CS 116x Winter 2015
Craig S. Kaplan

Module 04
Physics and Animation

Topics

- Newton's first law: moving things at constant speed (in one dimension)
- Newton's second law: applying forces
- Newton's third law: handling collisions
- Two-dimensional physics
- Using a physical simulation library
- Animation principles and easing

Readings

- *Getting Started*, Pages 91–96
- Imaginary Institute Tech Note 04, up to but not including "Using the Curves"
- Nature of Code, Chapters 1 and 2 (Chapter 5 is relevant too, but we'll use a different library)

Introduction

Physics? Haven't we suffered enough?

A basic intuition for natural (i.e., physically plausible) motion is extremely useful in many digital design contexts. As computation becomes cheaper, and we're able to pack more visually appealing effects into apps and operating systems, we're seeing more visual effects that could be interpreted as "physical". Here are a few examples:

- In a lot of touch-based interfaces (like mobile phones), scrolling doesn't stop dead at the end of a

document. Instead, you're able to "overshoot" the end of the document slightly. When you let go, the document "bounces back" to its end position as if on a spring. Touchpad scrolling is similar on many laptops.
- When you launch an application from the OSX Dock, or when a notification appears, the application's icon appears to bounce up and down in the dock in a physically plausible way.
- Websites play lots of CSS tricks to embed simple physics into browsing. For example, when you expand a tweet or the comments of a Google Plus post, the expansion starts slow, speeds up, and then slows down again when it's near completion. Here's a simple example.
- Many popular games rely directly on simple models of physics. In 2D, the most famous example is Angry Birds.

How can we explain the seeming enthusiasm for physical systems in games and user interface? There's certainly no need for it—it's just as easy, if not easier, to simulate motion on the screen that violates the laws of physics (teleportation, for starters).

Every physical object in the real world must obey the laws of physics[citation needed], and so your perception of motion is influenced at a very deep level by what's physically plausible. I would argue, then, that natural motion has greater appeal, looks more "organic", and is easier and faster to interpret. It also can provide us with subtle visual cues. When an object moves with acceleration and deceleration, we can anticipate the extent and duration of that motion. These same principles have formed the core of effective animation for nearly a hundred years.

Example sketch: Anticipation

At the outset, it may seem like we're in for a significant challenge in simulating the laws of physics. But there are a few reasons for hope:

- You have a lifetime of direct experience of physical laws, whether you're aware of it or not. Your intuition

will be a powerful aid in knowing whether your code is correct.
- The amount of physics we actually need is minimal. Newton's Three Laws of Motion account for basically all of our everyday experience of dynamics, and we don't even need to obey them perfectly. No quantum mechanics or relativity here!
- Computation (that is, simulation) takes a lot of the math out of physics. This will become more clear as we work our way through simple examples. If I throw a ball in the air, it take a bit of math to figure out how high it will go before it falls back down. But I can just simulate the ball's motion moment by moment (or frame by frame in Processing), and watch it find its peak on its own.
- Once we work through the basics of physical simulation, we'll simply hand the problem over to a library that will handle it much better than we ever could. At that point, physics becomes "just another feature" of Processing.

Newton's laws of motion

To a first approximation, the kinds of physical motion matter in computer software are a direct expression of *Newton's Laws of Motion*. Here they are, translated into English:

I. Every body persists in its state of being at rest or of moving uniformly straight forward, except insofar as it is compelled to change its state by force impressed.

II. The alteration of motion is ever proportional to the motive force impress'd; and is made in the direction of the right line in which that force is impress'd.

III. To every action there is always opposed an equal reaction: or the mutual actions of two bodies upon each other are always equal, and directed to contrary parts.

The most important thing Newton left out is *relativity*, which Einstein proposed in the early 1900s. Einstein's equations really do make Newton's wrong. But the amount of error is so tiny, and so irrelevant in everyday life, that we can ignore it completely except in very special circumstances.

Those are still pretty complicated statements. Let's take them one by one, simplify them, and see how they can be modelled with programming. In our first pass at applying these laws, we'll work in a one-dimensional universe: objects are constrained to move along a line.

The first law

The first law is usually written "An object in motion tends to remain in motion. An object at rest tends to remain at rest." Or, more succinctly, "Keep doing what you're doing".

Simulating an object at rest is easy: keep drawing it in the same location every frame. What about an object in motion? Let's assume, as Newton did for this law, that there are no forces that act on the object (no gravity, no friction, no wind, no built-in engine, etc.). This object will experience *uniform motion*: it will move equal distances in equal amounts of time. If we choose a unit of distance (inches, metres, furlongs) and a unit of time (seconds, hours, fortnights), we can say that the object is moving at a *constant speed*: some fixed number of distance units per time unit.

For simplicity, in Processing I will measure distance in *pixels* and time in *frames* (assuming my sketch runs at a roughly constant frame rate). I might then reckon speed in *pixels per frame*. In that case, how does an object's position change every time I draw a frame?

Assuming a constant frame rate can be dangerous. If the scene gets very complex, or the computer slows down, or internet bandwidth drops out in a multiplayer game, you'd like for the simulation not to lose track of time. For that, you could switch to a more complex *implicit* method. Instead of looking at the current frame number, use the current *time* (via the `millis()` function in Processing) to decide where to put everything. That's too tricky for this course, especially when we start to take forces into account.

```
float position = 0;
float speed = 3.77;

void setup()
{
  size( 800, 100 );
}

void step()
{
  position = ???
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( position, 50, 50, 50 );
}
```

Example sketch: FirstLaw

We can generalize from this example to a principle for physical simulation: *at every step, use speeds to update positions.* In light of this approach, it's also clear that "at rest" is just a special case of "in motion", in which we have `speed == 0.0`.

The second law

As far as I know, Newton's second law doesn't really have an informal statement in English. Instead, it's usually expressed as an equation, one of the most important in all of physics: $F=ma$. That is, force is the product of mass and acceleration.

In practice, it might be better to divide both sides by m, ending up with $a=F/m$. Written this way, the equation tells us that if a force is applied to an object, the force produces acceleration. The amount of

acceleration depends on the object's mass. The same force will produce less acceleration when applied to a heavy object than to a light object (you can kick a soccer ball farther than you can kick a bowling ball).

For now, let's ignore mass. We're only dealing with one object, and it turns out that an object's mass becomes much more important when it interacts (i.e., collides) with other objects. For now, then, we'll say that force determines acceleration directly. Let's look at some simple examples of how that works out in code.

Gravity

We'll start with a force we're all familiar with, and that few humans have escaped: gravity. Every bit of stuff in the universe pulls on every other bit with a tiny amount of gravitational force, but we're only interested in the simplest example: the earth's gravitational force on things near its surface.

The real equations that govern your gravitational relationship to the earth aren't worth writing down here. If we *did* write them down, we would quickly discover two important facts:

1. Your acceleration due to gravity doesn't depend on your mass. That is, a feather and a bowling ball will fall at the same rate.
2. The earth is so big, and you are so small, that as long as your altitude stays roughly constant, we can pretend you experience a constant force.

Putting that together, we arrive at a bit of physics that you may remember from high school: everything accelerates towards the earth at 9.8 m/s^2.

What's that in pixels per frame squared? That is, how do I convert distances and times to the correct scale for a Processing sketch? There's no one right answer to that question. If I draw a sketch that shows a circle dropping to the bottom of the window, I would choose a different acceleration if the circle is a meteorite crashing to the ground versus a ball bearing dropping onto a table. The best advice is to try a few

values and decide what looks best for your application.

How do we actually incorporate acceleration into a Processing sketch? Acceleration is nothing more than the rate at which speed changes. Recalling from the First Law that speed is the rate at which position changes, this suggests that we perform a similar update. Thus we arrive at a second principle for physical simulation: *at every step, use accelerations to update speeds*. This kind of step-by-step update turns out to be very easy to express in code.

```
float position = 0;
float speed = 0;
float accel = 0.2;

void setup()
{
  size( 100, 600 );
}

void step()
{
  // Update speed using acceleration
  speed = ???
  // Update position using speed
  position = ???
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( 50, position, 50, 50 );
}
```

Example sketch: SecondLaw

This code is really a general model for objects that experience any kind of constant acceleration. Gravity is just the most obvious kind.

Sliding friction

A hockey puck sliding on ice will eventually slow down (i.e., decelerate) and stop. It experiences friction with the ice surface—in every small moment of time, the puck loses some of its speed, as a bit of its energy is turned into heat.

The low-level mechanism of friction is complex. For our purposes, what matters is that friction produces deceleration, and that the amount of deceleration is proportional to the object's current speed. We therefore model sliding friction using a kind of *damping factor*.

```
float position = 0;
float speed = 10.0;

// Proportion of speed lost
float damping = 0.02;

void setup()
{
  size( 600, 100 );
}

void step()
{
  // Lose some speed due to friction
  speed = ???
  // Update position using speed
  position = ???
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( position, 50, 50, 50 );
}
```

Example sketch: SlidingFriction

Sliding friction becomes even more complicated as an object gets very close to stopping. It would probably be best to include some code of the form “when the speed goes below a minimum threshold, set it to zero”. (Exercise: do it!)

Springs

We use a simple model for an idealized spring, which captures a lot of the high-level behaviours of springs in the real world. A spring has a “rest length”: the length it wants to have. If you stretch it or squeeze it to some other length, it exerts a force to try to get back to its rest length. The force is proportional to how far the spring is from its rest length. (If you’ve ever tried a “bungee run” at a carnival, you know that each step is harder than one before it.) This simplified model is called Hooke’s Law, and is usually written in the form $F = -kx$. The constant k is a physical property of the spring called its *stiffness*. Remembering that we’re still ignoring mass, we can translate this law directly into a new simulation.

```
float position = 800;
float speed = 0.0;

float rest_length = 400.0;
float stiffness = 0.003;

void setup()
{
  size( 800, 100 );
}


void step()
{
  float disp = position - rest_length;
  speed = ???
  position = ???
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( position, 50, 50, 50 );
}
```

Example sketch: Spring

This spring never loses energy. It will oscillate back and forth forever, which looks unrealistic. We can model the spring slowly losing energy using the same damping calculation we used in the case of sliding friction.

Example sketch: DampedSpring

The third law

Newton’s third law is apparently about “the mutual actions of two bodies upon each other”. In most simple cases, that refers to *collisions*.

Modelling collisions is quite complicated. Even the collision of two circles in 2D can be hard to wrap your head around (if the game of Pool is any indication). Plus, to model collisions accurately we’d also need to start taking the masses of objects into account. We’re not going to try to implement that much complexity ourselves. But let’s have a look at one simple type of collision that we can simulate relatively easily: we’ll simulate a ball that bounces off the bottom of the screen.

When the ball meets the virtual floor, Newton’s law tells us that the ball imparts a force upon the floor, and the floor imparts an opposite force upon the ball. In theory, we’d have to simulate the floor accelerating downward away from the ball as a result! In practice, we effectively assume that the floor has infinite mass, so that it can never be made to move. As a result, the floor transfers all of the ball’s energy back into it, and the ball leaves with its speed reversed.

```
float position = 0;
float speed = 0;
float accel = 0.4;

void setup()
{
  size( 100, 600 );
}

void step()
{
  speed = ???
  position = ???

   if( position+25 > height ) {
    position = ???
    speed = ???
  }
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( 50, position, 50, 50 );
}
```

This code doesn't quite work: the ball won't return exactly to its original height. The main problem is that we're not detecting the exact moment of collision. There are a few ways to correct that, but they're fussy and not worth exploring here. In any collision code you implement, it's fine to use the simple speed-reversing method. If you want to see a fuller solution, look at the **ThirdLawAccurate** sketch.

Example sketch: ThirdLaw

As with springs, we usually factor in a damping factor so that at each bounce, the ball loses some of its energy.

In summary, there are five main steps involved in running a physics simulation of this kind:

1. **Calculate all the forces on each object.**
2. **Use the forces to update the speeds.**
3. **Use the speeds to update the positions.**
4. **Process any collisions.**
5. **Draw the current state of the world.**

There isn't necessarily one correct order in which to run these steps (theoretically, if the time difference between frames is small enough, it probably won't matter).

Two-dimensional dynamics

Obviously, there's only so much you can do that's visually interesting in one dimension. For the purposes of creating richer sketches, we should graduate to (at least) two dimensions.

In two dimensions, the position of an object (say, a circle) will be given by two numbers (*px*,*py*). It turns out that a similar pairing of *x* and *y* coordinates suffices to capture speed and acceleration too, with one small change: we usually speak of *velocity* rather than speed. Velocity is a *vector*, an arrow of some length that points in some direction. In two dimensions and higher, *speed* is the length of this arrow. Acceleration also becomes a vector.

We can actually get pretty far with just this simple reformulation. In fact, you can almost squint and ignore the vectors, and just pretend that you're running two separate simulations, one in *x* and one in *y.* With that philosophy, we might produce this revised sketch to demonstrate the First Law:

```
// Position
float px = 0;
float py = 0;
// Velocity
float vx = 4.0;
float vy = 5.3;

void setup()
{
  size( 500, 500 );
}

void step()
{
  px = ???
  py = ???
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( px, py, 50, 50 );
}
```

Example sketch: FirstLaw2D

Similarly, with the Second Law we can treat a force in terms of independent components that affect an object's x and y velocity components separately. For example, a thrown projectile will have its vertical position and velocity affected by gravity, while its horizontal speed will stay constant.

```
// position
float px = 50;
float py = 550;
// velocity
float vx = 2.0;
float vy = -15.0;
// constant acceleration
float ax = 0.0;
float ay = 0.3;

void setup()
{
  size( 400, 600 );
}

void step()
{
  // Update velocity using acceleration
  vx = ???
  vy = ???
  // Update position using velocity
  px = ???
  py = ???
}

void draw()
{
  // Run the physical simulation for
  // one step
  step();

  // Draw the world
  background( 80 );
  ellipse( px, py, 50, 50 );
}
```

Example sketch: SecondLaw2D

In a simple example like this one, it's also not too hard to implement collisions off all four walls of the sketch. (Exercise: do it!)

Things start to get more complicated when we attempt to deal with the Third Law, and handle 2D collisions. There are two main problems:

1. Calculating the effect of a general collision on the velocities of two bodies involves a lot of messy

geometry. Even a ball bouncing off of a wall that isn't horizontal or vertical uses some tricky math. Two general moving objects is even harder, but that's well beyond the scope of this course!

2. In one dimension, an object can collide only with the objects immediately to its right and its left. In two dimensions, there are many more ways that objects can collide. If you've got a bunch of objects floating around in your simulation, in theory you can test every pair of objects every frame for collisions. But that requires a lot of computation, and the amount goes up disproportionately with the number of objects (a computer scientist would call this a "quadratic time algorithm"). There are some very clever algorithms and data structures for organizing a scene to check for collisions efficiently, but that's well beyond the scope of this course!

And beyond that, there's a completely separate mode of rigid motion that we haven't talked about at all yet: rotation. Objects can spin in 2D. Spinning has its own versions of velocity and acceleration, and interacts with the three laws as we originally wrote them (for example, a collision can set a object spinning). It's safe to say, then, that we've reached the end of what can comfortably be accomplished by coding physics from first principles. It's time to bring out the big guns.

Physics engines

A physics engine is a library that keeps track of all the positions, velocities, accelerations, collisions, friction, and other properties of a simulated world, allowing you to inject a bunch of objects and then run the simulation step by step. There are many physics engines available to programmers. They are a big deal in the entertainment industry. Many 2D and 3D games rely on realistic physics, and many movies simulate physics as an essential part of visual effects sequences. In fact, in 2015 the authors of the open-source 3D physics engine Bullet Physics won a

> HCI researchers have also experimented with physics engines in desktop interfaces. Be sure to watch the famous "BumpTop" demo video.

technical academy award for their contribution to filmmaking.

There are several physics engines available in Processing. We're going to use one called Fisica. Fisica is a Processing "wrapper" around the Java library JBox2D. JBox2D is itself a Java translation of an earlier C++ library Box2D, a very popular 2D physics library perhaps most famously used in Angry Birds.

At this point, the task is to learn the coding practices of yet another library. First, make sure you have Fisica installed (via "Add Library..." in Processing). Now you can follow the recipe to add physical simulation to your sketch. The process will look a bit like what we did for ControlP5. First, tell Processing that you want to use Fisica in this sketch:

```
import fisica.*;
```

Next, declare a global variable that will represent the entire physics engine. The relevant type is `FWorld`.

```
FWorld world;
```

In the `setup()` function, we can now initialize the world and start adding objects to it. In ControlP5, we did so by sending "add" messages to a global object that I called `cp5`. With Fisica we'll create new objects directly using the magic word `new` in Processing. Fisica objects are all variations on the type `FBody`. We can then ask the world object to take ownership of the new body.

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

  Fisica.init(this);

  world = new FWorld();

  FBody circle = new FCircle( 50 );
  circle.setPosition( width/2, 25 );
  world.add( circle );
}
```

So far this sketch doesn't do anything. We still need to add a `draw()` function. When using the Fisica library, the draw() function will usually do two things: run the simulation for one time step, and then draw the world:

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

  world.step();
  world.draw();
}
```

Note that we never draw anything ourselves—we've basically handed control of the world over to Fisica. The library needs to know about all the objects in the world in order to track their dynamical properties, so we may as well allow the library to draw the world for us too.

As with ControlP5 Controllers, you can set a bunch of properties on Fisica bodies: physical properties like position and velocity, and also rendering properties like strokes and fills:

```
FBody circle = new FCircle( 50 );
circle.setPosition( width/2, 100 );
circle.setVelocity( 0, -200 );
circle.setFillColor( #406080 );
circle.setNoStroke();
```

Plus, there are some handy convenience functions for doing things like adding bounding walls to the sketch:

```
// Add a box around the world.
world.setEdges();

// But remove all but the bottom edge.
world.remove(world.left);
world.remove(world.right);
world.remove(world.top);
```

There are a few other standard objects that can be added to a Fisica world, most obviously boxes (`FBox`) and polygons (`FPoly`). And there are many properties that can be set both on bodies and on the world itself, to control forces like gravity, friction, restitution (energy recovery after a collision), and damping.

Fisica assumes by default that objects you create are subject to direct manipulation (you can pick them up and move them manually, temporarily "overriding" the laws of physics. Of course, you can disable that for whichever objects you want. Conversely, you can check manually whether a given point in the sketch is inside any objects in the world, which allows you to attach things like click events to bodies.

The best way to learn about these features is to browse the online documentation for Fisica and to look at many examples. The library itself comes with a set of standard examples; you can click on the icons on the Fisica web page to see the sketches in action and look at the source code for each one (a good starting point is "Buttons"). There are a few additional Fisica-related sketches at openprocessing.org. Fisica includes features we can't spend time on, like joints, chains, and even some blobby fluid objects. You can also track collisions and decide what to do when they happen. It's possible to create very elaborate (and *very* satisfying) sketches using this kind of physical simulation, up to and including programs with the complexity of Angry Birds. I encourage you to play around to discover more.

I'm probably lying—most likely, the demos won't run in your browser, because it's becoming increasingly difficult to launch Java applets in web browsers. If the demos don't work, click on "source code" and copy the demo's source code into a new Processing sketch. Most of them will work that way. You should also have the example sketches in the "libraries" sub-folder of your Processing sketchbook, though that might be hard to find.

Animation principles and digital design

Old animations (from the 1920s and 1930s) look pretty strange to our eyes. You can find lots of examples on YouTube—look for cartoons by Max Fleischer and Ub Iwerks, for example. While animators had figured out the basic illusion of motion, they hadn't yet developed the aesthetic conventions that make animation look natural and appealing.

On the other hand, I'm really intrigued by Cuphead, a forthcoming videogame with a visual style inspired by Fleischer.

It was Walt Disney (i.e., the actual person) who first codified principles of effective animation in the 1930s. A good reference for the ideas that came out of that studio is *Disney Animation: The Illusion of Life* by Thomas and Johnson (AKA "Frank and Ollie").

Computer animation went through a similar coming-of-age. In this case it was John Lasseter of Pixar who first discussed the use of Disney principles in computer animation in his paper "Principles of traditional animation applied to 3D computer animation". The effect of his philosophy was apparent in early Pixar shorts like Luxo, Jr.

The standard principles of effective computer animation are a beautiful intersection between computation and visual aesthetics. We don't have time to cover them all, but let me single out a few important ones.

Squash and Stretch

An object should deform to suggest the character of its motion. If we are animating a baseball being pitched and then hit, we might *stretch* the ball as it's flying through the air to emphasize its speed, and *squash* it at the moment it hits the bat to show its reaction to the bat's force. We exaggerate these effects to make them visible, but the result is appealing.

Off the top of my head, I'd say that stretch simulates the motion blur of a fast-moving object. Squash is real: a hit baseball *really does* deform briefly, as the back of the ball keeps moving even as the front is stopped by the bat.

Squash and stretch are useful to know about and to watch out for, but they're fairly difficult to implement

effectively: deforming objects is much harder than moving them around. I've created a hacked-together demonstration, but let's not stop to discuss how it works.

Example sketch: SquashStretch

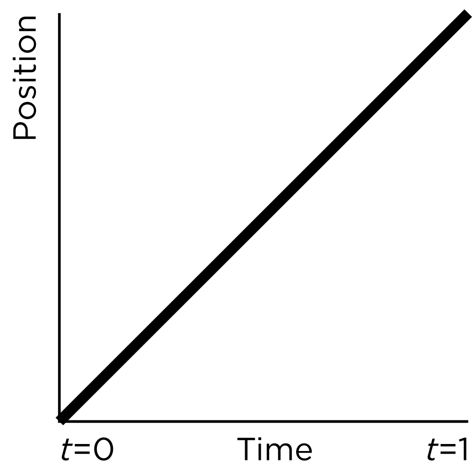
Slow in, slow out

Remember Processing's `lerp()` function for linear interpolation? When trying to animate the motion of an object from A to B, it's tempting to use straightforward linear interpolation at all intermediate positions. Assume that motion is controlled by a simple float variable `t` that ranges from 0 to 1.

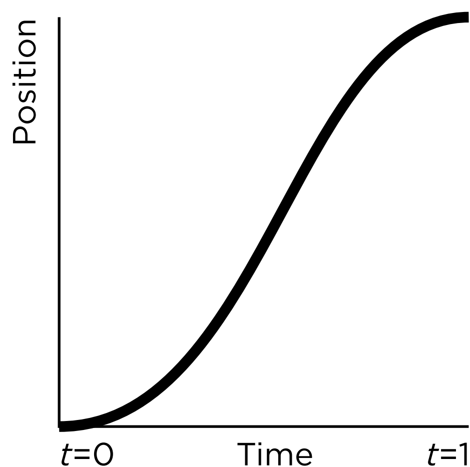
```
void draw()
{
  // Assume that you're moving a circle
  // from (startX,endX) to (startY,endY),
  // and t is a parameter value between
  // 0 and 1.
  ellipse(
    lerp( startX, endX, t ),
    lerp( startY, endY, t ),
    50, 50 );
  t = t + 0.01;
}
```

But that's not how objects move in the real world. An object must necessarily *accelerate* out of a resting position—it can't just start moving instantaneously. Similarly, objects don't stop dead when they reach their destinations: they must decelerate to a stop. Animators refer to this phenomenon as "slow in, slow out": when drawing intermediate frames in the animation of a motion, you don't space the positions of the object evenly for even differences in time. This is an animator's intuitive way of representing acceleration and deceleration. Slow in and out can make motion look more natural; and as I said at the start of these notes, it also allows us to form intuitive judgments about the character and endpoint of a physical motion.

When programmers transform the simple linear interpolation between two values of a variable (whether it's a point's *x* position, an object's size, transparency, etc.), they refer to the process as *easing*. There are many popular easing curves that achieve different aesthetic effects. Imagine drawing a graph showing time on the horizontal axis and an object's position on the vertical. Here's what simple linear interpolation looks like:



On the other hand, a slow in, slow out curve might look more like this:



See how the curve is "horizontal" at the start and end, giving it an overall S shape? That's the defining

characteristic of slow in, slow out. It guarantees that the object will appear to accelerate and decelerate.

It isn't necessarily obvious how to *define* curves that happen to have that shape. We won't try to do that in this course but we will at least try to use them. The trick is to "warp time": pass the value `t` defined above through the function that defines the easing curve before lerping in the usual way. In full generality, we might end up with code like this:

```
float ease( float t )
{
  // Transform t through some mathematical
  // function and return the new value.
  return ???
}

void draw()
{
  float et = ease( t );

  ellipse(
    lerp( startX, endX, et ),
    lerp( startY, endY, et ),
    50, 50 );
  t = t + 0.01;
}
```

The most common form of easing is "cubic easing" (for reasons you don't have to care about):

```
float ease( float t )
{
  return -2*t*t*t + 3*t*t;
}
```

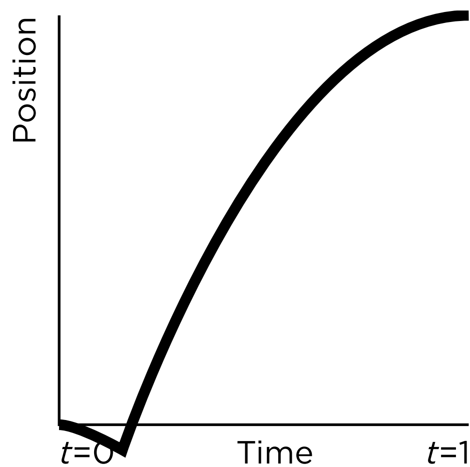
There are other easing curves, but I won't discuss their mathematical properties here. If you're interested, I've included a sample sketch that demonstrates some of them. You can also see a lot more of them in action at easings.net (a demonstration of many easing curves in Javascript).

Example sketch: Easing

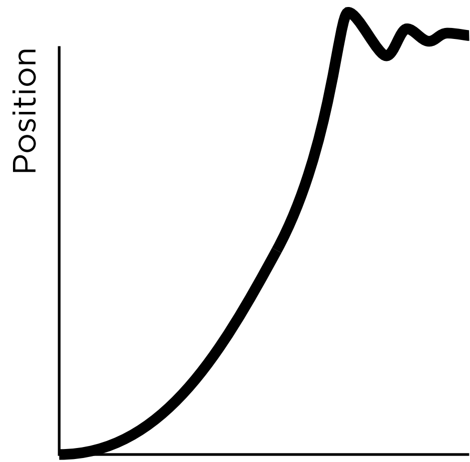
Anticipation and follow-through

Pure acceleration and deceleration aren't the only possible ways to begin and end a motion. In fact, they're a bit artificial, because objects rarely exhibit such perfect adherence to the laws of physics. Giving motions a bit more character can also help communicate the nature of a motion even more clearly than slow in, slow out.

For example, living beings usually "anticipate" motion. There's some kind of preparation or wind-up. Even with straight-line motion, a bit of anticipation can add a lot, and the initial backwards jerk can alert the user that something interesting is about to happen:



At the other end of the curve, it's hard for living beings to hit a target exactly. There's often some kind of overshooting, followed by some kind of correction. It's sometimes effective to simulate this behaviour using a kind of "bounce" easing curve:



Finally, we sometimes combine easing curves to obtain different effects at the start and end of a motion. This would correspond to gluing together the left half of one curve with the right half of another.