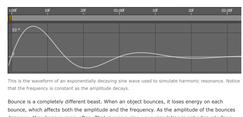
Realistic Bounce and Overshoot

Goldfish don't bounce.
—Bart Simpson

Sometimes you may want to embellish your animated motions with a bit of physical realism. For example, you may have a layer quickly scale up from zero to 100 percent and you want to add a little overshoot and oscillation and have it finally settle in at 100 percent. Another example would be if you have an object falling into frame and you want it to bounce a little when it hits bottom. These two scenarios seem similar, but they represent very different physical processes. Either of these simulations can be created with expressions, but it is important to pick the correct one. In this article, I'll cover these animation tools in detail and present some tips on how and when to apply them.

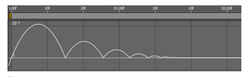
Bounce vs. Overshoot

In both bounce and overshoot scenarios, you are dealing with decaying amplitude. With overshoot, you are basically dealing with a harmonic oscillation, like you would have with a pendulum or a spring. That means the frequency remains the same (at the resonant frequency of the object) as the amplitude decays. This is often simulated with an exponentially decaying sine wave. It's a straightforward solution, although there is definitely a trick to getting the initial amplitude to match up correctly with the incoming animation).



This is the waveform of an exponentially decaying sine wave used to simulate harmonic resonance. Notice that the frequency is constant as the amplitude decays.

Bounce is a completely different beast. When an object bounces, it loses energy on each bounce, which affects both the amplitude and the frequency. As the amplitude of the bounces decrease, they happen more often. That means a sine wave simulation is not adequate for a bounce. In fact, the bounce waveform is actually a series of parabolas of decreasing amplitude. The object stops at the top of the bounce and then accelerates (due to the force of some gravity-like phenomenon), so the math involved is completely different.



This waveform is generated by a bounce simulation expression. Notice that the bounces occur more frequently as the object loses energy.

You might be tempted to simulate this bounce behavior by taking the absolute value of the oscillating sine wave (using the JavaScript `Math.abs()` function) and linking the frequency variable to a slider which you would keyframe to speed up. Don't do it. If you're tempted, please refer to my *Expression Speed and Frequency Control* article to see what's involved in doing it correctly.

Overshoot in Detail

Take a look at the basic expression for an exponentially decaying sine wave:

```
amp = 80;
freq = 1;
decay = 1;

t = time - inPoint;
amp*Math.sin(t*freq*Math.PI*2)/Math.exp(t*decay);
```

As it sits, this expression will trigger a decaying sine wave oscillation at the layer's In Point. The first three lines just set the parameters for the waveform: maximum amplitude of 80, frequency of one oscillation per second, and a decay (how fast the amplitude diminishes) value of one. Variable `t` is used to calculate the time since the layer's In Point. All the real math happens in the last line.

There are three things going on in the last line. The `Math.sin()` piece generates a sine wave of frequency `freq` that varies in amplitude between plus and minus one. The `Math.exp()` piece generates a curve that increases exponentially at a rate determined by variable `decay`. The sine wave gets multiplied by the amplitude variable (`amp`) and that result gets divided by the value of the exponential curve, resulting in the desired exponentially decaying sine wave.

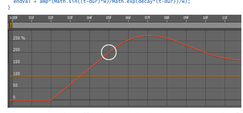
That's a handy expression, and sometimes it's all you need. More often though, you'll want to use the exponentially decaying sine wave to provide some oscillating overshoot at the end of another motion. The trick is to get the amplitude of the overshoot oscillation to match the incoming velocity. The way that you accomplish this depends on the nature of your animation. In some cases, you may have an animation where the incoming velocity is determined by the expression itself. For example, let's say you want a layer to scale up from zero to 200 percent over a short period of time and then overshoot a little and settle in at 200 percent. We'll set it up so the animation triggers at the layer's In Point. Here's a basic expression that will scale the layer from zero to 100 percent over one tenth of a second, starting at the layer's In Point:

```
t = time - inPoint;
startVal = [0,0];
endVal = [200,200];
dur = 0.1;
linear(t,0,dur,startVal,endVal);
```

Calculation Overshoot

Now we'll add some overshoot. In this case, since the incoming animation is generated by a `linear()` function, we can easily calculate the velocity coming into the overshoot. It turns out that the velocity is just the end value minus the start value, divided by the duration, which in this case would be `(endVal - startVal)/dur`. So we need to modify the scale-up expression so that it ramps up until it reaches `endVal`, then switches to the overshoot oscillation. The final expression looks like this:

```
freq = 3;
decay = 5;

t = time - inPoint;
startVal = [0,0];
endVal = [200,200];
dur = 0.1;
if (t < dur){
  linear(t,0,dur,startVal,endVal);
}else{
  amp = (endVal - startVal)/dur;
  w = freq*Math.PI*2;
  endVal + amp*(Math.sin((t-dur)*w)/Math.exp(decay*(t-dur)));
}
```



Notice that the transition from the `linear()` ramp to the overshoot oscillation (which happens at frame 5) matches perfectly.

The trick to making the velocities of the two animations match is the mysterious variable `w`. Here, `w` represents the angular velocity of the oscillation. Without getting into a lot of detail, it turns out that dividing the incoming velocity by the angular velocity of the oscillation gives an overshoot that matches perfectly. In practical terms this means that the higher the frequency of oscillation, the lower the resulting amplitude of the overshoot. One important thing to remember with overshoot is that you don't have direct control over the overshoot amplitude (the `amp` variable is calculated by the expression). If you want a larger overshoot, you need to either increase the incoming velocity, or reduce the oscillation frequency. It's worthwhile playing around with the expression to see the interaction between frequency, incoming velocity, and resulting overshoot amplitude.

This is an extremely useful and versatile expression. Here's a fun variation that you can use on a text layer to get the text characters to randomly scale up from zero to 100 percent, with overshoot. To get this to work, you would add a Scale Animator to your text layer, set the Scale value to zero percent, add an Expression Selector, (you can then delete the Range Selector), and finally, replace the default expression of the Expression Selector's Amount property with this:

```
freq = 3;
decay = 5;
maxDelay = 1.0;

seedRandom(textIndex,true);
myDelay = random(maxDelay);
t = time - (inPoint + myDelay);
startVal = [100,100];
endVal = [0,0];
dur = 0.1;
if (t < dur){
  linear(t,0,dur,startVal,endVal);
}else{
  amp = (endVal - startVal)/dur;
  w = freq*Math.PI*2;
  endVal + amp*(Math.sin((t-dur)*w)/Math.exp(decay*(t-dur)));
}
```

Here the overshoot expression has been combined with random text scaling.

You'll notice that the only real difference between this and the previous version of the expression are the lines that calculate a random delay variable (`myDelay`) based on the character's `textIndex` value. Using `textIndex` as the random seed ensures that each character will get a unique, random delay.

Here's another variation that you can use to get the 3D characters of text layer to sequentially swing into view with overshoot. First you need to move the anchor point of your text layer to the top of your text. You can do this with an *Anchor Point Animator*. Also, make sure you have enabled *per-character 3D*. Add a new *Animator* for *Rotation* (don't use the same one you used to adjust the *Anchor Point*). Set the value of X Rotation such that the text is rotated out of view (perpendicular to the screen). Add an Expression Selector and delete the Range Selector. Replace the default expression of the Expression Selector's Amount property with this:

```
freq = 2;
decay = 5;
delay = .15;
dur = .12;

myDelay = (textIndex-1)*delay;
t = time - (inPoint + myDelay);
startVal = 100;
endVal = 0;

if(t < dur){
  linear(t,0,dur,startVal,endVal);
}else{
  amp = (endVal - startVal)/dur;
  w = freq*Math.PI*2;
  endVal + amp*(Math.sin(t*w)/Math.exp(decay*t)/w);
}
```

3D characters rotate into view sequentially, with overshoot.

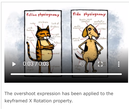
Keyframe Overshoot

Perhaps a more common overshoot application would be adding overshoot to a keyframed animation. This is actually fairly straightforward because the expression language gives you access to a property's velocity. In this case we'll use the `velocityAtTime()` function to get the incoming velocity at the most recent keyframe. This is the keyframe overshoot expression:

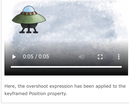
```
freq = 3;
decay = 5;

n = 0;
if (numKeys > 0){
  n = nearestKey(time).index;
  if (key(n).time > time) n--;
}
if (n > 0){
  t = time - key(n).time;
  amp = velocityAtTime(key(n).time - .001);
  w = freq*Math.PI*2;
  value + amp*(Math.sin(t*w)/Math.exp(decay*t)/w);
}else
  value
```

As before, the first two lines just define the variables that control the frequency and decay of the oscillation. The next section is a handy little routine that finds the most recent keyframe. The rest of the expression extracts the property's velocity at that keyframe and uses that velocity as the amplitude for the overshoot calculation. Note that the expression actually gets the velocity .001 seconds *before* the keyframe to ensure it picks up the incoming (and not the outgoing) velocity.

One of the really nice features of this expression is that it is property-agnostic, which means it should work, as-is, with almost any property that can be keyframed. See the two sidebar movies for two examples of keyframed animations where the expression provides overshoot. The first demonstrates a popular X Rotation overshoot, where the layers are keyframed to quickly rotate from 100 degrees to zero, with the expression providing the overshoot animation. The second demonstrates overshoot applied to the keyframed Position property.



The overshoot expression has been applied to the keyframed X Rotation property.



Here, the overshoot expression has been applied to the keyframed Position property.

Bounce Overview

Our ultimate objective here is to end up with an expression that will simulate a bounce back at the end of a keyframed motion. To understand how a bounce simulation works though, it's useful to start with a scenario that is probably more familiar—a projectile that bounces when it hits the ground/floor. To keep things simple, we'll limit it to two dimensions. When you launch a 2D projectile, there are a number of factors that come into play: the force of gravity, the elasticity of the object, the launch angle, the initial velocity, and sometimes friction. An expression to simulate this motion basically has to split the initial velocity into x and y components. Gravity works in the y direction. At each bounce, the object loses y velocity based on the elasticity and x velocity based on the friction. Taking all these factors into account gives you an expression for 2D bounce that looks like this:

```
elev = degreesToRadians(75);
v = 1000;
e = .7;
f = .15;
g = 5000;
nMax = 9;
tLaunch = 1;

vy = v*Math.sin(elev);
vx = v*Math.cos(elev);
if (time >= tLaunch){
  t = time - tLaunch;
  tCur = 0;
  segDur = 2*vy/g;
  tNext = segDur;
  d = 0; // x distance traveled
  nb = 0; // number of bounces
  while (tNext < t && nb <= nMax){
    d += vx*segDur;
    vy *= e;
    vx *= f;
    segDur *= e;
    tCur = tNext;
    tNext += segDur;
    nb++
  }
  if(nb <= nMax){
    delta = t - tCur;
    x = d + delta*vx;
    y = deltaT*vy - g*delta/2;
  }else{
    x = 0;
    y = 0;
  }
  value + [x,-y]
}else
  value
```

There are a few things to be aware of with this expression. The bounce parameters—launch angle (`elev`), initial velocity (`v`), elasticity (`e`), friction (`f`), and gravity (`g`)—are all defined at the top of the expression. Note that you also have to define the maximum number of bounces (`nMax`) to keep the expression from disappearing into an endless calculation of smaller and smaller bounces. This version of the expression also includes a variable to control the launch time (`tLaunch`).

Keyframe Bounce Back

Now we'll look at the expression that that is really the point of this section. This expression uses a property's velocity coming into a keyframe to calculate a bounce-back (in the direction opposite to the incoming animation) with a series of diminishing bounces. The expression uses most of the logic from the basic 2D bounce expression, except that there is no need to worry about launch angle and initial velocity (those are retrieved from the keyframe), or friction. This expression has been fashioned to work with most properties. See the sidebar movie for examples applied to the Scale, Position (2D and 3D), and Rotation properties. Here's the expression:

```
e = .7;
g = 5000;
nMax = 9;

n = 0;
if (numKeys > 0){
  n = nearestKey(time).index;
  if (key(n).time > time) n--;
}
if (n > 0){
  t = time - key(n).time;
  v = -velocityAtTime(key(n).time - .001)*e;
  vl = length(v);
  if (value instanceof Array){
    vu = (vl > 0) ? normalize(v) : [0,0,0];
  }else{
    vu = (v < 0) ? -1 : 1;
  }
  tCur = 0;
  segDur = 2*v/g;
  tNext = segDur;
  nb = 1; // number of bounces
  while (tNext < t && nb <= nMax){
    v *= e;
    segDur *= e;
    tCur = tNext;
    tNext += segDur;
    nb++
  }
  if(nb <= nMax){
    delta = t - tCur;
    value + vu*delta*(vl - g*delta/2);
  }else{
    value
  }
}else
  value
```

The bounce-back expression applied to the Position, Rotation, and Scale properties.

As before, you can control the bounce characteristics by adjusting the elasticity (`e`) and gravity (`g`) variables.

Further Exploration

In this article you've seen the difference between overshoot and bounce simulations. Hopefully that will help you in choosing the appropriate simulation for your own animations. You've also seen how to ensure that your overshoot simulations match up with your incoming animation.

Quick Links
- Overshoot in Detail
- Calculation Overshoot
- Keyframe Overshoot
- Bounce Overview
- Keyframe Bounce Back

A basic decaying sine wave oscillation applied to the Rotation property.

A basic bounce expression has been applied to the Rotation property.

Sine vs. Cosine

The JavaScript functions `Math.sin()` and `Math.cos()` both generate sinusoidal waves, but they serve different purposes for expression writers. The `Math.sin()` function starts at zero and increases in value. `Math.cos()` is 90 degrees out of phase with `Math.sin()`, which means it starts at maximum value and decreases.

You would generally choose `Math.cos()` when you want to start with your object in the fully extended, or fully rotated condition. For example, you might want to start a pendulum motion with the pendulum at maximum rotation rather than zero. We won't use it here, but it's a good tool to know about.

The overshoot animation matches the velocity of the incoming animation.

A 2D bounce simulation that uses launch angle, initial velocity, gravity, elasticity, and friction.