

A friendly game designer's tutorial on

How to make your damped springs stiffer, more stable, and easier to tune

Written by Michael Schmidt Nissen

Version 0.7.3, July 2019

Quick Definition Guide

x position

v velocity

a acceleration

F force

J impulse

m mass

k stiffness coefficient

c damping coefficient

t time

Δ delta

$$x = v\Delta t = a\Delta t^2 = \frac{F}{m}\Delta t^2 = \frac{J}{m}\Delta t^3$$

$$\frac{\Delta x}{\Delta t} = v = a\Delta t = \frac{F}{m}\Delta t = \frac{J}{m}\Delta t^2$$

$$\frac{\Delta x}{\Delta t^2} = \frac{\Delta v}{\Delta t} = a = \frac{F}{m} = \frac{J}{m}\Delta t$$

$$m \frac{\Delta x}{\Delta t^2} = m \frac{\Delta v}{\Delta t} = ma = F = \frac{J}{\Delta t}$$

$$m \frac{\Delta x}{\Delta t} = mv = ma\Delta t = F\Delta t = J$$

Abstract

In this article we will derive a modified but very simple damped spring equation that doesn't display any of the problems most game-developers associate with springs. From a game physics perspective, the spring equation combines the highest possible degree of stiffness and damping with great numerical stability. From a game designing and level editing perspective, the equation is easy to implement and tune, and it doesn't require the game designer to have any background knowledge about physics. The spring does not need to be re-tuned if parameters like particle mass or simulation time step are changed. This makes the improved spring equation a great tool for game designers and engine programmers alike.

The trouble with springs...

In the game development community, the damped spring based on Hooke's law of elasticity is often regarded as the black sheep of game physics, and a number of notable books, papers and websites warns the reader against implementing them. There are at least two good reasons for this.

The most straight-forward reason is that springs are notoriously difficult to tune. It can be very time consuming to adjust the stiffness and damping coefficients in order to make a simulation behave just right. For large spring systems the task can be daunting. If a single variable is changed, for instance a spring coefficient, the mass of a particle, or the physics engine time step, you might have to re-tune everything from scratch.

A deeper reason for avoiding springs is that they are prone to instability. Every once in a while, a system of connected springs and particles will blow up in your face without warning, and for no apparent reason – usually followed by a program crash. This happens because either simulation time step or spring coefficients have been set too high, or particle mass has been set too low, placing the simulation outside it's (usually unknown) range of stability.

The problem is that there's no simple or death-proof way to determine what combination of values adds up to a stable system. Thus, the game designer's ability to balance a spring system often relies on experience and “gut feeling”, which makes it look more like black magic than actual physics. This lack of transparency is why the game development community looks upon springs with a lot of suspicion.

Improving Hooke's infamous damped spring

In the following pages we will do a complete breakdown of the math behind damped springs and use it to reformulate the old equation in a way that makes sense for game developers. We will *only* be focusing on how springs work in the discrete time-stepping world of game physics, *not* how they work in real-life, which is a slightly different thing. First, let's take a look at the equation at hand:

$$F = -kx - cv$$

The equation is divided up into two parts. The first part calculates a spring

force that is linearly proportional to displacement x , and the second part calculates a damping force that is linearly proportional to velocity v . The value of the stiffness and damping coefficients k and c decide the magnitude of the forces.

Game designers usually want a spring to be as rigid as possible. That is, they want it to behave like a steel rod, and not like a soggy old rubber band. But what k and c values do we plug in to achieve this behaviour? 0.001? 1.337E42? For someone who doesn't understand the underlying math it is basically trial-and-error: Plug in a random value. If the spring behaves too sloppy, increase the value, and if the simulation becomes jittery or explodes, decrease it.

In the following section we will derive a spring equation that eliminates this challenge and allows the game designer to implement a spring that will always stay inside the safe region of stability and will reach its rest state in as little as one loop. To explain how this is done we first need to take a look at our integrator – the part of the program that computes new particle velocities and positions based on the forces applied to them. The most commonly used integrator is the so-called symplectic Euler or semi-explicit Euler method:

$$v_{(t+\Delta t)} = v_{(t)} + a_{(t)}\Delta t$$

$$x_{(t+\Delta t)} = x_{(t)} + v_{(t+\Delta t)}\Delta t$$

This integrator is popular because it is very easy to implement, and it displays great energy conservation and stability compared to similar algorithms. Please notice that velocity is expressed as $v = a \Delta t$, and that distance is expressed as $x = v \Delta t$.

Let's move on and look at a damped spring attached to a fixed point in one end and a particle in the other. Assume that the particle is displaced distance x from its rest position. Now let us try to find the force required to move it the opposite distance in one loop with the symplectic Euler algorithm. From Newton's 2nd law of motion we know that $a = F / m$, which give us the following relationship between distance and force:

$$x = -v\Delta t = -a\Delta t^2 = -\frac{F}{m}\Delta t^2$$

The minus sign simply means that we are going in the opposite direction of the displacement. Now we can isolate force:

$$F = -\frac{m}{\Delta t^2}x$$

And since the spring force in Hooke's law of elasticity is defined as:

$$F = -kx$$

It becomes immediately clear that:

$$k = \frac{m}{\Delta t^2}$$

Which is the exact stiffness coefficient value required to move the particle back to its rest position in one simulation loop. However, since we are not doing anything to stop the particle, it will keep oscillating back and forth through the rest position. We need to add damping, which is done with the second part of the spring equation.

Let's assume the particle has a velocity v that we want to eliminate. By calculating the amount of force needed to exactly counter this velocity, we can make the particle stop at the rest distance. In the Symplectic Euler integration algorithm we have the following relationship between velocity and force:

$$v = -a\Delta t = -\frac{F}{m}\Delta t$$

When isolating force we get:

$$F = -\frac{m}{\Delta t}v$$

And since the damping force is defined:

$$F = -cv$$

We immediately see by inspection that the damping coefficient is:

$$c = \frac{m}{\Delta t}$$

This is the exact damping coefficient value needed to make the spring stop the particle in one simulation loop. Now we can join the two parts and write out the complete damped spring equation in its most raw and basic form:

$$F = -\frac{m}{\Delta t^2}x - \frac{m}{\Delta t}v$$

This spring equation has some very interesting properties. The first thing we notice is the lack of coefficients. We've simply replaced k with $m/\Delta t^2$ and c with $m/\Delta t$. When implementing the equation we see that it really does work! The spring reaches rest length and stops dead without any oscillation in one loop, completely independent of particle position, velocity, mass, and physics engine time step.

This is the stiffest possible spring we can imagine, and it displays behaviour more similar to a rigid constraint than a soft, bouncy spring. We have found the steel rod spring we were looking for in the beginning of the chapter. The equation also has another interesting property. It simply cannot blow up no matter what values we throw at it. Practically speaking, we can regard the spring as being unconditionally stable.

We can refactor the equation a bit so it's easier to use in a programming context. Now time step, state, and mass are separated, which makes it easier to write re-useable code. Since physics engines normally run with a fixed time step, we can also pre-compute the $1/\Delta t$ and $1/\Delta t^2$ values and save a few precious CPU-cycles.

$$F = -(\frac{1}{\Delta t^2}x + \frac{1}{\Delta t}v)m$$

Re-introducing coefficients

Now we have a really powerful spring equation. It is easy to implement, very stable, and it reaches its rest-state in just one loop. But in our quest towards a better spring it has lost its springiness. We need to get the softness and bounciness back again, and for this purpose we will re-introduce stiffness and damping coefficients.

To avoid confusion, the new coefficients are named C_s and C_d . While the original coefficients could represent any positive numerical value, these both lie in the interval between zero and one.

$$F = -\left(\frac{1}{\Delta t^2} x C_s + \frac{1}{\Delta t} v C_d\right) m \quad [0.0 \leq C_s, C_d \leq 1.0]$$

As we can see, the new coefficients are simply the fraction of completely rigid behaviour we would like the spring to display. Soft, bouncy springs would usually have values in the range of 0.00001 – 0.001. In the other end of the scale, values of just 0.01 – 0.1 is enough to display rigid behaviour. Setting both values to 1.0 would of course still satisfy the spring in one loop.

Please notice that spring behaviour is determined *exclusively* by these two coefficients. Particle mass or simulation time step has literally *no* influence on how the spring behaves, and changing them – even during runtime – wouldn't make it necessary to re-tune the system. This makes the spring a much more powerful and developer-friendly tool that can be used safely by designers without any worries about possible future changes in game physics.

Interestingly, the spring will get *less* rigid and *less* stable if we increase the C_s or C_d values above one. This happens because the particles overshoot their target rest values, leading to undefined behaviour. If we keep increasing either or both of the coefficients, the system will start to jitter, and at some point it will ride the Neptune Express. In other words, we have – almost by accident – determined the exact upper limit for the two spring coefficients, which we can define:

$$s_{\max} = \frac{m}{\Delta t^2}, d_{\max} = \frac{m}{\Delta t}$$

This also allows us to simplify the spring a bit, making it almost as simple as

the equation we started out with:

$$F = -s_{\max} x C_s - d_{\max} v C_d$$

Now it also becomes clear why so many people dislike the Hooke's law spring equation. Not because it is inherently bad, but because it is so easy to accidentally plug in a coefficient value that doesn't make any sense. The equation derived in this paper always scales spring force to fit time step and mass, but the original equation does not.

This makes it very easy for the less physics-minded game designer to inadvertently plug in a coefficient value above the upper limits we have just derived. In the original equation it isn't even apparent that there is such a thing as an upper limit. I would like to stress the point that this lack of transparency in the original Hooke's law spring is the main reason why so many people have frustrating experiences with it.

Two connected particles with different mass

It is only slightly more complicated to constrain two free-moving particles with the improved spring. To do this, we need to introduce the concept of *reduced mass*. This is a quantity that can be used to compute interactions between two bodies as-if one body were stationary, which allows us to reuse the equation we've already derived. The reduced mass for two particles with mass m_a and m_b is defined:

$$m_{\text{reduced}} = \frac{m_a m_b}{(m_a + m_b)}$$

Since the inverse mass quantity is often pre-computed for other purposes, it can also be useful to define reduced mass like this:

$$m_{\text{reduced}} = \frac{1}{(1/m_a + 1/m_b)}$$

Unsurprisingly, the maximum coefficient values for two connected particles are:

$$s_{\max} = \frac{m_{\text{reduced}}}{\Delta t^2}, d_{\max} = \frac{m_{\text{reduced}}}{\Delta t}$$

Which gives us the following equation to work with:

$$F = -(\frac{1}{\Delta t^2} x C_s + \frac{1}{\Delta t} v C_d) m_{\text{reduced}}$$

This spring equation will move two connected particles of any given mass to rest distance and make them stand still relative to each other in one loop. However, since angular momentum is conserved, the particles may rotate around each other, which will make the bodies come to rest at a larger distance, depending on how fast they spin.

Impulse-based spring

Today, most physics engines are based on impulses, or direct changes in velocities, rather than forces and acceleration. The spring equation described above works just as well if we redesign it to work at impulse-level. Since impulse is defined $J = F \Delta t$, the symplectic Euler integration algorithm needs a little update:

$$v_{(t+\Delta t)} = v_{(t)} + \frac{J_{(t)}}{m}$$

$$x_{(t+\Delta t)} = x_{(t)} + v_{(t+\Delta t)} \Delta t$$

The maximum coefficient values for the impulse based spring are:

$$s_{\max} = \frac{m_{\text{reduced}}}{\Delta t}, d_{\max} = m_{\text{reduced}}$$

This gives us the following impulse based spring to work with. When used with the integration algorithm shown above, this equation returns the numerically exact same result as the force based spring we derived earlier in the paper.

$$J = -(\frac{1}{\Delta t} x C_s + v C_d) m_{\text{reduced}}$$

And that pretty much sums it up. Now I think it's about time we got our hands dirty and turned all the math into a working piece of code.

From math to code

I am using C++ syntax in the following code snippets, but for the sake of brevity and clarity I have kept them unoptimized and as simple as possible. With a little effort you should be able to make them run in your own favourite programming language. First let's define the particle class, which should be pretty self-explanatory:

```
class Particle
{
    vec2  position;           //
    vec2  velocity;          //
    vec2  impulse;           //
    float inverse_mass;       // Set to zero for infinite mass = static object.
};
```

Next we have the spring class, which is equally simple:

```
class Spring
{
    vec2    unit_vector;      // Normalized distance vector.
    float    rest_distance;   //
    float    reduced_mass;    //
    Particle* particle_a;     // Pointer to endpoint particle a.
    Particle* particle_b;     // Pointer to endpoint particle b.
};
```

Then we have the function that updates particle state. This part of the code implements the symplectic Euler integration algorithm and it is important to understand that the spring equation is tailored to be used with this update method. If you swap this out with another algorithm, it will probably break the logic of the spring equation.

```

void ComputeNewState( Particle* P )
{
    // Compute new velocity from impulse, scaled by mass.
    P->velocity += P->impulse * P->inverse_mass;

    // Compute new position from velocity.
    P->position += P->velocity * TIMESTEP;

    // Reset impulse vector.
    P->impulse = Vec2( 0.0, 0.0 );
}

```

Finally there is the function that computes and applies the spring impulse to its two endpoint particles. This is the core of our physics engine, and this is where the spring equation dwells.

```

void ComputeSpringImpulse( Spring* S )
{
    // First we calculate the distance and velocity vectors.
    vec2 distance = S->particle_b->position - S->particle_a->position;
    vec2 velocity = S->particle_b->velocity - S->particle_a->velocity;

    // We normalize the distance vector to get the unit vector.
    S->unit_vector = distance.normalize();

    // Now we calculate the distance and velocity errors.
    float distance_error = S->unit_vector.dot( distance ) - S->rest_distance;
    float velocity_error = S->unit_vector.dot( velocity );

    // Now we use the spring equation to calculate the impulse.
    float distance_impulse = C_S * distance_error * INVERSE_TIMESTEP;
    float velocity_impulse = C_D * velocity_error;
    float impulse = -( distance_impulse + velocity_impulse ) * S->reduced_mass;

    // Finally, we apply opposite equal impulses to the endpoint particles.
    S->particle_a->impulse -= impulse * S->unit_vector;
    S->particle_b->impulse += impulse * S->unit_vector;
}

```

Limitations and Pitfalls

When connecting multiple springs and particles into larger bodies, we run into the same trouble as with any other type of constraint. Rather than cooperating, they tend to compete against each other, and this spells trouble. When a spring moves two particles to satisfy distance and velocity, it usually means dissatisfying one or several other springs. It is outside the scope of this article to dig deeply into this problem, but I would like to provide a bit of advice on how to prevent the worst disasters.

When two or more springs are connected to the same particle, which is the case in any kind of rope, mesh, or squishy body, setting the coefficients to the maximum value of 1.0 will lead to stability problems. Although the spring equation is stable when particles are connected to just one spring, this is sadly not the case for higher number of springs. After some lengthy tampering I have worked out that a safe upper limit for both the stiffness and damping coefficient is:

$$C_{\max} \approx \frac{1}{(n+1)}$$

Where n denotes the highest number of springs attached to any of the two particles connected by the spring. So for example, in a rope where any particle is connected by at most two springs, C_s and C_d can both safely be set to 0.33, and in a square mesh, where any particle is at most connected by four springs, they can be set to 0.2.

Conclusion and future work

As promised, I have managed to derive a very stable, rigid, and user-friendly spring equation over the course of a few pages. The math behind the equation is surprisingly simple, which makes me wonder why it hasn't been described or implemented decades ago. In the next article, we will look further into the mentioned limitations when implementing large systems of interconnected springs and improve it even further using an iterative approach with warm starting.

About the author

Michael Schmidt Nissen lives in Denmark and currently makes a living as a historical blacksmith. He extracts iron and steel from local ores using the same techniques as his Viking ancestors. Programming, especially small games and physics simulations, has been a beloved hobby ever since Michael learned to code in the golden era of the Commodore 64 and Amiga 500.

If you enjoyed reading this article, please consider giving it some constructive feedback and share it with your colleagues. Also, if you use the spring equation in a game or demo, please give the author credit for it.

Links

https://en.wikipedia.org/wiki/Hooke%27s_law

https://en.wikipedia.org/wiki/Newton%27s_laws_of_motion

https://en.wikipedia.org/wiki/Reduced_mass

https://en.wikipedia.org/wiki/Semi-implicit_Euler_method

<http://gafferongames.com/game-physics/spring-physics/>