# Soft Constraints

Reinventing the Spring

Erin Catto

Title page

Hello everyone!

My name is Erin Catto and I want to thank you for coming to my tutorial.

The topic of my presentation is soft constraints.

First let me give you a little background about myself.

# Who am I?

Who am I?

My first job in the game industry was writing the physics engine for Tomb Raider: Legend at Crystal Dynamics. The engine was used to create all the physics puzzles you find in the game. The same engine lives on today in Lara Croft: Guardian of Light and the upcoming Deus Ex 3.

After working at Crystal Dynamics, I went to Blizzard and wrote a custom physics engine for Diablo3 called Domino. Domino handles the destruction and ragdolls you see in the game. Domino is now used by multiple titles at Blizzard.

In my spare time I have been working on the Box2D open source engine. This engine is widely used in the independent games community. Box2D is used in Crayon Physics, Limbo, and several iPhone games.

# The mystery of the magic formulas

From the Open Dynamics Engine (ODE) manual:

$CFM = 1 / (h\ k + c)$
$ERP = h\ k / (h\ k + c)$

"In fact, ERP and CFM can be selected to have the same effect as any desired spring and damper constants."

h: time step
k: spring stiffness
c: damping factor

Teaser

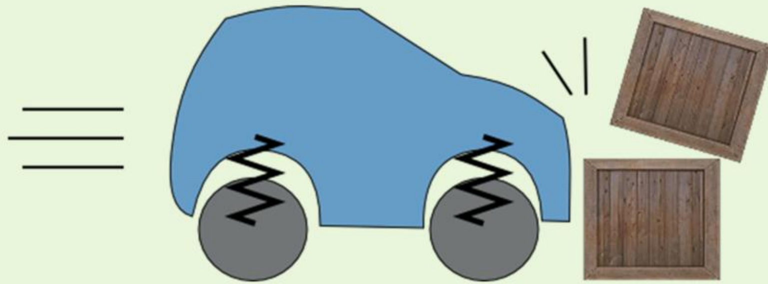
Have you heard of the Open Dynamics Engine? It is called ODE for short. ODE is probably the first open source 3D physics engine that is suitable for games.

A few years ago I was reading the ODE manual and I came across these strange parameters called CFM and ERP. These parameters are used to soften the constraints between rigid bodies. They basically appear to be fudge factors.

In the manual I found these nice formulas that relate ERP and CFM to the time step h, spring stiffness k, and damping factor c. At first, I just assumed this was just hackery without any real mathematical underpinning. But I was wrong, and today I'm going to show you the solid math behind these magic formulas.

But first, let's see why this topic is important for games.

# Games often need to use rigid constraints and springs



Setting

Suppose you are working on a game with vehicles. You probably want the vehicle to have a springy suspension. It would be nice if the vehicle could crash into things, such as a stack of boxes. Or you might want the vehicle to drive across a suspension bridge made of rigid bodies. There are many possibilities for combining rigid bodies and springs.

Rigid bodies and springs both have their place. Rigid bodies excel at representing collision and friction. Springs excel at absorbing and storing energy.

# Physics programmers must combine springs and constraints

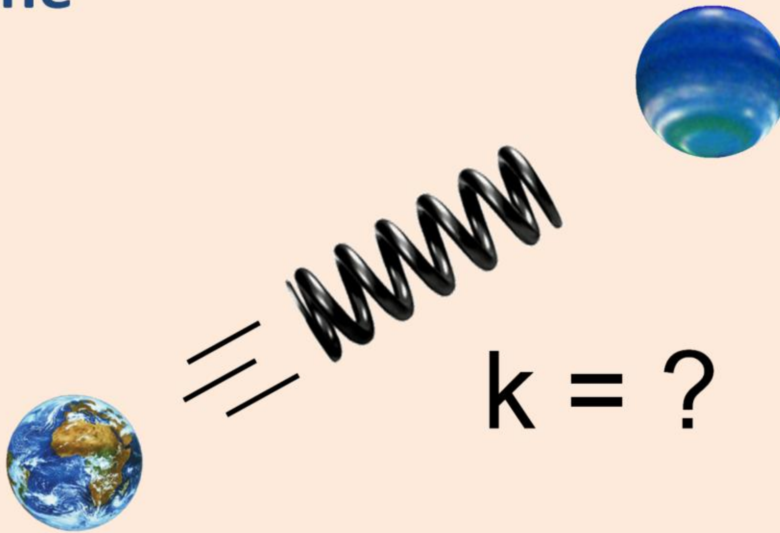
```
void SolveTimeStep()
{
    ApplySpringForces();
    SolveConstraints();
}
```

Role

So we have to simulate springs and constraints together. This seems quite easy. We just apply some spring forces to the rigid bodies and then let the constraint solver do its thing.

If all goes well, we get a nice simulation where springs, rigid bodies, and constraints are all interacting well together. Often this is exactly what you get.

# Springs can blow up and are difficult to tune



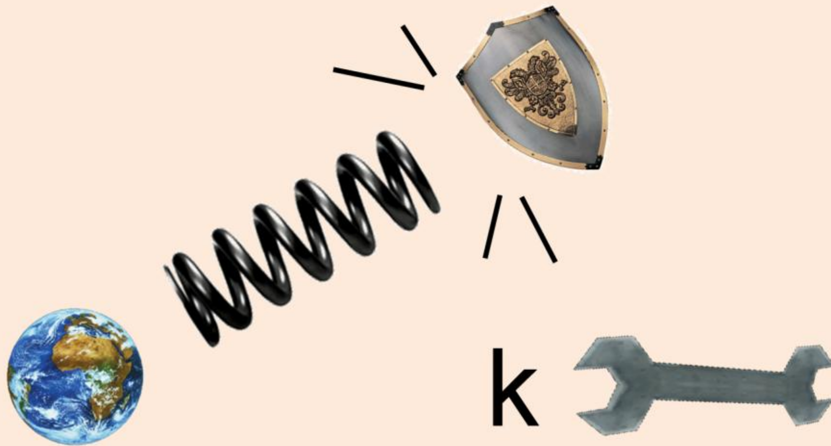

What challenges do I face?

Unfortunately, springs have two big problems. First, numerical instability can cause stiff springs can blow up and send your simulation to Neptune.

Second, the spring stiffness k is difficult to tune. Quite often tuning springs is a trial and error process that is unfit for large scale development.

In other words, springs can be a real nightmare for physics programmers and designers.

# We want springs that are stable and easy to tune




Where do I want to be?

Wouldn’t it be nice to use springs and not have to worry about your simulations going unstable?

Wouldn’t it be nice to provide springs to game designers that are easy to tune?

As you might guess, these are the goals of this presentation.

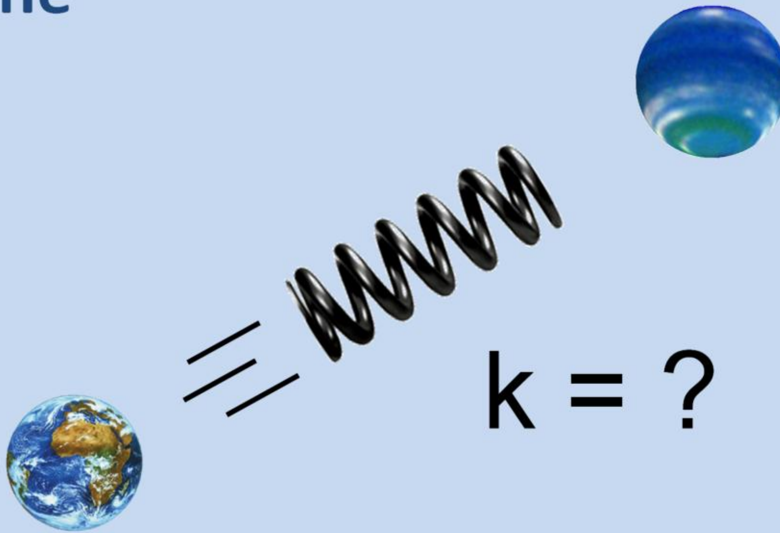
## Use soft constraints instead of springs

Call to Action

If you are concerned about spring stability and tuning, then you should consider using soft constraints.

What are soft constraints? You can think of them as Buddha springs. They behave like springs without stability problems and they integrate easily with rigid body constraints. Also, I will show you a method for tuning soft constraints easily.

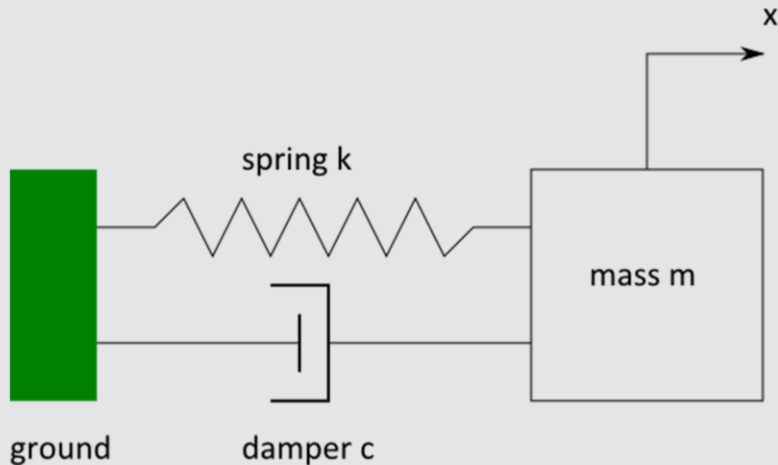
So let’s get into some key reasons why soft constraints are a good solution.




1

I claimed that springs can blow up and are difficult to tune. I should back this up some more. Let's explore good old springs a bit before we dive into soft constraints. As you'll see, understanding springs will help us understand soft constraints.

# We can understand springs by studying the harmonic oscillator



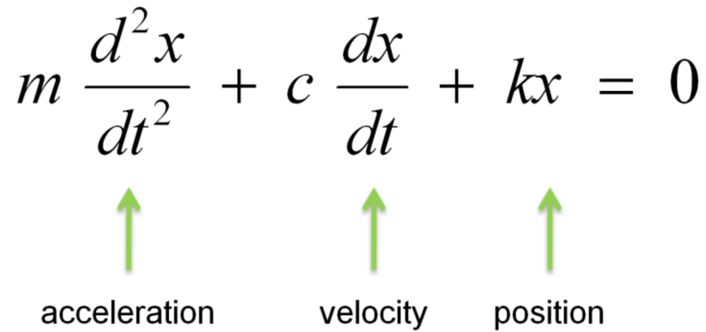

1:1

I’ve been working on physics a long time, so when I look at a harmonic oscillator it is like catching up with an old friend. Sad but true.

So what is the harmonic oscillator? First we start with a mass that can only move in along a single axis, the x-axis in this case. Second we add a ground point that does not move and can support any force. Then we attach a spring and damper between the mass and ground.

The spring acts to maintain the position of the mass along the x-axis. The damper acts to reduce the velocity of the mass. By adjusting the spring and damper constants, we can get many different behaviors.

# The harmonic oscillator has a well known differential equation




1:1:1

To understand how the harmonic oscillator moves, we need its differential equation. Here we have the well known equation of motion for the harmonic oscillator, which is just an expression of Newton's law, force equals mass times acceleration. In this case the spring and damper are the forces.

# Replace c and k with the damping ratio and angular frequency

$$\frac{d^2x}{dt^2} + 2\zeta\omega\frac{dx}{dt} + \omega^2 x = 0$$

$$2\zeta\omega = \frac{c}{m} \qquad \omega^2 = \frac{k}{m}$$

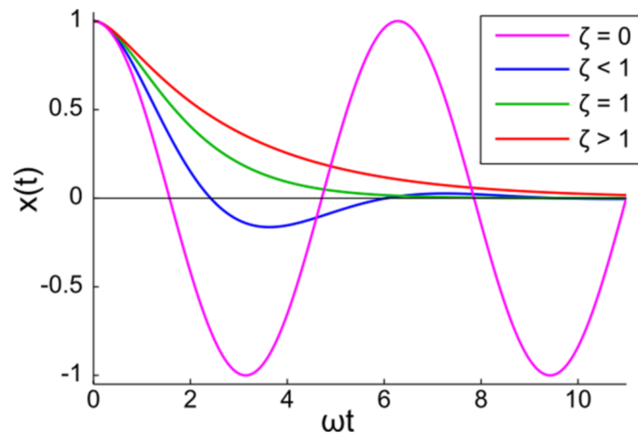
1:1:2

We can understand the motion of the harmonic oscillator by introducing the damping ratio (zeta) and angular frequency (omega). So we have reduced the number of constants from 3 to 2. However, the differential equation remains the same.

The damping ratio is dimensionless and controls the amount of oscillation in the solution.

The angular frequency has units of radians per second and controls the rate of oscillation.

# The damping ratio and frequency describe the behavior




1:1:3

We can now look at the solutions of the differential equation for various damping ratios.

A small damping ratio allows the mass to oscillate back and forth unhindered.

A larger damping ratio causes the oscillation to decay to zero over time. If the damping ratio is less than one, then there will be some oscillation. This system is said to be under-damped.

Once the damping ratio hits one, all oscillation is gone. This is called critical damping. Larger values just slow down the mass further.

The angular frequency just controls the number of oscillations per second. We can get a quicker response by using a larger angular frequency.

# Choose your numerical integrator wisely

1:2

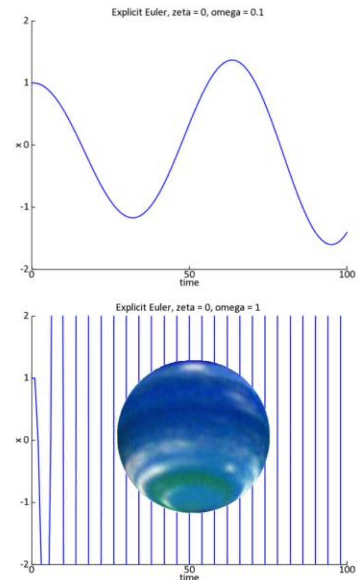
The harmonic oscillator has an exact solution. Unfortunately we can rarely use the exact solution most scenarios that involve multiple dimensions, multiple bodies, and constraints.

So we have to use a numerical integrator to solve the differential equations for our simulations. A numerical integrator is an algorithm for taking the current position and velocity of the system and predicting a future position and velocity. Usually we make the predictions over small time steps to keep errors small and to provide timely transforms for rendering.

There are many choices for numerical integration. I'll show you a few integrators and you will see that some integrators are clearly better than others.

# The explicit Euler integrator is fast but unstable

$$x_2 = x_1 + hv_1$$
$$v_2 = v_1 - h\omega^2 x_1$$




1:2:1

This is the classic explicit Euler integrator. Here we using it to solve the harmonic oscillator with zero damping.

We use a time step h and update the position and velocity step by step using these formulas.

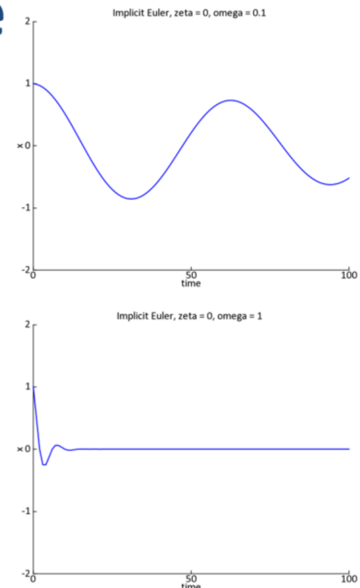
It is amazing that this integrator is ever considered because it always blows up (when damping is zero). In the bottom figure we have a mildly stiff spring that is on its way to Neptune (and back again).

The reason it blows up is because it extrapolates position and velocity based on the current slope. This causes overshoot and the stiffer the spring, the more overshoot. Extrapolation is cheap, but it is wrong more often than right.

Stay far, far, away from explicit Euler.

# The implicit Euler integrator is slow but unconditionally stable

$$x_2 = x_1 + h v_2$$
$$v_2 = v_1 - h\omega^2 x_2$$




1:2:3

The implicit Euler integrator uses the updated position and velocity. This creates some difficulty because we don't have the updated position and velocity. So these equations are implicit. We have to solve them.

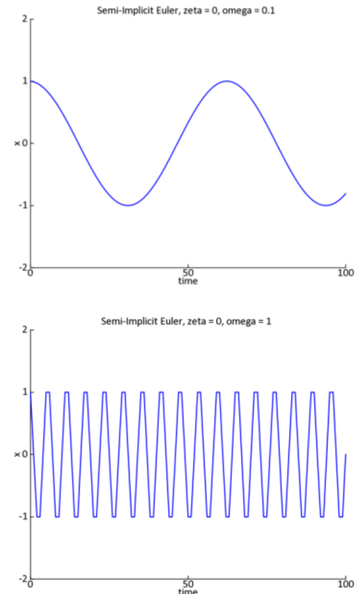
In this case the equations are linear, so solving them is not a big deal. In more general cases we are looking at multi-dimensional non-linear equations that are expensive to solve.

If we go through the pain of solving the implicit Euler formula, we are rewarded with an extremely stable solution. The larger the time step, the more energy is absorbed. There is no limit to the maximum time step in terms of stability. However, larger time steps may make solving the equations more difficult.

Implicit Euler is generally not used for rigid body simulation due to the excessive expense.

# The semi-implicit Euler integrator is fast and stable enough

$$
\begin{aligned}
x_2 &= x_1 + h v_2 \\
v_2 &= v_1 - h\omega^2 x_1
\end{aligned}
$$



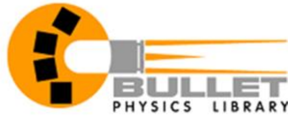

1:2:2

We can make a small change to explicit Euler to get a much better result. By advancing the position based on the updated velocity, we get the semi-implicit Euler integrator. We already have the updated velocity from the first equation, so the algorithm is fast.

As you can see, semi-implicit Euler is stable and conserves energy.

Semi-implicit Euler will eventually blow up if you take big time steps. A general rule is to take at least 4 time steps per period of oscillation. For example, if the oscillation frequency is 60Hz, then you shouldn't take time steps slower than 15Hz.

# Semi-implicit Euler is used by most rigid body engines



Semi-implicit Euler is the integrator of choice for most physics engines. It is cheap, stable, and works well with rigid body constraints. In fact, most modern constraint solvers are designed to work with semi-implicit Euler.

So this creates a bit of trouble when we try to integrate springs into a physics engine. The springs will be stable at low stiffness values, but they can become unstable if they are too stiff. We would like a spring that is always stable inside the constraint solver framework. As you will see this need is answered by soft constraints.

# Designers should never choose the spring stiffness

k = 1

k = 10

k = 0.1

k = 0.0000001

k = 10000000

k = asdf!@$(@#$

1:3

Have you ever tried to tune springs in a real game? I've seen it in practice and it is bad.

Usually you start with 1 and then multiply or divide by 10 until you get into the right ballpark. However, along the way, your simulation can start blowing up. We may be able to recover from this in simple systems. However, if there are multiple springs and bodies it may be quite hard to tune them all simultaneously.

Another problem is that the desired stiffness may not be stable. At this point, you have no choice but to reduce the time step, leading to a significant performance penalty. Typically the whole physics scene is stepped together, so the whole system can become inefficient due to one stiff spring.

This kind of trial and error is not friendly to designers and often requires significant programmer intervention.

# The units for c and k are not intuitive

$$c = \frac{mass}{time}$$

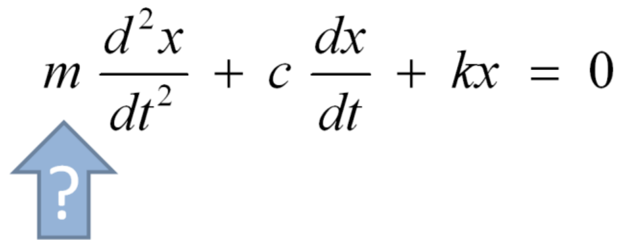
$$k = \frac{mass}{time * time}$$

1:3:1

Here’s one explanation for the difficulty in tuning the spring and damper constants: they have units that are baffling to most humans.

Therefore, we should think of c and k as engineering values, not game values.

# Selecting c and k is pointless without the mass




1:3:2

Another problem with tuning c and k is the mass. Often in games the mass is not known at the time c and k need to be tuned.

However, the damping ratio and frequency depend on the mass. Also, we may establish the damping and spring constants at one point and then later the mass changes, invalidating c and k.

Also, for rigid bodies the mass is seen by the spring is not simply the mass of the body, it also includes the effect of rotational inertia.

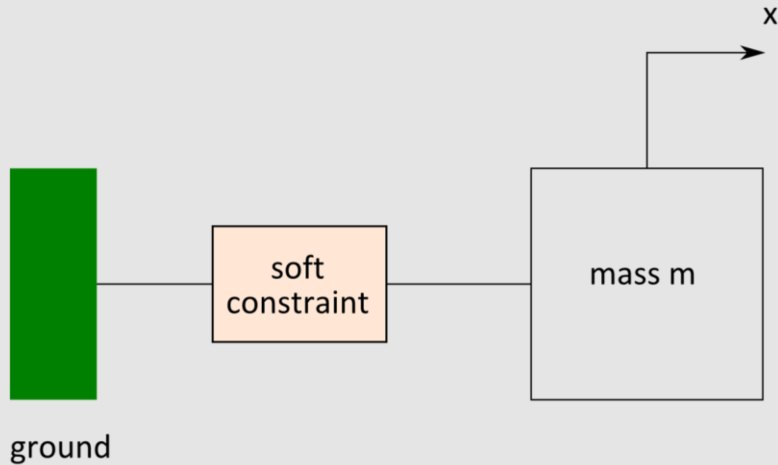
# Soft constraints are stable

2

Let’s talk about these mysterious Buddha springs. The nice thing about these springs is that they are stable.

Mathematically, they are not springs at all, but rather they are modified rigid constraints. I’ll now walk you through the development of soft constraints and show you why they are stable. You’ll even learn a bit about those magic formulas I showed earlier.

# We can understand soft constraints in one dimension



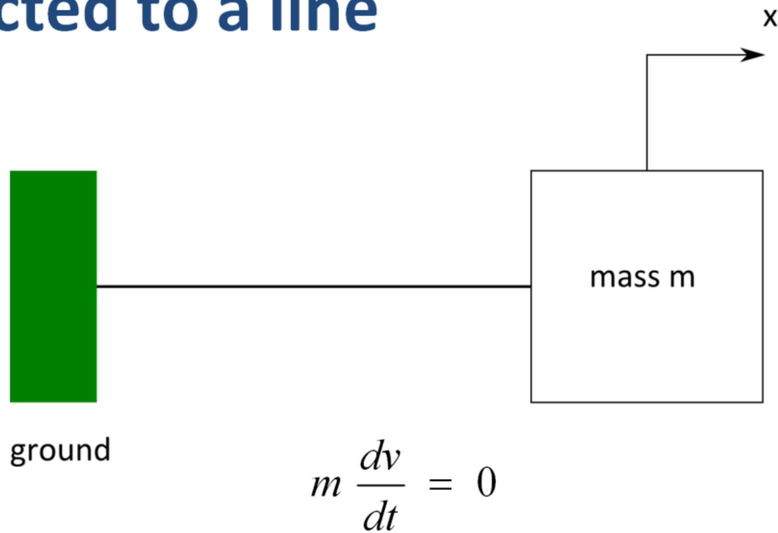

2:1

We can start to understand soft constraints by taking our harmonic oscillator and ripping out the spring and damper.

Instead we put we put in a mysterious box that contains the soft constraint.

In the following I will describe what is in the box.

# Consider a point mass that is restricted to a line

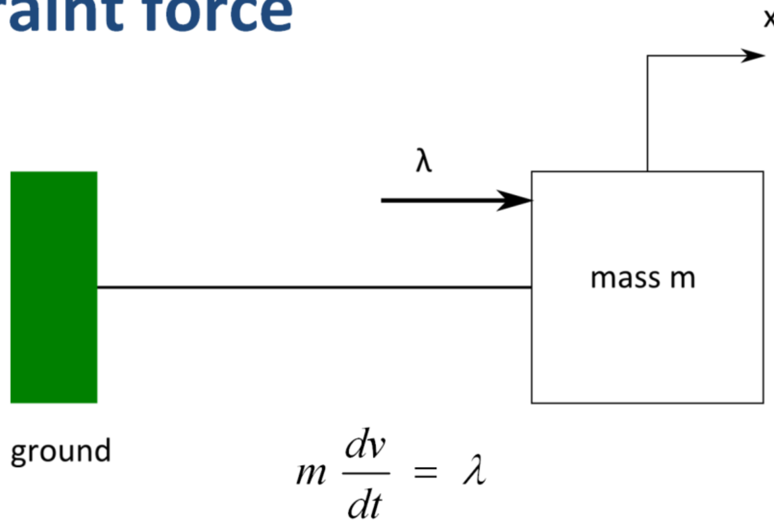



$$m \frac{dv}{dt} = 0$$

2:1:1

First we start with our mass that is free to move along the x-axis. There are no forces so the acceleration is zero.

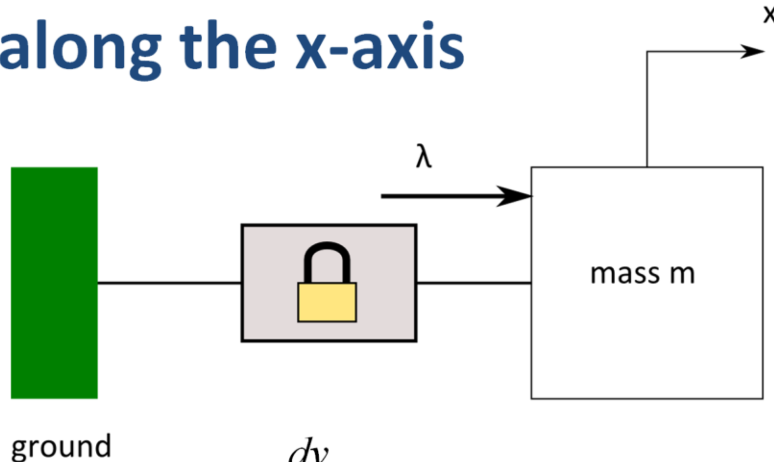
# Halt motion along the line using a constraint force




2:1:2

Now we bring in a constraint force lambda that acts along the x-axis. The job of this constraint is to halt motion along the x-axis.

# The rigid constraint prevents motion along the x-axis




$$m \frac{dv}{dt} = \lambda$$

$$v = 0$$

2:1:3

The constraint force seeks to keep the x-axis velocity at zero.

So we have a constraint force lambda and a simple velocity constraint that says v = 0.

This is now the simplest rigid constraint I can imagine.

# The rigid constraint can be softened with two parameters

$$m \frac{dv}{dt} = \lambda$$

$$v + \frac{\beta}{h} x + \gamma \lambda = 0$$

2:2

Now we are going to make the constraint much more interesting by adding some mathematical magic.

The acceleration equation stays the same.

However, we modify the velocity constraint by adding two terms with two new constants: beta and gamma. Recall that h is the time step.

# Gamma softens the velocity constraint

$$v + \gamma\lambda = 0$$

2:2:2

The second parameter, gamma, is even stranger than beta.

Basically the purpose of gamma is to soften the rigid constraint.

Gamma feeds the constraint force into the velocity constraint. So how could this possibly help?

Well, feeding back the constraint force allows the velocity to be non-zero. For example suppose some force, such as gravity, is pulling on the mass in the positive x-direction. Then the velocity would tend to be positive. The constraint force will resist and it will be negative. Suppose gamma is a small positive number. Then the feedback term will diminish the apparent positive velocity. This in turn reduces lambda.

Eventually v and lambda find a balance that depends on the value of gamma. In general, larger values of gamma make the constraint softer.

Gamma cannot do the job alone. We need beta to keep the position centered on the origin.

# Beta controls the position error and stores energy

$$v + \frac{\beta}{h} x = 0$$

2:2:1

Let’s try to make sense of these new terms.

The constant beta serves to feed back the position error to the velocity. So if the mass has move from zero, the velocity will be adjusted to return the mass to zero. This sometimes called Baumgarte stabilization, after the inventor of the technique.

So instead of trying to compute a force to drive the velocity to zero, we will have a slightly different constraint force that adjusts the velocity to remove the position error. With this adjustment, the system can now store energy, like a spring.

This is similar to a spring, but it is not the same. With a spring we apply a force to affect the acceleration. Here we are trying to adjust the velocity and the constraint force will be determined to satisfy this velocity constraint.

This subtle, yet crucial difference should not be overlooked.

# We solve the system using semi-implicit Euler

$$v_2 = v_1 + \frac{h}{m}\lambda$$

$$x_2 = x_1 + hv_2$$

$$v_2 + \frac{\beta}{h}x_1 + \gamma\lambda = 0$$

2:2:3

We can solve the soft constraint using our integrator of choice, semi-implicit Euler.

We have a velocity and position update. We need the velocity constraint to solve for lambda.

We could probably do some direct analysis of the stability of this system, but there is a better way that is indirect.

# Softness parameters are related to spring and damping coefficients

$$\begin{bmatrix} c \\ k \end{bmatrix} \Leftrightarrow \begin{bmatrix} \gamma \\ \beta \end{bmatrix}$$

2:3

It turns out that the softness parameters can be related to the damping and spring constants.

At first glance, this must seem unreasonable. I just told you that springs are bad and soft constraints are so much better. Why would we want to relate soft constraints to those awful springs?

There is a reason!

# Compare the spring-damper and constrained solutions

Spring-Damper

$$v_2 = \frac{v_1 - \frac{hk}{m} x_1}{1 + \frac{hc}{m} + \frac{h^2 k}{m}}$$

Implicit Euler

Soft Constraint

$$v_2 = \frac{v_1 - \frac{\beta}{m\gamma} x_1}{1 + \frac{h}{m\gamma}}$$

Semi-implicit Euler

2:3:1

We talked about one integrator that was always stable, no matter the stiffness of the spring. That is the implicit Euler!

Using the implicit Euler formulas, we can compute the velocity update.

We can do the same thing for the soft constraint with semi-implicit Euler.

When we put these formulas side-by-side we notice some interesting similarities. These similarities allow use to derive some formulas relating the softness parameters to the spring and damper constants.

# We can solve for gamma and beta by matching coefficients

Spring-Damper

$$v_2 = \frac{v_1 - \frac{hk}{m} x_1}{1 + \frac{hc}{m} + \frac{h^2 k}{m}}$$

Implicit Euler

Soft Constraint

$$v_2 = \frac{v_1 - \frac{\beta}{m\gamma} x_1}{1 + \frac{h}{m\gamma}}$$

Semi-implicit Euler

2:3:1.5

# The matching solutions show that soft constraints are stable

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

2:3:2

And here are those formulas.

So if we use these formulas, we can get a solution to the soft constraint system that is identical to the implicit Euler solution of the harmonic oscillator.

To me this is a profound result.

Since the implicit Euler solution is always stable, so is the soft constraint.

Notice that gamma and beta each depend on both c and k. So we cannot say that beta is purely like a spring and gamma is not purely a damper.

Also gamma and beta are positive when c and k are positive.

Notice also what happens when k becomes really stiff: gamma goes to zero and beta goes to one.

These formulas are also invertible, so we can compute c and k from gamma and beta. The limitation is that gamma must be non-zero.

Note: we need to adjust gamma when the physics engine deals with impulses instead of forces. This just involves dividing gamma by h.

# The mystery of ERP and CFM (SOLVED)

From the Open Dynamics Engine (ODE) manual:

$CFM = 1 / (h\,k + c)$
$ERP = h\,k / (h\,k + c)$

$$\gamma = CFM$$
$$\beta = ERP$$

”In fact, ERP and CFM can be selected to have the same effect as any desired spring and damper constants.”

Teaser Resolved

So now we have answered the ODE riddle. Gamma is CFM and beta is ERP. And now we know how to derive these formulas.

Now I could have simply asked the ODE developer about these formulas. But where would be the fun in that?

# Any velocity constraint can be softened using two parameters

$$Jv + \frac{\beta}{h} C\left(x\right) + \gamma\lambda = 0$$

2:3:3

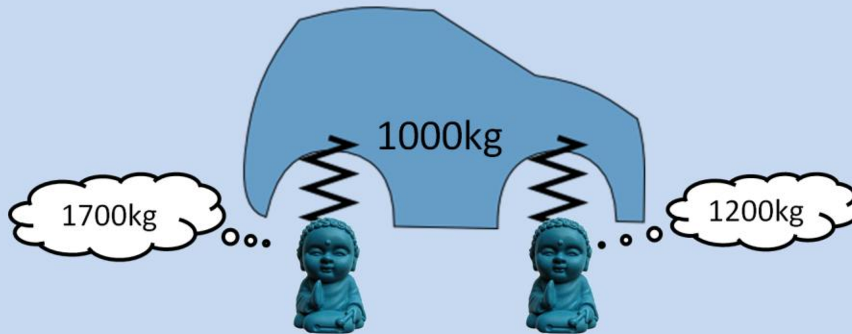
Now we have studied the harmonic oscillator and the soft constraint analog. However, soft constraints are not limited to 1D translation. Any rigid constraint can be modified to be soft.

So you can have a soft distance constraint, soft angular constraint, and so on.

Here is the general formula for a 1D constraint. We now have the constraint Jacobian J and the position error C. Everything else works the same.

I recommend looking at the implementation in Box2D for details.

# The effective mass makes tuning soft constraints easy




3

At the beginning of this presentation I claimed that springs are difficult to tune, in part because the mass seen by springs is not well known.

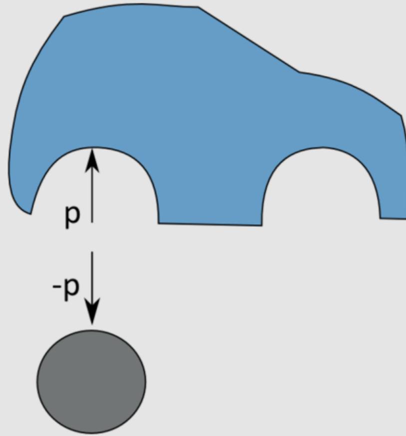
In the realm of rigid constraints, the effective mass fills this role. Using the effective mass makes tuning soft constraints easy.

The effective mass is mass seen by the constraint. So for each constraint the effective mass will likely be different.

We already saw that rotational inertia plays a big part in determining the mass seen by a spring. The same holds true for soft constraints. The effective mass seen by a constraint depends on the position and direction of the constraint. For example, these Buddhas see a different effective mass because they are located at different positions.

Note that the effective mass has nothing to do with weight. We do have rotational inertia, but there is no such thing as rotational weight. The mass seen by the Buddhas does not necessarily add up to the total mass.

# The effective mass is the mass *seen* by the constraint impulse




$$m_{eff} \, v_{rel} = p$$

3:1

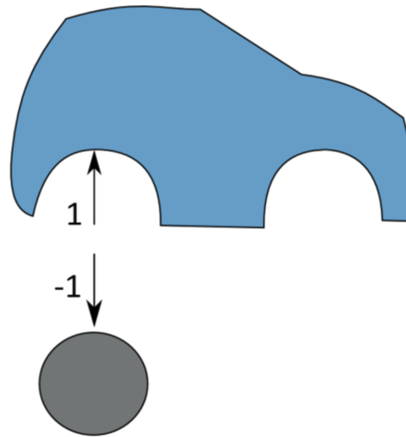
I said that the effective mass is the mass seen by the constraint. In particular it is the mass seen by the constraint impulse.

In this slide lambda represents the constraint impulse, not the force. This is not a big deal because an impulse is just the force times the time step.

Treating lambda as an impulse we write our intended relationship. Suppose the chassis and wheel begin at rest. Then effective mass time the relative velocity is equal to the constraint impulse.

We can use the formula to perform a mental experiment to determine the effective mass.

# Apply a unit impulse along the constraint axis

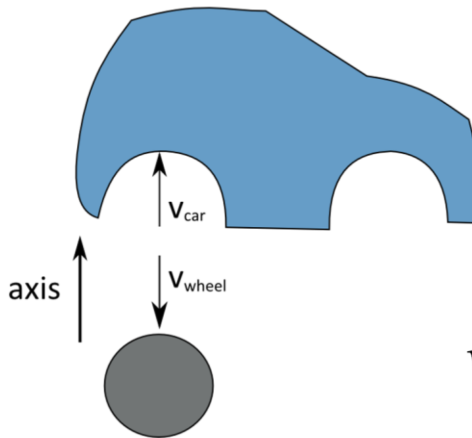



3:1:1

First we apply a unit impulse to the wheel and chassis.

The applied impulses are equal in magnitude but in the opposite direction along the constraint axis. This is going to add some velocity to the car body and wheel.

# Measure the relative velocity along the constraint axis




$$v_{rel} = \left(v_{car} - v_{wheel}\right) \bullet axis$$

3:1:2

Now we measure the resulting velocities of the car and wheel along the axis. We need the velocity at the constraint anchor points, so you'll need to use a cross product along with the center of mass velocity and angular velocity.

Then we subtract the wheel velocity from the car velocity and dot the difference with the constraint axis. So we now have a value for the relative velocity.

# The effective mass is the inverse of the relative velocity

$$m_{eff} = \frac{p}{v_{rel}} = \frac{1}{v_{rel}}$$

3:1:3

We can now just solve for the effective mass by dividing the impulse by the relative velocity.

The nice thing about this approach is that we side step the need for constraint Jacobians. You can actually write a constraint solver using this technique.

Now if you didn't follow this derivation, I recommend reading these slides again later and I have also included a reference at then end of the presentation.

# The effective mass is already computed in most solvers

$$m_{eff} = \frac{1}{JM^{-1}J^T}$$

3:2

In most physics engines we already have the effective mass because we need it to solve the constraints.

This is the most general formula for a 1D constraint. The formula uses the constraint Jacobian and mass matrix. You can find details for this in the references.

So we don’t need to do any extra work to get the effective mass.

# Use the effective mass to compute the softness parameters on the fly

$$\begin{bmatrix} \omega \\ \zeta \\ m_{eff} \end{bmatrix} \Rightarrow \begin{bmatrix} \beta \\ \gamma \end{bmatrix}$$

3:3

Now that we have the effective mass we can compute the softness parameters from using the harmonic oscillator constants: frequency, damping ratio, and mass.

The mass comes automatically, leaving just the frequency and damping ratio to be tuned.

# The designer selects the frequency and damping ratio

$$\begin{bmatrix}\omega \\ \zeta\end{bmatrix}$$

3:3:1

So imagine we give a soft constraint to a designer. We ask the designer for the frequency and damping ratio.

This can be simplified further. We can get the frequency by asking for the response speed in frames (e.g. on a 60Hz basis).

For the damping ratio we can simply enumerate a few values: free oscillation, some oscillation, or no oscillation.

# Use the effective mass to compute the spring-damper coefficients

$$k = m_{eff}\omega^2$$
$$c = 2m_{eff}\zeta\omega$$

3:3:2

At run time we compute the effective mass seen by the constraint.

We combine the effective mass with the frequency and damping ratio to compute the spring and damper constants.

These formulas come from the harmonic oscillator.

# The spring-damper coefficients lead to softness parameters

$$\gamma = \frac{1}{c + hk}$$

$$\beta = \frac{hk}{c + hk}$$

3:3:3

Finally we use our magic formulas to compute the softness parameters.

Springs are reliably stable with implicit Euler

Modern rigid body engines use semi-implicit Euler

# Buddha springs bridge the gap between implicit and semi-implicit Euler

implicit Euler

stable

semi-implicit Euler

fast

The answer is: Buddha springs!

## Making them easy to tune will make your designers love you

Stability and tunability work together to make designers love you.

# References

- Downloads: http://box2d.org
- ODE manual: http://opende.sourceforge.net/wiki/index.php/Manual_(All)
- Explaining the rigid body solver: http://tuxedolabs.blogspot.com/2010/08/explaining-rigid-body-solver.html
- Soft impulses: http://bulletphysics.org/Bullet/phpBB3/viewtopic.php?f=4&t=1354
- Claude Lacoursière: http://physics.hardwire.cz/mirror/urn_nbn_se_umu_diva-1143-2__fulltext.pdf

You can get other presentations and the Box2D code at box2d.org.

The ODE manual is still online where you can see the original magic formulas for CFM and ERP.

I have also listed a nice blog post by Dennis Gustafsson that explains an intuitive method for computing the effective mass.

I have included a link to a forum topic where I explain how to use softness in a sequential impulse constraint solver.