"EXPONENTIAL FUNCTIONS FOR NON-LINEAR CONNECTIONS"

SOFTWARE: CROSS-PLATFORM / SOFTIMAGE XSI
RIGGING TUTORIAL
LEVEL: EXPERT

Written by Christoph Schinko, 2007

1. INTRODUCTION

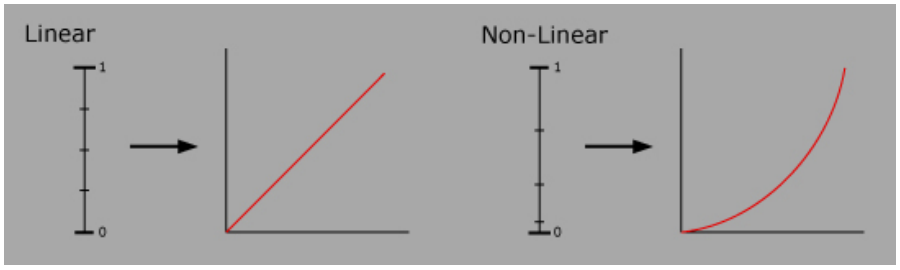

Figure 1: Linear vs. non-linear link

Linking parameters via expressions is simple and powerful: no extra objects, f-curves or constraints. Being simply math behind the scenes, they don't need to be taken care of and can do extraordinary things for the 3D artist.

However, sometimes the stiff, linear link we get from expressions is not exactly what we're looking for. Especially when rigging a character, where bone-rotations or shape-blends are driven by the surrounding rig-structure, we often need a more organic connection, a non-linear link. Something that, shown as a graph, looks like a quadratic or exponential function, rather than a straight line. [Fig.1]

That way, bones wouldn't start rotating so abruptly when the rig activates them, and shapes could finally work hand in hand with bone rotations. Unfortunately, XSI's "link deform with orientation" feature cannot be trusted for complex rigs, as the links created are not editable later on.

2. A LINEAR CONNECTION

Our example for the day: The foot of a character. When the foot is lifted a certain amount (25°), the pants should start deforming to drape across the shoe. Due to the pants' topology, this couldn't be done with point weighting, so a correctional shape was modelled instead - now to be linked to the rotation of the foot. [Fig.2]

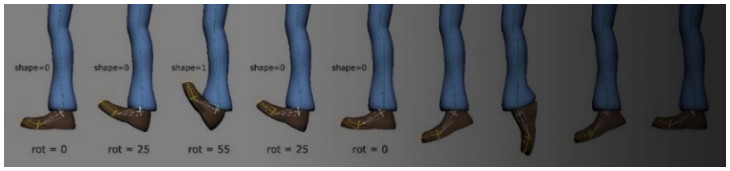

Figure 2: The characters foot, bones & different lifting stages with the correctional shape blending in

The usual approach is straight forward. We put an expression on the weighting parameter of our shape in the animation mixer and drive it through the rotation of the bone. As we're dealing with a rotational range from 25°-55°, we need to transform that input to match it to a shape weight range of 0-1. To do that, we'll subtract 25 from our rotation (giving us a range of 0°-30°), and divide that by 30.

shape weight = (rot – 25) / 30

As we don't want the trousers to deform before the foot's rotation reaches 25°, we'll tell the expression only to kick in when the foot bone reaches that value.

cond(rot < 25, 0, "OurExpression")

Or, for XSI to understand us:

cond(footBone.kine.local.rotz < 25, 0, (footBone.kine.local.rotz – 25) / 30)

What we have now is working, but as suspected, a linear interpolation is not really doing the trick for us.

Somewhere along the foot's rotation of 35°, the trousers will start intersecting with the shoes geometry, and the resulting motion will generally have an unnatural feel to it. [Fig.3]

What we need is the ability to edit that link we've just created, make it flexible. We need to be able to bend it, so that we can adjust that link until it gives us a smooth-looking motion and the geometries do not intersect. In this case: When the foot rotation hits 25°, the trousers' correctional shape needs to start blending in very slowly at first, and then faster and faster until its peak at 55°.

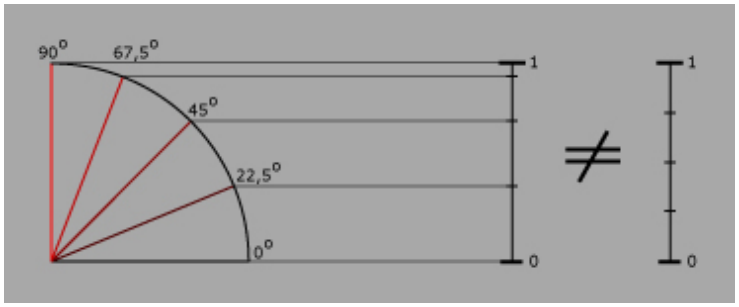

Figure 3: Why rotations and linear blends don't work together

Note: Of course one can use the foot's rotation as a multiplier for the expression (outcome * footRotation and so forth), but it's very tedious and also hardly editable later on. Furthermore, if you have to do this multiple times for your rig, and have to deal with different rotation-ranges and -directions every time, it'll take up a lot of time and is therefore not recommended.

Rather, let's try to keep our function independent of the foot's rotation, and come up with a mathematical function for that non-linearity we're after.

If we can get this to work, we have an adjustable curve, entirely without keys or time-dependent parameters – we still have just a simple expression that will keep on working for us no matter what we do to the rig or character.

3. A NON-LINEAR CONNECTION

Alright! First, let's look at exponential function graphs, they seem to be the closest to what we want: a slow start at the beginning of the blend, growing faster and faster until it peaks. [Fig.4]

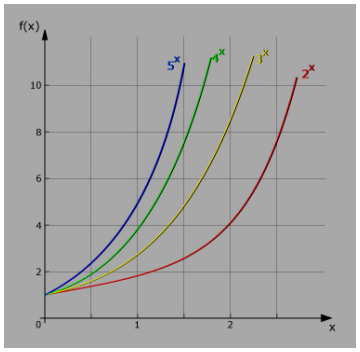

Figure 4: Exponential functions

Let's start by choosing one of those graphs as a starting point; 3^x looks like a curve that would fit well for our task. The curve looks steep enough at an x-value of 2, so we'd like to use the functions' behaviour from x=0 to x=2. This is the only part of the curve we'll use. We simply cut off the rest by transforming its x-value range much like before; "x" being the incoming rotation of the driving bone.

Our example:
bone rotation: *25° – 55°*
we want: *x = 0 – 2*

Much like before:
x = (rot – 25) / 15

This gives us the following curve:

$f(x) = 3^x = 3^{(rot - 25) / 15}$

The first thing we'll notice is that our graph starts at a f(x)-value of 1, as do all other exponential functions (or any number, for that fact) with an exponent of 0.

We'll simply subtract 1 from the result, giving us the following [Fig.5]:

$f(x) = 3^x - 1 = 3^{(rot - 25) / 15} - 1$

This moves down the whole curve, which now starts at 0 (as does our shape) and ends at 8 (which we know how to handle). Just divide the function by 8 and we get a range from 0-1; perfect for shape blending and a ton of other rigging tasks!

$f(x) = (3^{(rot - 25) / 15} - 1) / 8$

So far, so good, but just out of curiosity: what if we wanted to use a **different base** for our function? Maybe 5^x?

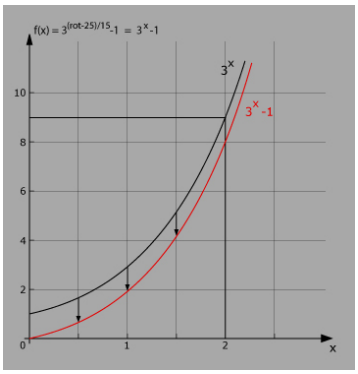

Figure 5: The 3^x function passing through 0

How can we generally normalize **all** exponential functions to suit our need, sticking to our range from x=0 to x=2?

We will need to divide each function differently, in order to get our 0-1 range. Let's look at some examples [Fig.6]:

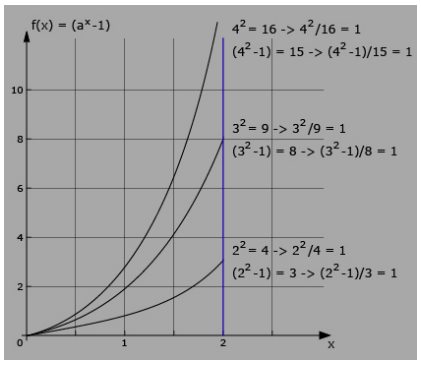

Figure 6: Normalizing all functions by division

Looking at the above picture, it becomes obvious pretty quickly that each exponential function needs to be divided through the square of its base, subtracted by 1 , (a^2-1), in order to get a result of 1 at the end of our rotation range (x=2).

→ We can now freely exchange the base of our exponential function and therefore get to use: a^x

Taking our function from before (3^x – 1), divided by what we just figured out (a^2-1), results in:

$f(x) = (a^x - 1) / (a^2-1)$

This gives us an exponential function in which we can change the base and always get a curve that passes through the same start and end points (0,0 and 2,1) - much like two hinges, if you will - and can be bent freely between those two by adjusting the base value *a*. We can therefore cover a whole field of curves, from a strongly bent line to

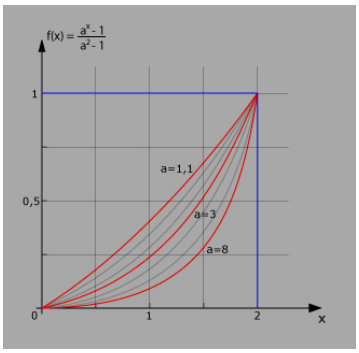

Figure 7: All functions passing through the same points

one close to linearity (with the base towards 1). [Fig.7]

Now our linked shape will slowly blend in with the growing rotation of a bone, and blend in ever faster until it is fully present. *a* will be delivered either by a null object ("baseNull"), a custom parameter, or is simply hard-coded; whatever suits your rigging needs.

The expression for our shape weight:

*pow(BaseNull, (footBone – 25) / 15) - 1 / (BaseNull * BaseNull – 1)*

or

*pow(BaseNull.kine.local.posy, (footBone.kine.local.rotz – 25) / 15) - 1 / (BaseNull.kine.local.posy * BaseNull.kine.local.posy – 1))*

Looks like we've accomplished what we set out to do, but if we look a bit further, we'll realise that this function can easily be enhanced to give us an even more powerful tool.

4. ADVANCED NON-LINEAR BEHAVIOUR

Inverting our curve to slowly **descent** first, is simply a matter of taking the negative results from our curve (mirroring it) and offsetting them by +1 (moving it up). [Fig.8]

*f(x) = **(-1)** * previousExpression **+ 1***

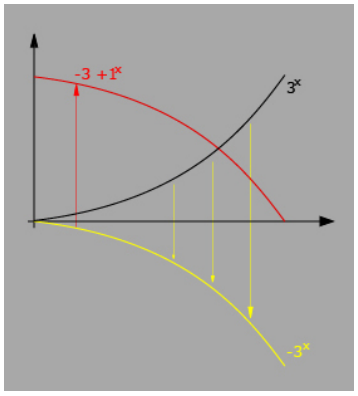

Figure 8: A new curve type

Introducing a new parameter ("switch") enables us to switch between those two behaviours. Is the parameter set to 1, we'll get the initial function behaviour. Setting it to -1 will result in the new graph.

The expression would look like this:

*switch * (previousExpression) – (switch – 1) / 2*

or, more confusingly:

*switch.kine.local.posy * (pow(BaseNull.kine.local.posy, (footBone.kine.local.rotz – 25) / 15) - 1 / (BaseNull.kine.local.posy * BaseNull.kine.local.posy – 1)) – (switch.kine.local.posy – 1) /2*

With this setup, the switch parameter (+1 or -1) defines which of the two curves will be created in the expression-link, and the base-parameter adjusts the curvature of the function graph. This base parameter has to be >1 in order to work (1 will result in a constant line).

Finally, let's look at two more curve types that might come in handy during the rigging process, depending on the situation and rotation direction

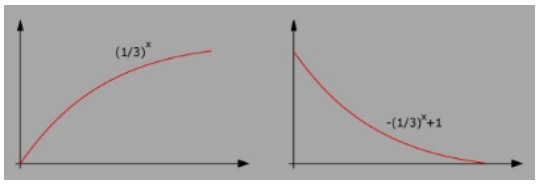

Figure 9: Two more curve types

you're working on. This time the curves are starting off steep and slow in towards the end of the curve. [Fig.9]

For this, the base of our function must be set to 1 divided by the base, so instead of 3^x, it must be $1/3^x$, or better yet: $0.33^{x!!}$

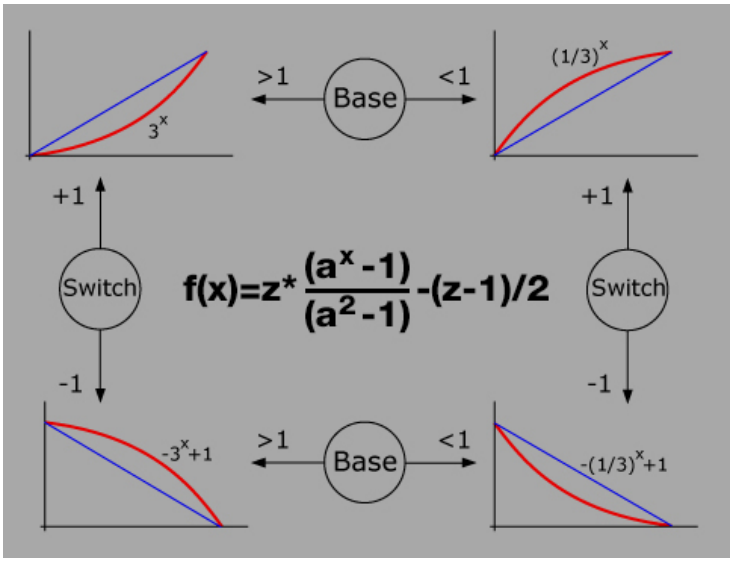

Figure 10: All 4 possible connection types and how they're linked

This means we don't have to change anything to our setup to do this, we simply set the base parameter to a value smaller than 1!

With our existing two parameters, the expression can be used to reach all four different types of graphs, and their respective areas towards linearity. [Fig.10]

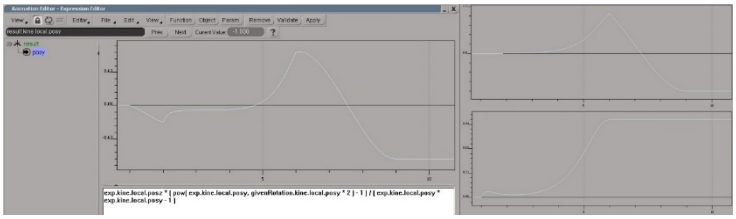

Figure 11: Connection curves with animated custom parameters

5. NEAT LITTLE PACKAGES

The two custom parameters created to control the links' behaviour can either be left in the scene, or eventually be hard-coded (the ideal value of the parameter actually typed into the expression, the parameter deleted) once a non-linear expression works well for the intended cause.

Although there are other ways to organically link parameters, most solutions are either hard to control or become overly complicated and instable within the rig, especially when working with more than one rotation axis.
The approach presented here is clean and easy to control, and more so, the behaviour can be adjusted visually, right in the viewport.

Note: The outcome of the function can always be previewed by activating *View>ShowGraph* in the expression editor, and get this: the parameters could even be animated to create most complex, non-linear links, as shown below, all that with our little expression! [Fig.11]

Hopefully this tutorial has been helpful and inspired to experiment a bit with expressions and exponential functions. If you have any feedback, please feel free to e-mail me at any time. Enjoy!

Written by Christoph Schinko in 2007
office[@]christoph-schinko.com
www.christoph-schinko.com