The Iterative Exponential Curve

A.G. van Oostrum [wiskunde.org] – September 27, 2024

Introduction

In this document, we describe the *iterative exponential curve*, a specific implementation of a decaying exponential. This curve has many applications, such as interactive computer programs where some sort of *interpolation* is needed. We will also look at its mathematical underpinnings, featuring a simple introduction to *generating functions*. We assume the reader is familiar with basic programming and high school level mathematics.

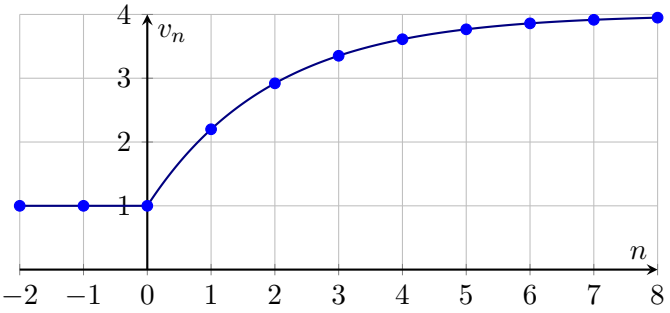
The Curve

Basic Structure

The curve we're about to describe comes about in situations where we have some value which we iteratively update:

```
v += (d-v)*r;
```

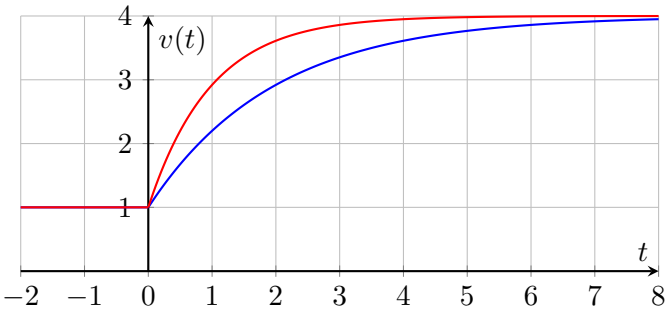
Here, v is a variable storing the current *value*, d is the *destination* value and r is some number (often between 0 and 1) specifying the *rate*. We can call v_n the value after n steps, with $v_n = v_0$ if $n < 0$ (so that v_0 represents the 'baseline' value). Below is a plot when $v_0 = 1, d = 4, r = 0.4$:



Here we can see both the actual computed values (blue circles), as well as the underlying continuous curve (which we'll derive later). As we can see, the values look like samples of a flipped decaying exponential curve, which starts at v_0 and approaches d. Before we'll proof this, we'll take a look at calculating r.

Timestep-Dependence

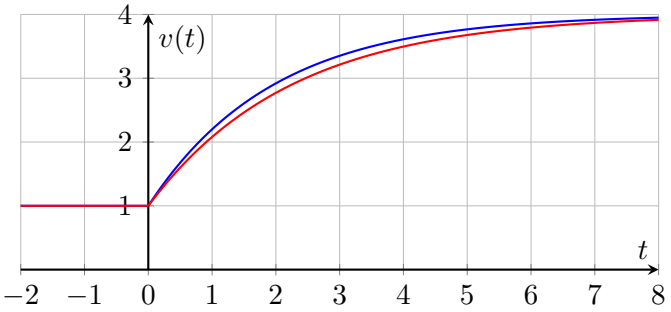
Something that's immediately obvious from the above implementation, is the timestep-dependence when viewed as a function of *time*. That is, if we perform the above update logic twice as many times per second, the resulting curve will be squished by a factor of 2. In the following graph, the red curve is the result of doing exactly that.



Do note that the x-axis and y-axis now represent *time* and the *continuous value* respectively, so that the two curves can be compared. This *timestep-dependence* is often undesirable. One might attempt to fix it as follows:

```
v += (d-v)*r*dt;
```

Here, dt represents the *deltatime*, i.e. the time between each consecutive step. However, this doesn't work, as the following graph shows, where the blue curve has $dt = 1$ and the red curve has $dt = 0.5$:



In order to really understand what's going on and how to fix it, we'll have to analyse the *update recurrence* more rigorously.

Mathematical Analysis

Update Recurrence

Let's first look at the simplest situation: $v_0 = 0$ and $d = 1$. Once we've solved this case, we'll be able to transform it into all other cases. Mathematically, the corresponding recurrence is now:

$$\begin{aligned} v_0 &= 0 \\ v_n &= v_{n-1} + (1 - v_{n-1}) \cdot r \qquad \text{if } n > 0 \end{aligned}$$

From now on, we'll only focus on $n \geq 0$, since $n < 0$ is trivial. The recursion simply states what we did in code, and can be rewritten as:

$$\begin{aligned} v_0 &= 0 \\ v_n &= r + (1 - r) \cdot v_{n-1} \qquad \text{if } n > 0 \end{aligned} \tag{1}$$

This slightly different form will help us later.

Generating Functions

In order to *solve* this recurrence (i.e. find a formula for v_n that isn't defined in terms of itself, called the *direct formula*), we must use a technique called *generating functions*. The main idea behind generating functions is to view a sequence of numbers as a *formal power series*, which basically means an (infinite) polynomial we're never actually evaluating and thus can manipulate more freely. If we're given a sequence of numbers, say $(a_0, a_1, a_2, \dots)$, the corresponding generating function is:

$$A(z) = a_0 + a_1 z + a_2 z^2 + \cdots = \sum_{n=0}^{\infty} a_n z^n$$

If the sequence is finite, say it contains N numbers, we can simply interpret $a_n = 0$ for $n \geq N$, in which case:

$$A(z) = \sum_{n=0}^{N-1} a_n z^n$$

Now let's look at a simple example: the sequence of infinitely many 1's. The corresponding generating function is:

$$Z(z) = \sum_{n=0}^{\infty} z^n$$

Normally we wouldn't be able to simplify this in general, however, because z is not *actually* a number (it's an *indeterminate*), we can use the following trick:

$$z \cdot Z(z) = \sum_{n=0}^{\infty} z^{n+1} = \sum_{n=1}^{\infty} z^n = Z(z) - z^0 = Z(z) - 1$$

This implies:

$$(z-1)Z(z) = -1$$

Which finally implies:

$$Z(z) = \sum_{n=0}^{\infty} z^n = \frac{1}{1-z} \tag{2}$$

This is a very important identity! It's worth noting what exactly happened: the summation from $n = 0$ to ∞ really means that we're summing over all $n \geq 0$. When we changed the z^{n+1} to z^n, we did so by substituting n with $n - 1$, so that z^{n+1} becomes $z^{n-1+1} = z^n$. By doing so, the summation condition turns into $n - 1 \geq 0$ and thus $n \geq 1$. Now, of course, we haven't formally justified why we can just *do* these things. However, that's not the aim of this document, and is best left for another time.

Solving the Recurrence

While there are *many* more interesting things to learn about generating functions, we now already know enough to solve the recurrence. Indeed, let's take a look at the generating function of our values v_n:

$$V(z) = \sum_{n=0}^{\infty} v_n z^n \tag{3}$$

We need two key observations: (a) $v_0 = 0$, so we can ignore the first term, and (b) we can substitute recurrence (1) *inside* the summation:

$$V(z) = \sum_{n=1}^{\infty} v_n z^n = \sum_{n=1}^{\infty} (r + (1-r)v_{n-1}) z^n$$

Let's put on the algebraic autopilot, which starts off splitting the sum:

$$\begin{aligned} V(z) &= \sum_{n=1}^{\infty} rz^n + \sum_{n=1}^{\infty} (1-r)v_{n-1} z^n \\ &= rz \sum_{n=1}^{\infty} z^{n-1} + (1-r) \sum_{n=0}^{\infty} v_n z^{n+1} \\ &= rz \sum_{n=0}^{\infty} z^n + (1-r)z \sum_{n=0}^{\infty} v_n z^n \\ &= \frac{rz}{1-z} + (1-r)zV(z) \end{aligned}$$

In the last step, we used equation (2). Now $V(z)$ is expressed in terms of itself, which kind of makes sense, since v_n is recursively defined! This can now be written as:

$$(1 - (1-r)z)V(z) = \frac{rz}{1-z}$$

And therefore:

$$V(z) = \frac{rz}{(1-z)(1-(1-r)z)}$$

All right, but how does this help us? Well, we can use the well-known technique of *partial fraction decomposition*:

$$V(z) = \frac{A}{1-z} + \frac{B}{1-(1-r)z} = \frac{A(1-(1-r)z) + B(1-z)}{(1-z)(1-(1-r)z)} = \frac{A + B + z(-(1-r)A - B)}{(1-z)(1-(1-r)z)}$$

Remember now that we're not solving for z, but rather this is an identity, so we must have:

$$\begin{aligned} A + B &= 0 \\ -(1-r)A - B &= r \end{aligned}$$

The solution is $A = 1, B = -1$ (which is easily checked), so:

$$V(z) = \frac{1}{1-z} - \frac{1}{1-(1-r)z}$$

We know that the first term is just $Z(z) = \sum_{n=0}^{\infty} z^n$ by equation (2), but what about the second term? Well, this is just $-Z((1-r)z)$ and therefore:

$$V(z) = \sum_{n=0}^{\infty} z^n - \sum_{n=0}^{\infty} ((1-r)z)^n = \sum_{n=0}^{\infty} (1 - (1-r)^n) z^n$$

Comparing this to our initial definition of $V(z)$ (equation (3)), we can at last conclude:

$$v_n = 1 - (1-r)^n \tag{4}$$

Basic Transformations

We've now found a direct formula for v_n when $v_0 = 0$ and $d = 1$ (the 'standard form'). In this section, we'll say that c is a curve with parameters c_0 and d_c, so the standard form is v with $v_0 = 0$ and $d_v = 1$.

The next obvious question is: what if we allow for arbitrary parameters? Well, let w be the curve with arbitrary w_0 and d_w. Using induction, we will show that $w_n = w_0 + (d_w - w_0)v_n$, which will yield us the general formula for values of the iterative exponential curve.

For $n = 0$, the equation clearly holds. Assume it holds for some $n \geq 0$, i.e. $w_n = w_0 + (d_w - w_0)v_n$. Combining this with the recurrence $w_{n+1} = w_n + (d_w - w_n)r$ gets us:

$$\begin{aligned} w_{n+1} &= w_0 + (d_w - w_0)v_n + (d_w - w_0 - (d_w - w_0)v_n)r \\ &= w_0 + (d_w - w_0)(v_n + (1 - v_n)r) = w_0 + (d_w - w_0)v_{n+1} \end{aligned}$$

Through induction, the equation holds for all integers $n \geq 0$. By equation (4), we can conclude:

$$w_n = w_0 + (d - w_0)(1 - (1 - r)^n)$$

Ergo: every iterative exponential curve is a shifted and scaled version of the 'standard form'. In particular, this means curves with the same base value and destination but different rates can be compared by just comparing the standard forms with the corresponding rates, which is exactly how we can solve the timestep-dependence.

Applying Our Results

Timestep-Independence

As we just explained, we can solve the timestep-dependence by solving it for the particular case when $v_0 = 0$ and $d = 1$. Let's say dt is the timestep, then $t = n \cdot dt$ and $n = t/dt$. Using equation (4), we can express the *continuous* value as:

$$v(t) = 1 - (1 - r)^{t/dt}$$

Let's say we fix the value at $t = \tau$, then we can solve for r:

$$\begin{aligned} v(\tau) &= 1 - (1 - r)^{\tau/dt} \\ (1 - r)^{\tau/dt} &= 1 - v(\tau) \\ 1 - r &= (1 - v(\tau))^{dt/\tau} \\ r &= 1 - (1 - v(\tau))^{dt/\tau} \end{aligned}$$

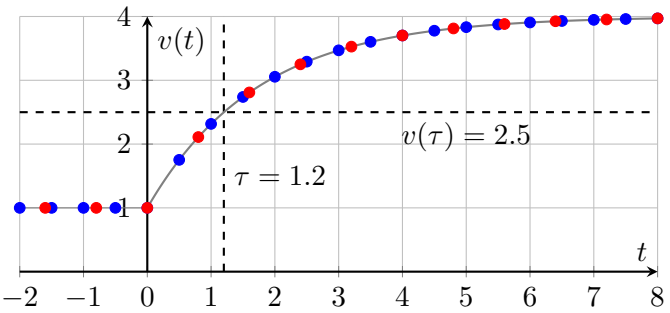
That's it! Everytime dt changes, we just recompute r. For example, if τ specifies the time to reach the half-point of the curve (i.e. $v(\tau) = 0.5$), then:

$$r = 1 - (0.5)^{dt/\tau} = 1 - 2^{-dt/\tau} \tag{5}$$

This also works if the timestep changes mid-curve, since the recursive definition of v can be interpreted as if a new curve starts every frame with the same d but a different v_0. In the general case, formula (5) computes the r such that $v(\tau) = v_0 + 0.5(d - v_0)$, i.e. 'halfway' to the destination. Therefore, the code in a realtime frame-based program such as a game might look like:

```
// beginning of frame: compute r
r = 1 - pow(2, -dt/t_half);

// later in frame: update v
v += (d-v)*r;
```

Here `t_half` is just τ when $v(\tau)$ is the halfway point. The graph below shows the computed values for $v_0 = 1, d = 4, \tau = 1.2$ for $dt = 0.5$ (blue) and $dt = 0.8$ (red):



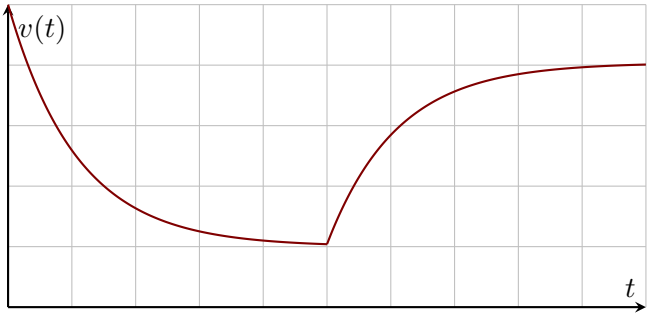
As we can see, both values lie on the same underlying continuous curve. We've fixed the time-dependence.

Self-Correction

One of the nice properties of this curve, is that it is *self-correcting*, which means that we can change any value at any point in time, and the curve will adapt. In particular, we can change v itself, or d, or even r (via τ).

One example where v might change, is if we have some UI where the user can drag 2D objects, which snap back to their original position when let go. In that case, d is the original position and we set v equal to the mouse position while dragging. While we aren't dragging, we execute the iterative update code. (In this case, v and d are actually 2D vectors, but we'd just implement the curve on both components.)

In other cases, d might change, for instance if we're animating a health-bar in a videogame. v could then represent the 'animated health', while d is the 'actual health'. Upon tacking damage, d gets decreased by the amount we take damage, but upon healing, d gets increased by how much we heal. The result might look something like this:



In this graph, we see another nice feature of this implementation: the initial curve is very sharp, which gives an immediate sense of feedback, while the later parts are smooth, which gives the feeling of a gentle 'landing'.

Approaching d

Let's take a look at the parameter r. We can confine r to be $0 < r \leq 1$, since other values don't really make sense (when $r = 0$, the value never changes). When $r = 1$, the change is instantaneous, which can actually be very useful in situations where we want to 'toggle' interpolation. For instance, let's say we are animating for a UI, we might do:

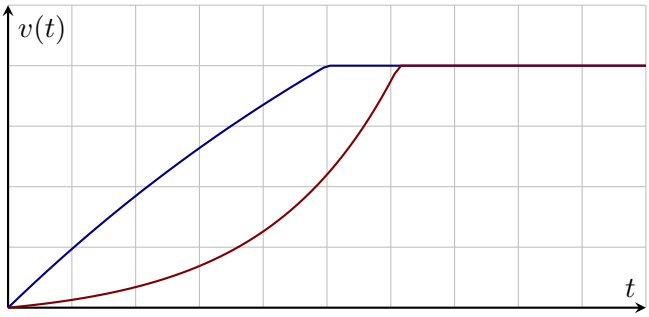
```
if (do_animate) r = 1 - pow(2, -dt/t_half);
else r = 1;

// later
v += (d-v)*r;
```

This would allow us to simply toggle animations on or off, for instance via a settings menu. Now, if instead $0 < r < 1$, the value 1 (in general: d) is never *exactly* reached, since for all n, we have $0 \leq 1 - (1 - r)^n < 1$. However, we quickly converge to a given threshold, which is actually enough. For instance, let's calculate how long it takes to fall within 0.1% error, which means $v_n \geq 0.999$:

$$\begin{aligned} 1 - (1 - r)^n &\geq 0.999 \\ (1 - r)^n &\leq 0.001 \\ n &\geq \log(0.001)/\log(1 - r) \end{aligned}$$

Usual parameters might be $dt = 1/60$ and $r = 1 - 2^{-(1/60)/0.1} = 0.109...$, so that it takes 0.1 seconds to reach halfway, which gives a snappy feel if used for animation-type effects. This gives: $n \geq \log(0.001)/\log(1 - 0.109...) = 59.794....$ Because n must be integer, the result is $t = n \cdot dt = 60/60 = 1$, i.e. after 1 second we are within 0.1% error. This might seem a lot, given that we are already halfway at 0.1 seconds, but it's really not, since most people won't percieve 0.1% error and in most use cases, 1 second isn't a lot of time.

If these results are still undesirable, they might be mitigated even further by other techniques, such as clamping, an example of which is given below.



The blue curve has $0 < r < 1$ as usual, however the red curve has $r < 0$. Normally, this would result in an unbounded curve, but because we clamp it anyway, this is actually a viable strategy for creating curves with a smooth start and hard landing, if that's needed. The implementation gets slightly more complex though, as the choice of `min` or `max` for the clamping depends on whether the destination is *below* or *above* the curve.

Conclusion

In conclusion: we've developed and analysed the very versatile *iterative exponential curve* and we've seen that it has some nice properties. In many situations, this curve is an excellent choice for responsive, yet smooth feeling animations or other interpolations. Although the interpolation destination is never exactly reached, it's hardly ever a problem and if it is, can be mitigated by simple methods.

Addenda

1

It's a fun exercise to use generating functions to solve the Fibonacci recurrence given by:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \qquad \text{if } n > 1 \end{aligned}$$

2

In the field of *Digital Signal Processing* (DSP), there is the so-called *z-transform*, whose definition is very similiar to the way we defined generating functions: given discrete-time signal values x_k, the signal's z-transform is:

$$\mathcal{X}(z) = \sum_{k=0}^{\infty} x_k z^{-k}$$

(Sometimes, the summation starts at $k = -\infty$, depending on if the signal can extend 'into the past'.) One might wonder why z is raised to the negative power instead. Well, one of the reasons is because it maps the *frequency spectrum* of the signal to the *unit circle* in the *z-plane*:

$$X(\omega) = \mathcal{X}(e^{i\omega})$$

In a nutshell, this is because substituting $z = e^{i\omega}$ turns the definition of $\mathcal{X}$ into a *Discrete-time Fourier transform* computation. However, this is a whole 'nother can of worms!