Trig and forces: the pendulum

Do you remember reading about Newton's laws of motion from a couple sections back? We are just about ready to convert those laws into running code. After all, it's been nice learning about triangles and tangents and waves, but really, the core of this course is about simulating the physics of moving bodies. Let's take a look at how trigonometry can help us with this pursuit.

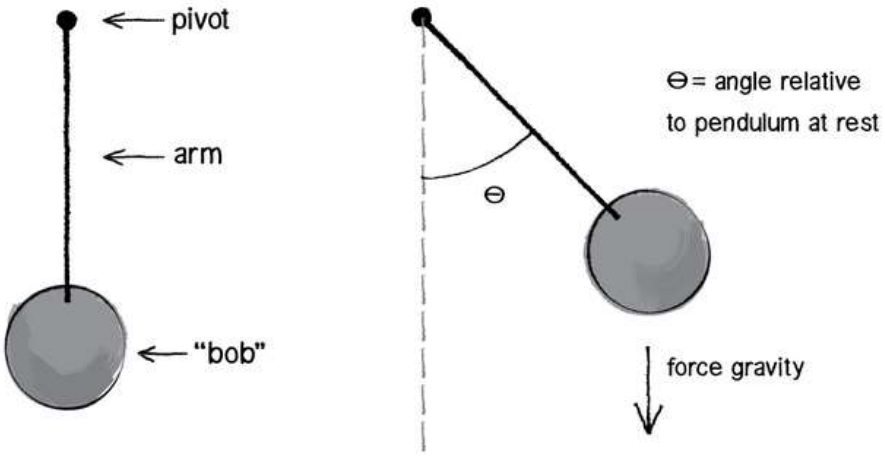


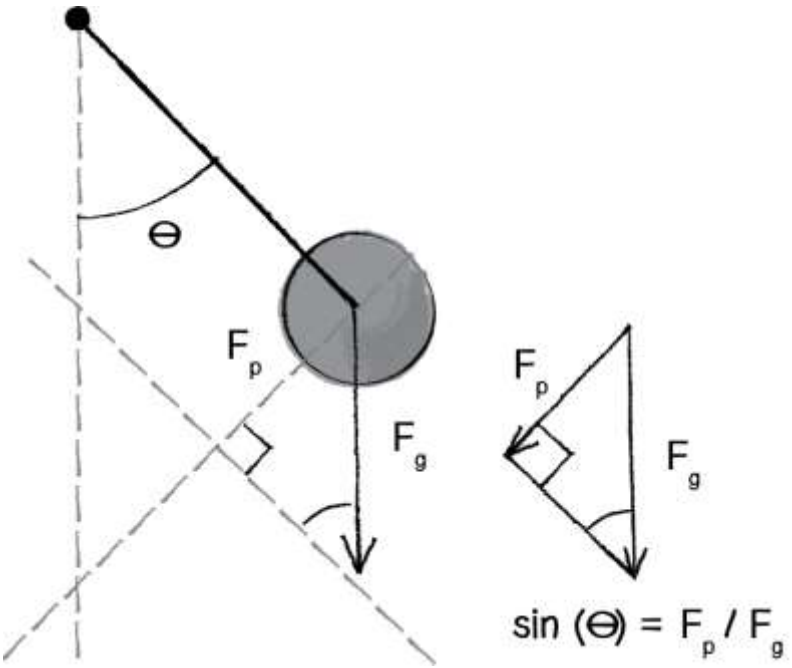
Diagram of pendulum with angles

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—the program canvas.

In the Forces section, we learned how a force (such as the force of gravity shown 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 the force at the angle of the pendulum's arm (relative to a pendulum at rest with an angle of 0).

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.



We can see that the force of the pendulum (F_pF_pF, start subscript, p, end subscript) should point perpendicular to the arm of the pendulum in the direction that the pendulum is swinging. 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. Since the force of gravity (F_gF_gF, start subscript, g, end subscript) 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:

$$sine(\theta) = \frac{F_p}{F_g}$$ *sine*(*θ*)=*FgFp*s, i, n, e, left parenthesis, theta, right parenthesis, equals, start fraction, F, start subscript, p, end subscript, divided by, F, start subscript, g, end subscript, end fraction

Therefore:

$$F_p = F_g \times sine(\theta)$$ *Fp*=*Fg*×*sine*(*θ*)F, start subscript, p, end subscript, equals, F, start subscript, g, end subscript, times, s, i, n, e, left parenthesis, theta, right parenthesis

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.

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

The good news is that with Newton's second law, we know that there is a relationship between force and acceleration, namely $$F = M \times A$$ *F*=*M*×*A*F, equals, M, times, A, or$$A = F / M$$ *A*=*F*/*M*A, equals, F, slash, M, and we can use that relationship with the formula above to figure out the angular acceleration. See if you can follow this:

Starting with:
pendulum force = force due to gravity * sine(θ)

Then we divide the right side by mass, to come up with the acceleration, based on Newton's second law:
pendulum angular acceleration = (force due to gravity * sine(θ)) / mass

Then we realize we can just divide the force due to gravity by mass, and that's the same thing as acceleration due to gravity, so we'll just substitute

that:

pendulum angular acceleration = acceleration due to gravity * sine (θ)

Ta-da! We now have a way to calculate the angular acceleration.

This is a good time to remind ourselves that we're ProcessingJS 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.

angular acceleration = gravity * sine(θ)

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 in which we remind ourselves that the purpose of the course 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 a 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` object. Let's think about all the properties we've encountered in our pendulum discussion that the object will need to keep track of:

- arm length

- angle
- angular velocity
- angular acceleration

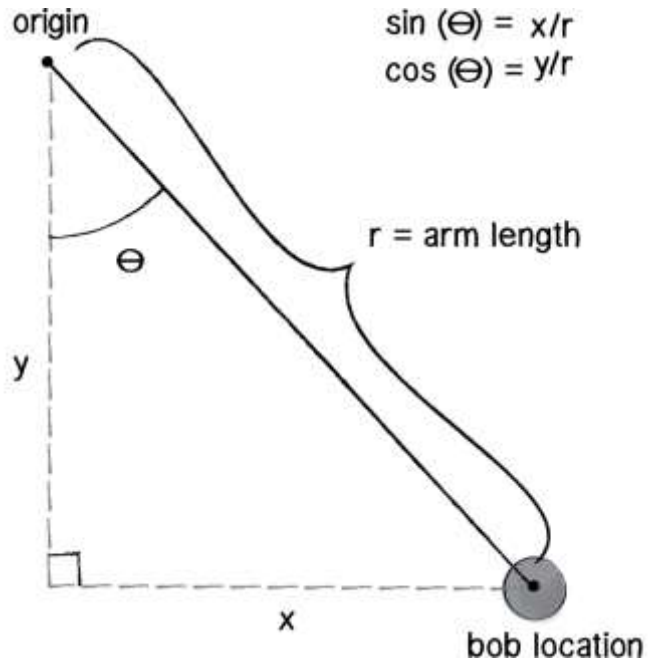
  Plus we'll also want to specify where the pendulum is hanging from, so we could start with a constructor like this:

```
var Pendulum  = function(origin, armLength) {
    this.origin = origin;
    this.armLength = armLength;

    this.angle = PI/4;
    this.aVelocity = 0.0;
    this.aAcceleration = 0.0;
};
```

We'll also need to write an `update()` method to update the pendulum's angle according to our formula…

```
Pendulum.prototype.update = function() {
    // Arbitrary constant
    var gravity = 0.4;
    // Calculate acceleration
    this.aAcceleration = -1 * gravity * sin(this.angle);
    // Increment velocity
    this.aVelocity += this.aAcceleration;
    // Increment angle
    this.angle += this.aVelocity;
};
```



…as well as a `display()` method to draw the pendulum in the window. This 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 position)? This may be getting a little tiring, but the answer, yet again, is trigonometry. Let's reference the diagram to the left.

The origin is just something we make up, as is the arm length. Let's say we construct our pendulum like so:

```
var p = new Pendulum(new PVector(100, 10), 125);
```

We're storing the current angle on the `angle` property. So relative to the origin, the pendulum's position is a polar coordinate: *(r,angle)*. And we need it to be Cartesian. Luckily for us, we spent some time in the Angles section deriving the formula for converting from polar to Cartesian. In that section, our angle was relative to the horizontal axis, but here, it's relative to the vertical axis, so we end up using `sin()` for the x position and `cos()` for the y position, instead of `cos()` and `sin()`, respectively. And

so, we can calculate the position relative to the origin using that conversion formula, and then add the origin position to it:

```
this.position = new PVector(
   this.armLength * sin(this.angle),
   this.armLength * cos(this.angle));
this.position.add(this.origin);
stroke(0, 0, 0);
fill(175, 175, 175);
line(this.origin.x, this.origin.y, this.position.x, this.position.y);
ellipse(this.position.x, this.position.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 1. 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 `armLength`. For a more involved explanation, visit The Simple Pendulum website.

```
this.aAcceleration = (-1 * gravity / this.armLength) * sin(this.angle);
```

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 did in the Forces section), 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:

```
this.aVelocity *= this.damping;
```

Putting everything together, we have the following example. We've added a bit of functionality to make it easy to drag the bob and drop it from different heights, too. Try it out!

```
// A Simple Pendulum Object
// Includes functionality for user to click and drag the pendulum

angleMode = "radians";

// This constructor could be improved to allow a greater variety of pendulums
var Pendulum  = function(origin, armLength) {
  this.origin = origin;
  this.armLength = armLength;
  this.position = new PVector();
  this.angle = PI/4;

  this.aVelocity = 0.0;
  this.aAcceleration = 0.0;
  // Arbitrary damping
```

```
    this.damping = 0.995;
    // Arbitrary ball radius
    this.ballRadius = 48.0;
    this.dragging = false;
};

Pendulum.prototype.go = function() {
    this.update();
    this.display();
};

Pendulum.prototype.update = function() {
    // As long as we aren't dragging the pendulum, let it swing!
    if (!this.dragging) {
        // Arbitrary constant
        var gravity = 0.4;
        // Calculate acceleration (see: http://www.myphysicslab.com/pendulum1.html)
        this.aAcceleration = (-1 * gravity / this.armLength) * sin(this.angle);
        // Increment velocity
        this.aVelocity += this.aAcceleration;
        // Arbitrary damping
        this.aVelocity *= this.damping;
        // Increment angle
        this.angle += this.aVelocity;
    }
};
```

```
Pendulum.prototype.display = function() {
   // Polar to cartesian conversion
   this.position = new PVector(
     this.armLength * sin(this.angle),
     this.armLength * cos(this.angle));
   this.position.add(this.origin);
   stroke(0, 0, 0);
   strokeWeight(2);
   // Draw the arm
   line(this.origin.x, this.origin.y, this.position.x, this.position.y);
   fill(175, 175, 175);
   if (this.dragging) {
      fill(0, 0, 0);
   }
   // Draw the ball
   ellipse(this.position.x, this.position.y, this.ballRadius, this.ballRadius);
};

// The methods below are for mouse interaction

// This checks to see if we clicked on the pendulum ball
Pendulum.prototype.handleClick = function(mx, my) {
   var d = dist(mx, my, this.position.x, this.position.y);
   if (d < this.ballRadius) {
      this.dragging = true;
   }
};
```

```
// This tells us we are not longer clicking on the ball
Pendulum.prototype.stopDragging = function() {
  this.aVelocity = 0; // No velocity once you let go
  this.dragging = false;
};

Pendulum.prototype.handleDrag = function(mx, my) {
  // If we are dragging the ball, we calculate the angle between the
  // pendulum origin and mouse location
  // we assign that angle to the pendulum
  if (this.dragging) {
    // Difference between 2 points
    var diff = PVector.sub(this.origin, new PVector(mx, my));
    // Angle relative to vertical axis
    this.angle = atan2(-1*diff.y, diff.x) - PI/2;
  }
};

var p = new Pendulum(new PVector(width/2, 0), 175);

var draw = function() {
  background(255);
  p.go();
};

mousePressed = function() {
  p.handleClick(mouseX, mouseY);
```

```
};

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

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