Gravitational attraction

Probably the most famous force of all is gravity. We humans on earth think of gravity as an apple hitting Isaac Newton on the head. Gravity means that stuff falls down. But this is only our experience of gravity. In truth, just as the earth pulls the apple towards it due to a gravitational force, the apple pulls the earth as well. The thing is, the earth is just so massive that it overwhelms all the gravity interactions of every other object on the planet. Every object with mass exerts a gravitational force on every other object. And there is a formula for calculating the strengths of these forces, as depicted in the diagram below:

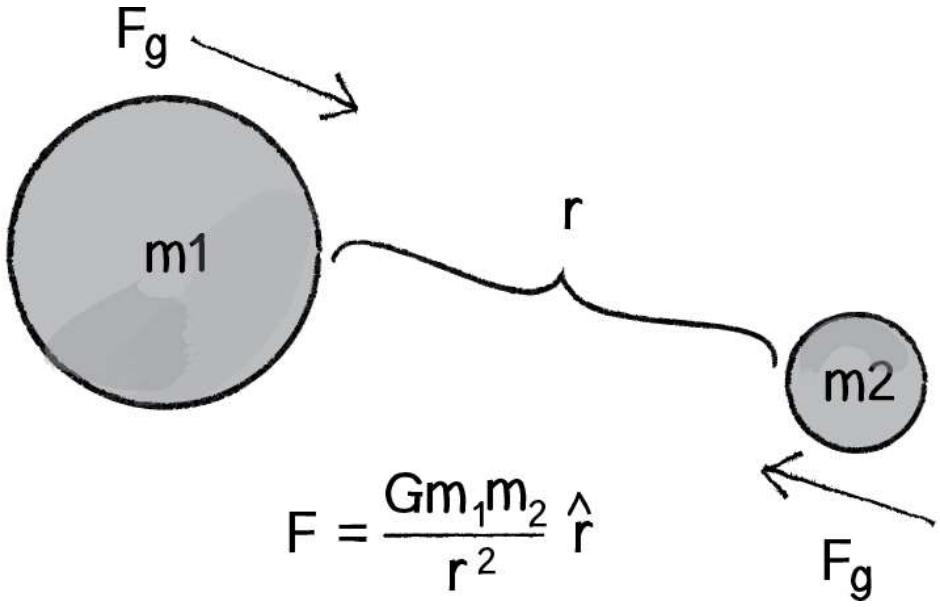


Diagram of gravitational forces between two spheres

Let's examine this formula a bit more closely.

- `F` refers to the gravitational force, the vector we ultimately want to compute and pass into our `applyForce()` function.
- `G` is the *universal gravitational constant*, which in our world equals 6.67428 x 10^-11 meters cubed per kilogram per second squared. This is a pretty important number if your name is Isaac Newton or Albert Einstein. It's not an important number if you are a ProcessingJS programmer.

Again, it's a constant that we can use to make the forces in our world weaker or stronger. Just making it equal to one and ignoring it isn't such a terrible choice either.

- m_1m_1m, start subscript, 1, end subscript and m_2m_2m, start subscript, 2, end subscript are the masses of objects `1` and `2`. As we saw with Newton's second law (\vec{F} = M\vec{A}$\vec{F} = M\vec{A}$F, with, vector, on top, equals, M, A, with, vector, on top), mass is also something we could choose to ignore. After all, shapes drawn on the screen don't actually have a physical mass. However, if we keep these values, we can create more interesting simulations in which "bigger" objects exert a stronger gravitational force than smaller ones.
- \hat{r}$\hat{r}$^ refers to the unit vector pointing from object `1` to object `2`. As we'll see in a moment, we can compute this direction vector by subtracting the location of one object from the other.
- r^2r^2r, start superscript, 2, end superscript refers to the distance between the two objects squared. Let's take a moment to think about this a bit more. With everything on the top of the formula—`G`, `m_1`, m_2m_2m, start subscript, 2, end subscript—the bigger its value, the stronger the force. Big mass, big force. Big `G`, big force. Now, when we divide by something, we have the opposite. The strength of the force is inversely proportional to the distance squared. The *farther away* an object is, the *weaker* the force; the *closer*, the *stronger*.

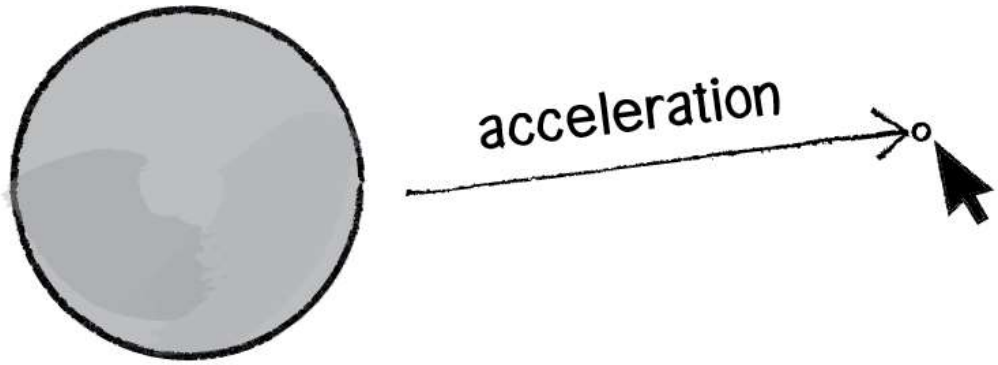
Hopefully by now the formula makes some sense to us. We've looked at a diagram and dissected the individual components of the formula. Now it's time to figure out how we translate the math into ProcessingJS code. Let's make the following assumptions.

We have two objects, and:

1. Each object has a `PVector` location: `location1` and `location2`.

2. Each object has a numeric mass: `mass1` and `mass2`.
3. There is a numeric variable `G` for the universal gravitational constant.
   Given these assumptions, we want to compute a `PVector force`, the force of gravity. We'll do it in two parts. First, we'll compute the direction of the force\hat{r}*r*^ in the formula above. Second, we'll calculate the strength of the force according to the masses and distance.

   Remember when we figured out how to have an object accelerate towards the mouse? We're going to use the same logic here.



   A vector is the difference between two points. To make a vector that pointed from the circle to the mouse, we simply subtracted one point from another:

```
var dir = PVector.sub(mouse, location);
```

   In our case, the direction of the attraction force that object 1 exerts on object 2 is equal to:

```
var dir = PVector.sub(location1, location2);
```

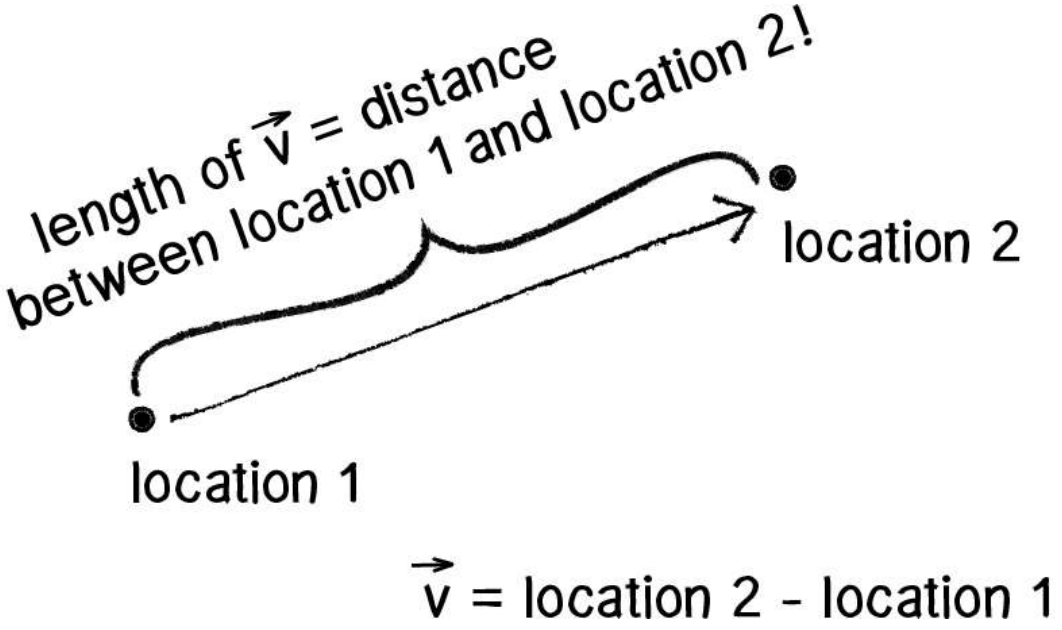
   Don't forget that since we want a unit vector, a vector that tells us about direction only, we'll need to normalize the vector after subtracting the locations:

```
dir.normalize();
```

OK, we've got the direction of the force. Now we just need to compute the magnitude and scale the vector accordingly.

```
var m = (G * mass1 * mass2) / (distance * distance);
dir.mult(m);
```

The only problem is that we don't know the distance. `G`, `mass1`, and mass2 were all givens, but we'll need to actually compute distance before the above code will work. Didn't we just make a vector that points all the way from one location to another? Wouldn't the length of that vector be the distance between two objects?



Well, if we add just one line of code and grab the magnitude of that vector before normalizing it, then we'll have the distance.

```
// The vector that points from one object to another
var force = PVector.sub(location1, location2);

// The length (magnitude) of that vector is the distance between the two
objects.
var distance = force.mag();
```

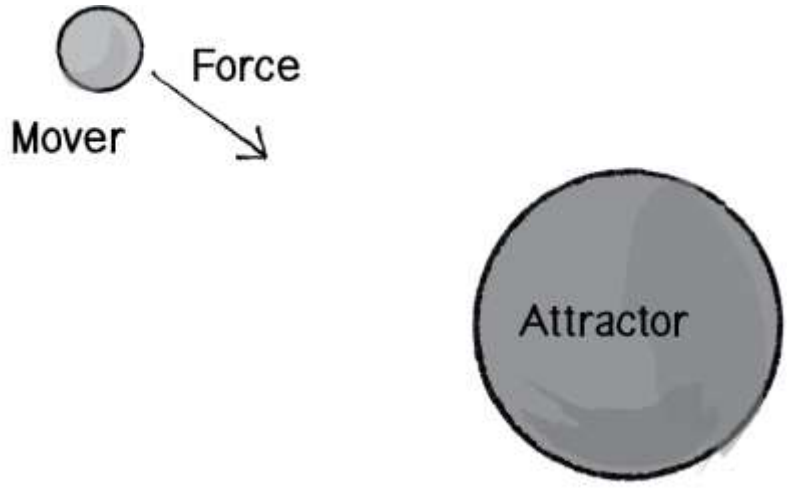
```
// Use the formula for gravity to compute the strength of the force.
var strength = (G * mass1 * mass2) / (distance * distance);

// Normalize and scale the force vector to the appropriate magnitude.
force.normalize();
force.mult(strength);
```

Note that I also renamed the `PVector` "dir" as "force." After all, when we're finished with the calculations, the `PVector` we started with ends up being the actual force vector we wanted all along.

Now that we've worked out the math and the code for calculating an attractive force (emulating gravity), we need to turn our attention to applying this technique in the context of an actual ProcessingJS program. Earlier in this section, we created a simple `Mover` object—an object with `PVector`'s location, velocity, and acceleration as well as an `applyForce()`. Let's take this exact class and put it in a program with:

- A single `Mover` object.
- A single `Attractor` object (a new object type that will have a fixed location).

  The `Mover` object will experience a gravitational pull towards the `Attractor` object, as illustrated in the diagram.



We can start by making the new `Attractor` object very simple—giving it a location and a mass, along with a method to display itself (tying mass to size).

```
var Attractor = function() {
    this.position = new PVector(width/2, height/2);
    this.mass = 20;
    this.G = 1;
    this.dragOffset = new PVector(0, 0);
    this.dragging = false;
    this.rollover = false;
};

// Method to display
Attractor.prototype.display = function() {
    ellipseMode(CENTER);
    strokeWeight(4);
    stroke(0);
    fill(175, 175, 175, 200);
    ellipse(this.position.x, this.position.y, this.mass*2, this.mass*2);
};
```

After defining that, we can create an instance of the `Attractor` object type.

```
var mover = new Mover();
var attractor = new Attractor();

draw = function() {
    background(50, 50, 50);

    attractor.display();
    mover.update();
```

```
    mover.display();
};
```

This is a good structure: a main program with a `Mover` and an `Attractor` object. The last piece of the puzzle is how to get one object to attract the other. How do we get these two objects to talk to each other?

There are a number of ways we could do this, architecturally. Here are just a few possibilities.

Task	Function
1. A function that receives both an`Attractor` and a `Mover`:	`attraction(a, m);`
2. A method in the `Attractor` object that receives a `Mover`:	`a.attract(m);`
3. A method in the `Mover` object that receives an `Attractor`:	`mover.attractedTo(a);`
4. A method in the `Attractor` object that receives a `Mover` and returns a `PVector`, which is the attraction force. That attraction force is then passed into the `Mover`'s`applyForce()` method.	`var f = a.calculateAttraction(m); mover.applyForce(f);` |

It’s good to look at a range of options for making objects talk to each other, and you could probably make arguments for each of the above possibilities. Let's start by discarding the first one, since an object-oriented approach is really a much better choice over an arbitrary function not tied to either the `Mover` or `Attractor` objects. Whether you pick option 2 or option 3 is the difference between saying “The attractor attracts the mover” or “The mover is attracted to the attractor.” Number 4 seems the

most appropriate, at least in terms of where we are in this course. After all, we spent a lot of time working out the `applyForce()` method, and I think our examples will be clearer if we continue with the same methodology.

In other words, where we once had:

```
var f = new PVector(0.1, 0); // Made up force
mover.applyForce(f);
```

We will now have:

```
var f = a.calculateAttraction(m); // Attraction force between two objects
mover.applyForce(f);
```

And so our `draw()` function can now be written as:

```
draw = function() {
    background(50, 50, 50);

    // Calculate attraction force and apply it
    var f = a.calculateAttraction(m);
    mover.applyForce(f);

    attractor.display();
    mover.update();
    mover.display();
};
```

We're almost there. Since we decided to put the `calculateAttraction()` method inside of the `Attractor` object type, we'll need to actually write that function. The function needs to receive a `Mover` object and return a PVector. And what goes inside that function? All of that nice math we worked out for gravitational attraction!

```
Attractor.prototype.calculateAttraction = function(mover) {

    // What's the force's direction?
    var force = PVector.sub(this.position, mover.position);
    var distance = force.mag();
    force.normalize();

    // What's the force's magnitude?
    var strength = (this.G * this.mass * mover.mass) / (distance *
distance);
    force.mult(strength);

    // Return the force so it can be applied!
    return force;
};
```

And we're done. Sort of. Almost. There's one small kink we need to work out. Let's look at the above code again. See that symbol for divide, the slash? Whenever we have one of these, we need to ask ourselves the question: What would happen if the distance happened to be a really, really small number or (even worse!) zero??! Well, we know we can't divide a number by 0, and if we were to divide a number by something like 0.0001, that is the equivalent of multiplying that number by 10,000! Yes, this is the real-world formula for the strength of gravity, but we don't live in the real world. We live in the *ProcessingJS world*. And in the ProcessingJS world, the mover could end up being very, very close to the attractor and the force could become so strong the mover would just fly way off the screen. And so with this formula, it's good for us to be practical and constrain the range of what distance can actually be. Maybe, no matter where the `Mover` actually is, we should never consider it less than 5 pixels or more than 25 pixels away from the attractor.

```
distance = constrain(distance, 5, 25);
```

For the same reason that we need to constrain the minimum distance, it's useful for us to do the same with the maximum. After all, if the mover were to be, say, 500 pixels from the attractor (not unreasonable), we'd be dividing the force by 250,000. That force might end up being so weak that it's almost as if we're not applying it at all.

Now, it's really up to you to decide what behaviours you want. But in the case of, "I want reasonable-looking attraction that is never absurdly weak or strong," then constraining the distance is a good technique.

Let's put it all together now in one program. The `Mover` object type hasn't changed at all, but now our program includes an `Attractor` object, and code that ties them together. We've also added code to the program to control the attractor with a mouse, so that it's easier to observe the effects.

```
// Attractor: An object type for a drag-able attractive body in our world

var Attractor = function() {
  this.position = new PVector(width/2, height/2);
  this.mass = 20;
  this.G = 1;
  this.dragOffset = new PVector(0, 0);
  this.dragging = false;
  this.rollover = false;
};


Attractor.prototype.calculateAttraction = function(mover) {
  // Calculate direction of force
  var force = PVector.sub(this.position, mover.position);
  // Distance between objects
```

```
  var distance = force.mag();
  // Limiting the distance to eliminate "extreme" results
  // for very close or very far objects
  distance = constrain(distance, 5, 25);
  // Normalize vector
  force.normalize();
  // Calculate gravitional force magnitude
  var strength = (this.G * this.mass * mover.mass) / (distance * distance);
  // Get force vector --> magnitude * direction
  force.mult(strength);
  return force;
};

// Method to display
Attractor.prototype.display = function() {
  ellipseMode(CENTER);
  strokeWeight(4);
  stroke(0);
  if (this.dragging) {
    fill(50, 50, 50);
  } else if (this.rollover) {
    fill(100, 100, 100);
  } else {
    fill(175, 175, 175, 200);
  }
  ellipse(this.position.x, this.position.y, this.mass*2, this.mass*2);
};
```

```
// The methods below are for mouse interaction

Attractor.prototype.handleHover = function(mx, my) {
  var d = dist(mx, my, this.position.x, this.position.y);
  if (d < this.mass) {
    this.rollover = true;
  } else {
    this.rollover = false;
  }
};

Attractor.prototype.handlePress = function(mx, my) {
  var d = dist(mx, my, this.position.x, this.position.y);
  if (d < this.mass) {
    debug("setting dragging to true");
    this.dragging = true;
    this.dragOffset.x = this.position.x - mx;
    this.dragOffset.y = this.position.y - my;
  }
};

Attractor.prototype.handleDrag = function(mx, my) {
  debug("should we be dragging?" + this.dragging);
  if (this.dragging) {
    this.position.x = mx + this.dragOffset.x;
    this.position.y = my + this.dragOffset.y;
  }
```

```
};


Attractor.prototype.stopDragging = function() {
   debug("setting dragging to false");
   this.dragging = false;
};
var Mover = function() {
   this.position = new PVector(400, 50);
   this.velocity = new PVector(1, 0);
   this.acceleration = new PVector(0, 0);
   this.mass = 1;
};
  Mover.prototype.applyForce = function(force) {
   var f = PVector.div(force,this.mass);
   this.acceleration.add(f);
};
  Mover.prototype.update = function() {
   this.velocity.add(this.acceleration);
   this.position.add(this.velocity);
   this.acceleration.mult(0);
};
Mover.prototype.display = function() {
   stroke(0);
   strokeWeight(2);
   fill(255, 255, 255, 127);
   ellipse(this.position.x, this.position.y, this.mass*16, this.mass*16);
};
```

```
Mover.prototype.checkEdges = function() {
    if (this.position.x > width) {
        this.position.x = width;
        this.velocity.x *= -1;
    } else if (this.position.x < 0) {
        this.velocity.x *= -1;
        this.position.x = 0;
    }
    if (this.position.y > height) {
        this.velocity.y *= -1;
        this.position.y = height;
    }
};
var mover = new Mover();
var attractor = new Attractor();
var draw = function() {
  background(50, 50, 50);
  var force = attractor.calculateAttraction(mover);
  mover.applyForce(force);
  mover.update();
   attractor.display();
  mover.display();
};

var mouseMoved = function() {
    attractor.handleHover(mouseX, mouseY);
};
```

```
var mousePressed = function() {
  attractor.handlePress(mouseX, mouseY);
};
var mouseDragged = function() {
  attractor.handleHover(mouseX, mouseY);
  attractor.handleDrag(mouseX, mouseY);
};
var mouseReleased = function() {
  attractor.stopDragging();
};
```

And we could, of course, expand this example using an array to include many Mover objects, just as we did with friction and drag. The main change that we've made to the program is to adjust our `Mover` object to accept mass, x, and y (as we've done in the past), initialize an array of randomly placed `Mover`s, and loop over that array to calculate the attraction force on each of them, each time:

```
var movers = [];
var attractor = new Attractor();

for (var i = 0; i < 10; i++) {
    movers[i] = new Mover(random(0.1, 2), random(width), random(height));
}

draw = function() {
    background(50, 50, 50);

    attractor.display();
    for (var i = 0; i < movers.length; i++) {
        var force = attractor.calculateAttraction(movers[i]);
```

```
        movers[i].applyForce(force);

        movers[i].update();
        movers[i].display();
    }
};

// Attractor: An object type for a drag-able attractive body in our world
var Attractor = function() {
  this.position = new PVector(width/2, height/2);
  this.mass = 20;
  this.G = 1;
  this.dragOffset = new PVector(0, 0);
  this.dragging = false;
  this.rollover = false;
};
Attractor.prototype.calculateAttraction = function(mover) {
  // Calculate direction of force
  var force = PVector.sub(this.position, mover.position);
  // Distance between objects
  var distance = force.mag();
  // Limiting the distance to eliminate "extreme"
  // results for very close or very far objects
  distance = constrain(distance, 5, 25);
  // Normalize vector
  force.normalize();
  // Calculate gravitional force magnitude
  var strength = (this.G * this.mass * mover.mass) / (distance * distance);
```

```
  // Get force vector --> magnitude * direction
  force.mult(strength);
  return force;
};

// Method to display
Attractor.prototype.display = function() {
  ellipseMode(CENTER);
  strokeWeight(4);
  stroke(0);
  if (this.dragging) {
    fill(50, 50, 50);
  } else if (this.rollover) {
    fill(100, 100, 100);
  } else {
    fill(175, 175, 175, 200);
  }
  ellipse(this.position.x, this.position.y, this.mass*2, this.mass*2);
};

// The methods below are for mouse interaction

Attractor.prototype.handleHover = function(mx, my) {
  var d = dist(mx, my, this.position.x, this.position.y);
  if (d < this.mass) {
    this.rollover = true;
  } else {
```

```
        this.rollover = false;
    }
};

Attractor.prototype.handlePress = function(mx, my) {
    var d = dist(mx, my, this.position.x, this.position.y);
    if (d < this.mass) {
        debug("setting dragging to true");
        this.dragging = true;
        this.dragOffset.x = this.position.x - mx;
        this.dragOffset.y = this.position.y - my;
    }
};

Attractor.prototype.handleDrag = function(mx, my) {
    debug("should we be dragging?" + this.dragging);
    if (this.dragging) {
        this.position.x = mx + this.dragOffset.x;
        this.position.y = my + this.dragOffset.y;
    }
};

Attractor.prototype.stopDragging = function() {
    debug("setting dragging to false");
    this.dragging = false;
};
var Mover = function(mass, x, y) {
```

```
  this.position = new PVector(x, y);
  this.velocity = new PVector(1, 0);
  this.acceleration = new PVector(0, 0);
  this.mass = mass;
};

Mover.prototype.applyForce = function(force) {
  var f = PVector.div(force,this.mass);
  this.acceleration.add(f);
};

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

Mover.prototype.display = function() {
  stroke(0);
  strokeWeight(2);
  fill(255, 255, 255, 127);
  ellipse(this.position.x, this.position.y, this.mass*16, this.mass*16);
};

Mover.prototype.checkEdges = function() {
  if (this.position.x > width) {
    this.position.x = width;
```

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

var movers = [];
var attractor = new Attractor();

for (var i = 0; i < 10; i++) {
    movers[i] = new Mover(random(0.1, 2), random(width), random(height));
}

var draw = function() {
    background(50, 50, 50);

    attractor.display();
    for (var i = 0; i < movers.length; i++) {
        var force = attractor.calculateAttraction(movers[i]);
        movers[i].applyForce(force);

        movers[i].update();
```

```
      movers[i].display();
    }
};
var mouseMoved = function() {
    attractor.handleHover(mouseX, mouseY);
};

var mousePressed = function() {
    attractor.handlePress(mouseX, mouseY);
};

var mouseDragged = function() {
    attractor.handleHover(mouseX, mouseY);
    attractor.handleDrag(mouseX, mouseY);
};

var mouseReleased = function() {
    attractor.stopDragging();
};
```