Kinematic Timing Curves: Cartoon Physics with Ease

Arnie Cachelin
Apple

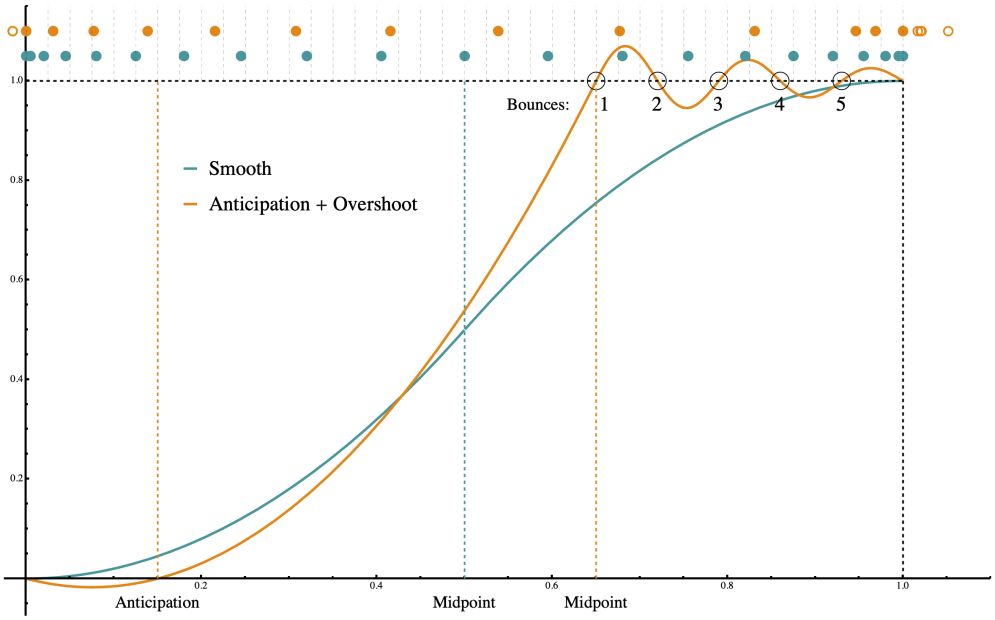


Figure 1. Animation timing curve controlled by three parameters, anticipation, midpoint, and bounces, shown here as a smooth inertial curve and a cartoon physics curve with anticipation and overshoot. The dots at the top are snapshots of the motion at equal time steps, so the spacing is proportional to the speed.

Abstract

We analyze animators' "cartoon physics" and apply appropriate constraints and boundary conditions to Newton's second law, to derive a simple model with intuitive parameters that reproduces the effects of inertia, anticipation, and overshoot commonly found in both traditional animation and modern interactive graphics. Our model takes the form of a normalized timing curve that is computationally efficient and should be easy to add to systems for producing motion graphics, mechanical visualization, and user interface animations. In its simplest form, the model reproduces a standard smooth step curve, but with an alternate formulation derived from basic kinematics.

1. Introduction

Traditionally, an animator making an object move from point A to point B would not make the object instantly spring to motion with a constant speed, then stop abruptly. That wouldn't look natural. Rather, they would ease the object into and out of its motion, so it would speed up gradually from point A, then slow to a smooth stop at point B. That looks natural because objects have inertia; they have to gradually accelerate to get moving and they can't stop instantly, but need some braking distance. So animators make things accelerate and decelerate, to represent the physical effect of inertia [Lasseter 1987].

For convenience, many computer animation systems have encapsulated this behavior into generic ease-in/out timing curves, which can be applied automatically to motions. By *timing curve* (or, often, *ease curve*) we mean functions $x(t)$ defined for $0 \leq t \leq 1$ with values also between 0 and 1 (mostly), where the horizontal axis represents the fraction of the entire animation duration, and the vertical axis shows the fraction of the progress from state A to state B. This is represented in Figure 1. Because the curve is normalized, it can be scaled to accommodate motions of any duration and preserve the character of the motion independently of the exact end points. By defining them in a normalized space, these 1D curves can be applied to timing on arbitrary (3D) transformations, or any other property animation with fixed start and end times.

Though ease-in/out effects are standard in the interpolated keyframe animation that artists would use to do actual cartoons, the timing curve is better suited to simple animated state transitions. As a result, timing curves are widely used to animate interactive visual elements of modern user interfaces (UIs), where studies show better user experience with elements that are animated to behave as if they were physical objects [Thomas and Calder 2001]. In fact, the inertia-simulating ease-in/out curve is a standard feature in most UI libraries, and has been for years. Some UI toolkits even include anticipation and overshoot effects, basic elements of cartoon animation.

Because timing curves are both standard and general, it is useful to express animation tropes beyond inertia using a timing curve. One of these effects is anticipation, where the object pulls back a bit before going forward. Another familiar effect is overshoot with oscillation, where the object does not manage to stop at point B, but keeps going then snaps back, possibly oscillating back and forth about point B a few times, as if it were caught by an elastic net, or spring. This is often referred to as *follow through*. These effects happen in nature because of inertia and elastic restoring forces; animators exaggerate them to bring their characters to life, and designers of interactive graphics apply them to make their pixels feel substantial.

By solving the basic physics of motion for inertia and restoring forces and applying normalizing boundary conditions, we have created a simple model with manageable parameters that can express these behaviors. This model extends the stan-

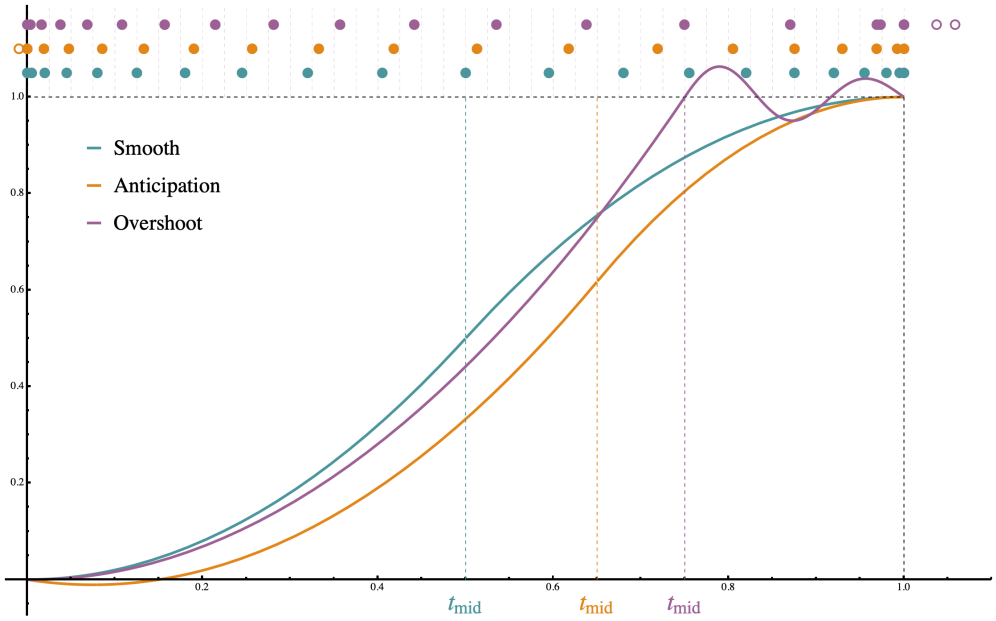


Figure 2. A variety of expressive timing curves can be derived from the three parameters anticipation, midpoint, and bounces.

dard no-parameter timing curve with three easy-to-visualize parameters: anticipation, midpoint, and bounces. These parameters can give rise to a wide variety of expressive behaviors, as seen in the timing curves in Figures 1 and 2. Because the model is derived from Newtonian physics, the various behaviors produced will be physically plausible, and thus realistic. We propose a new, physically based timing curve that reproduces these animation effects. Specifically our contributions are the following:

- A physically based smooth ease curve that is easy to compute and can serve as ground truth for evaluating other curves.
- A closed-form, generalized solution for the animation effects of inertia, anticipation, and overshoot: a class of motion curves that can be challenging to reproduce by hand, but simple enough that a physics simulation would be overkill.
- A simple expression that reproduces these effects and smoothly blends between them while maintaining physical realism. Controlled by three intuitive parameters, it allows users to define physically correct procedural motions while maintaining a high degree of control.

2. Related Work

Smooth motion interpolation with timing curves is intimately related to the piecewise interpolation of control points that form splines. Hermite-basis polynomials

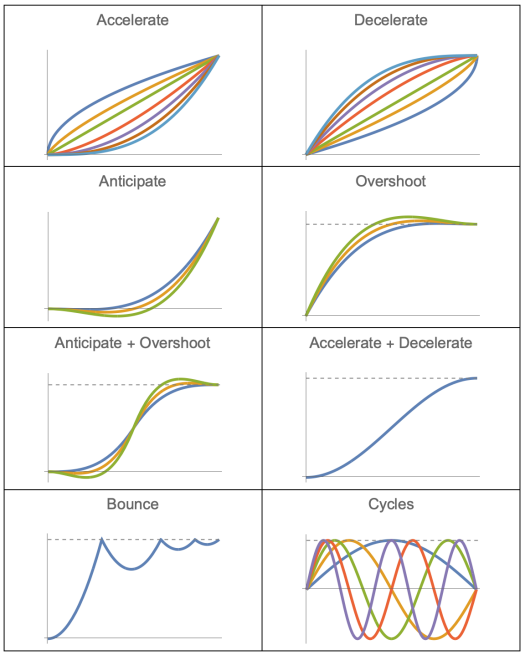
have been used effectively to fit a cubic spline to interpolate a set of input points [Plass and Stone 1983; Kochanek and Bartels 1984]. Kochanek used them to define a multi-segment interpolating spline specifically for animation, with artist-friendly parameters that carefully limited speed discontinuities in service of physically realistic animation.

Wiggly Splines [Kass and Anderson 2008] introduced a multi-segment spline curve that can transition from a classic smooth curve to an oscillator with controlled frequency. It is based on space-time constraints [Witkin and Kass 1988], a least-action formulation for physics-based animation with multiple constraints. It is very similar to this work in spirit, in the sense of applying physics to get analytic solutions to replace, reduce, or art-direct simulation-based animations. These multi-segment interpolation schemes are far more elaborate and complex than the following very focussed contribution, which abstracts two specific analytic solutions into a very simple but general timing curve.

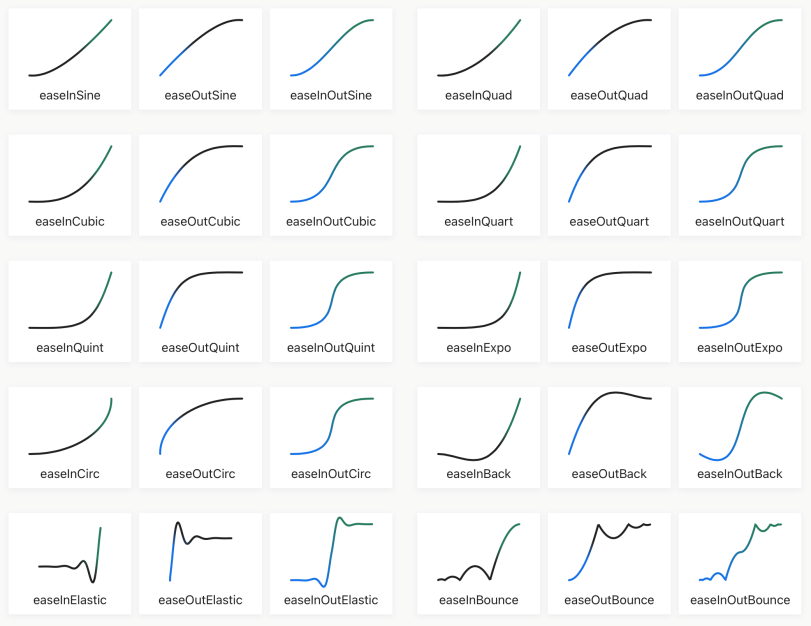
The Cartoon Animation Filter [Wang et al. 2006] proposed a filter-based process for automatically applying anticipation and overshoot to motions. Analogous to the unsharp filter in image processing, the Cartoon Animation Filter exaggerates the motion by subtracting a smoothed version of itself. With a single parameter to control the strength of the effect and automatic processing of potentially large numbers of keyframes, the filter is well adapted to bulk processing, if overkill for the simple animations where a timing curve could suffice.

Simple timing-curve–based animation is ubiquitous in graphical user interfaces, controlling animations from widget motion to element transparency in an effort to make graphical objects behave like physical objects. For years, UI toolkits [Hudson and Stasko 1993; Chang and Ungar 1993] have supported anticipation, ease-in/out, and follow through, sometimes explicitly invoking cartoon animation. A generation later, user studies demonstrated benefits of cartoon animation in user interfaces [Thomas and Calder 2001]. At present, UI animations generally at least use a smoothed ease-in/out timing curve for motion, and many APIs offer a variety of other effects as well. The Android UI API, for example, includes nine interpolator subclasses at this writing, which offer a number of combinations of anticipation, overshoot, and inertia (Figure 3a). The cross-platform toolkit Qt includes 30 preset timing curves, offering canned combinations of animation effects, both plausible and otherwise, as shown in Figure 3b.

Apple's UIKit Dynamics [Apple 2022c] provides constant acceleration animation, `UIPushBehavior`, as well as a damped oscillator to snap an element to a position, `UISnapBehavior`. It also offers a number of physics-based parametric animations, but the developer has to compose them together. This is a complex, flexible parametric system that can also be constrained to produce only physical results, though not without doing some physics. An alternative API for moving visual



(a) *Android uses preset interpolators, some with an adjustable parameter.*



(b) *Cross-platform toolkit Qt offers 30 preset curves, also available on Easings.net [Sitnik and Solovev 2022].*

Figure 3. Preset timing curves are primarily fixed combinations of inertia, anticipation, overshoot, and oscillation. Our system reproduces most of these effects with a simple function that smoothly blends between them while maintaining physical realism.

elements [Apple 2022a] exposes the initial velocity and the damping ratio, the damping relative to critical damping (with no oscillation). Though compact, the damping ratio parameter entangles the duration and oscillations, and is not easily visualized. The lower-level CoreAnimation framework provides a damped oscillator model for element animation. The `CASpringAnimation` API [Apple 2022b] exposes the mass, spring stiffness, damping, and initial velocity, and provides read-only access to a computed settling duration, which is dependent on all the physical parameters. In both cases, the initial velocity parameter allows developers to append the damped oscillation motion to terminate a previous motion, and thus construct a motion with inertia and overshoot. Although developers can tune this system for real-world objects, they have to use trial and error or an additional mathematical model to control the overshoot duration and bounces.

Android [Google Developers 2022a; Google Developers 2022b; Cogito Learning 2013] has taken a different approach to the problem of physical animation for UI elements by offering a set of fixed combinations in the form of `Interpolator` subclasses. They have the classic ease-in/out curve (`AccelerateDecelerateInterpolator`), with no parameters, as well as separate *Accelerate* and *Decelerate* only classes with a single parameter. Though the two separate functions are parabolic, the combined ease curve is a cosine ($y = (1 - cos(\pi t))/2$). There are also separate *Anticipate* and *Overshoot* interpolator classes, as well as the combined *Anticipate* + *Overshoot* version. These each have a *tension* parameter that modulates the effect. The *Anticipate* + *Overshoot* interpolator has an additional but redundant parameter, *extraTension*, which is the amount by which to multiply the tension. The `AnticipateOvershoot Interpolator` is the closest to our curve in intention certainly. It is based on a symmetric, two-section cubic curve:

$$y = \begin{cases} 0.5\left((C+1)(2t)^3 - C(2t)^2\right) & \text{for } t < 0.5, \\ 0.5\left((C+1)(2t-2)^3 \; + \; C(2t-2)^2\right) + 1 & \text{for } t \geq 0.5, \end{cases} \tag{1}$$

where $C = \textit{tension} \times \textit{extraTension}$.

This interpolator is locked to a single bounce, with anticipation and overshoot amounts always symmetric. Interestingly, with a zero parameter value, this interpolator returns a cubic smooth ease curve that is different than the *Accelerate* + *Decelerate* ease curve:

$$y = \begin{cases} 4t^3 & \text{for } t < 0.5, \\ 4t^3 - 12t^2 + 12t - 3 & \text{for } t \geq 0.5. \end{cases} \tag{2}$$

The `BounceInterpolator` produces three nicely damped parabolic bounces against 1.0, and the `CycleInterpolator` generates a sine wave with variable frequency, but there is no damped oscillator.

Qt [Qt 2009] takes the application of preset timing curves even further, offering 30 presets with few parameters. Their `QEasingCurve` class provides in, out, and

in+out combinations of both bounces and damped oscillators with overshoot, as well as a variety of ease in, out, and in+out curves using polynomials of second through fifth order, exponentials, *and* a cosine. The anticipate and overshoot and combo (`InBack`, `OutBack`, and `InOutBack`) are similar to those in Android, but the damped oscillators offer amplitude and frequency controls. In addition, the Qt API supports easing curves defined by custom cubic Bézier segments or by segments of Kochanek's TCB splines, the former supporting Qt's ability to reproduce the CSS/CoreAnimation ease curve standard.

These systems all offer ways to add anticipation, inertia, and overshoot to the motion visual element, either by building your own from components or by selecting from a handful of fixed curves with varying degrees of physical plausibility. They follow two basic paths: one requires developers to piece together physical solutions using their old physics textbooks or *Wikipedia*, and the other offers a wide variety of fixed curve shapes that are predominantly nonphysical. A large number of choices cannot completely compensate for the difficulty of producing a physically correct motion. None of these models attempt to guarantee physically correct motion nor include all the elements in a single continuous model with parameters that are easy to visualize, and thus explain and understand. Our model does not include a bounce, though the procedure for replacing the deceleration section with a bounce would be straightforward. One good argument for the preset curves system is the simplicity of choosing from a list of pretty good choices, rather than fiddling with sliders. Having a simple base model doesn't preclude the use of preset curves; it just makes them easier.

The default form of our timing curve looks like the standard smooth interpolation curve that has made its way into animation systems including CSS [MDN 2011], Apple CoreAnimation [Apple 2006], and Android [Jackson 2013]. It is also found in graphics languages from RenderMan [Upstill 1989] to GLSL [Rost 2005], to Metal [Apple Developer 2014], which all refer to it as `SmoothStep()`. In these languages it is primarily used for shading surfaces, where smooth transitions are used to minimize aliasing artifacts. Though `SmoothStep()` is a form of Hermite-basis interpolation, Perlin, whose bias and gain functions are similar to `SmoothStep()` [Perlin and Hoffert 1989], has offered an alternate polynomial smoothing curve [Perlin 2002] whose speed and acceleration *both* go to zero at the end points. Both our curve and the Hermite curve only have zero speed at the ends.

3. Derivation

Typically, ballistic trajectories are calculated by splitting the trajectory in half, calculating how long the initial velocity can carry an object up, and doubling that to find the time and thus distance of flight. We'll do the same thing, though not in the same

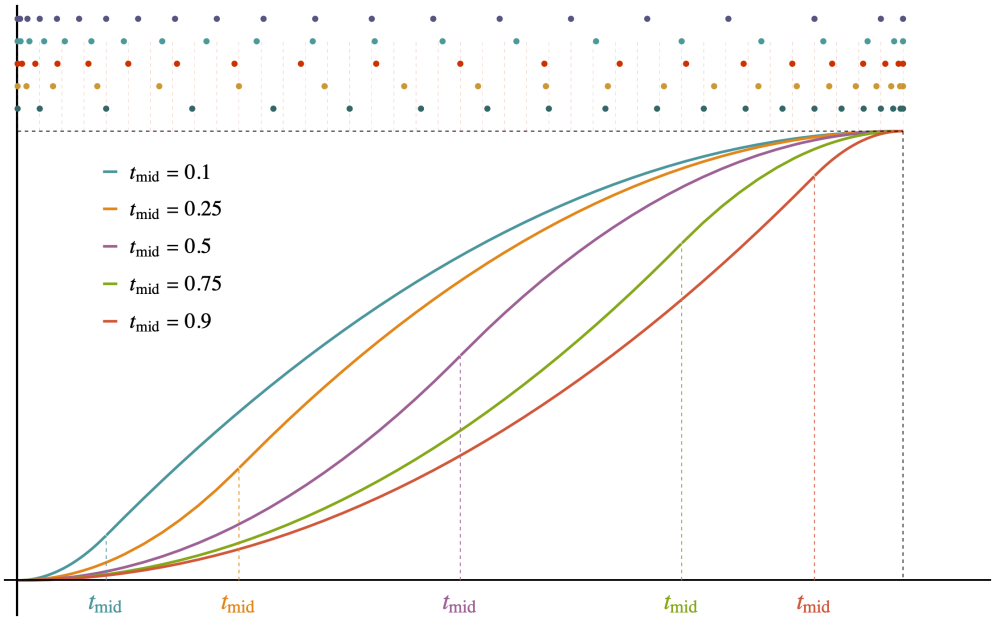


Figure 4. Adjusting the nominal midpoint, t_{mid}, controls the curve shape, emphasizing either ease-in or ease-out to suit the requirements of the animation.

order. In this case the first half will accelerate, and the second will decelerate, both at a constant rate. The timing curve thus has two phases, an *ease out* that accelerates out from the start point, and an *ease in* that decelerates into the end point. For clarity, we will use the subscripts A and D to denote these acceleration and deceleration phases, respectively. The standard timing curve switches from acceleration to deceleration right in the middle. Adjusting the dividing point between these two phases gives great control over the shape of the curve, so we will use this dividing point, t_{mid}, as an input to our model (Figure 4). The acceleration section covers the motion from time $t = 0$ to $t = t_{\text{mid}}$. The deceleration section contains any overshoot and settling-in oscillations, and it goes from t_{mid} to $t = 1.0$.

In summary, at $t = 0$ we apply a constant acceleration; at t_{mid}, we put on the brakes. If we are making a smooth stop, we brake with a constant deceleration. In the overshoot case we brake with linear restoring force, like a spring, which results in a damped harmonic oscillator. To join the acceleration and deceleration phases smoothly, we assure that the boundary between these sections of the curve are continuous and have matching speed. In the overshoot case, we tune the frequency so the value crosses 1.0 when the time reaches 1.0.

3.1. Acceleration

The acceleration-phase motion, $x_A(t)$, follows the simple kinematics of constant acceleration. The equation of motion expresses position as a function of time, governed

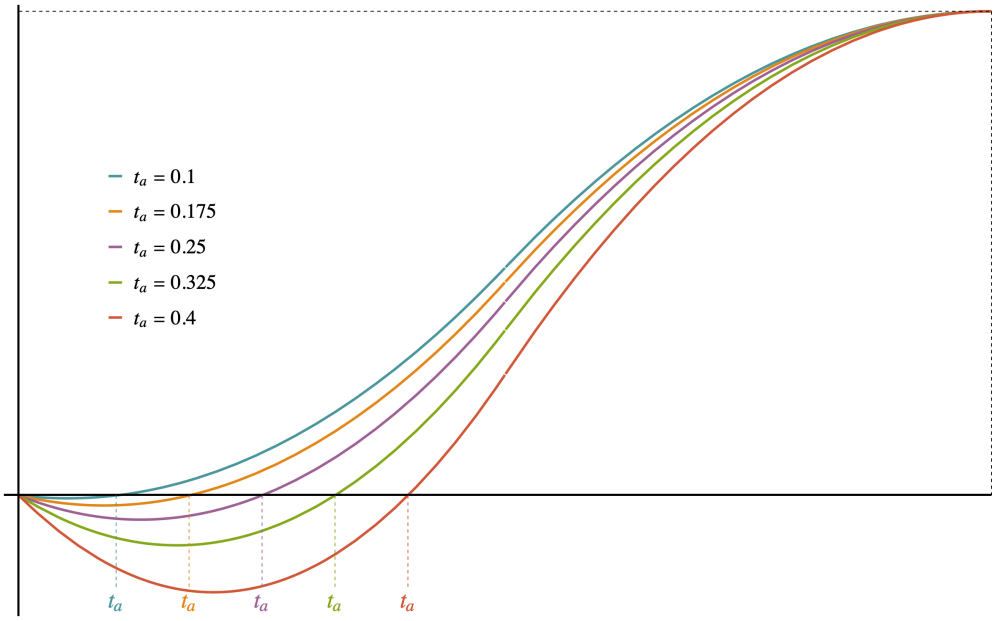


Figure 5. Variations on the anticipation time of the two-parameter timing curve, with the midpoint parameter locked at 0.5.

by the initial velocity V_0 and the acceleration a:

$$x_A(t) = V_0 t + \frac{1}{2} a t^2. \tag{3}$$

We can see the anticipation effect in Figure 5, where the motion reverses briefly and dips below zero. This is only possible when the initial velocity is negative, which in turn requires that the acceleration is positive. The shape of the curve is thus determined by the balance of the negative initial velocity and the acceleration against it. We can take advantage of this observation to replace the initial velocity parameter, V_0, with a directly visible, easy-to-understand parameter, the *anticipation time*, t_a, defined as the point at which the motion comes back above zero. If t_a is 0, obviously there is no anticipation effect:

$$V_0 + \frac{a t_a}{2} = 0 \quad \Longrightarrow \quad V_0 = \frac{-a t_a}{2}.$$

We can now recast our primary equations to use t_a:

$$x_A(t) = \frac{a}{2}(t - t_a)t. \tag{4}$$

The velocity at time t can thus be written as

$$v_A(t) \equiv \frac{d}{dt} x_A(t) = a\left(t - \frac{t_a}{2}\right). \tag{5}$$

To normalize the timing curve values, we just divide by x_{max}, the farthest the acceleration phase will reach:

$$x_{\text{max}} \equiv x_A(t_{\text{max}}) = \frac{a}{2}\,(t_{\text{max}} - t_a)\,t_{\text{max}}.$$

The normalized acceleration-phase position, $X_A(t) \equiv x_A(t)/(x_A(t_{\text{max}}))$, is greatly simplified, as a drops out. This leaves a normalized timing curve with anticipation, specified by just two parameters, t_a and t_{mid}:

$$X_A(t) = \frac{t\,(t - t_a)}{t_{\text{max}}\,(t_{\text{max}} - t_a)}, \tag{6}$$

with corresponding velocity

$$V_A(t) \equiv \frac{d}{dt}X_A(t) = \frac{2t - t_a}{t_{\text{max}}\,(t_{\text{max}} - t_a)}. \tag{7}$$

As expected, normalizing the motion removes the last scale-dependent term. Equations (6) and (7) are normalized equations of unitless motion for acceleration with anticipation. They give us the position and speed of the acceleration phase with parameter t_{max} representing the point at which the acceleration crosses the $x = 1$ line. Any deceleration phase motion that we want to append must match both of these values at the boundary where they join to preserve C^1 continuity.

3.2. Deceleration

3.2.1. Smooth Stop

A smooth stop reaches zero velocity gradually. The deceleration stage, starting by definition at the midpoint, t_{mid}, must now dissipate speed so that it goes to 0 as t reaches 1.0. We achieve this by applying a constant deceleration, d, over the braking period sufficient to just overcome the velocity at the end of the acceleration:

$$v_A(t_{\text{mid}}) + d(1 - t_{\text{mid}}) = 0 \quad \Longrightarrow \quad d = \frac{-v_A(t_{\text{mid}})}{1 - t_{\text{mid}}}.$$

At time $t = t_{\text{mid}}$, the position is $x_{\text{mid}} \equiv x_A(t_{\text{mid}})$ with speed $v_{\text{mid}} \equiv v_A(t_{\text{mid}})$, so d becomes

$$d = \frac{-v_{\text{mid}}}{1 - t_{\text{mid}}},$$

and the full equation of motion for $t_{\text{mid}} \le t \le 1$, the deceleration phase denoted by subscript D, becomes

$$x_D(t) = x_{\text{mid}} + (t - t_{\text{mid}})v_{\text{mid}} - \frac{(t - t_{\text{mid}})^2 v_{\text{mid}}}{2(1 - t_{\text{mid}})}$$

or

$$x_D(t) = x_{\text{mid}} + (t - t_{\text{mid}})\left(1 - \frac{t - t_{\text{mid}}}{2(1 - t_{\text{mid}})}\right)v_{\text{mid}}. \tag{8}$$

Using the definitions of x_{mid} and v_{mid},

$$x_{\text{mid}} = \frac{a}{2} t_{\text{mid}}(t_{\text{mid}} - t_a),$$
$$v_{\text{mid}} = \frac{a}{2}(2t_{\text{mid}} - t_a),$$

we can rewrite $x_D(t)$ using only a, t_a, and t_{mid}:

$$x_D(t) = \frac{a}{2}\left(t_{\text{mid}}(t_{\text{mid}} - t_a) + (t - t_{\text{mid}})(2t_{\text{mid}} - t_a)\left(1 - \frac{t - t_{\text{mid}}}{2(1 - t_{\text{mid}})}\right)\right).$$

Because the position and speed of both curves match at t_{mid}, when we normalize x_D we have to divide both parts of the smooth stop motion by $x_D(1)$:

$$X_S(t) = \frac{1}{x_D(1)} \begin{cases} x_A(t), & t \le t_{\text{mid}}, \\ x_D(t), & t_{\text{mid}} < t \le 1, \end{cases}$$

where $X_S(t)$ is the normalized smooth stop curve.

Replacing x_{mid} and v_{mid} and evaluating at $t = 1$ gives us $x_D(1)$, the constant we need to normalize the curve value:

$$x_D(1) = \frac{a}{2}\left(\frac{t_a}{2} + \frac{t_{\text{mid}} t_a}{2} - t_{\text{mid}}\right).$$

When we work this all out, a and thus t_{max} cancel out of the normalized timing curve, as expected:

$$X_{DS}(t) \equiv \frac{x_D(t)}{x_D(1)} = \frac{t^2(t_a - 2t_{\text{mid}}) - 2t(t_a - 2t_{\text{mid}}) + (t_a - 2)t_{\text{mid}}^2}{(t_{\text{mid}} - 1)(t_a t_{\text{mid}} + t_a - 2t_{\text{mid}})}. \tag{9}$$

Combining this with the equation for the acceleration phase motion X_{AS} yields

$$X_{AS}(t) \equiv \frac{x_A(t)}{x_D(1)} = \frac{2t(t_a - t)}{t_a t_{\text{mid}} + t_a - 2t_{\text{mid}}}. \tag{10}$$

We can express the smooth stop motion using only anticipation and midpoint values (Figure 6):

$$X_S(t) = \begin{cases} X_{AS}(t), & t \le t_{\text{mid}}, \\ X_{DS}(t), & t_{\text{mid}} < t \le 1. \end{cases} \tag{11}$$

3.2.2. Overshoot

We can model the cartoon physics overshoot effect as a linear restoring force like a spring, where the force is proportional to the offset distance. To reproduce the effect, the constant deceleration is replaced by a deceleration that increases with distance. The resulting motion is described by a simple sine wave with amplitude A and frequency f cycles per second. To match the position boundary condition, the sine

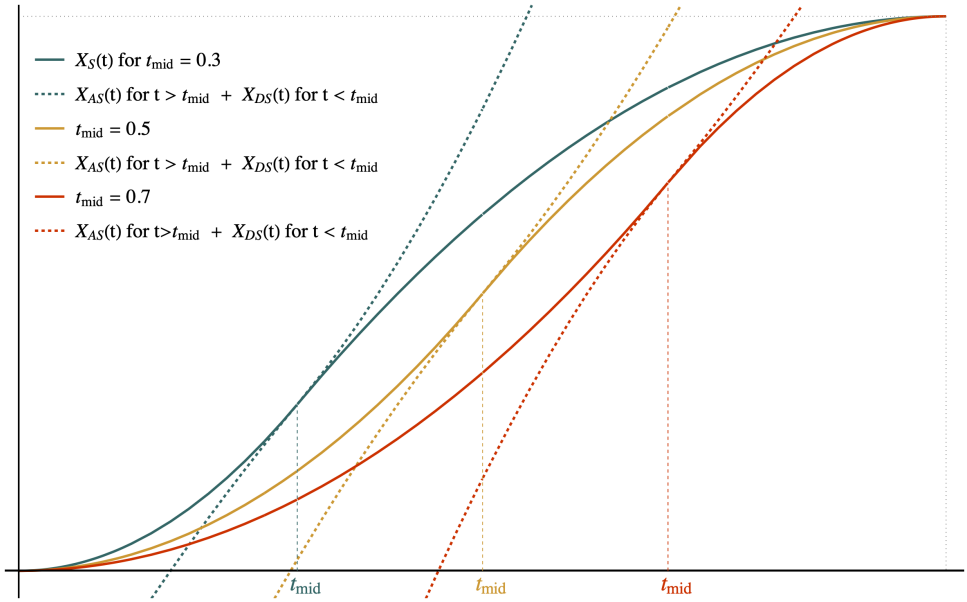


Figure 6. Matched pairs of acceleration and deceleration curves: the dotted sections are the unused continuations of the two curves. Each pair matches position and tangent at t_{mid}, assuring C^1 continuity.

should pass through zero at $t = t_{\text{mid}}$. We assure this by shifting the sine so it is 0 at t_{mid}:

$$x(t) = x_{\text{max}} + A \sin\left(2\pi f (t - t_{\text{mid}})\right).$$

The speed at the end of the acceleration phase must also be matched to the speed of the next section to keep things moving smoothly. The speed at t_{mid} is v_{mid}. Because the overshoot oscillator is a sine function, its derivative is a cosine. We set the sine to be 0 at t_{mid}, which means that the cosine is 1 and the speed, $v_D(t_{\text{mid}})$, is at its maximum, as it should be:

$$v_D(t) \equiv \frac{d}{dt} x_D(t) = 2\pi f A \cos(2\pi f\,(t - t_{\text{mid}})).$$

To match the speed at the curve start where $t = t_{\text{mid}}$, we set it equal to $2\pi f A$:

$$v_D(t_{\text{mid}}) = v_{\text{mid}} = 2\pi f A \cos(2\pi f * 0) = 2\pi f A,$$

then we use this to compute an appropriate amplitude:

$$A = \frac{v_{\text{mid}}}{2\pi f},$$

$$x_D(t) = x_{\text{max}} + \frac{v_{\text{mid}}}{2\pi f} \sin(2\pi f\,(t - t_{\text{mid}})).$$

Though the frequency is an obvious, visible parameter for an oscillator, its effects are entangled with the duration of the deceleration phase, and if we want the motion

to stop at 1.0 at the end, we will need to tune the frequency carefully. We propose instead a more animator-friendly parameter that is orthogonal to the timing and easy to visualize, namely, the number of overshoot bounces. Rather than tuning the frequency, the animator specifies the number of bounces, and we compute the frequency so that the correct number of cycles fits exactly into the span from t_{mid} to 1.0. We define bounces as the number of times the value crosses the 1.0 line. No crossings corresponds to a smooth stop without any overshoot, one crossing means that the value overshoots and snaps back once, and so on.

As there are two bounces per cycle of the oscillator and the period of oscillation is $T = 1/f$, the time to make B bounces is $B \cdot T/2$. To fit that into the deceleration span that is 1.0–t_{mid}, we set them to be equal, then solve for the frequency, f:

$$B\frac{T}{2} = 1 - t_{\text{mid}} = \frac{B}{2f} \quad \Longrightarrow \quad f = \frac{B}{2\,(1 - t_{\text{mid}})}. \tag{12}$$

Of course, the oscillations should be damped in order to die down over time. The standard damped harmonic oscillator is modulated by an exponential decay like $e^{-\gamma t}$, corresponding to a fractional loss of energy per cycle. The visible effect of damping is the reduction of the overshoot amplitude per bounce. We express this as the ratio of peaks in the sine at successive bounces:

$$\frac{e^{-\gamma(t+T/2)}}{e^{-\gamma t}} = e^{-\gamma T/2}. \tag{13}$$

We make the assumption that the damping results in a constant fractional reduction per bounce in order to replace γ with a more readily visualizable parameter. By assuring that γ is inversely proportional to the bounce period $T/2$, we can expose the per-cycle decay exponent, k,

$$\gamma \equiv \frac{2k}{T} = 2kf = \frac{kB}{(1 - t_{\text{mid}})}, \tag{14}$$

so that the ratio of amplitudes, $e^{-\gamma T/2}$, simply becomes e^{-k}.

Adjusting the decay exponent tunes the overshoot amplitude falloff, but because the curve hits 1.0 at the end by design of the sine function even without damping, we can tune the damping to taste (Figure 7). Indeed, we can set the decay exponent as a constant of the model, rather than a free parameter, as the value of limited adjustability is outweighed by the cognitive burden of another number to adjust. We will carry k around for a while, for completeness, but feel free to replace it with $1/4$, which gives a snappy 22% loss per bounce.

The overshoot deceleration motion with damped oscillator is thus

$$x_{DO}(t) = x_{\text{max}} + \frac{a}{2}\left(\frac{(1 - t_{\text{mid}})(2t_{\text{mid}} - t_a)}{\pi B}\sin\left(\pi B\frac{t - t_{\text{mid}}}{1 - t_{\text{mid}}}\right)e^{-kB\frac{t-t_{\text{mid}}}{1-t_{\text{mid}}}}\right),$$

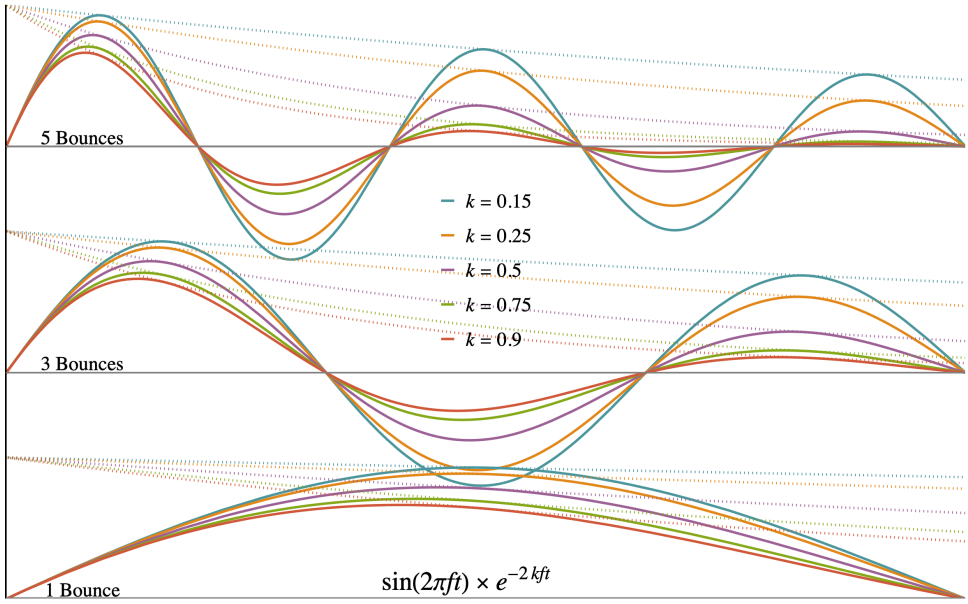


Figure 7. Different values for the decay exponent cause different attenuation of later bounces. In particular, values above 0.5 lead to very flat final bounces

where

$$x_{\max} = x_{AO}(t_{\text{mid}}) = \frac{a}{2}(t_{\text{mid}} - t_a)t_{\text{mid}}$$

As before, we have to normalize both halves of the motion by dividing $x_{\max}$, removing a, and expressing the curve in three parameters:

$$X_{DO}(t) \equiv \frac{x_{DO}(t)}{x_{\max}}. \tag{15}$$

So, the expression for the overshoot motion's deceleration phase, $X_{DO}(t)$, becomes

$$X_{DO}(t) = \left(1 + \frac{(1 - t_{\text{mid}})(2t_{\text{mid}} - t_a)}{\pi B(t_{\text{mid}} - t_a)t_{\text{mid}}} \sin(\pi B \frac{t - t_{\text{mid}}}{1 - t_{\text{mid}}}) e^{-kB\frac{t-t_{\text{mid}}}{1-t_{\text{mid}}}}\right) \tag{16}$$

or

$$X_{DO}(t) = \left(1 + \frac{(1 - t_{\text{mid}})(2t_{\text{mid}} - t_a)}{\pi B(t_{\text{mid}} - t_a)t_{\text{mid}}} \sin(\pi B t_d) e^{-kBt_d}\right), \tag{17}$$

where $t_d \equiv (t - t_{\text{mid}})/(1 - t_{\text{mid}})$ is a relative deceleration-phase time.

Recall Equation (4):

$$x_A(t) = \frac{a}{2}(t - t_a)t.$$

Thus, in the overshoot case,

$$X_{AO}(t) \equiv \frac{x_A(t)}{x_{\max}} = \frac{(t - t_a)t}{(t_{\text{mid}} - t_a)t_{\text{mid}}},$$

which matches the normalized acceleration curve, as expected (Figure 8).

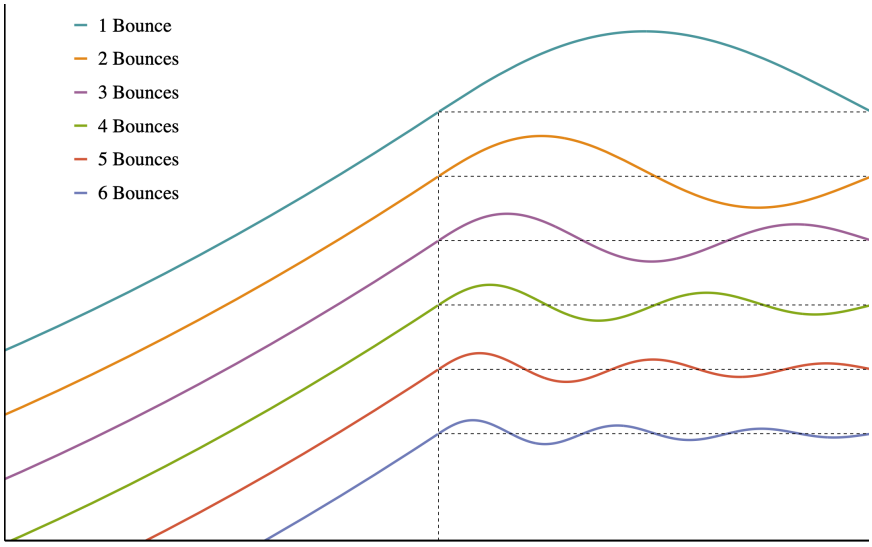


Figure 8. Overshoot deceleration curves, vertically offset for clarity.

Taken all together, we have an expressive timing function model derived from physical principles with three independent parameters: the anticipation time, the midpoint time, and the bounces.

The bounces parameter determines which of the two solutions to use:

$$X(t) = \begin{cases} X_S(t), & B = 0, \\ X_O(t), & B > 0, \end{cases}$$

where both the smooth and overshoot cases are also split into two halves:

$$X_S(t) = \begin{cases} \dfrac{2t(t_a - t)}{t_a t_{\text{mid}} + t_a - 2t_{\text{mid}}}, & t \le t_{\text{mid}}, \\ \dfrac{t^2(t_a - 2t_{\text{mid}}) - 2t(t_a - 2t_{\text{mid}}) + (t_a - 2)t_{\text{mid}}^2}{(t_{\text{mid}} - 1)(t_a t_{\text{mid}} + t_a - 2t_{\text{mid}})}, & t_{\text{mid}} < t \le 1, \end{cases} \tag{18}$$

$$X_O(t) = \begin{cases} \dfrac{t\,(t - t_a)}{t_{\text{mid}}\,(t_{\text{mid}} - t_a)}, & t \le t_{\text{mid}}, \\ 1 + \dfrac{(1 - t_{\text{mid}})(2t_{\text{mid}} - t_a)}{\pi B(t_{\text{mid}} - t_a)t_{\text{mid}}} \sin\left(\pi B \dfrac{(t - t_{\text{mid}})}{1 - t_{\text{mid}}}\right) e^{-\frac{B}{4}\frac{(t - t_{\text{mid}})}{(1 - t_{\text{mid}})}}, & t > t_{\text{mid}}. \end{cases} \tag{19}$$

4. Conclusion

4.1. Comparison of Smooth Step Curves

A timing curve smoothly interpolating between an input between 0 and 1 and an output in that range is such a useful thing that it has been included in graphics languages

from RenderMan to GLSL. The curve is a staple of procedural animation, where it is generally preferred to linear animation for simple parameter transitions as well as motions. This is apparent not just in motion graphics or animation systems, but even in the interface animations in modern graphical user interfaces. The Hermite function behind the standard smooth step curve also forms the basis for much keyframe interpolation as well.

The default behavior of our timing curve with no added effects (i.e., no anticipation or bounces, midpoint at 50%) reproduces the standard ease-in/out smooth curve. This was not a design goal of our curves, nor is the precise polynomial form of the curve obvious. Rather we can now see specifically how the standard smooth step motion arises from the physics of inertia.

Our smooth stop curves simplify significantly when we use default values; for example, with zero anticipation, the adjustable-midpoint smooth curve from Figure 4 looks like this:

$$X(t) = \begin{cases} \dfrac{t^2}{t_{\text{mid}}}, & t \le t_{\text{mid}}, \\ \dfrac{t^2 - 2t + t_{\text{mid}}}{t_{\text{mid}} - 1}, & t > t_{\text{mid}}. \end{cases}$$

And with the mid point at 0.5, the smooth curve becomes

$$X(t) = \begin{cases} 2t^2, & t \le 0.5, \\ -2t^2 + 4t - 1, & t > 0.5. \end{cases}$$

For comparison, the Hermite-basis polynomial smoothing immortalized in the GLSL `SmoothStep()` function, among many others, is

$$x(t) = 3t^2 - 2t^3.$$

SmootherStep is a variation on this function due to Perlin. It boasts zero first and second derivatives at both ends, which is particularly useful for texturing. It has the form

$$x(t) = 6t^5 - 15t^4 + 10t^3.$$

CSS and CoreAnimation both use a cubic Bézier curve with end points at $(0, 0)$ and $(1, 1)$ and standard tangent positions of $(0.42, 0)$ and $(0.58, 1)$. Sine function segments like those used in Android are also popular [Parent 2012; Google Developers 2022a].

By comparison our function performs well, falling between Hermite and Perlin (Figure 9), while also providing a rich set of extended behaviors with only a few additional parameters. It is interesting that our relatively elaborate model should also reproduce this most basic smooth timing curve.

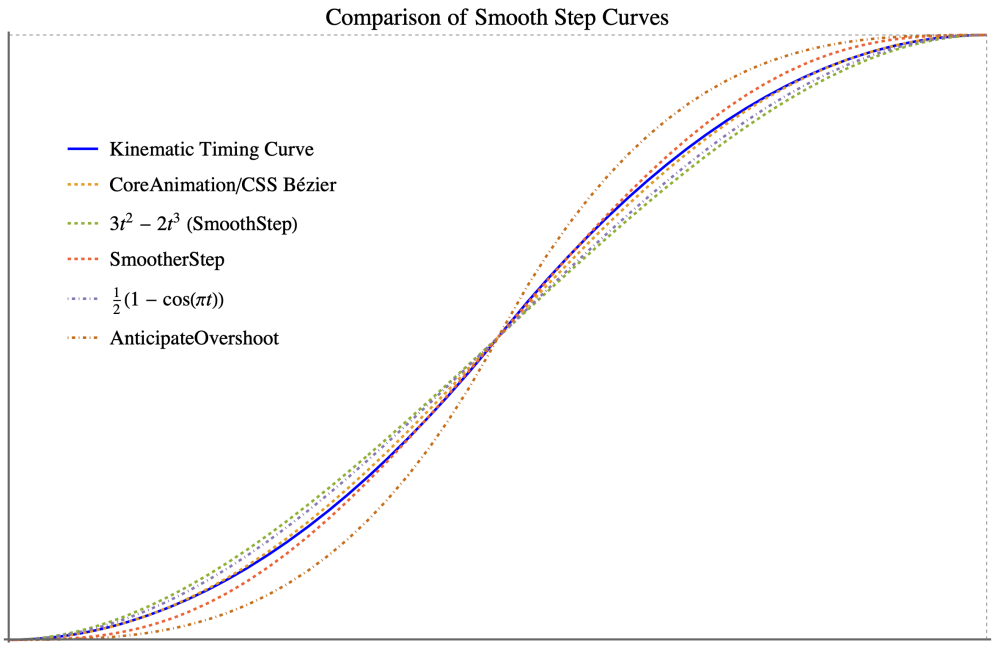


Figure 9. Our curve is very similar to standard smooth transition functions, falling between SmoothStep and Perlin's SmootherStep. The closest standard curve is the CSS cubic Bézier, which was presumably adjusted by eye for realism.

4.2. Results and Directions

Source code implementing the algorithm is listed in the appendix. As with the other implementations, the computations are simple polynomials, and sometimes a sine and an exponential. These are all relatively fast to compute on modern hardware, and fast approximations to the transcendental functions should work well enough for most purposes. One optimization that suggests itself is a truncated series expansion of the product of the sine and exponential for the damped oscillator, essentially treating it as a complex exponential.

The bounce animation in Qt and Android, like the damped oscillator, is a fixed replication of a physical result. It should be possible to make a third deceleration-phase option using the physics of bounces, following much the same procedure as we used for the damped oscillator. In this case, each bounce is a pair of constant accelerations resulting in a quadratic form, much like the ease-in/out curve.

We have have applied simple kinematics to the problem of replicating traditional "cartoon physics" used by animators to make motions expressive yet natural. By developing physical models of two stages of an animated motion, imposing continuity constraints at boundaries, then normalizing the motion to remove dependence on scale or specific timing, we derived a very general model for expressive parametric animation. As a bonus we gain insight into the classic smooth step curve, which might be said to approximate the result of constant acceleration and deceleration.

Acknowledgements

Greg Duquesne and Adam O'Hern were partners in the discussion of interactive graphics animations and physics that led to this work. Alex Coburn, Nafees Bin Zafar, and Deepak Tolani provided invaluable guidance in shaping the final draft. Without the support and encouragement of Novaira Masood and Apple, this work would not exist.

A. Appendix: Source Code

```
// Smooth timing curve value
float Xs(float t, float ta, float tmid) {
    float tam = ta - tmid - tmid; // ta - 2tmid

    float xa = (2.0*t*(ta - t) / (ta*tmid + tam));
    float xd = ((t - 2.0)*t*tam + (ta - 2.0)*tmid*tmid);
    xd /= ((tmid-1.0)*(ta*tmid + tam));

    return t<tmid ? xa : xd;
}

// Overshoot timing curve value
float Xo(float t, float ta, float tmid, float B) {
// Terms independent of t:  can be precomputed
    float tma = tmid - ta;
    float td = 1.0 - tmid;

// Time-dependent part
    if(t<tmid) {
        float xa = t*(t - ta) / (tmid*tma);
        return xa;
    }
// amp can be precomputed
    float amp = td*(tmid + tma)/(tmid*tma*B*M_PI);
    float xd = amp * sin(B*M_PI*(t - tmid)/td);
    xd *= exp(-(t - tmid)*(B/(4.0*td)));
    xd += 1.0;
    return xd;
}

// Timing curve with anticipation, ta, midpoint, tmid, and bounces, B
float KinematicTiming(float t, float ta, float tmid, int B) {
    return B>=1 ? Xo(t,ta,tmid,(float)B) : Xs(t,ta,tmid);
}
```

Listing 1. C source code implementing the Kinematic Timing Curve.

Index of Supplemental Materials

- KinematicTimingSamples.gif: Animated GIF examples.
- KinematicTimingCurve.c: Sample code

References

APPLE DEVELOPER. 2014. *Metal Shading Language Specification*. Apple, Inc., Cupertino, CA. URL: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf. 28

APPLE, 2006. kCAMediaTimingFunctionEaseInEaseOut. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/quartzcore/kcamediatimingfunctioneaseineaseout?language=objc. 28

APPLE, 2022. animateWithDuration:delay:usingSpringWithDamping:initialSpringVelocity:options:animations:completion:. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/uikit/uiview/1622594-animatewithduration/. 27

APPLE, 2022. CASpringAnimation. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/quartzcore/caspringanimation?language=objc. 27

APPLE, 2022. UISnapBehavior. *Apple Developer Documentation*. URL: https://developer.apple.com/documentation/uikit/uisnapbehavior?language=objc. 25

CHANG, B.-W., AND UNGAR, D. 1993. Animation: From cartoons to the user interface. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, Association for Computing Machinery, New York, UIST '93, 45–55. URL: https://doi.org/10.1145/168642.168647. 25

COGITO LEARNING, 2013. Android animations tutorial 5: More on interpolators. *Cogito Learning*, October 24. URL: http://cogitolearning.co.uk/2013/10/Android-animations-tutorial-5-more-on-interpolators/. 27

GOOGLE DEVELOPERS, 2022. AccelerateDecelerateInterpolator. *Android Developers Documentation*. URL: https://developer.android.com/reference/android/view/animation/AccelerateDecelerateInterpolator. 27, 37

GOOGLE DEVELOPERS, 2022. AnticipateOvershootInterpolator. *Android Developers Documentation*. URL: https://developer.android.com/reference/android/view/animation/AnticipateOvershootInterpolator. 27

HUDSON, S. E., AND STASKO, J. T. 1993. Animation support in a user interface toolkit: Flexible, robust, and reusable abstractions. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology*, Association for Computing Machinery, New York, UIST '93, 11. URL: https://dl.acm.org/doi/10.1145/168642.168648. 25

JACKSON, W. 2013. *Pro Android Graphics*. Apress, New York. URL: https://www.oreilly.com/library/view/pro-android-graphics/9781430257851/. 28

KASS, M., AND ANDERSON, J. 2008. Animating oscillatory motion with overlap: Wiggly Splines. In *ACM SIGGRAPH 2008 Papers*, Association for Computing Machinery, New York, 28:1–28:8. URL: https://dl.acm.org/doi/10.1145/1399504.1360627. 25

KOCHANEK, D. H. U., AND BARTELS, R. H. 1984. Interpolating splines with local tension, continuity, and bias control. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, SIGGRAPH '84, 33–41. URL: https://dl.acm.org/doi/10.1145/800031.808575. 25

LASSETER, J. 1987. Principles of traditional animation applied to 3D computer animation. *ACM SIGGRAPH Computer Graphics 21*, 4 (Aug.), 35–44. URL: https://doi.org/10.1145/37402.37407. 23

MDN, 2011. transition-timing-function. *MDN Web Docs*. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/transition-timing-function. 28

PARENT, R. 2012. *Computer Animation: Algorithms and Techniques*, 3 ed. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, Amsterdam. URL: https://books.google.com/books?id=ZNZ3XIGeMkgC. 37

PERLIN, K., AND HOFFERT, E. M. 1989. Hypertexture. *ACM SIGGRAPH Computer Graphics 23*, 3 (July), 253–262. URL: https://doi.org/10.1145/74334.74359. 28

PERLIN, K. 2002. Improving noise. *ACM Transactions on Graphics 21*, 3 (July), 681–682. URL: https://doi.org/10.1145/566654.566636. 28

PLASS, M., AND STONE, M. 1983. Curve-fitting with piecewise parametric cubics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, SIGGRAPH '83, 229–239. URL: https://doi.org/10.1145/800059.801153. 25

QT, 2009. QEasingCurve class. *Qt Documenation*, Qt Core 5.15.6. URL: https://doc.qt.io/qt-5/qeasingcurve.html. 27

ROST, R. J. 2005. *OpenGL Shading Language*, 2 ed. Addison-Wesley, Upper Saddle River, NJ. URL: https://www.oreilly.com/library/view/opengl-shading-language/0321334892/. 28

SITNIK, A., AND SOLOVEV, I., 2022. Easing functions cheat sheet. URL: https://easings.net/. 26

THOMAS, B. H., AND CALDER, P. 2001. Applying cartoon animation techniques to graphical user interfaces. *ACM Transactions on Computer-Human Interaction 8*, 3 (Sept.), 198–222. URL: https://doi.org/10.1145/502907.502909. 23, 25

UPSTILL, S. 1989. *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, Upper Saddle River, NJ. URL: https://www.pearson.ch/HigherEducation/Addison-Wesley/EAN/9780201508680/RenderMan-Companion-The. 28

WANG, J., DRUCKER, S. M., AGRAWALA, M., AND COHEN, M. F. 2006. The Cartoon Animation Filter. *ACM Transactions on Graphics 25*, 3 (July), 1169–1173. URL: https://doi.org/10.1145/1141911.1142010. 25

WITKIN, A., AND KASS, M. 1988. Spacetime constraints. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, Association for Computing Machinery, New York, SIGGRAPH '88, 159–168. URL: https://doi.org/10.1145/54852.378507. 25

Author Contact Information

Arnie Cachelin
Apple
1 Infinite Loop
Cupertino, CA
acachelin@apple.com

Arnie Cachelin, Kinematic Timing Curves: Cartoon Physics with Ease, *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 3, 22–42, 2022
http://jcgt.org/published/0011/03/02/

Received:	2021-11-10		
Recommended:	2022-02-17	Corresponding Editor:	Joe Geigel
Published:	2022-07-19	Editor-in-Chief:	Marc Olano