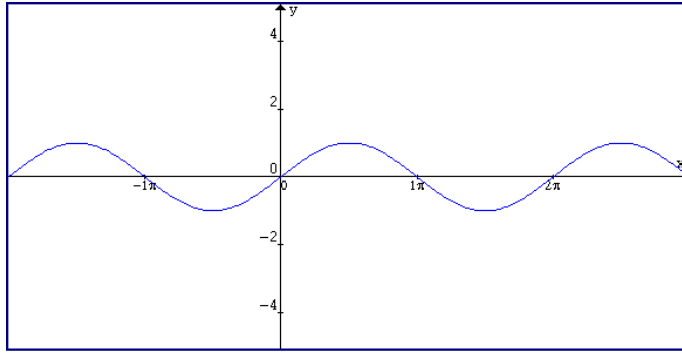
Exercise: Simulate the spaceship in the game Asteroids. In case you aren't familiar with Asteroids, here is a brief description. A spaceship (represented as a triangle) floats in two dimensional space. The left arrow keys turns the spaceship counter-clockwise, the right clockwise. The space bar applies a "thrust" force in the direction the spaceship is pointing. See the screenshot below.



3.6 Oscillation: Amplitude and Period

Are you amazed yet? We've seen 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 $y = \sin(x)$.

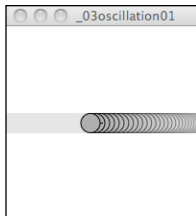


[REDRAW THIS]

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. Pluck a guitar string, swing a pendulum, bounce 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 this will follow the same methodology we applied to Perlin noise in the introduction.)

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.



[for print, need a better illustration, canvas for HTML5]

This is what is known as simple harmonic motion (or to be fancier: “the periodic sinusoidal oscillation of an object”). This is 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 x.x), 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 world we live in, the Processing world, 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. What is time in our Processing world? I mean, certainly we could say we want the circle to oscillate every 3 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 2PI radians—i.e., it will start at 0, repeat at 2PI, 4PI, 6PI, 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 at those values:

frameCount	frameCount / period	TWO_PI * frameCount / period
0	0	0
60	0.5	PI
120	1	TWO_PI

frameCount	frameCount / period	TWO_PI * frameCount / period
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 2 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 a Processing example that oscillates the x location of a circle with an amplitude of 100 pixels and a period of 120 frames.

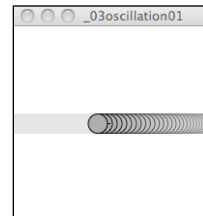
Example 3.x Simple Harmonic Motion

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

void draw() {
  background(255);

  float period = 120;
  float amplitude = 100;
  float x = amplitude * cos(TWO_PI * frameCount / period);  $$ Calculating horizontal location
                                                             according to formula for simple
                                                             harmonic motion

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



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: Create a simulation of a weight (sometimes referred to as a “bob”) hanging from the top of the window by a spring using the sine function. Use the map() function to calculate the vertical location of the bob. Later this chapter, we’ll see how to recreate this same simulation by modeling the forces of a spring according to Hooke’s law.

3.6 Oscillation: Angular Velocity

Understanding the concepts of oscillation, amplitude, frequency/period is important and 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.x. 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.x Simple Harmonic Motion II

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

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

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.x: Oscillator objects

```
class Oscillator {

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

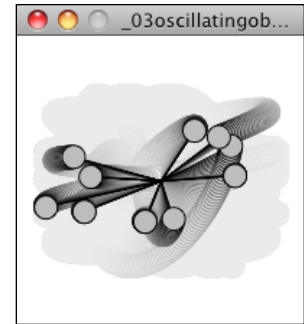
  Oscillator() {
    angle = new PVector();
    $$ Random velocities and amplitudes
    velocity = new PVector(random(-0.05,0.05),random(-0.05,0.05));
    amplitude = new PVector(random(width/2),random(height/2));
  }

  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);                      $$ Drawing the oscillator as a line connecting a circle
    popMatrix();
  }
}
```

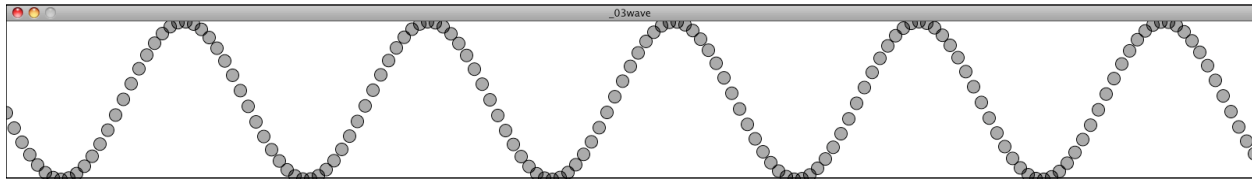


Exercise: Try initializing each Oscillator object with velocities and amplitudes that are not random to create some sort of regular pattern. Can you make the oscillators appear to be the legs of a insect-like creature?

Exercise: Incorporate angular acceleration into the Oscillator object.

3.7 Waves

If you're asking yourself, "Um, this is all great and everything, but what I really want is to just 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 10 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.

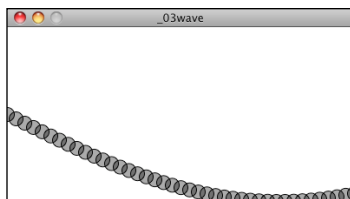
```
for (int x = 0; x <= width; x += 10) {

    float y = amplitude*sin(angle);    $$ 1) Calculate the y location according to
                                       amplitude and sine of the angle.

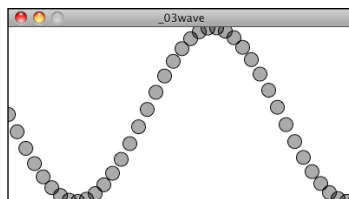
    ellipse(x,y+height/2,16,16);      $$ 2) Draw a circle at the (x,y) location

    angle += angleVel;                $$ 3) Increment the angle according to angular
                                       velocity.
}
```

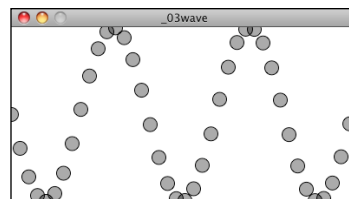
Let's look at the results with different values for thetaVel:



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.

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

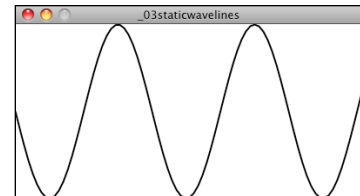
```
size(400,200);
background(255);
smooth();
```

```
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."

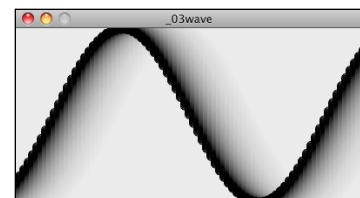
While it's a nice thought, it doesn't work. If you look at the wave, the righthand edge doesn't match the lefthand; where it ends in one cycle of *draw()* can't be where it starts in the next. Instead, what we need to do is have a variable dedicated entirely to tracking what value of angle the wave should start with. This angle (which we'll call "startAngle") increments with its own angular velocity.

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

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

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

```
  startAngle += 0.02;
  float angle = startAngle;
```



\$\$ In order to move the wave, we start at a different theta value each frame

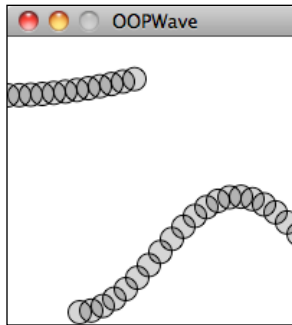
```

for (int x = 0; x <= width; x += 10) {
  float y = map(sin(angle), -1, 1, 0, height);
  stroke(0);
  fill(0, 50);
  ellipse(x, y, 10, 10);
  angle += angleVel;
}
}

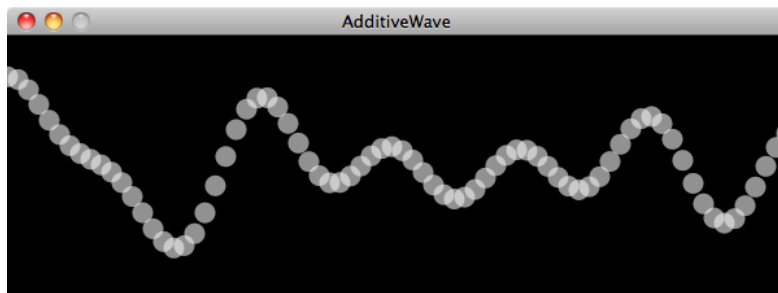
```

Exercise 3.x: Try using the Perlin noise function instead of sine or cosine.

Exercise 3.x: 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.)



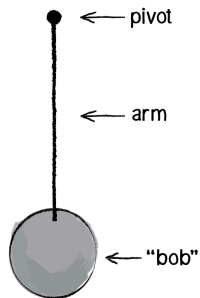
Exercise 3.x: More complex waves can be produced by the values of multiple waves together. Create a sketch that implements this, as in the screenshot below.



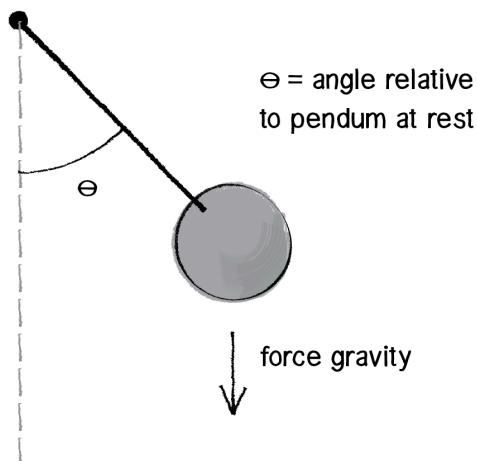
3.8 Trigonometry and Forces: The Pendulum

Do you miss Newton's laws of motion? I know I sure do. Well, lucky for you, it's time to bring it all back home. After all, it's been nice learning about triangles and tangents and waves, but the core of this book is really simulating the physics of moving bodies. Let's take a look at how trigonometry can help us with this pursuit.

A pendulum is a bob suspended from a pivot. Obviously a real-world pendulum would live in a 3D space, but we're going to look at a simpler scenario, a pendulum in a 2D space—a Processing window.

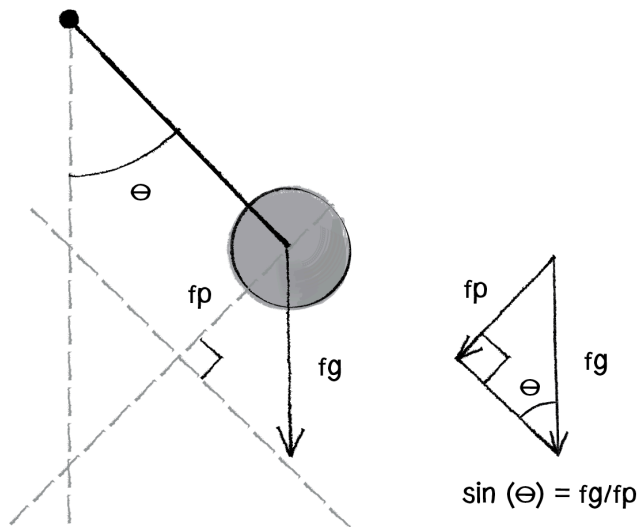


In Chapter 2, we learned how a force (such as the force of gravity in the diagram above) causes an object to accelerate. $F = M * A$ or $A = F / M$. In this case, however, the pendulum bob doesn't simply fall to the ground because it is attached by an arm to the pivot point. And so, in order to determine its *angular* acceleration, we not only need to look at the force of gravity, but also at the angle of the pendulum's arm (relative to a pendulum at rest with an angle of zero).



And this is why we're here in this chapter. In the above case, since the pendulum's arm is of fixed length, the only variable in the scenario is the angle. We are going to simulate the pendulum's motion through the use of angular velocity and acceleration. The angular acceleration will be calculated using Newton's second law with a little trigonometry twist.

Let's zoom in on the right triangle from the pendulum diagram.



[ADD TENSION TO THIS DIAGRAM OR MAYBE THE ONE RIGHT BEFORE?]

After all if there were no arm, the bob would just fall straight down. It's the tension force of the arm that keeps the bob accelerating towards the pendulum's rest state. We therefore can see that the force of the pendulum (F_p) should point perpendicular to the arm of the pendulum in the direction that the pendulum is swinging. The force of gravity (F_g) points downward. By making a right triangle out of these two vectors, we've accomplished something quite magnificent. We've made the force of gravity the hypotenuse of a right triangle and separated the vector into two components, one of which represents the force of the pendulum. Since sine equals opposite over hypotenuse, we have:

$$\sin(\theta) = F_p / F_g$$

therefore:

$$F_p = F_g * \sin(\theta)$$

Lest we forget, we've been doing all of this with a single question in mind: What is the angular acceleration of the pendulum? Once we have the angular acceleration, we'll be able to apply our rules of motion to find the new angle for the pendulum.

$$\begin{aligned} \text{angular velocity} &= \text{angular velocity} + \text{angular acceleration} \\ \text{angle} &= \text{angle} + \text{angular velocity} \end{aligned}$$

The good news is that with Newton's second law, we know that there is a relationship between force and acceleration, namely $F = M * A$, or $A = F / M$. So if the force of the pendulum is equal to the force of gravity times sine of the angle, then:

$$\text{angular acceleration of pendulum} = \text{acceleration due to gravity} * \sin(\theta)$$

This is a good time to remind ourselves that we're Processing programmers and not physicists. Yes, we know that the acceleration due to gravity on earth is 9.8 meters per second squared. But this number isn't relevant to us. What we have here is just an arbitrary constant (we'll call it "gravity"), one that we can use to scale the acceleration to something that *feels* right.

$$\text{angular acceleration} = \text{gravity} * \sin(\theta)$$

Amazing. After all that, the formula is so simple. You might be wondering, why bother going through the derivation at all? I mean, learning is great and all, but we could have easily just said: Hey, the angular acceleration of a pendulum is some constant times the sine of the angle. This is just another moment to remind ourselves that the purpose of the book is not to learn how pendulums swing or gravity works. The point is to think creatively about how things can move about the screen in an computationally-based graphics system. The pendulum is just a case study. If you can understand the approach to programming a pendulum, then however you choose to design your onscreen world, you can apply the same techniques.

Of course, we're not finished yet. We may be happy with our simple, elegant formula, but we still have to apply it in code. This is most definitely a good time to practice our object-oriented programming skills and create a Pendulum class. Let's think about all the properties we've encountered in our pendulum discussion that the class will need:

- arm length
- angle
- angular velocity
- angular acceleration

```
class Pendulum {  
  
    float r;           // Length of arm  
    float angle;       // Pendulum arm angle  
    float aVelocity;    // Angular velocity  
    float aAcceleration; // Angular acceleration
```

We'll also need to write a function *update()* to update the pendulum's angle according to our formula. . .

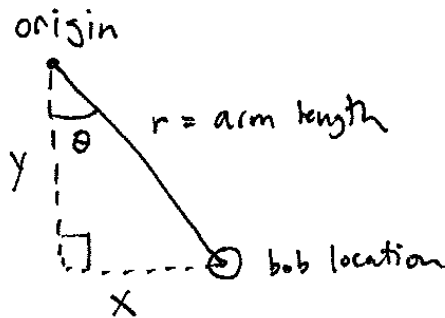
```
void update() {  
    float gravity = 0.4;           // Arbitrary constant  
    aAcceleration = -1 * G * sin(angle); // Calculate acceleration according to our formula
```

```

    aVelocity += aAcceleration;           // Increment velocity
    angle += aVelocity;                   // Increment angle
}

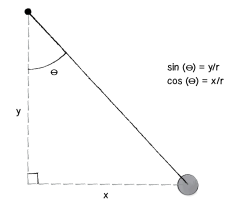
```

...as well as a function *display()* to draw the pendulum in the window, which begs the question: “Um, where do we draw the pendulum?” We know the angle and the arm length, but how do we know the x,y (Cartesian!) coordinates for both the pendulum’s pivot point (let’s call it “origin”) and bob location (let’s call it location)? This may be getting a little tiring, but the answer, yet again, is trigonometry.



$$\sin(\theta) = x/r$$

$$\cos(\theta) = y/r$$



[NEW ILLUSTRATION NEEDS ORIGIN, ARM LENGTH, BOB LOCATION]

The origin is just something we make up, as is the arm length. Let’s say:

```

PVector origin = new PVector(100,10);
float r = 125;

```

We’ve got the current angle stored in our variable “angle”. So relative to the origin, the pendulum’s location is a polar coordinate: (r,angle). And we need it to be Cartesian. Luckily for us, we just spent some time (Section 3.x) deriving the formula for converting from polar to Cartesian. And so:

```

PVector location = new PVector(r*sin(angle),r*cos(angle));

```

Since the location is relative to wherever the origin happens to be, we can just add origin to the location PVector:

```

location.add(origin);

```

And all that remains is the little matter of drawing a line and ellipse (you should be more creative, of course).

```

stroke(0);
fill(175);
line(origin.x,origin.y,location.x,location.y);
ellipse(location.x,location.y,16,16);

```


Before we put everything together, there's one last little detail I neglected to mention. Let's think about the pendulum arm for a moment. Is it a metal rod? A string? A rubber band? How is it attached to the pivot point? How long is it? What is its mass? Is it a windy day? There are a lot of questions that we could continue to ask that would affect the simulation. We're living, of course, in a fantasy world, one where the pendulum's arm is some idealized rod that never bends and the mass of the bob is concentrated in a single, infinitesimally small point. Nevertheless, even though we don't want to worry ourselves with all of the questions, we should add one more variable to our calculation of angular acceleration. To keep things simple, in our derivation of the pendulum's acceleration, we assumed that the length of the pendulum's arm is one. In fact, the length of the pendulum's arm affects the acceleration greatly, the longer the arm, the slower the acceleration. To simulate a pendulum more accurately, we divide by that length, in this case r . (For a more involved explanation, visit: <http://calculuslab.deltacollege.edu/ODE/7-A-2/7-A-2-h.html>.)

```
aAcceleration = (-1 * G * sin(angle)) / r;
```

Finally, a real world pendulum is going to experience some amount of friction (at the pivot point) and air resistance. With our code as is, the pendulum would swing forever, so to make it more realistic we can use a “damping” trick. I say “trick” because rather than model the resistance forces with some degree of accuracy (as we looked at in Chapter 2), we can achieve a similar result by simply reducing the angular velocity during each cycle. The following code reduces the velocity by 1% (or multiplies it by 99%) during each frame of animation:

```
aVelocity *= 0.99;
```

Putting everything together, we have the following example (with the pendulum beginning at a 45 degree angle).

Example 3.x: Swinging Pendulum

```
Pendulum p;
```

```
void setup() {
  size(200,200);
  smooth();
  p = new Pendulum(new PVector(width/2,10),125);
}
```

\$\$ We make a new Pendulum object with an origin location and arm length

```
void draw() {
  background(255);
  p.go();
}
```

```
class Pendulum {
  PVector location; // Location of pendulum ball
  PVector origin; // Location of arm origin
  float r; // Length of arm
  float angle; // Pendulum arm angle
  float aVelocity; // Angle velocity
  float aAcceleration; // Angle acceleration
  float damping; // Arbitrary damping amount
}
```

\$\$ Many many variables to keep track of the pendulum's various properties

```

Pendulum(PVector origin_, float r_) {
    origin = origin_.get();
    location = new PVector();
    r = r_;
    angle = PI/4;

    aVelocity = 0.0;
    aAcceleration = 0.0;
    damping = 0.995;
}

void go() {
    update();
    display();
}

void update() {
    float gravity = 0.4;
    aAcceleration = (-1 * gravity / r) * sin(angle);
    // $$ Formula we worked out for angular acceleration

    aVelocity += aAcceleration;
    angle += aVelocity;
    // $$ Standard angular motion algorithm

    aVelocity *= damping;
    // $$ Apply some damping
}

void display() {
    location.set(r*sin(angle),r*cos(angle),0);
    // $$ Where is the bob relative to the origin?
    // Polar to Cartesian coordinates will tell us!

    location.add(origin);

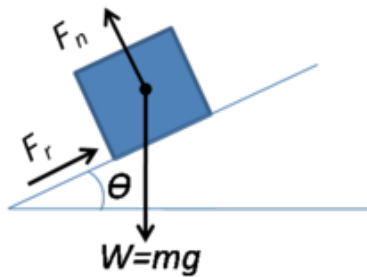
    stroke(0);
    line(origin.x,origin.y,location.x,location.y);
    // $$ The arm
    fill(175);
    ellipse(location.x,location.y,16,16);
    // $$ The bob
}
}

```

(Note that the version of the example posted on the web site has additional code to allow the user to grab the pendulum and swing it with the mouse.)

Exercise: String together a series of pendulums so that the endpoint of one is the origin point of another. (Note that doing this may produce intriguing results but will be wildly inaccurate physically. Simulating an actual double-pendulum involves sophisticated equations which you can read about here: <http://scienceworld.wolfram.com/physics/DoublePendulum.html>).

Exercise: Examine the following diagram [NOTE REDO DIAGRAM, NOT ACTUAL DIAGRAM].

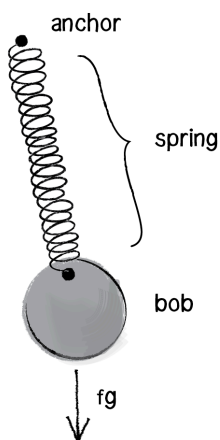


Using trigonometry, what is the magnitude of the “normal” force (the force perpendicular to the incline on which the box rests)? Note that, as indicated, the “normal” force is a component of the force of gravity.

Exercise part 2: Create an example that simulates the box sliding down the incline with friction. Note that the magnitude of the friction force is equal to the normal force.

3.10 Spring Forces

In Section 3.x, we looked at modeling simple harmonic motion by mapping the sine wave to a pixel range. Exercise 3.x asked you to use this technique to create a simulation of a spring hanging from a bob. While using the *sin()* function is a quick-and-dirty, one-line-of-code way of getting something up and running, it won't do if what we really want is to have a bob hanging from a spring in a two-dimensional space that responds to other forces in the environment (wind, gravity, etc.) To accomplish a simulation like this (one that is identical to the pendulum example, only now the arm is a springy connection), we need to model the forces of a spring using PVector.



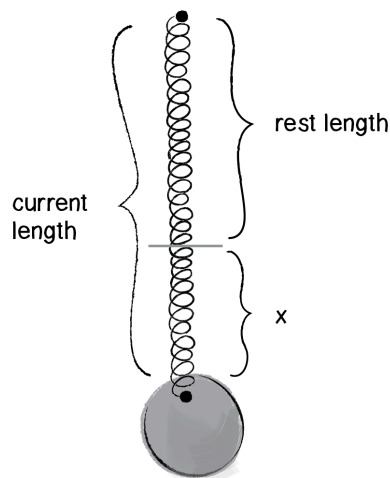
The force of a spring is calculated according to Hooke's law, named for Robert Hooke, a British physicist who developed the formula in 1660. Hooke originally stated the law in Latin: "Ut tensio, sic vis" or "As the extension, so the force" *[from wikipedia]*. Let's think of it this way:

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

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

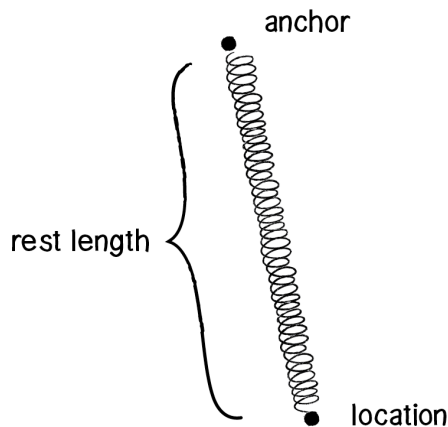
$$F_{\text{spring}} = -k * x$$

- k is constant and its value will ultimately scale the force. Is the spring highly elastic or quite rigid?
- x refers to the displacement of the spring, i.e. the difference between the current length and the rest length. The rest length is defined as the length of the spring in a state of equilibrium.



[DIAGRAM SHOULD SAY $x = \text{CURRENT LENGTH} - \text{REST LENGTH}$]

Now, remember, force is a vector so we need to calculate both magnitude and direction. Let's look at one more diagram of the spring and label all the givens we might have in a Processing sketch.



Let's establish the following three variables as related to the diagram above.

```
PVector anchor;
PVector location;
float restLength;
```

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

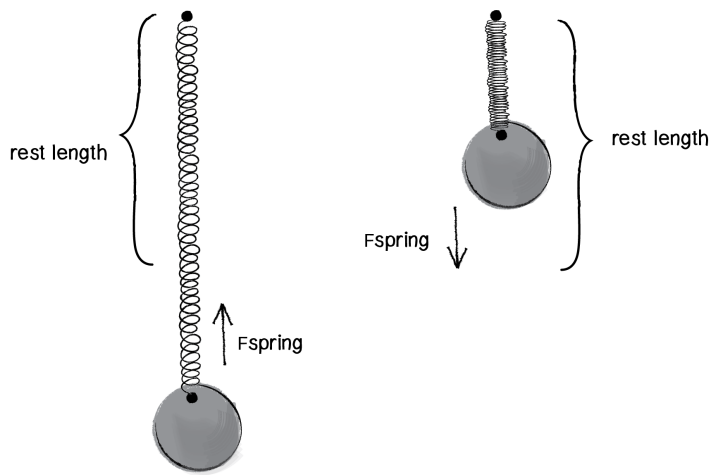
```
float k = 0.1;
```

x is perhaps a bit more difficult. We need to know the "difference between the current length and the rest length." The rest length is defined as the variable "restLength." What's the current length? The distance between the anchor and the bob. And how can we calculate that distance? How about the magnitude of a vector that points from the anchor to the bob? (Note this is exactly the same process we employed when calculating distance in Example 2.x: gravitational attraction.)

```
PVector dir = PVector.sub(bob, anchor);
float currentLength = dir.mag();
float x = restLength - currentLength;
```

\$\$ A vector pointing from anchor to bob gives us the current length of the spring.

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



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

```
float k = 0.1;                                $$ Magnitude of spring force according to Hooke's law
PVector force = PVector.sub(bob, anchor);
float currentLength = dir.mag();
float x = restLength - currentLength;

force.normalize();                             $$ Direction of spring force (unit vector)

force.mult(-1 * k * x);                       $$ Putting it together, direction and magnitude!
```

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

```
Bob bob;

void setup() {
  bob = new Bob();
}

void draw() {
  PVector gravity = new PVector(0,1);          $$ Our chapter 2 "make-up-a-gravity-force"
```

```

bob.applyForce(gravity);

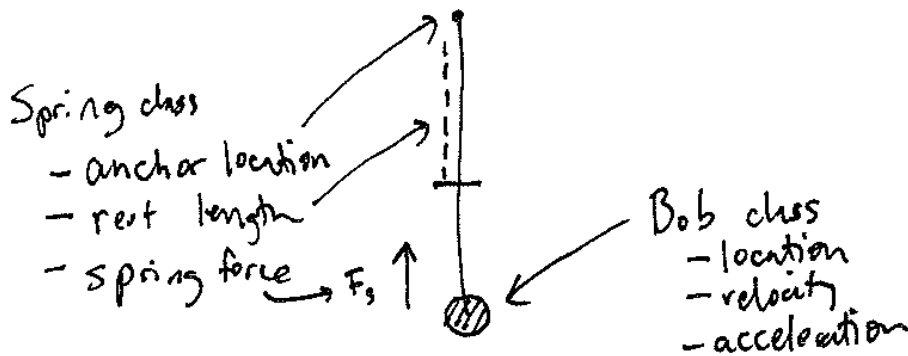
PVector springForce = _____???? $$ We need to also calculate and apply a spring
                                force!

bob.applyForce(spring);

bob.update();                $ Our standard update() and display() functions
bob.display();
}

```

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



Our goal here is to write a nice-looking main program as follows:

```

Bob bob;
Spring spring;                $$ Adding a Spring object

void setup() {
  bob = new Bob();
  spring = new Spring();
}

void draw() {
  PVector gravity = new PVector(0,1);
  bob.applyForce(gravity);

  spring.connect(bob);        $$ This new function in the Spring class will take care of
                              computing the force of the spring on the bob

  bob.update();
  bob.display();
  spring.display();
}

```

You may notice here that this is quite similar to what we did in Example 2.x, the Attractor. There, we said something like:

```
PVector force = attractor.attract(mover);
mover.applyForce(force);
```

The analogous situation here with a spring would be:

```
PVector force = spring.connect(bob);
bob.applyForce(force);
```

Nevertheless, in this example all we said was:

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

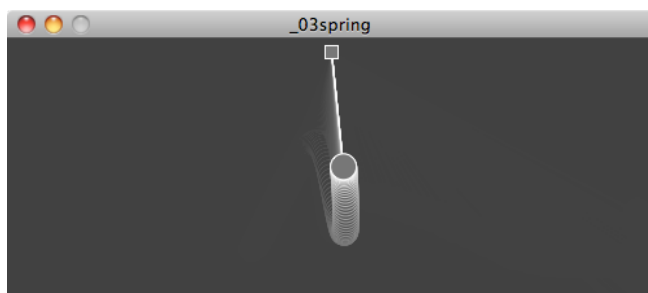
What gives? Why don't we need to call ***applyForce()*** on the bob? The answer is, of course, that we *do* need to call ***applyForce()*** on the bob. Only instead of doing it in ***draw()***, we're just demonstrating that a perfectly reasonable (and sometimes preferable) alternative is to ask the ***connect()***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

```
void connect(Bob b) {
  PVector force = some fancy calculations

  b.applyForce(force);    $$ The function connect() takes care of calling applyForce
                           and therefore doesn't have to return a vector to the calling
                           area.
}
```

Why do it one way with the Attractor and another way with the Spring? When we were first learning about forces, it was a bit clearer to show all the forces being applied in the main ***draw()*** loop, and hopefully this helped you learn about force accumulation. Now that we're more comfortable with that, perhaps it's simpler to embed some of the details inside the objects themselves.

Let's take a look at the rest of the elements in the Spring class.



Example 3.x: a Spring connection

```
class Spring {

  PVector anchor;          $$ We need to keep track of the spring's anchor location

  float len;               $$ Rest length and spring constant variables
  float k = 0.1;
```



```

// Constructor
Spring(float x, float y, int l) {    $$ The constructor initializes the anchor point
                                     and rest length

    anchor = new PVector(x,y);
    len = l;
}

// Calculate spring force
void connect(Bob b) {                $$ Here is our implementation of Hooke's Law
    PVector force = PVector.sub(b.location,anchor);  $$ Get a vector pointing from anchor to Bob
                                                    location

    float d = force.mag();
    float stretch = d - len;          $$ Calculate the displacement between
                                        distance and rest length

    force.normalize();                 $$ Direction and magnitude together!
    force.mult(-1 * k * stretch);

    b.applyForce(force);               $$ call applyForce() right here!
}

void display() {                      $$ Draw the anchor
    fill(100);
    rectMode(CENTER);
    rect(anchor.x,anchor.y,10,10);
}

void displayLine(Bob b) {             $$ Draw the spring connection between Bob
                                     location and anchor

    stroke(255);
    line(b.location.x,b.location.y,anchor.x,anchor.y);
}
}

```

The full code for this example is included on the book web site, and the web version also incorporates two additional features: (1) Bob includes functions for mouse interactivity so that it can be dragged around the window. (2) The Spring object includes a function to constrain the connection's length between a minimum and a maximum.

Exercise: Before running to see the example online, take a look at this constrain function and see if you can fill in the blanks.

```
void constrainLength(Bob b, float minlen, float maxlen) {
    PVector dir = PVector.sub(_____, _____);
    float d = dir.mag();

    if (d < minlen) {
        dir.normalize();
        dir.mult(_____);
        b.location = PVector.add(_____, _____);
        b.velocity.mult(0);
    } else if (_____) {
        dir.normalize();
        dir.mult(_____);
        b.location = PVector.add(_____, _____);
        b.velocity.mult(0);
    }
}
```

Annotations:

- Is it too short? (points to `d < minlen`)
- Vector pointing from Bob to anchor. (points to `PVector.sub(_____, _____)`)
- Is it too long? (points to `_____` in the `else if` condition)
- Keep location within constraint (points to `b.location = PVector.add(_____, _____);` in both branches)

Exercise: Create a system of multiple bobs and spring connections. How would you have a Bob connected to a Bob with no fixed anchor?

The Eco-System Project:

Step 3 Exercise:

Take one of your creatures and incorporate oscillation into its motion. You can use the Oscillator class from Example 3.x as a model. The Oscillator object, however, oscillates around a single point (the middle of the window). Try oscillating around a moving point. In other words, design a creature that moves around the screen according to location, velocity, and acceleration. But that creature isn't just a static shape, it's an oscillating body. Consider tying the speed of oscillation to the speed of motion. Think of a butterfly's flapping wings or the legs of an insect. Can you make it appear that the creature's internal mechanics (oscillation) drive its locomotion? For a sample, check out the "AttractionArrayWithOscillation" example with the code download.