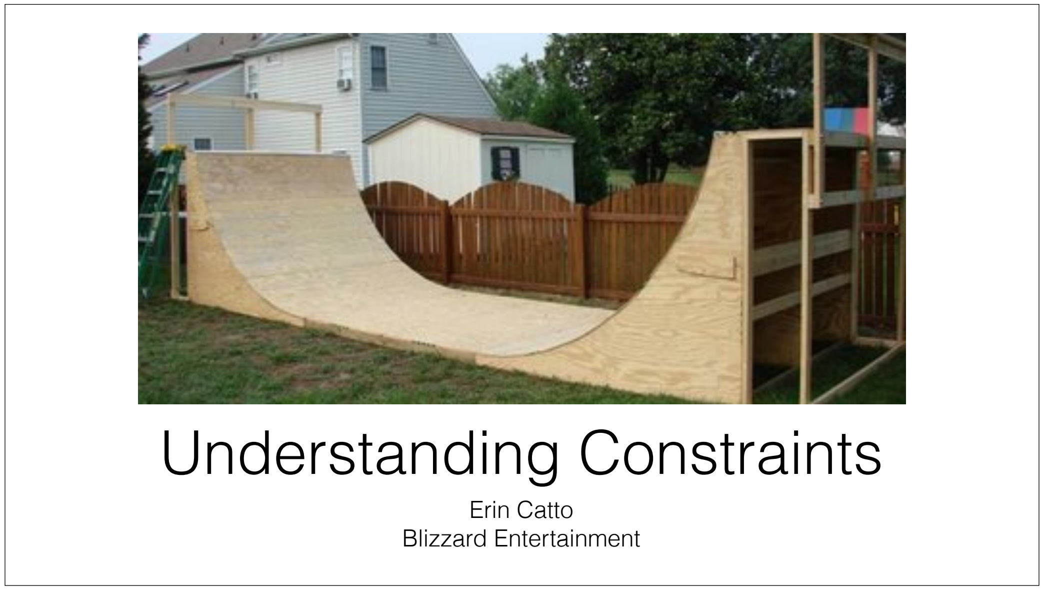
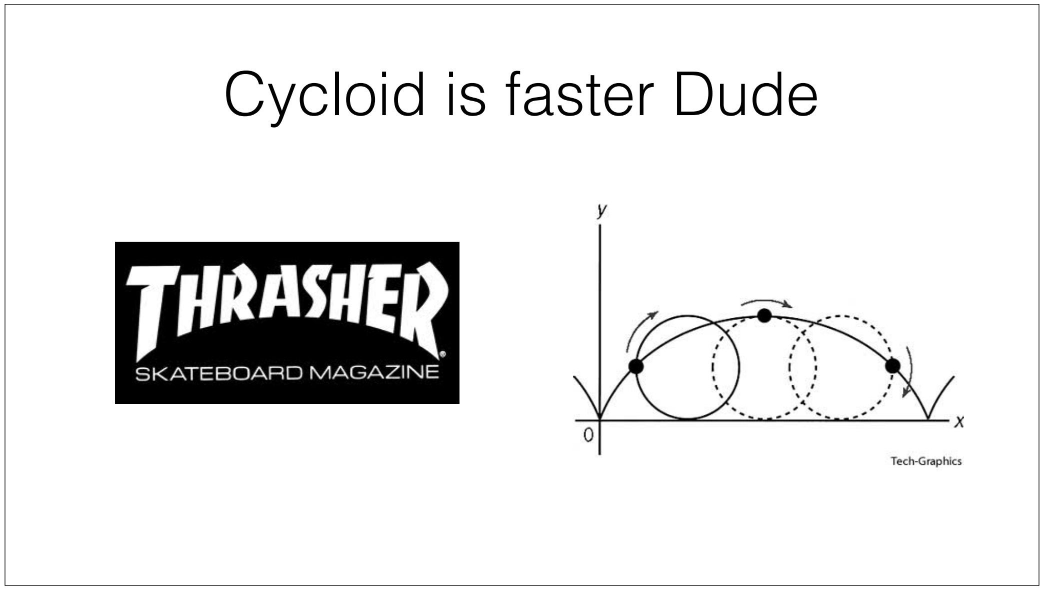


Understanding Constraints

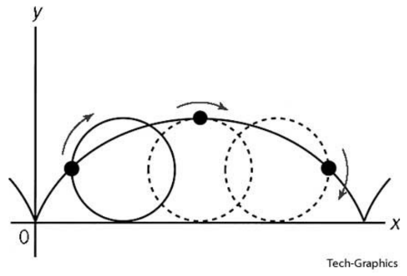
Erin Catto
Blizzard Entertainment

This talk is about constraints, the kind used in game physics. I started my talk last year with a story about Legos. People seemed to like that, so I have another story to start my talk this year. A long time ago when I was a teenager, long before game physics was a thing, I fell in love with skateboarding. I even built a skateboard half-pipe ramp in the backyard of my parent's house in Ohio. This is not the actual ramp, but it looked a lot like this.

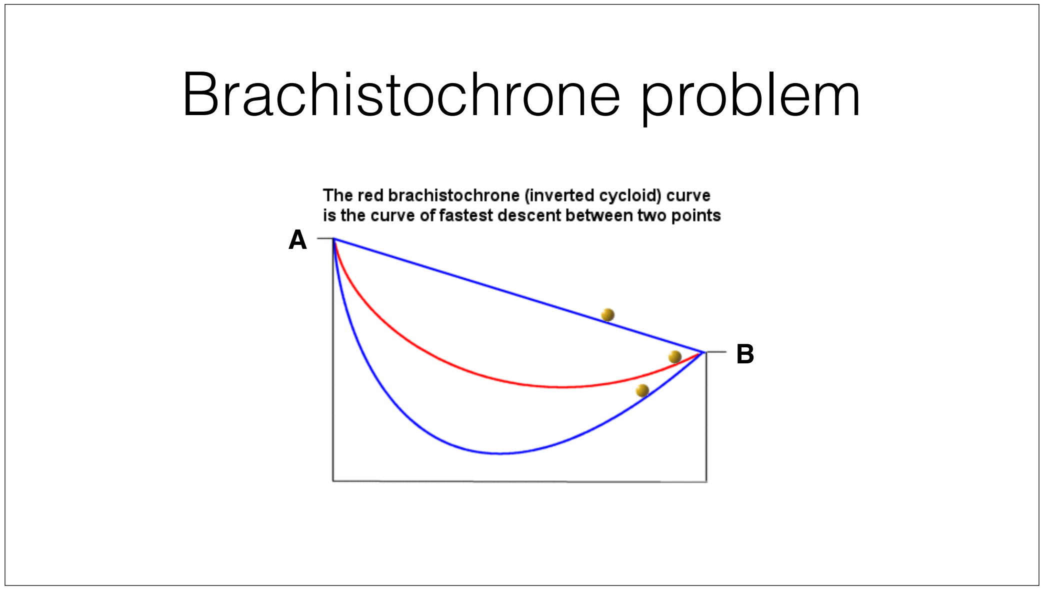


Cycloid is faster Dude

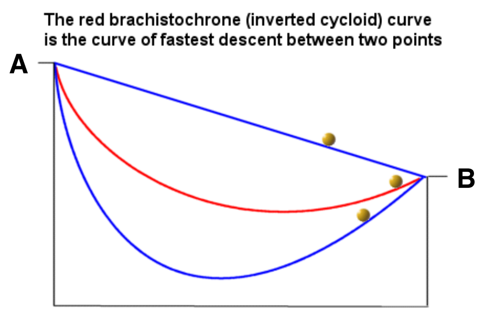
THRASHER
SKATEBOARD MAGAZINE



Back then I used to buy Thrasher magazine every month and I came across an article I found fascinating. The article said that a half-pipe ramp could be made faster by changing its shape. Normally a half-pipe is built as two quarter circles with a flat part in between. The article claimed a faster ramp could be made in the shape of a cycloid. A cycloid is a curve formed by tracing a point on a circle as it rolls without slipping.
I wondered how this could be true. I didn't have the mathematical knowledge to understand this until many years later.

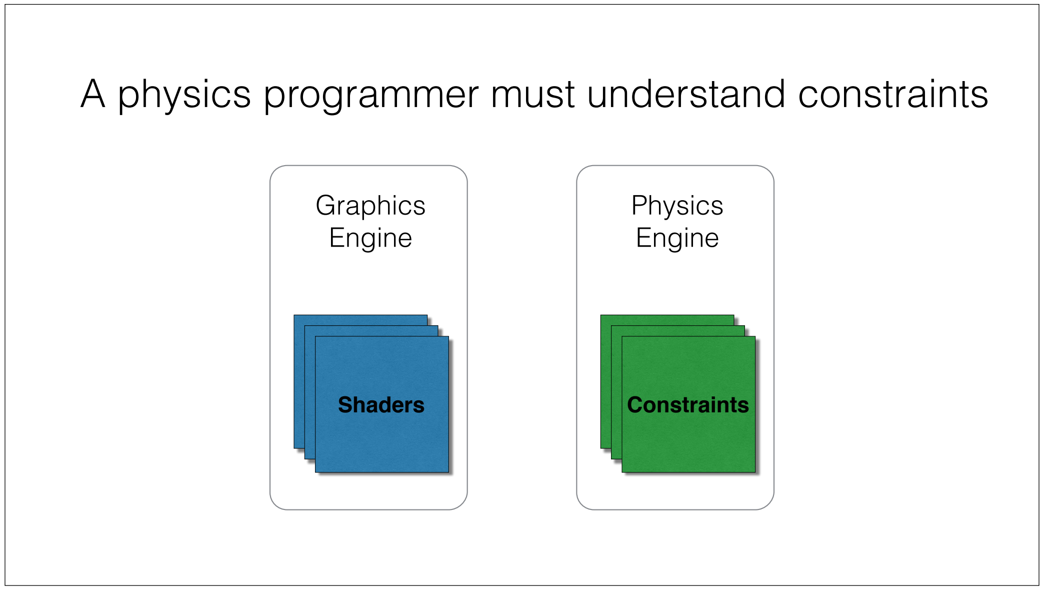


Brachistochrone problem


The red brachistochrone (inverted cycloid) curve
is the curve of fastest descent between two points
A
B

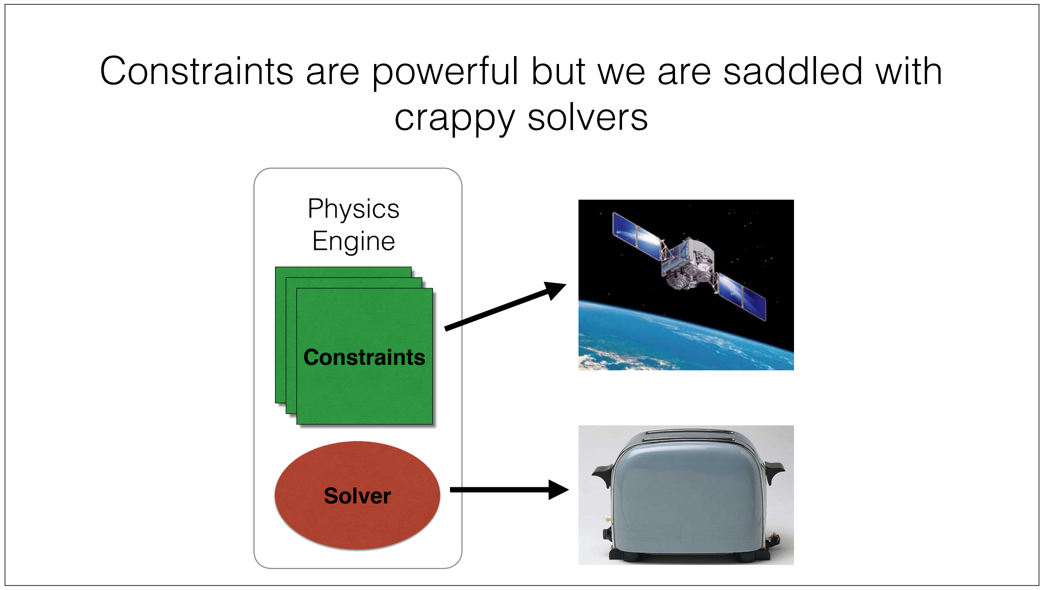
In grad school I was studying an advanced topic called The Calculus of Variations. In my book I found the Brachistochrone problem. Say you want to move a particle subject to gravity from point A to some lower point B. The fastest way to move that particle is along an inverted Cycloid. You can derive this using The Calculus of Variations. So I finally understood what I had read about in Thrasher magazine many years before.

At this point I realized that math can solve some really useful problems. But even more interesting, I realized that understanding math was its own reward. And I feel the same way about my research into game physics. Understanding constraints is incredibly useful, but understanding constraints is also its own reward.

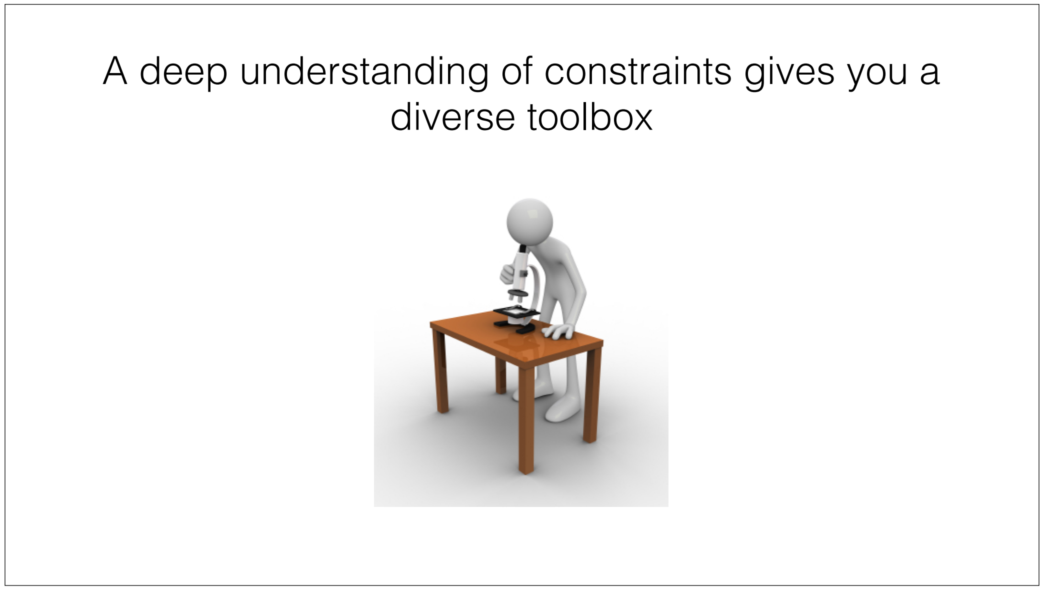


A physics programmer must understand constraints

As physics programmers it is our job to understand constraints. Constraints are as fundamental to game physics as shaders are to graphics programming. With shaders, you can use the standard recipes for lighting, shadows, etc., or you can make up your own shaders and create something unique. Just like shaders, you can use the standard constraints like revolute, prismatic, etc., or you can make up your own constraints and create something unique. Constraints are an area of physics programming where we get to show our knowledge and creativity.

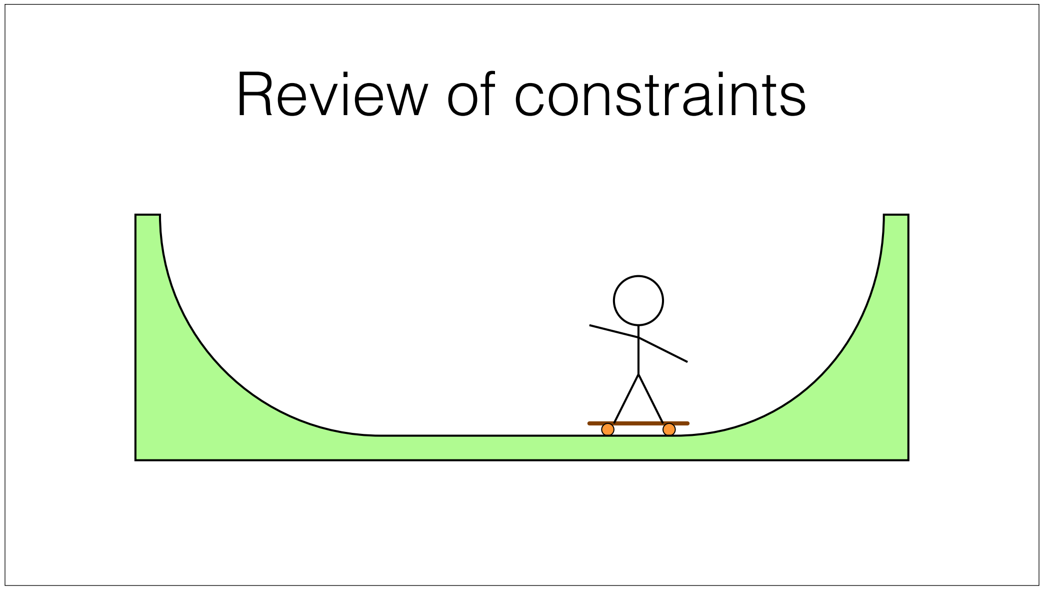

Constraints are powerful but we are saddled with crappy solvers
Physics Engine
Constraints
Solver

We can model and program our constraints perfectly. Good enough to drive a robot arm on an assembly line or launch a satellite into space. Unfortunately for games, we do not have enough cycles to solve constraints accurately. Solving constraints precisely requires cubic time and quadratic space. For games we make due with linear time and space, so we are constantly wrestling with our solvers. The better you understand constraint solvers, the better chance you'll have of creating robust simulations in your game.

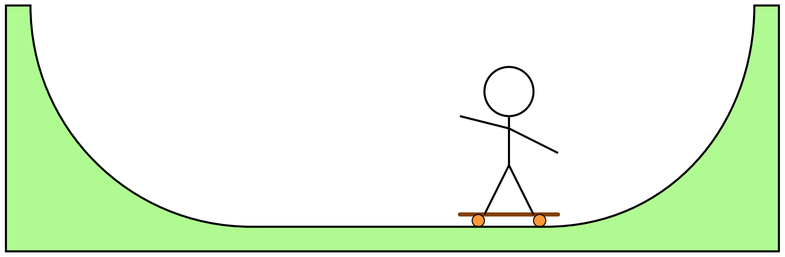


A deep understanding of constraints gives you a diverse toolbox

You can use standard constraints and solvers to create rag dolls and destruction. If you dig deeper you can create motorized rag dolls and soft constraints. But you can go even further: you can create block solvers, position solvers, and character solvers.

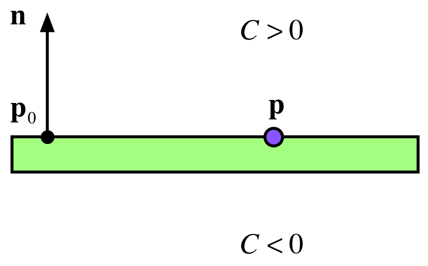


Review of constraints



I've been talking about constraints at the GDC since 2005, so I hope you are familiar with constraints and their ingredients. Nevertheless I will do a bit of review.
Everything starts with a position constraint function.
Imagine a skateboarder riding in this half-pipe. The skateboard is constrained to the surface of the ramp, but it can move freely along that surface. This is a position constraint.

Position constraint for bottom of ramp



$$C(\mathbf{p}) = 0$$

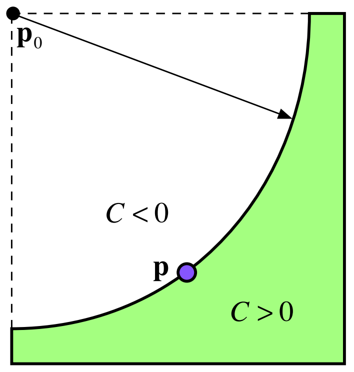
$$C(\mathbf{p}) = \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0)$$

Let’s build the position constraint C for the bottom of the ramp.

For simplicity, consider the skateboard to be a particle at position p. The position constraint C is a scalar function of p. It must be equal to zero when the constraint is satisfied. Otherwise it must be non-zero.

The bottom of the ramp is flat. So I have defined C to be the signed distance of the particle from the plane. Notice that C is positive if p is above the plane and negative if p is below the plane.

Position constraint for circular section

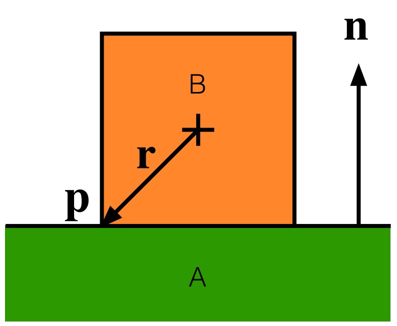


$$C(\mathbf{p}) = \left\| \mathbf{p} - \mathbf{p}_0 \right\| - r$$

The position constraint has a different form on the curved part of the ramp. Now the point p is restricted to the circle centered at p0. So I defined C to be the distance of p from p0 minus the radius r.

In many cases it is useful to think of C as an implicit surface. In this case C is an implicit representation of a circle.

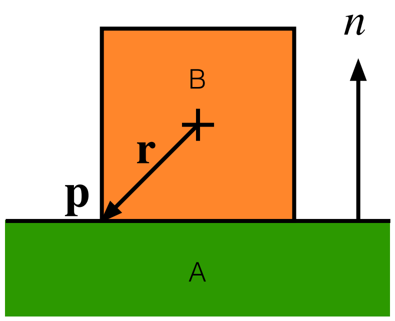
Recap: contact constraint



$$C = \left(\mathbf{p}_B - \mathbf{p}_A\right) \cdot \mathbf{n}$$

Now lets look at a constraint that is ubiquitous in game physics: the contact constraint. I have a box B on a fixed plane A and I'm going derive the contact constraint for the lower left corner. The contact constraint says that the contact points on the touching bodies don't move relative to each other along the contact normal vector n. For now, I'm treating this as a bilateral constraint. Later I will discuss inequality constraints.

Contact velocity constraint



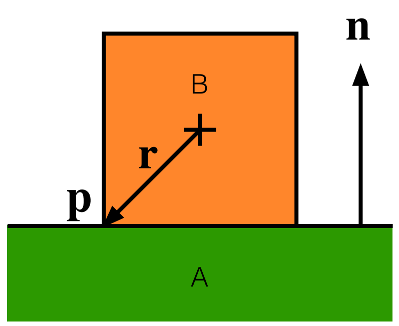
$$C = (\mathbf{p}_B - \mathbf{p}_A) \cdot \mathbf{n}$$
$$\dot{C} = (\dot{\mathbf{p}}_B - \dot{\mathbf{p}}_A) \cdot \mathbf{n} + (\mathbf{p}_B - \mathbf{p}_A) \cdot \dot{\mathbf{n}}$$

So we have the position constraint, but as you will see, I also need the velocity constraint for the solver.

I compute the time derivative of the position constraint to get the velocity constraint. The chain rule of differentiation produces two terms.

The dot above a variable denotes differentiation with respect to time.

Simplification



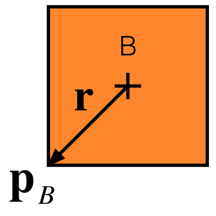
$$\dot{C} = \left(\dot{\mathbf{p}}_B - \cancel{\dot{\mathbf{p}}_A}\right)\cdot\mathbf{n} + \left(\cancel{\mathbf{p}_B - \mathbf{p}_A}\right)\cdot\dot{\mathbf{n}}$$

$$\dot{C} = \dot{\mathbf{p}}_B\cdot\mathbf{n}$$

I can now make a couple simplifications. Let’s assume body A is ground. Then the velocity of A is zero.

I can also throw out the second term because pA and pB are coincident.

Center of mass velocity



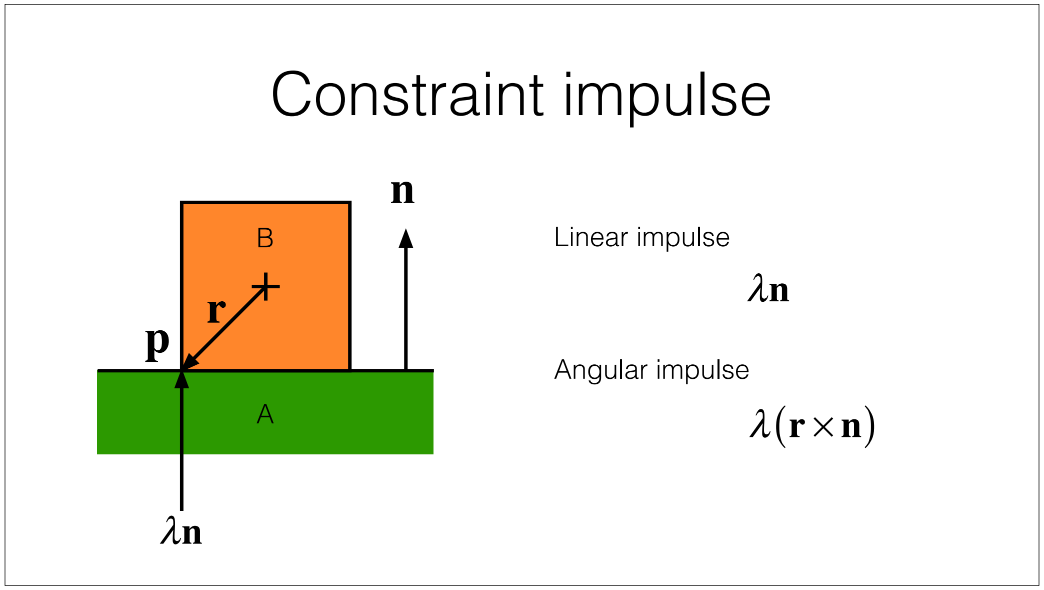
$$\boxed{\dot{\mathbf{p}}_B = \mathbf{v}_B + \boldsymbol{\omega}_B \times \mathbf{r}}$$

$$\dot{C} = \left(\mathbf{v}_B + \boldsymbol{\omega}_B \times \mathbf{r}\right) \cdot \mathbf{n}$$

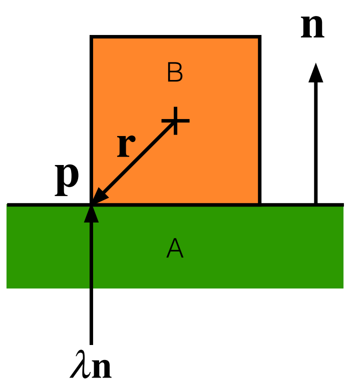
The solver works deals with the center of mass velocity.

Therefore, I need to express the contact velocity constraint in terms of the center of mass velocity v.

I can express the velocity of the contact point in terms of the center of mass velocity using this standard formula that uses the angular velocity omega and the radius vector r.



Constraint impulse



Linear impulse

$$\lambda\mathbf{n}$$

Angular impulse

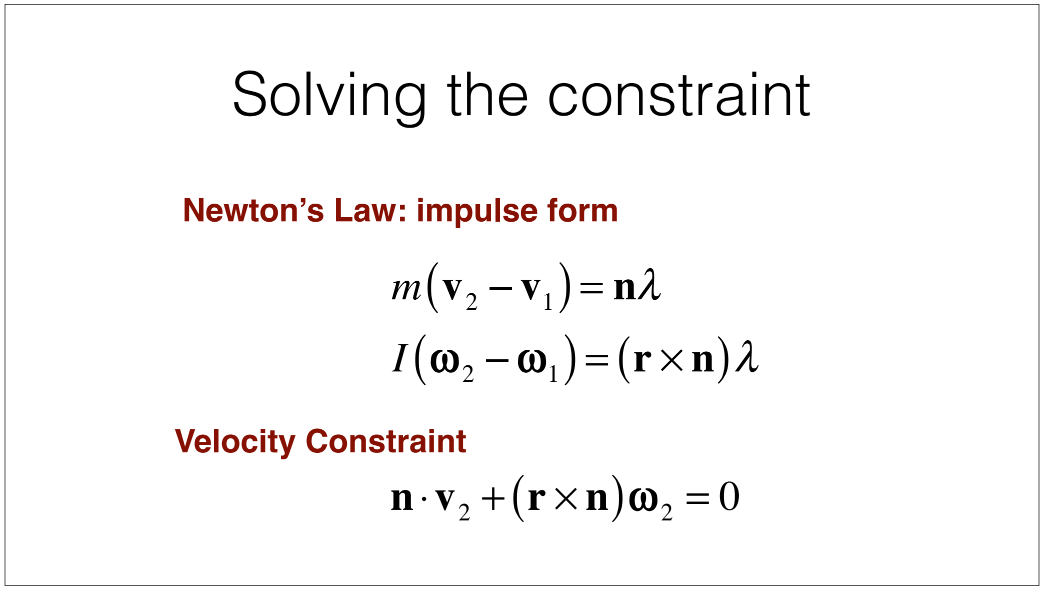
$$\lambda(\mathbf{r}\times\mathbf{n})$$

Now I have the contact velocity constraint and I need a way to ensure the velocity constraint is satisfied. The solver works by applying impulses to steer velocities such that the velocity constraints are satisfied. So lets setup the contact impulse so we can feed it to the solver.

The contact impulse is applied along the contact normal at the contact point p. The signed magnitude of the impulse is lambda.

I need to compute a value for lambda so that the velocity constraint is satisfied. This will help keep the box from sinking into the floor.

The contact impulse does not line up with the center of mass so the box experiences a linear impulse and an angular impulse.



Solving the constraint

Newton's Law: impulse form

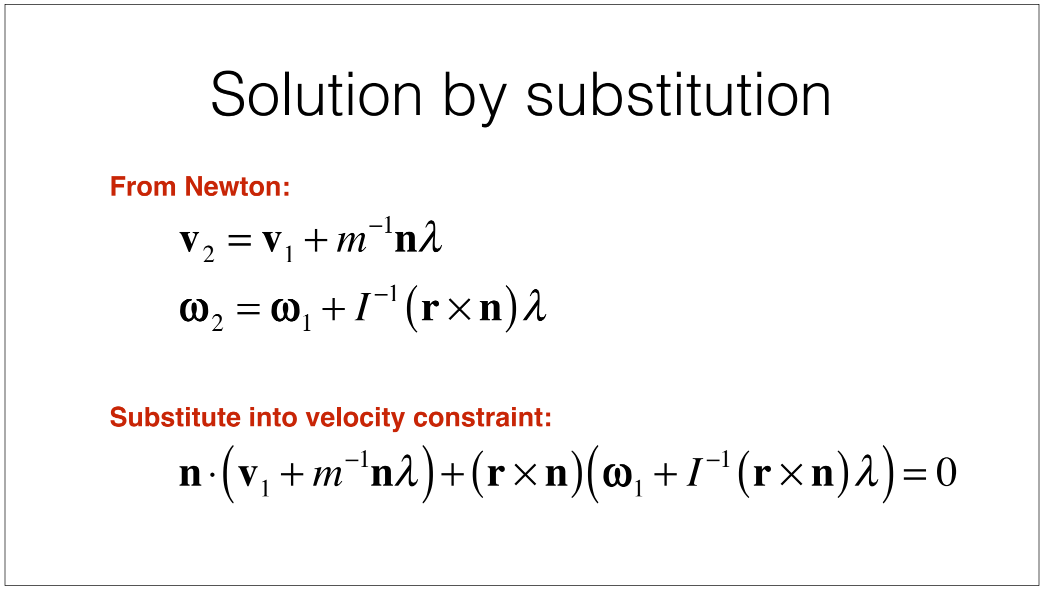
$$m(\mathbf{v}_2 - \mathbf{v}_1) = \mathbf{n}\lambda$$

$$I(\boldsymbol{\omega}_2 - \boldsymbol{\omega}_1) = (\mathbf{r} \times \mathbf{n})\lambda$$

Velocity Constraint

$$\mathbf{n} \cdot \mathbf{v}_2 + (\mathbf{r} \times \mathbf{n})\boldsymbol{\omega}_2 = 0$$

The question now is how to compute a value of lambda so that the velocity constraint is satisfied.
I need to use Newton's law of motion to solve for lambda. The first equation is the linear equation of motion that relates the change in linear velocity to the impulse. The second equation is the angular equation of motion involving the inertia tensor I and the change in angular velocity. The third equation is the velocity constraint in terms of v2 and omega2.
Now we have enough equations to solve for lambda.



Solution by substitution

From Newton:

$$\mathbf{v}_2 = \mathbf{v}_1 + m^{-1}\mathbf{n}\lambda$$

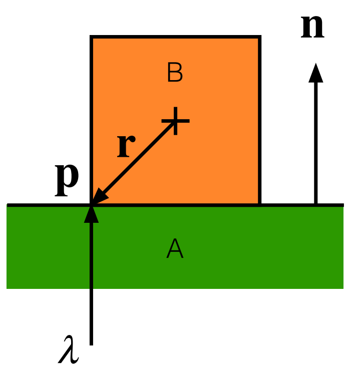
$$\boldsymbol{\omega}_2 = \boldsymbol{\omega}_1 + I^{-1}(\mathbf{r}\times\mathbf{n})\lambda$$

Substitute into velocity constraint:

$$\mathbf{n}\cdot\left(\mathbf{v}_1 + m^{-1}\mathbf{n}\lambda\right) + (\mathbf{r}\times\mathbf{n})\left(\boldsymbol{\omega}_1 + I^{-1}(\mathbf{r}\times\mathbf{n})\lambda\right) = 0$$

Using the equations from the previous slide: solve for v2 and omega2 in terms of lambda. Next, I can then eliminate v2 and omega2 from the third equation. This results in an equation where lambda is the only unknown. So I can solve for lambda.

Impulse formula and effective mass



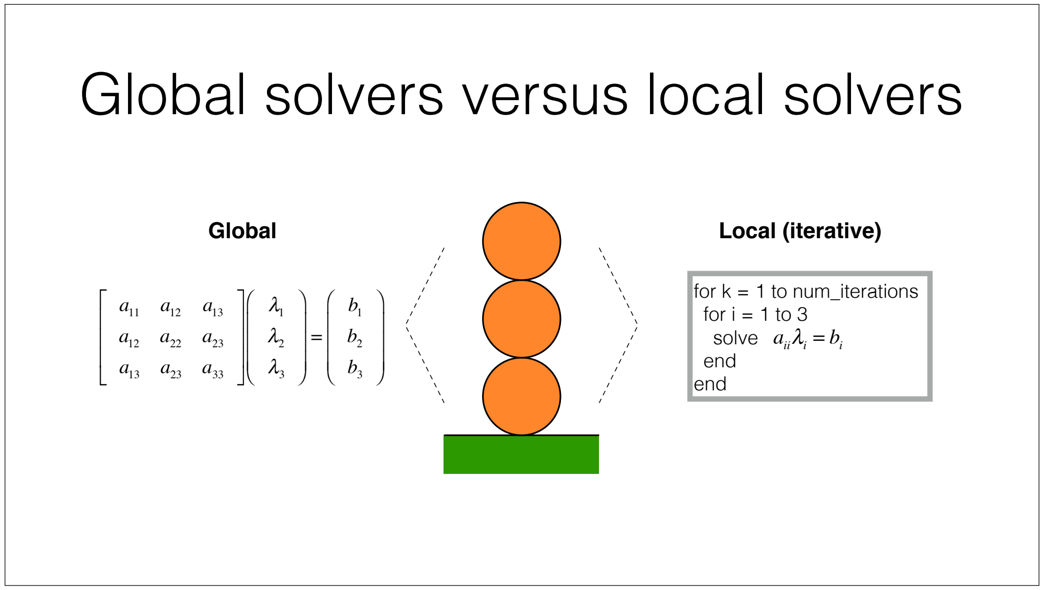
$$\boxed{\lambda = -m_{eff}\dot{C}_1}$$

(impulse = mass * velocity)

$$\boxed{m_{eff} = \frac{1}{m^{-1} + I^{-1}(\mathbf{r}\times\mathbf{n})^2}}$$

(effective mass)

I now have a formula for computing lambda in terms of the initial velocity constraint error and a new grouping of terms called the *effective mass*. The effective mass is the inertia projected onto the constraint axis and represents the inertial resistance seen by the constraint impulse. Notice that the effective mass has units of mass.



Global solvers versus local solvers

Global

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Local (iterative)

```
for k = 1 to num_iterations
 for i = 1 to 3
  solve  $a_{ii}\lambda_i = b_i$
 end
end
```

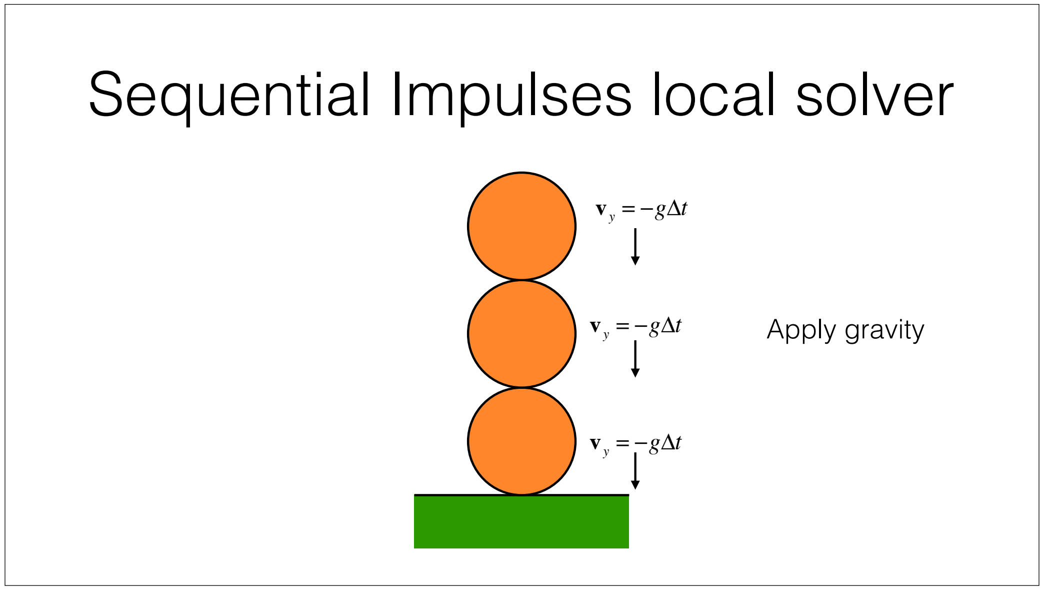
So now we know how to solve a single constraint. How about multiple constraints?

Solving multiple constraints is difficult because impulses applied at one constraint effect the errors in other constraints. Constraints should be globally consistent. However, computing the global solution requires building and factoring a matrix. If there are a hundreds constraint, you will have to build and factor a 100x100 matrix. This will likely be too slow for most games.

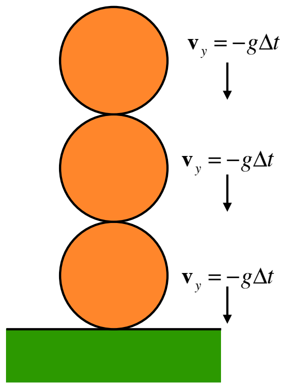
Therefore many physics programmers use local solvers. Local solvers solve one constraint at a time with the hope that the global solution will converge given enough iterations.

Iterative solvers use linear space and time, so they are practical for games. However we often do not give the solver enough iterations to converge. Often we allow for 10 or less iterations. This often gives us a low accuracy solution.

We can often mix global and local solvers to obtain a better result. I'll talk about this later in the context of block solvers.

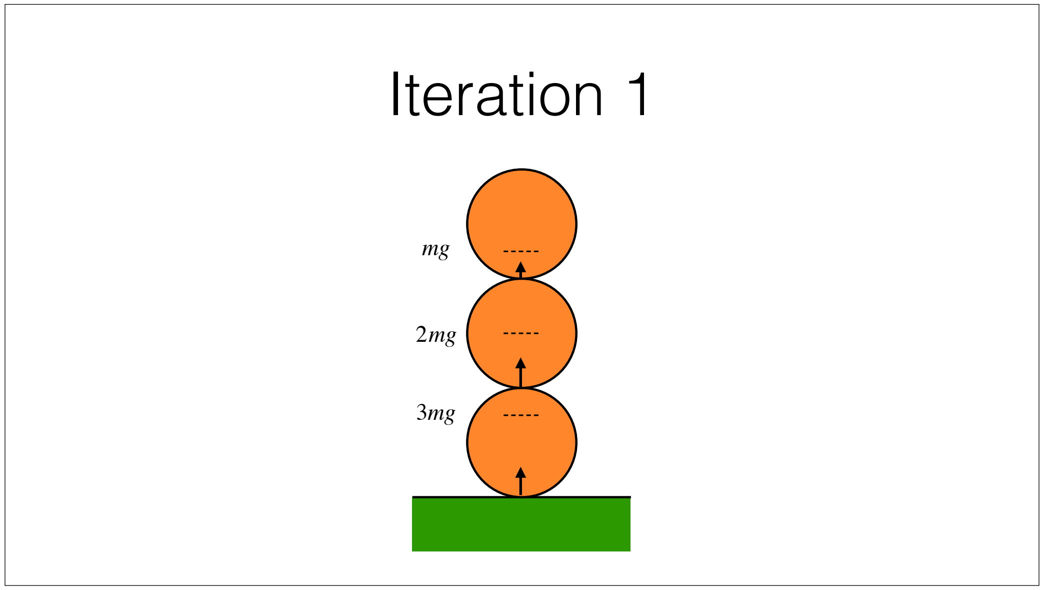


Sequential Impulses local solver

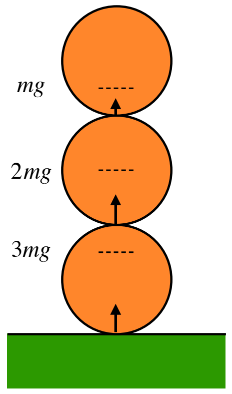

$\mathbf{v}_y = -g\Delta t$
$\mathbf{v}_y = -g\Delta t$
$\mathbf{v}_y = -g\Delta t$

Apply gravity

I use a local solver called Sequential Impulses. Here is an example of how the Sequential Impulses works. Suppose we have a stack of circles that are all at rest. Gravity is pushing the circles down but the stack is blocked by a ground plane, so we need to compute the contact impulses to counteract gravity.
At the beginning of the time step, I apply gravity and each circle starts moving down with the same velocity.

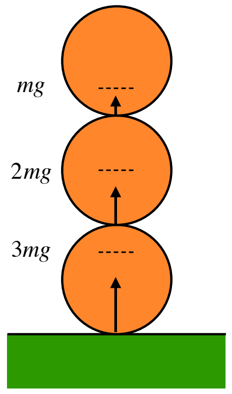


Iteration 1

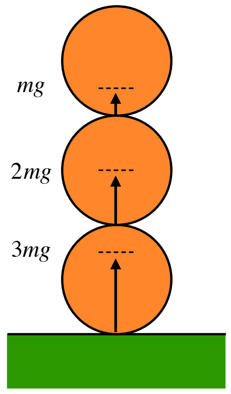


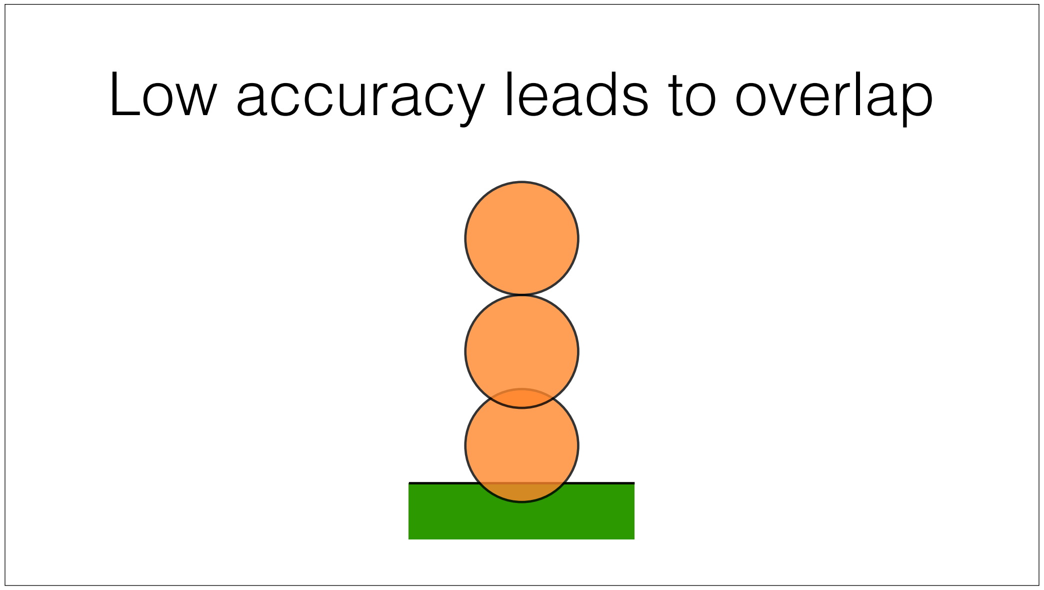
Each contact impulse needs to be large enough to support the weight of the circles above (indicated by the dashed lines).
The SI algorithm solves one constraint at a time in arbitrary order. After one iteration there is significant error in the contact impulses. I can continue to iterate to improve the impulses.

Iteration 2



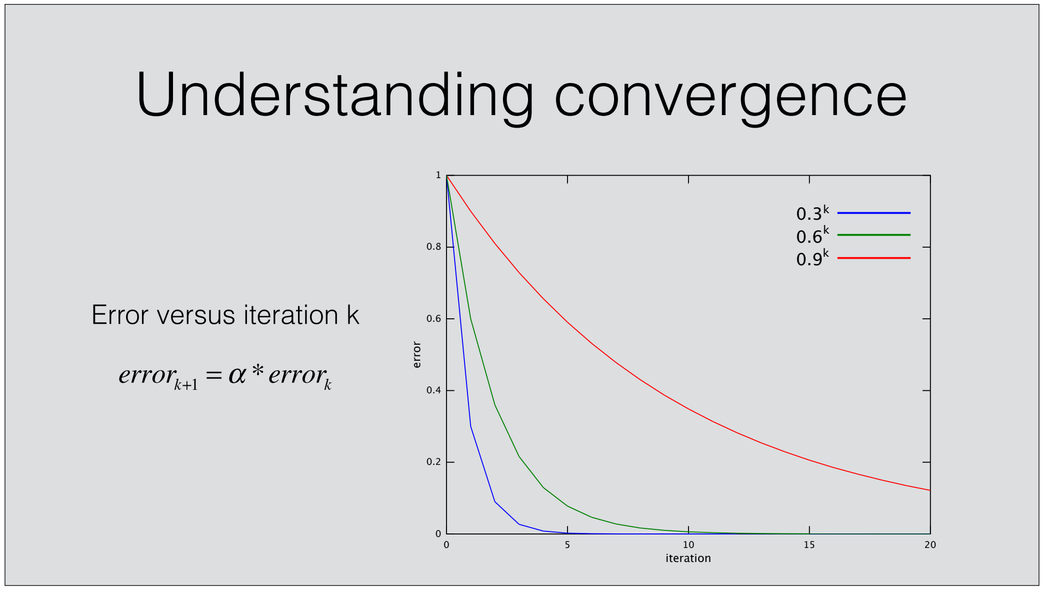
Iteration 3





Low accuracy leads to overlap

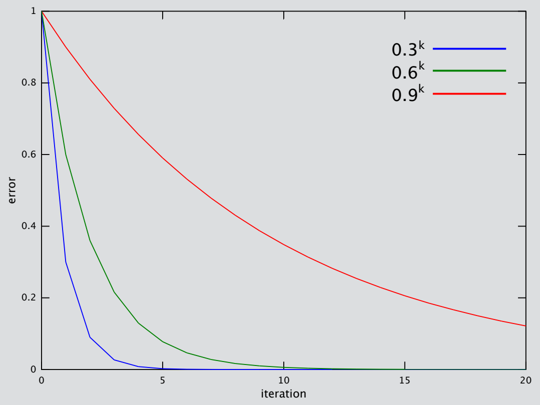
Suppose I terminate iterations before the impulses converge. This means the circles are still moving into each other and into the ground plane. This results in overlap after I update the positions.



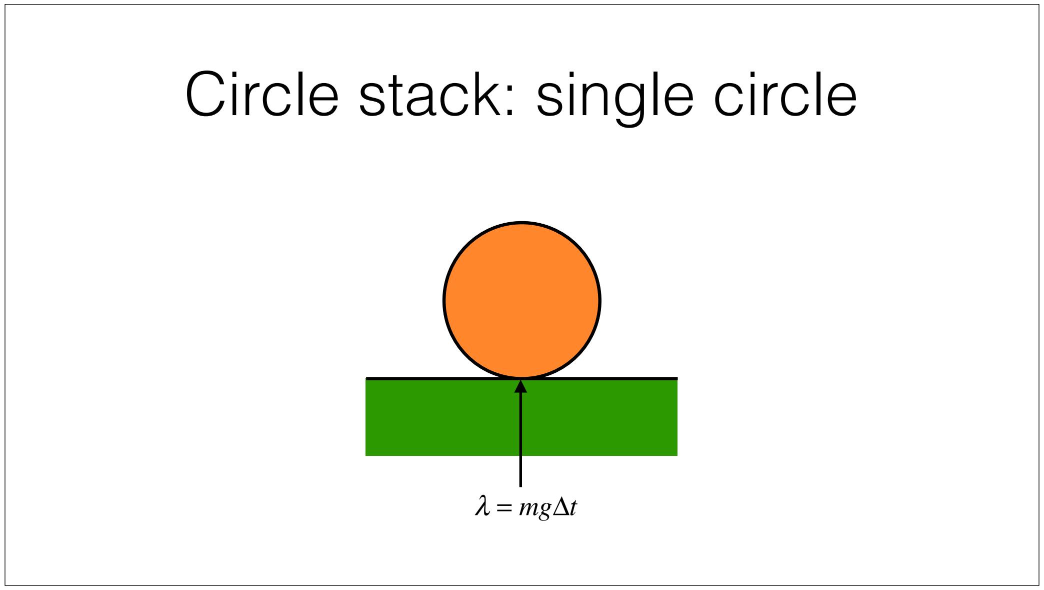
Understanding convergence

Error versus iteration k

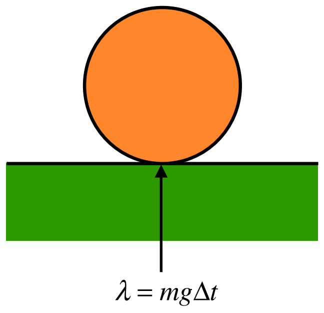
$$error_{k+1} = \alpha * error_k$$



Since we tend to use iterative solvers for game physics, we need to understand convergence. I usually associate convergence with an multiplicative decay. Each iteration the solver reduces the error by some fraction alpha. Convergence may be fast in some cases (small alpha) or slow in other cases (large alpha). As physics programmers, we need to look for the large alpha cases.



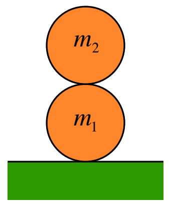
Circle stack: single circle

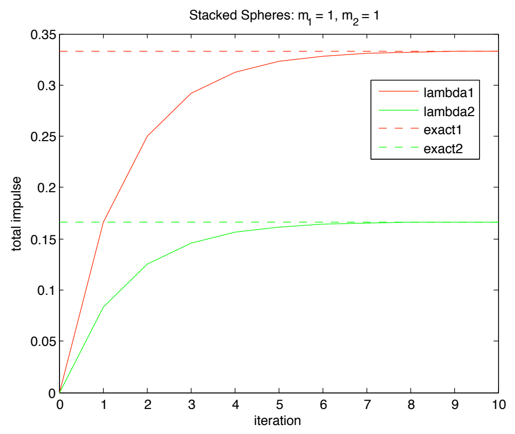


I'm going to explore convergence by stacking circles. Circle stacks are nice because they reduce the problem to one dimension.

With a single circle we only need one iteration because there are no competing constraints.

Circle stack: two circles

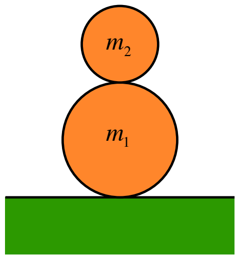


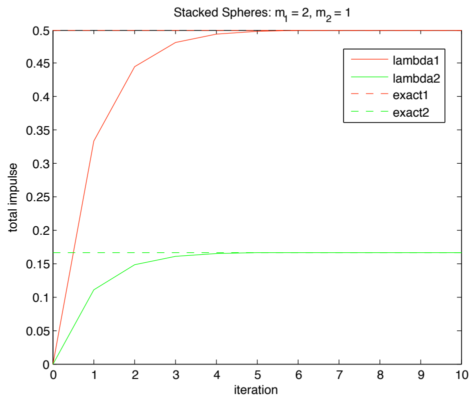


Lets look at the convergence for two circles. I created a Matlab script that runs the SI algorithm on a vertical stack of circles. You can download this script from box2d.org and you can run it in the free software Octave.

In this case the script shows that SI needs 5 iterations to get to 95% accuracy. Not bad!

Circle stack: light on heavy

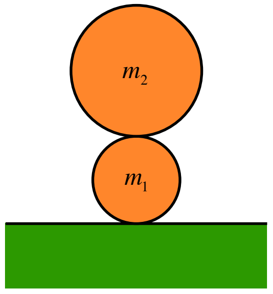


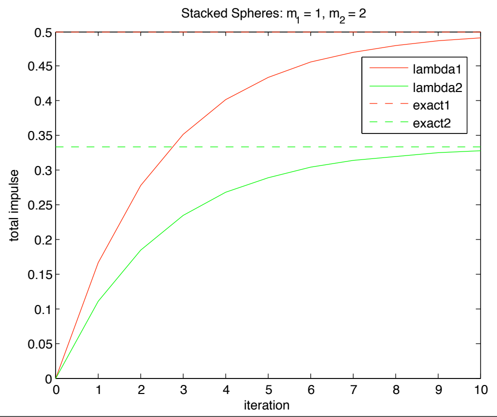


Convergence depends on the relative mass of the circles. Consider a 0.5 kg circle on top of a 1 kg circle.

Convergence got better! SI only needs 3 iterations for 95% accuracy.

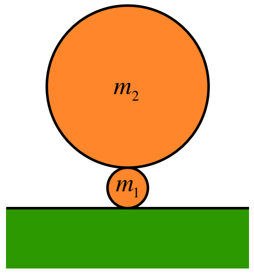
Circle stack: heavy on light

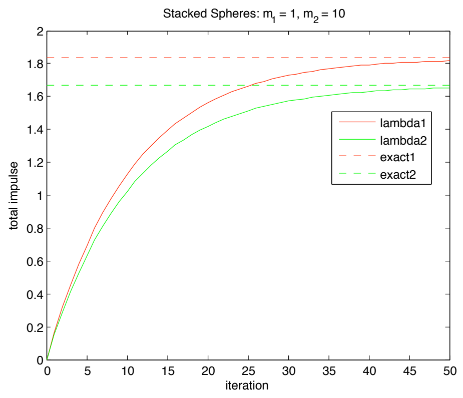




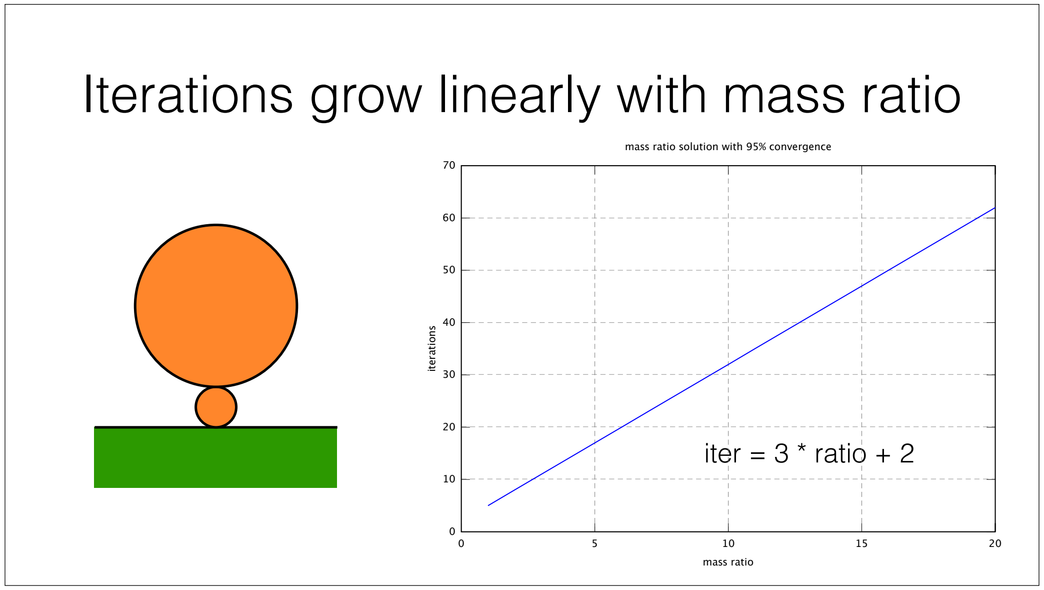
If the previous slide gave you hope, the current one will make you sad. If I stack a 1 kg circle on top of a 0.5 kg circle convergence gets much worse. Now you know why I said that SI is crappy. There are scenarios where its convergence is really poor.

Circle stack: very heavy on light

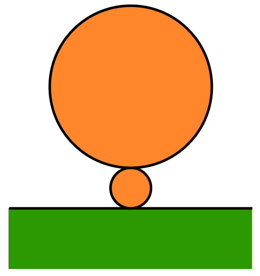


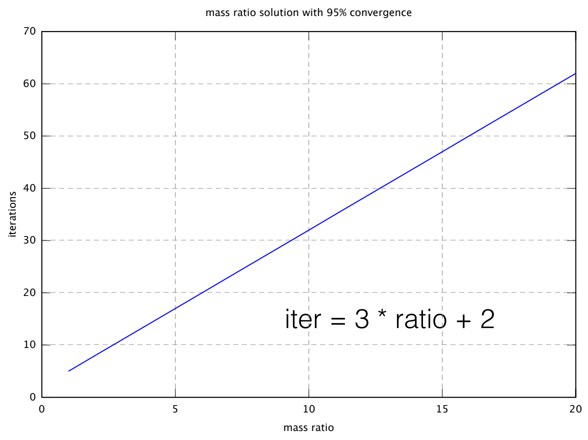


For a mass ratio of 10:1 I need around 50 iterations for 95% convergence.



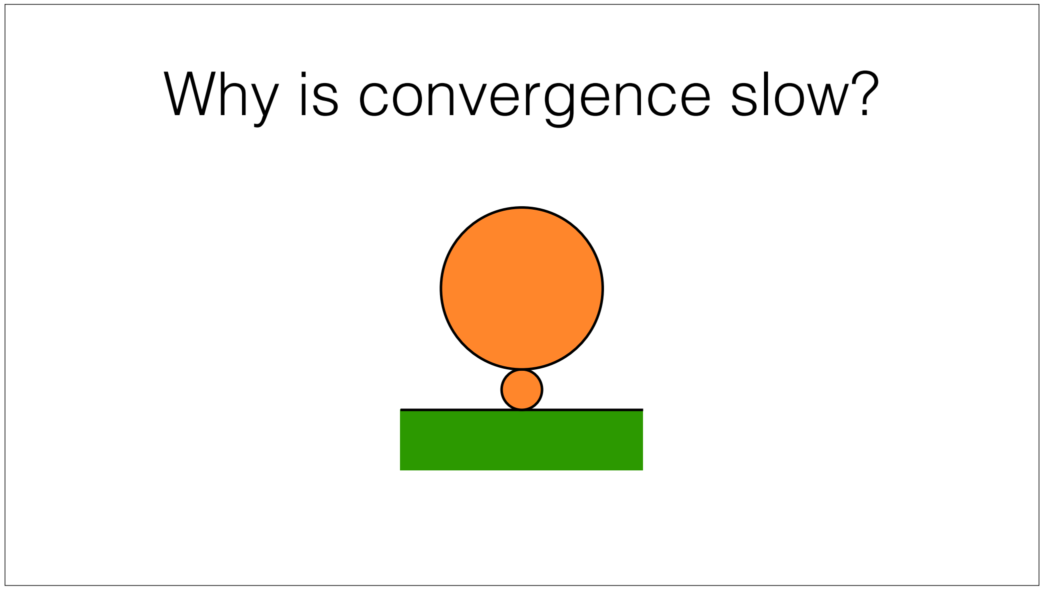
Iterations grow linearly with mass ratio



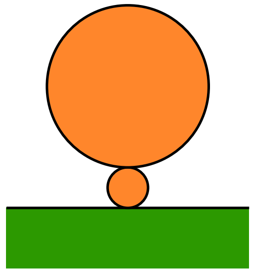


So how does convergence vary with mass ratio? I ran a bunch of simulations that computed the number of iterations needed to get 95% convergence for many mass ratios. The result is a linear graph!

I ran a linear regression on the data and found a surprisingly simple result: the number of iterations needed is 3 times the mass ratio plus 2. So the minimum iterations needed for any case is 2. If you double the mass ratio, you will needly roughly double the number of iterations.



Why is convergence slow?



Why is convergence slow in some some cases and fast in other cases?

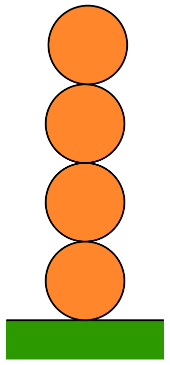
Lets look closely at the heavy on light case. The exact load at the upper contact point is the weight of the large circle. The exact load at the lower contact point is the combined weight of both circles. If we remove the large circle, the load on the lower contact point is just the weight of the small circle. This is a much lower load.

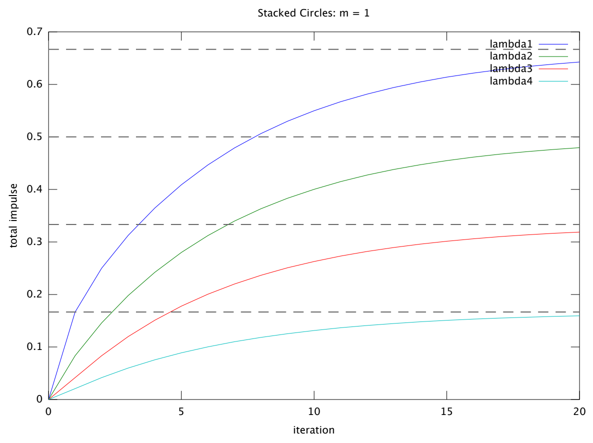
But that lower load is just what the lower contact point expects. The lower contact point doesn't know about the large circle on top.

Initially both circles are moving down under gravity at the same velocity. Assume that each iteration I solve the lower contact first then the upper contact second. When I solve the lower contact I compute a small impulse that will arrest the motion of the small circle.

Then I solve the upper contact. The large circle has the same downward velocity due to gravity that the small circle had. Also, the upper contact doesn't know the small circle is restricted by the ground plane. So when I solve the upper contact the small circle receives a downward velocity and the large circle is slowed down just a little bit.

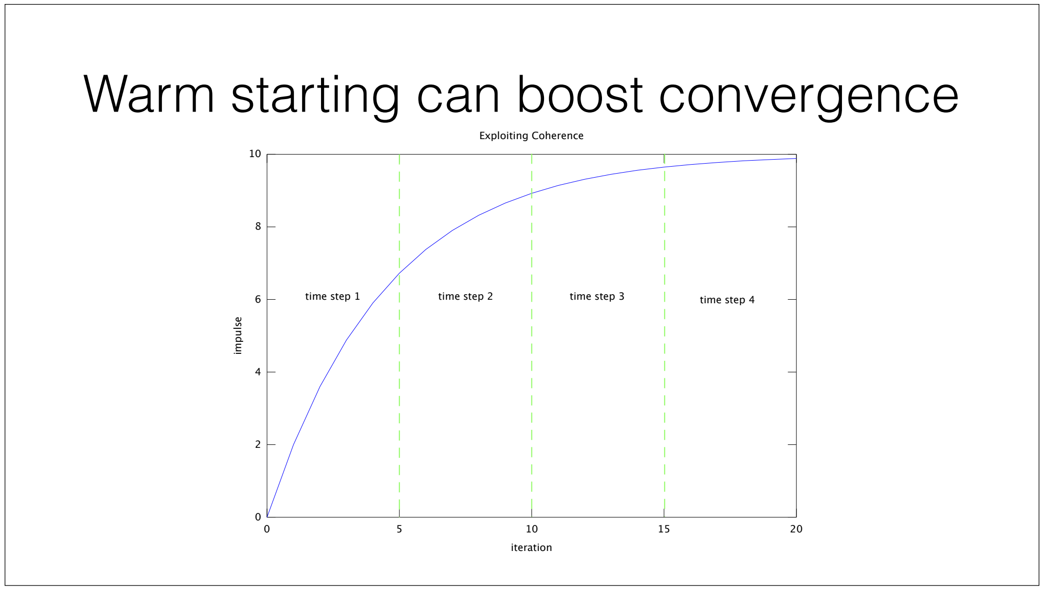
Tall stacks are challenging too



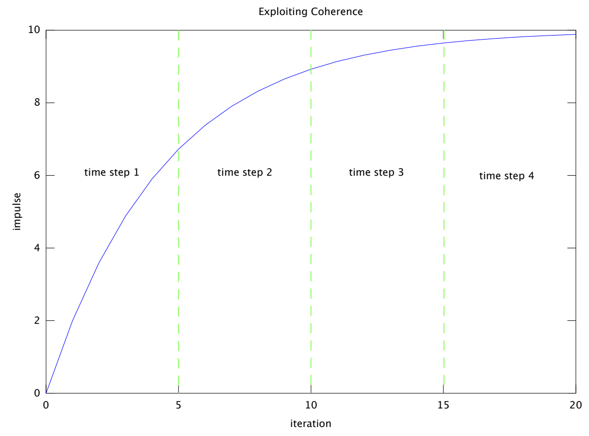


If a single circle requires just one iteration and two circles require 5 iterations, then how many iterations do we need for 4 circles? According to this graph I need 20 iterations to get to 95% accuracy.

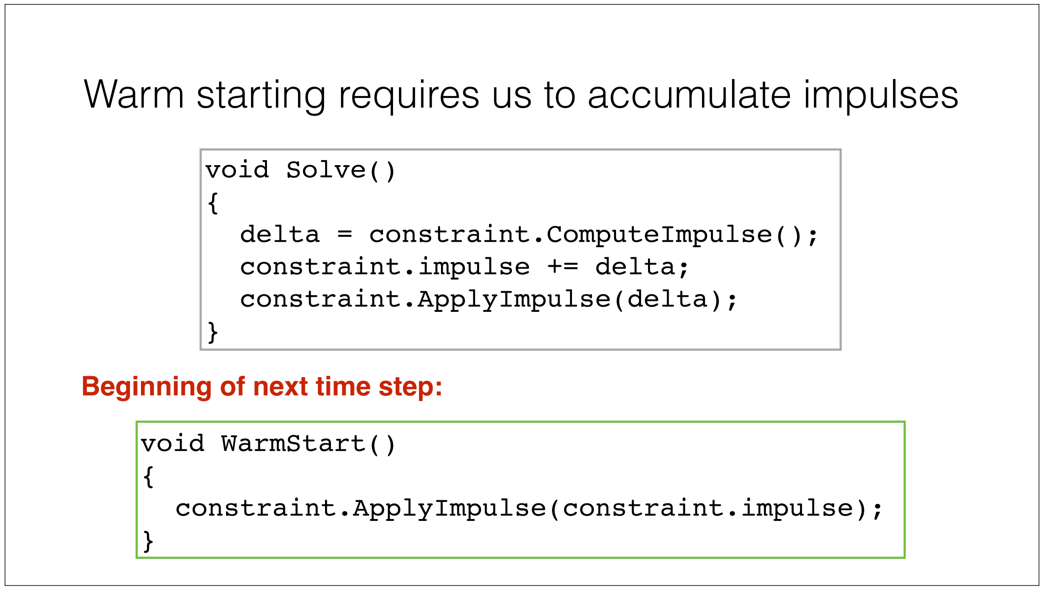
It takes more iterations for the weight of each circle to propagate down the stack.



Warm starting can boost convergence



So far things don't look good for local solvers. However, there is some hope. We are not solving one frame in isolation. We generally have some coherence from frame to frame. We can exploit this to improve the solution. The idea is that we can distribute the many iterations needed for convergence over several time steps. To do this we need to use a technique called *warm starting*.



Warm starting requires us to accumulate impulses

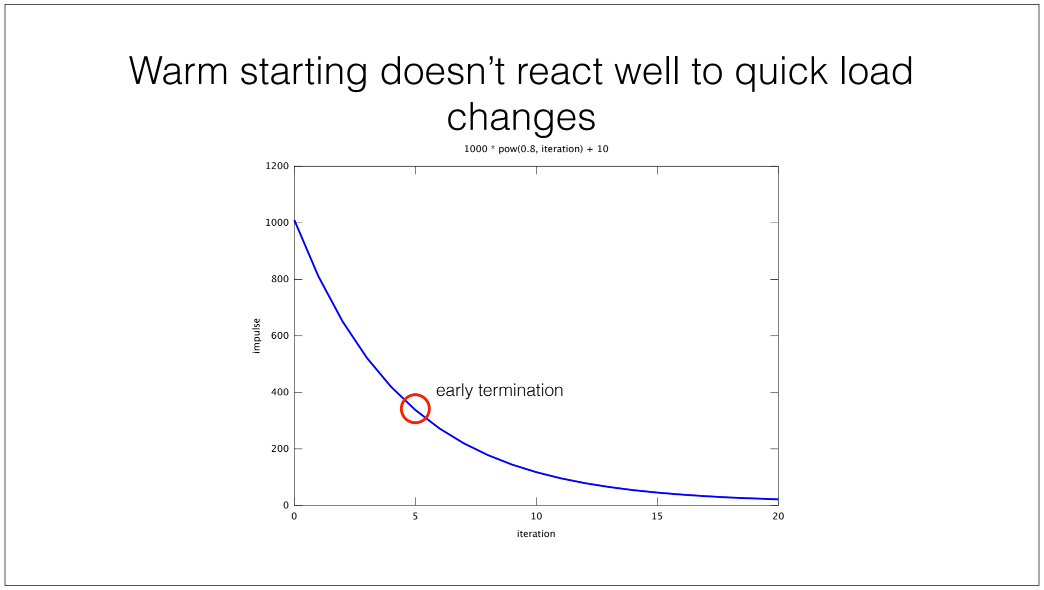
```
void Solve()
{
  delta = constraint.ComputeImpulse();
  constraint.impulse += delta;
  constraint.ApplyImpulse(delta);
}
```

Beginning of next time step:

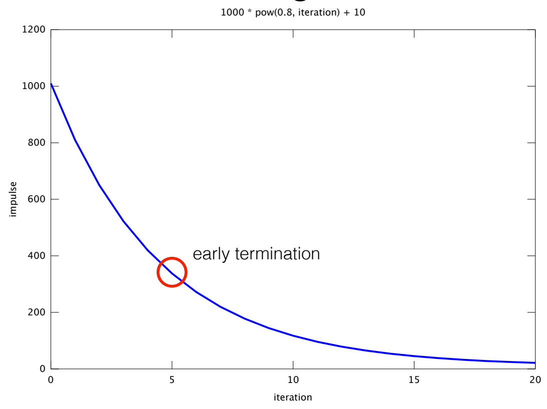
```
void WarmStart()
{
  constraint.ApplyImpulse(constraint.impulse);
}
```

As we iterate SI, we apply many small impulses that add up to the accumulated frame impulse. We can preserve this accumulated impulse and use it for warm starting.

Since we have the accumulated impulse for each constraint at the end of the time step, we can use this to initialize SI the next time step. First we apply the accumulated impulse from the previous time step, then we start applying incremental impulses to improve the solution.

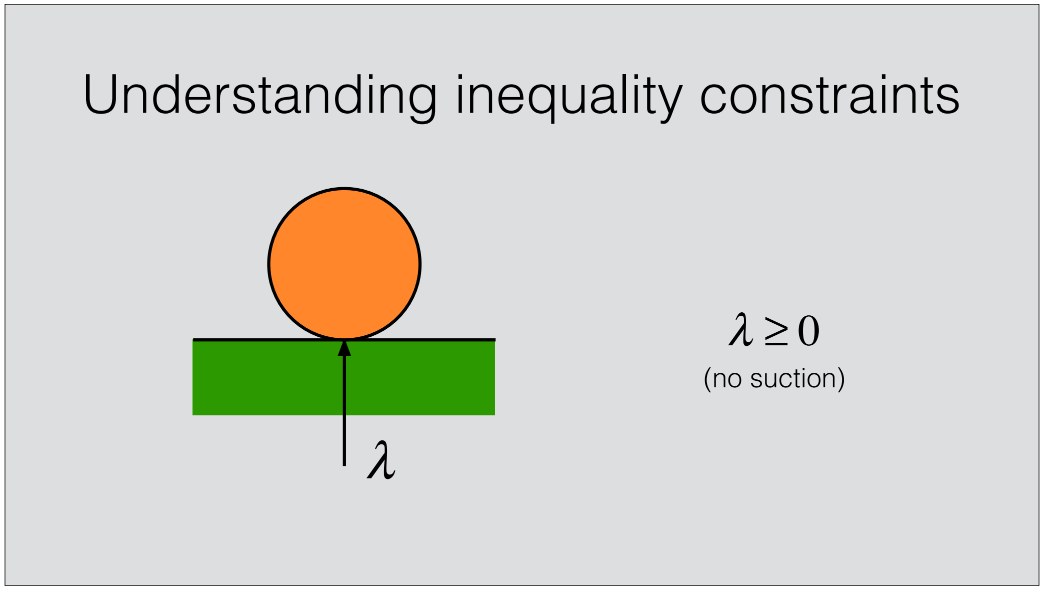


Warm starting doesn't react well to quick load changes

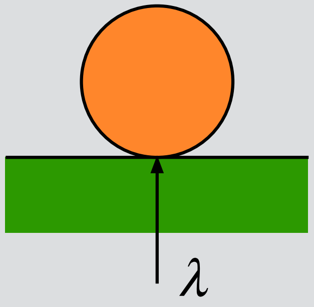


Overall, warm starting improves convergence. However, it does have a down side. If a large load is applied the accumulated impulse will build up to a large value. If the load is suddenly removed, there may not be enough iterations in a single time step to sufficiently reduce the accumulated impulse. In this case the affected rigid bodies may bounce.

(Demos of warm starting)



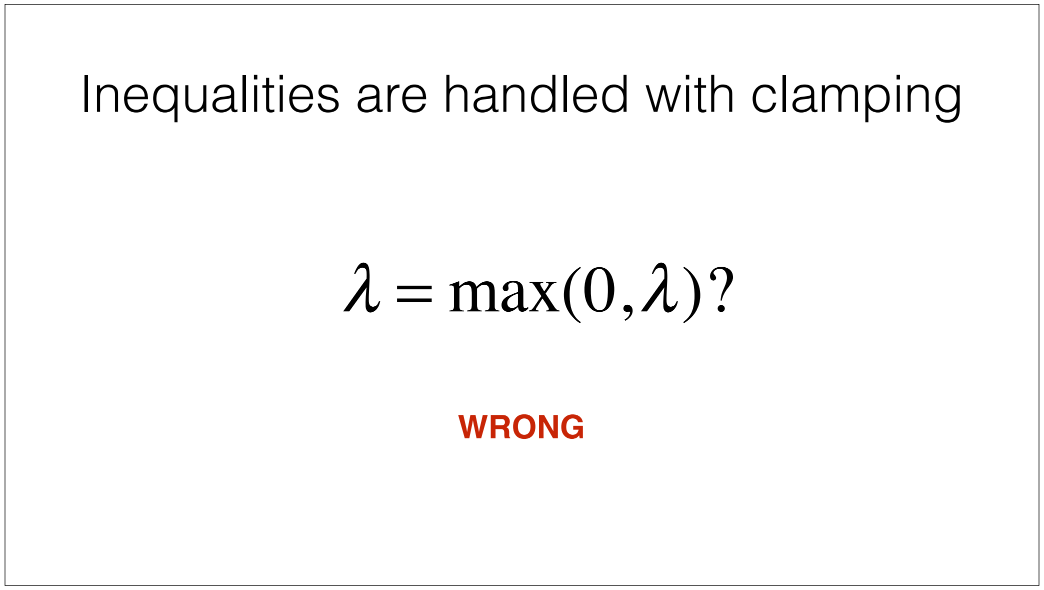
Understanding inequality constraints



$\lambda \geq 0$

(no suction)

Inequality constraints like contacts require special handling. In this case the constraint can push but can't pull. In the context of SI, it is not hard to handle inequality constraints. We simply have to clamp the impulse so that it is never negative (score +1 for SI).

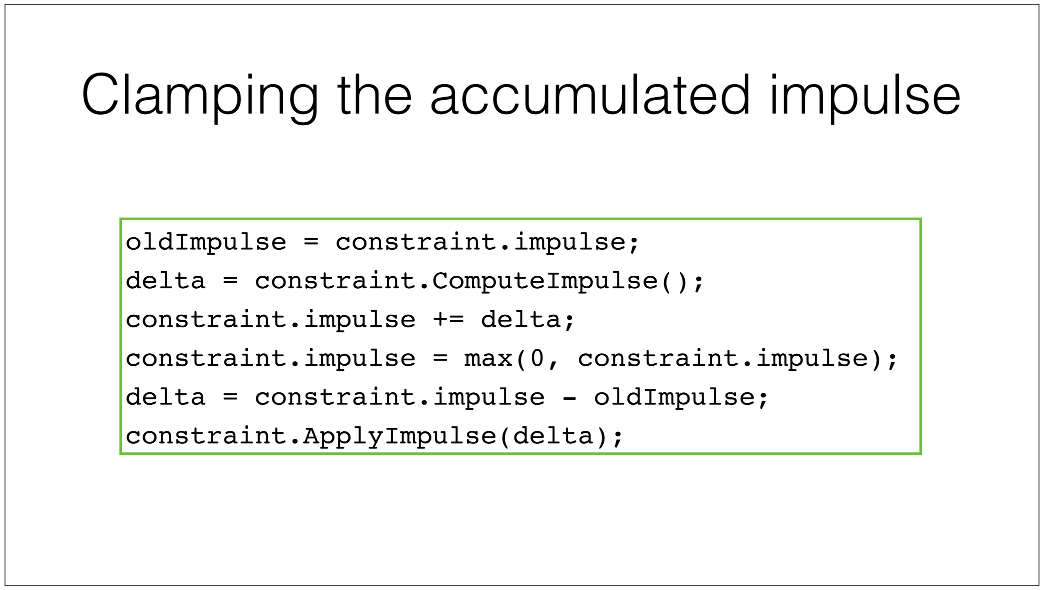


Inequalities are handled with clamping

$$\lambda = \max(0, \lambda)?$$

WRONG

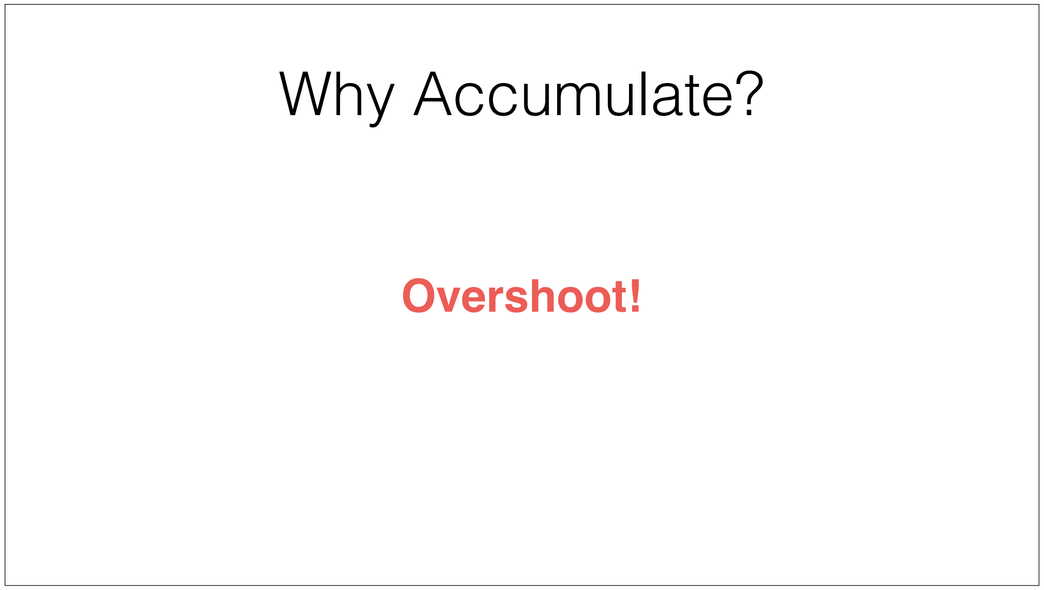
During SI iteration we compute incremental impulses at each constraint. Should we clamp the incremental impulse for inequality constraints? No! We should clamp the accumulated impulses. This allows incremental impulses to be negative.



Clamping the accumulated impulse

```
oldImpulse = constraint.impulse;
delta = constraint.ComputeImpulse();
constraint.impulse += delta;
constraint.impulse = max(0, constraint.impulse);
delta = constraint.impulse - oldImpulse;
constraint.ApplyImpulse(delta);
```

Here is how you clamp the accumulated impulse. First, make a copy of the accumulated impulse. Next, compute the incremental impulse by solving the constraint. Now add the incremental impulse to the accumulated impulse. Then clamp the accumulated impulses. Finally recompute the incremental impulse to account for the actual change in the accumulated impulse. We then apply the incremental impulse to the bodies involved in the constraint.



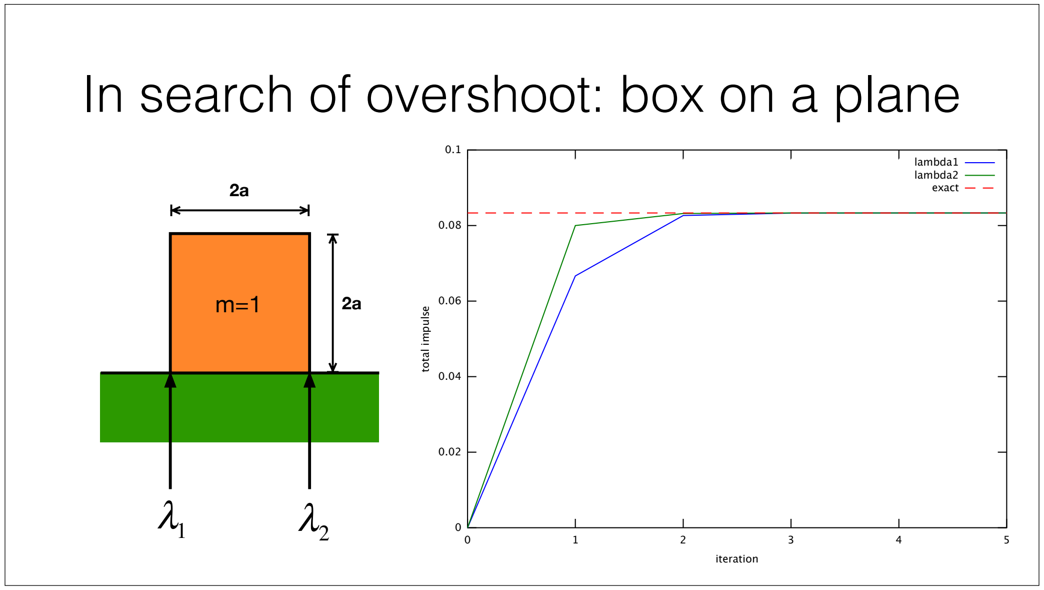
Why Accumulate?

Overshoot!

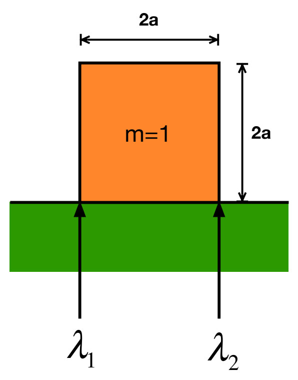
Clamping the accumulated impulse allows the incremental impulse to be negative while the accumulated impulse is positive.

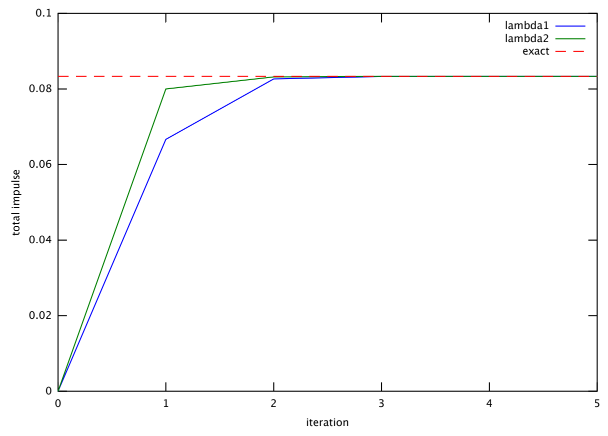
If we use warm starting, we clearly need to clamp the accumulated impulse because the contact impulse needs to decrease to zero as objects separate.

How about the case where we don't use warm starting? This boils down to the question: does the accumulated impulse ever overshoot during one time step? If it does overshoot, we need to allow for some negative incremental impulses to reduce the overshoot.



In search of overshoot: box on a plane





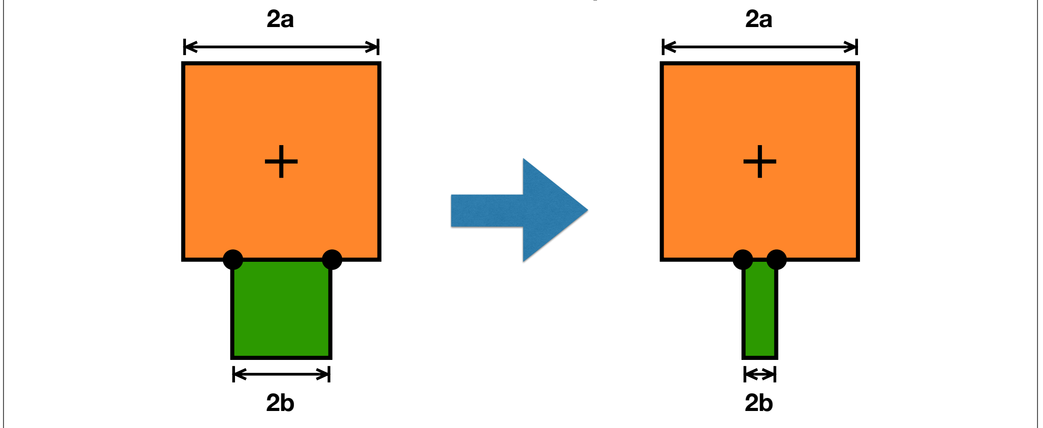
Let's look for overshoot!

Consider a box on a plane. If we graph the impulses over several iterations, we see that the accumulated impulses increase monotonically (they never decrease). So in this case we do not need to clamp the accumulated impulse.

Actually, we do not need any clamping at all if warm starting is not used (our initial guess is zero).

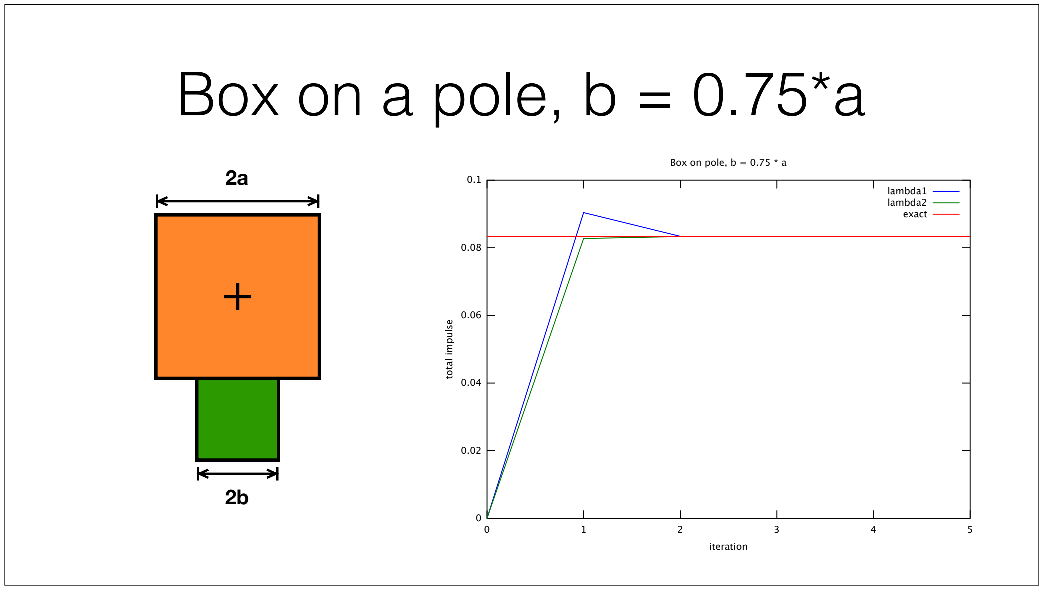
Demo: I wrote another Matlab/Octave script that I used to generate this graph.

Box on a pole

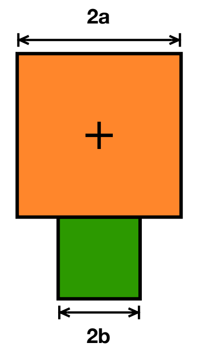


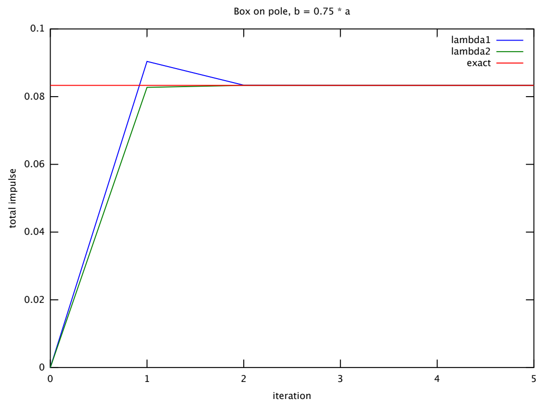
Now suppose I put the box on top of a narrow pole. As the pole gets super narrow, the scenario becomes closer to a box resting on a point. If there were a single point, all the weight of the box would be supported by that single point. However, with a narrow pole there are two contact points.

So when I solve the first contact point it tries to support almost all the weight of the box. This happens because the first point doesn't know about the second point. When I solve the second contact point it sees that the box is almost fully supported, but there is a small rotation, so only a small impulse is applied. But in the exact solution, each contact point must carry half the weight of the box. So the first accumulated impulse must decrease while the second accumulated impulse must increase. Otherwise the box will begin to rotate.

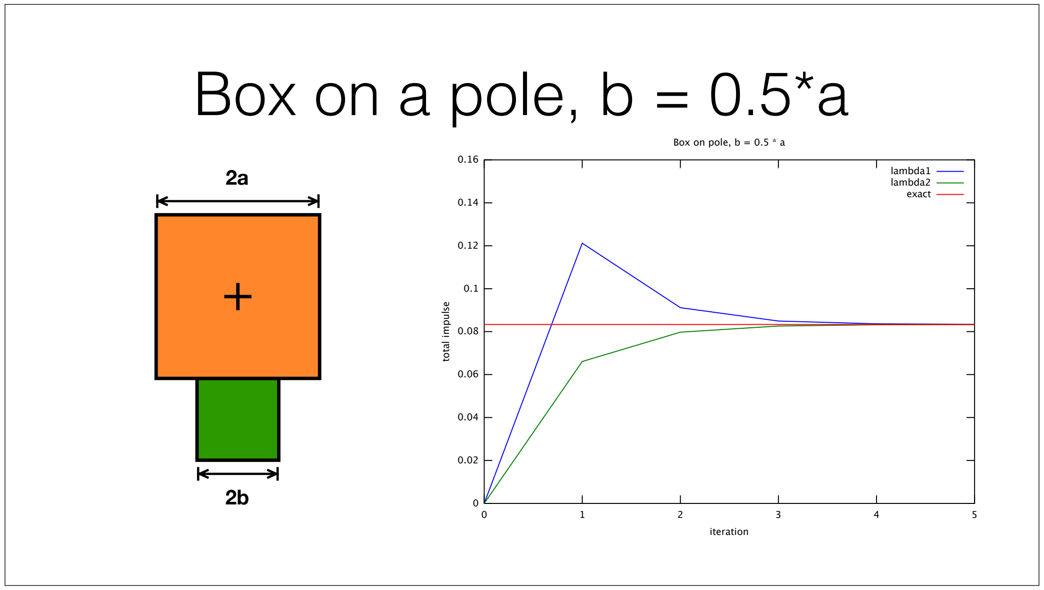


Box on a pole, b = 0.75*a

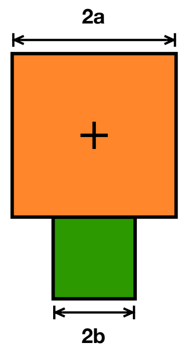


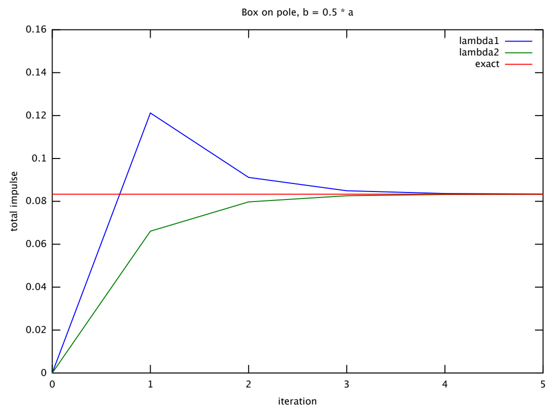


Even with b at 75% of a, there is some overshoot. However, I still get good convergence in 2 iterations.



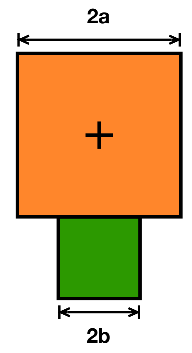
Box on a pole, b = 0.5*a

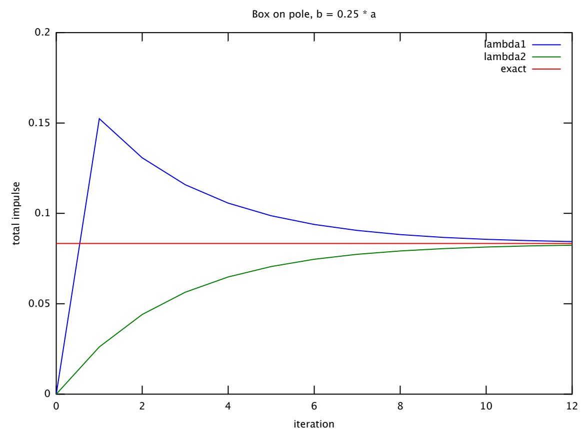




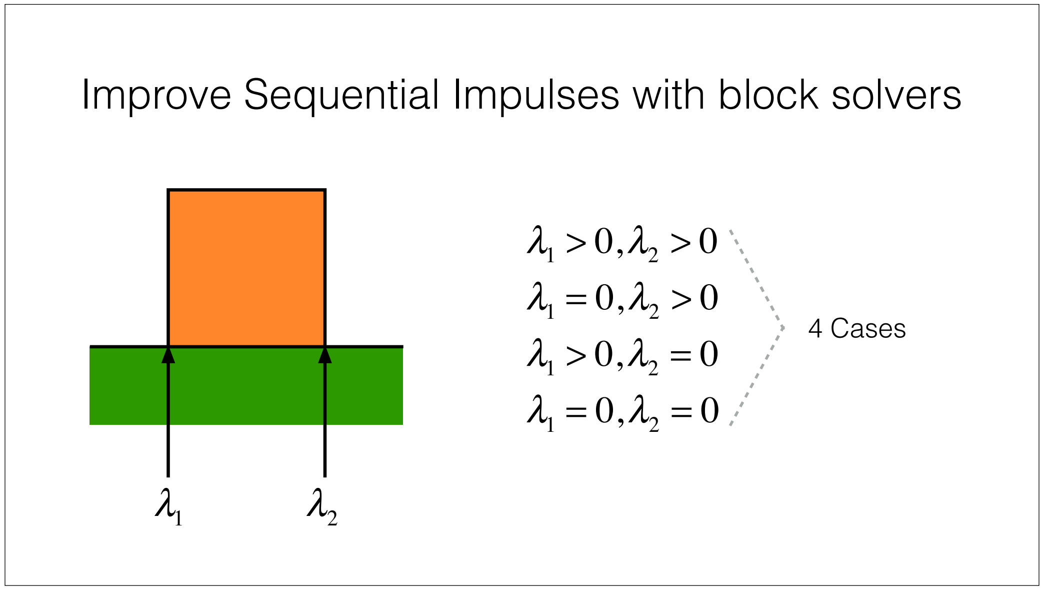
With b at 50% of a, there is more overshoot and I need a couple more iterations for convergence.

Box on a pole, b = 0.25*a

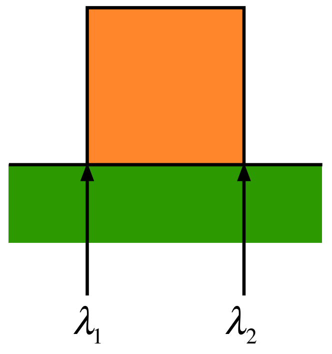




With b at 25% of a, there is significant overshoot and I need around 12 iterations for convergence. So not only is overshoot important in this case, we have also found a yet another scenario where the convergence of sequential impulses is poor!



Improve Sequential Impulses with block solvers



$$\left.\begin{array}{l}\lambda_1 > 0, \lambda_2 > 0 \\ \lambda_1 = 0, \lambda_2 > 0 \\ \lambda_1 > 0, \lambda_2 = 0 \\ \lambda_1 = 0, \lambda_2 = 0\end{array}\right\rangle \text{4 Cases}$$

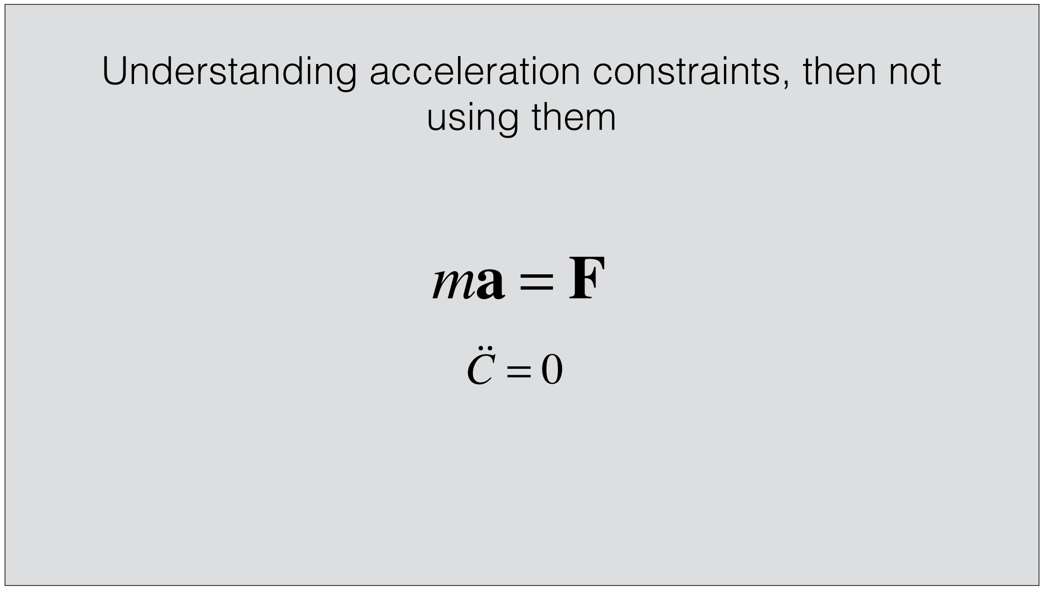
<Demo with and without block solver>

Global solvers are are too expensive, but they give great results.

We can solve a small number of constraints simultaneously to improve SI. For example, in 2D collision between convex shapes there is one or two contact points. In the case of two contact points, we can gain a lot of stability by solving both normal constraints simultaneously.

Inequality constraints can be active or inactive. An active contact point has a positive load while an inactive contact point has zero load. So with 2 contact points we have 4 cases. So first I try to solve both contact points as active. If one lambda is negative, I set lambda1 to zero and solve for lambda2. If lambda2 comes out negative, then I set lambda2 to zero and solve for lambda1. If lambda1 is negative then I set both lambdas to 0.

I don't have time to get into all the details of this, but you can find it in the Box2D code.



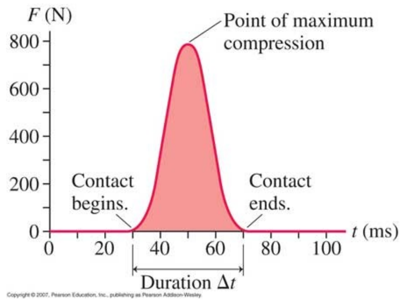
Understanding acceleration constraints, then not using them

$$m\mathbf{a} = \mathbf{F}$$

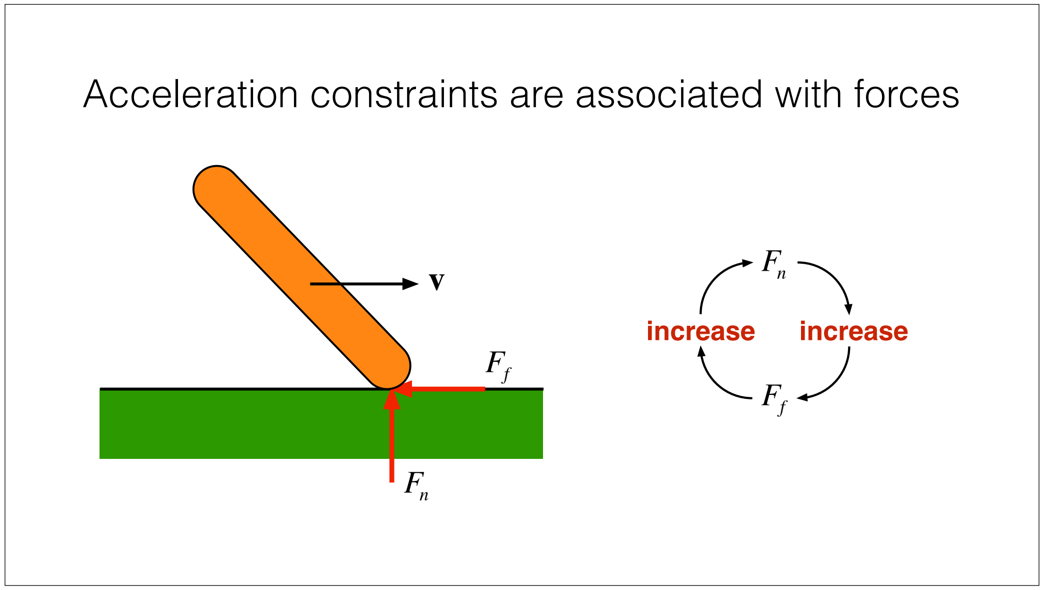
$$\ddot{C} = 0$$

Newton’s law of motion uses acceleration. Are there constraints on acceleration? Yes there are!

Acceleration constraints are associated with forces

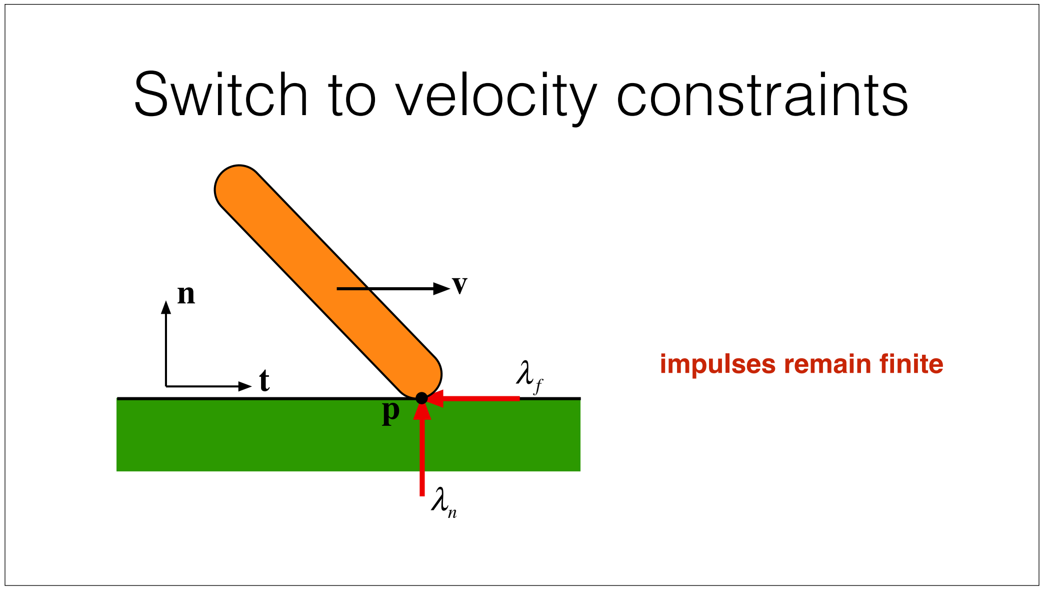


This image shows a realistic depiction of a collision force. This is drawn for the case where the collision occurs over some small period of time. In rigid body simulation, collision occur instantaneously, leading to infinite forces. So we can’t use accelerations for collisions, only for resting contact.


Acceleration constraints are associated with forces

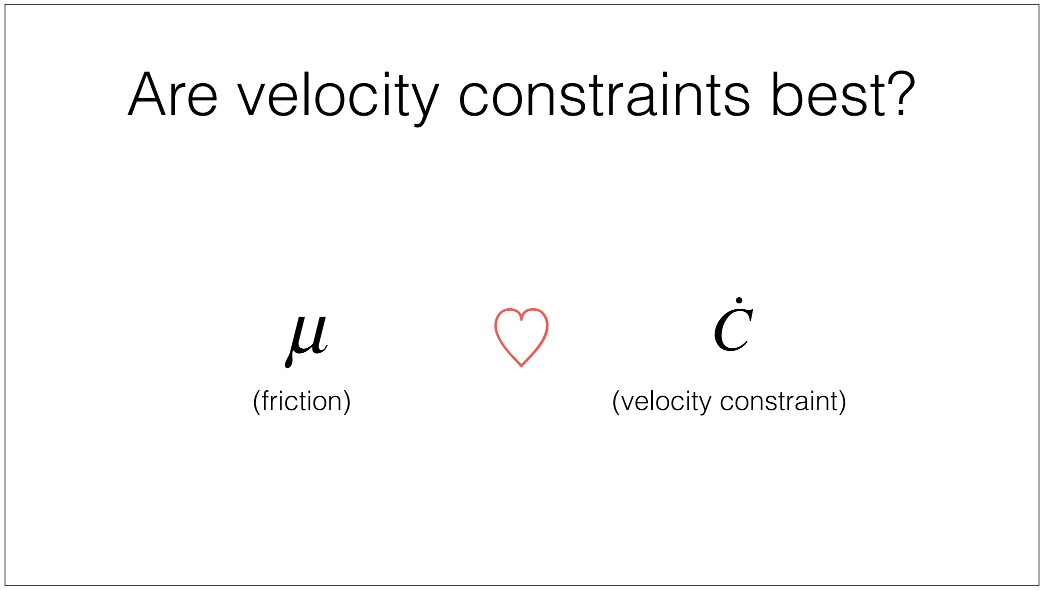
However, things get worse. Even without collision, friction forces can spike to infinity. Consider a sliding rod. As the rod slides, friction causes the rod to dig into the ground, increasing the normal force. But this causes friction to increase, leading to a positive feedback loop where friction blows up.

For this reason, game physics programmers abandoned acceleration constraints long ago.



Switch to velocity constraints

If we switch to velocity constraints, the friction impulse remains finite because velocity constraints don’t distinguish between collisions and sustained contact. The friction impulse can stop the contact point completely in one time step while remaining finite.

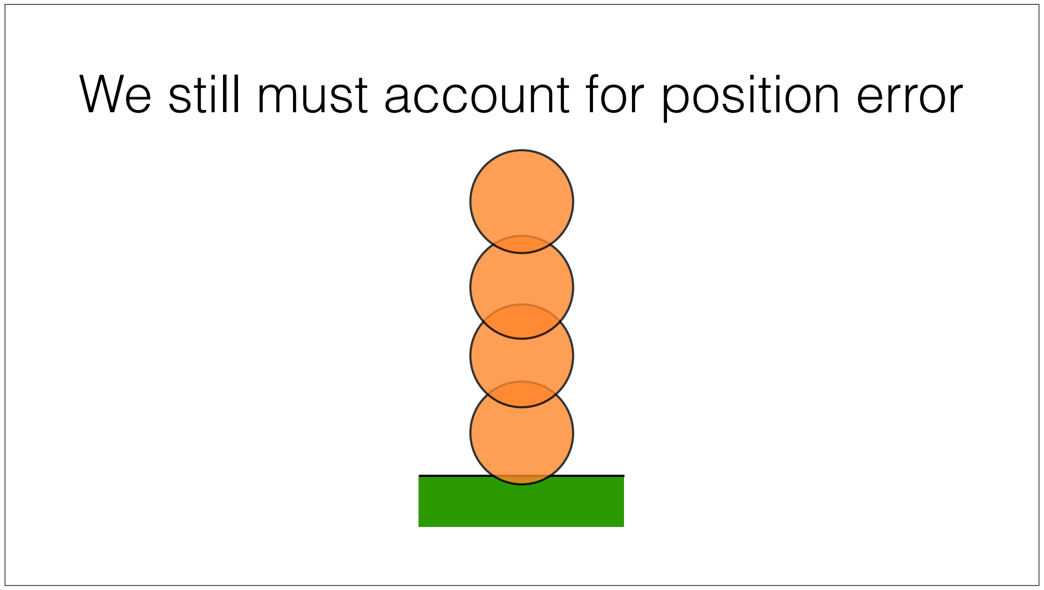


Are velocity constraints best?

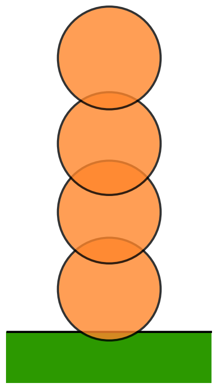
μ ♡ $\dot{C}$

(friction) (velocity constraint)

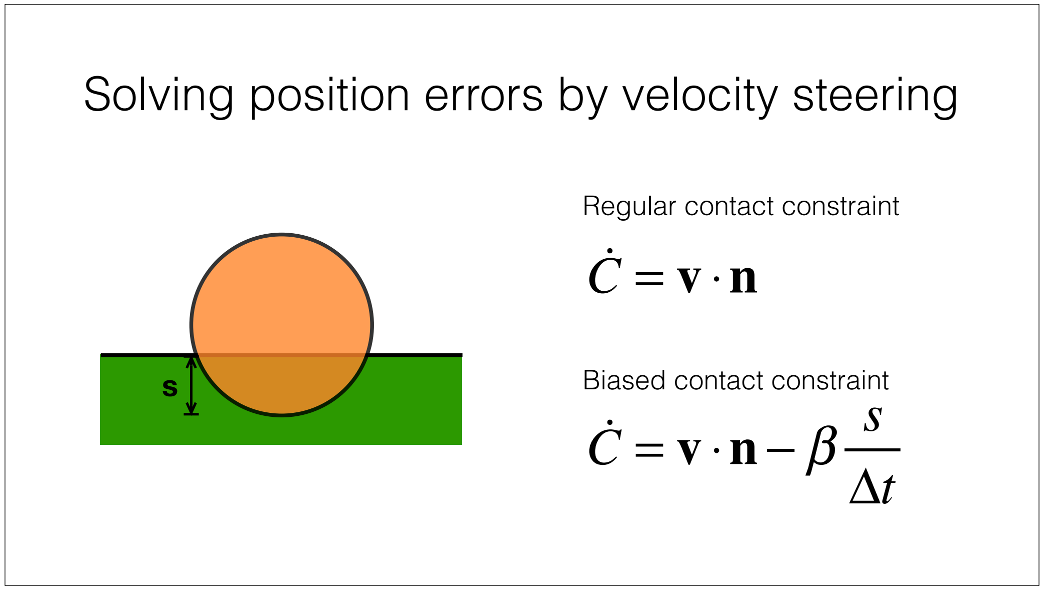
Velocity constraints seem to be the sweet spot for rigid body simulation. The impulses we compute are always finite and friction is easy to simulate because friction only constrains velocity, not position.



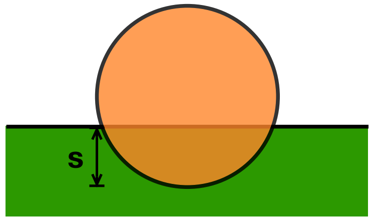
We still must account for position error



We can never have a perfect velocity solution due to the nonlinearity of rotation. Even without rotation, we may not allow enough iterations for the velocity solution to converge. So we will likely get some errors in the position constraints. The Sequential Impulse algorithm doesn't care about these position errors. So we need some additional machinery to deal with position errors.



Solving position errors by velocity steering



Regular contact constraint

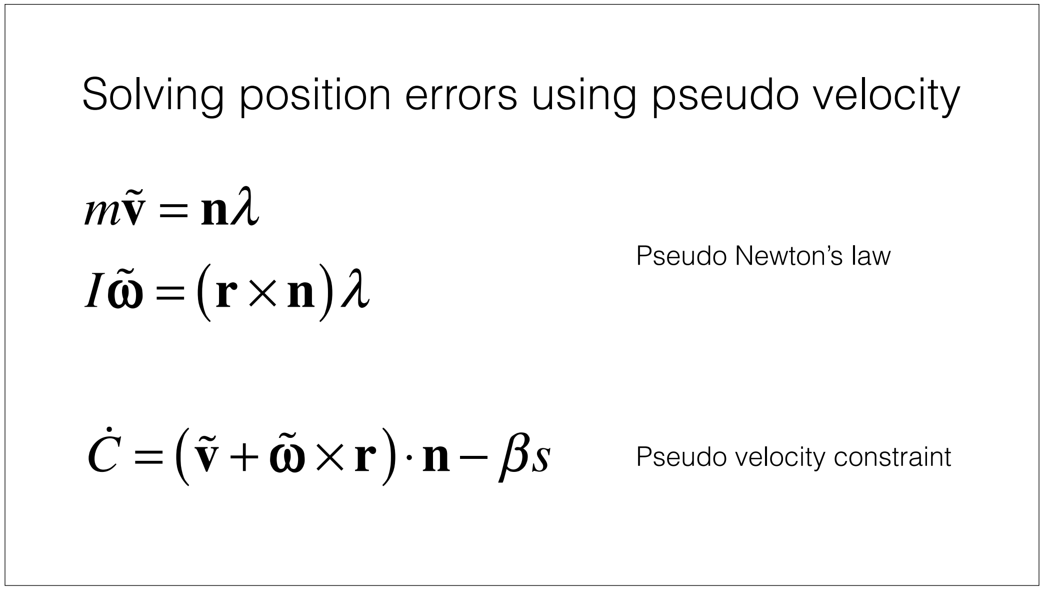
$$\dot{C} = \mathbf{v} \cdot \mathbf{n}$$

Biased contact constraint

$$\dot{C} = \mathbf{v} \cdot \mathbf{n} - \beta \frac{s}{\Delta t}$$

A typical technique for correcting position errors is to feed the position errors into the velocity constraint solver. We can add a velocity bias that steers the rigid bodies towards a configuration with less position error. The velocity bias involves a tuning factor beta, the signed distance s, and the time step.

This works well in many cases but it has some downsides. First, we must scale down the velocity bias so the bodies don't overshoot the position target. This is done with the scaling factor beta. Second, the velocity bias adds kinetic energy to the simulation and this is may cause instabilities.



Solving position errors using pseudo velocity

$$m\tilde{\mathbf{v}} = \mathbf{n}\lambda$$
$$I\tilde{\boldsymbol{\omega}} = \left(\mathbf{r}\times\mathbf{n}\right)\lambda$$

Pseudo Newton’s law

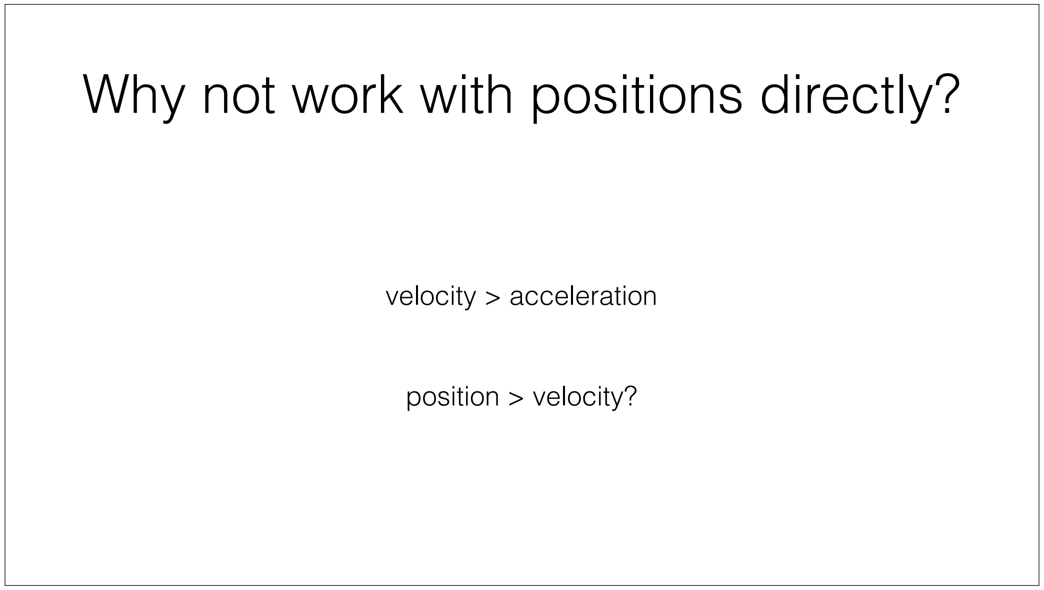
$$\dot{C} = \left(\tilde{\mathbf{v}} + \tilde{\boldsymbol{\omega}}\times\mathbf{r}\right)\cdot\mathbf{n} - \beta s$$

Pseudo velocity constraint

Steering the velocity is effective for correcting position error. We can remove the stability problems by using pseudo velocities in the position correction. These pseudo velocities are not part of the state and don’t persist across frames. For position correction I use the same constrained equations of motion to compute pseudo velocities as I do to compute real velocities except: there are no external forces (e.g. no gravity), and I don’t include velocity only constraints like friction or motors. In the process of computing the pseudo velocities, I compute pseudo impulses. I don’t need warm starting for these pseudo impulses because they are not loaded by external forces. The pseudo impulses become zero when the position error is zero.

I still use the scaling factor beta because nonlinearities can cause the solver to overshoot. I typically use a beta value of 0.2.

This method is also called NGS, non-linear Gauss Seidel. It is an iterative method for solving nonlinear position constraints. As NGS proceeds, positions are modified. Therefore, you must compute the position error each time you solve a constraint. This ensures the pseudo velocities point in the correct direction.

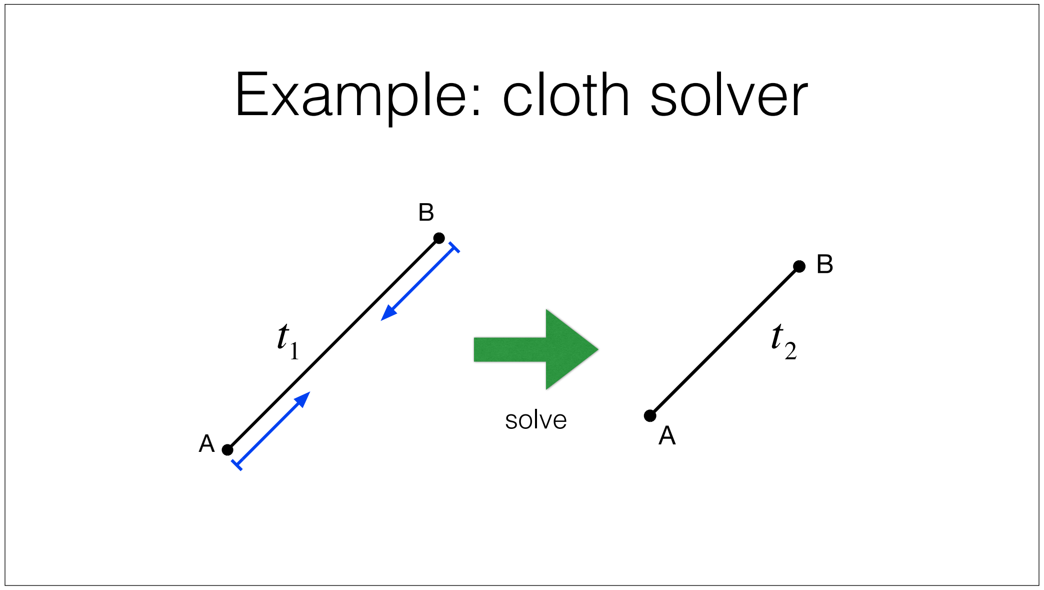


Why not work with positions directly?

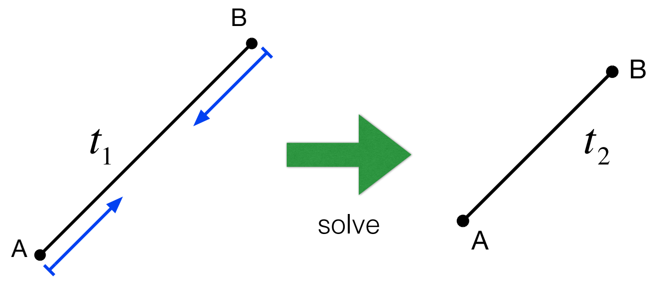
velocity > acceleration

position > velocity?

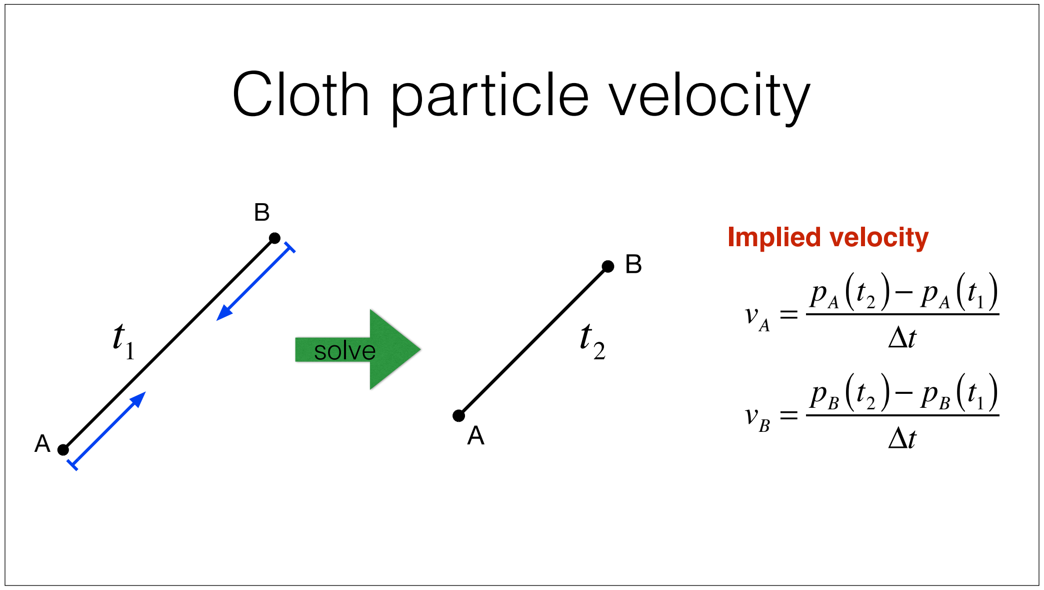
I already discussed how removing accelerations constraints leads to increased robustness (since contact forces can be infinite). In Box2D I have two solvers: one that solves velocity constraints and one that solves position constraints. So why not remove velocity constraints and just work with position constraints?



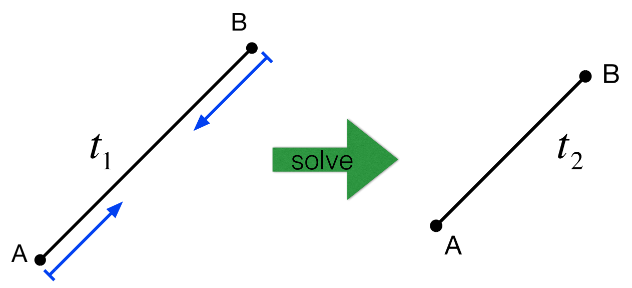
Example: cloth solver



Typical cloth solvers work with position constraints. The primary position constraints in cloth is the distance constraints. Even though distance constraints are non-linear, we can solve them exactly by moving the points together or apart along the connecting line as needed. Usually the adjustments are mass weighted so that light particles move more than heavy particles.



Cloth particle velocity



Implied velocity

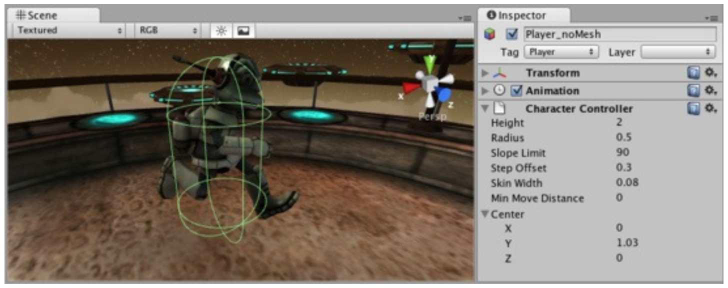
$$v_A = \frac{p_A(t_2) - p_A(t_1)}{\Delta t}$$

$$v_B = \frac{p_B(t_2) - p_B(t_1)}{\Delta t}$$

Cloth solvers treat velocity as a side effect. At the beginning of the time step I cache the particle positions. Then I apply gravity and distance constraints. When the constraint solver iterations are done, I update the particle velocity using the actual particle movement divided by the time step.

Cloth solvers are very robust, but it is not clear how to apply friction since there are no velocity constraints. Another issue with position only solvers is that you can get larger position errors. The velocity solver can prevent most interpenetration before it occurs.

Example: character solver



Player_noMesh

Tag Player Layer

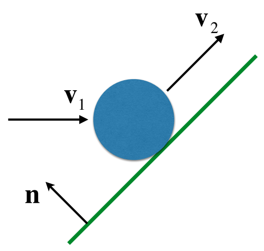
Character Controller

Height	2
Radius	0.5
Slope Limit	90
Step Offset	0.3
Skin Width	0.08
Min Move Distance	0
Center	
X	0
Y	1.03
Z	0

Character solvers are used to move player avatars around in the game world. The character solver's main job is to prevent the character from running through walls and other solid objects. Usually the character is surrounded with a simple collision shape, like a sphere or a capsule. Character solvers are agnostic about physics. Character physics is a subjective topic that are likely non-realistic for gameplay reasons. For example, characters may reverse direction instantly or have movement control even when jumping.

(image from Unity manual)

Character movement: velocity solver

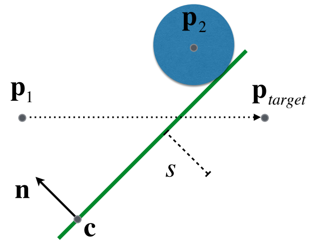


$$\boxed{\mathbf{v}_2 = \mathbf{v}_1 - (\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n}}$$

In many games the character is represented by a symmetric shape, such as a sphere or capsule. As such, we usually don’t need the character to rotate. Therefore, the character solver only needs to deal with translation. Furthermore, we can usually get away with using only plane constraints. Each time step, we move the character until it hits an obstruction and then solve for that contact plane. This process is repeated until the end of the time step.

Many games use a velocity type solver for characters. They start with some initial velocity, then clip the velocity each time a plane is encountered. This type of solver can suffer from numerical precision issues. For example, the clipped velocity may have a small error and the character continues moving slightly into a contact plane.

Character movement: position solver



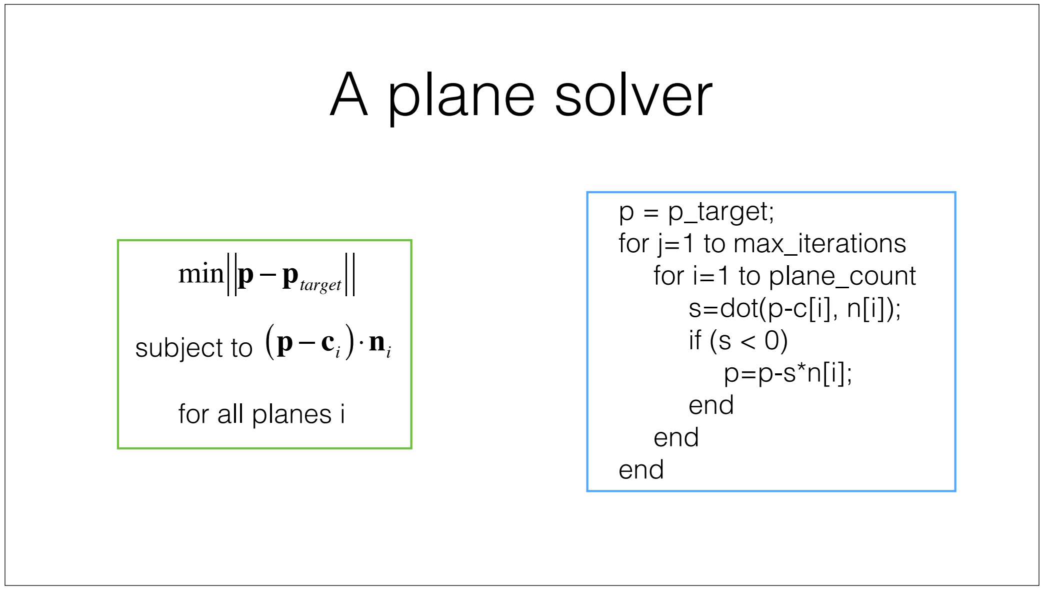
$$s = (\mathbf{p}_{target} - \mathbf{c}) \cdot \mathbf{n}$$
$$\mathbf{p}_2 = \mathbf{p}_{target} - s\mathbf{n}$$

Another approach is to use a position solver for characters. I developed this technique working on Tomb Raider. I collect collision planes as the character moves using *shape casts*.

Shape casts is like a ray cast, except instead of moving a point a long a line and looking for collisions, a shape cast moves a shape along a line looking for collisions. You can use the time of impact algorithm I discussed last year for doing generic shape casts.

Here's how the algorithm works. I first scan the environment for any overlaps. These used to create a list of collision planes. I then move the character to the target point and then apply NGS until the character is not moving much. This is solver point1. Then I perform a sweep from the start point to solve point1. Any new collision planes are added to the list. Again I move the character to the target point and then apply NGS to the new set of collision planes to get solve point2. I keep repeating this process until consecutive solve points are close together.

Hopefully you can see that this method is more geometrically robust than the velocity method.



A plane solver

$$\min \left\| \mathbf{p} - \mathbf{p}_{target} \right\|$$

subject to $(\mathbf{p} - \mathbf{c}_i) \cdot \mathbf{n}_i$

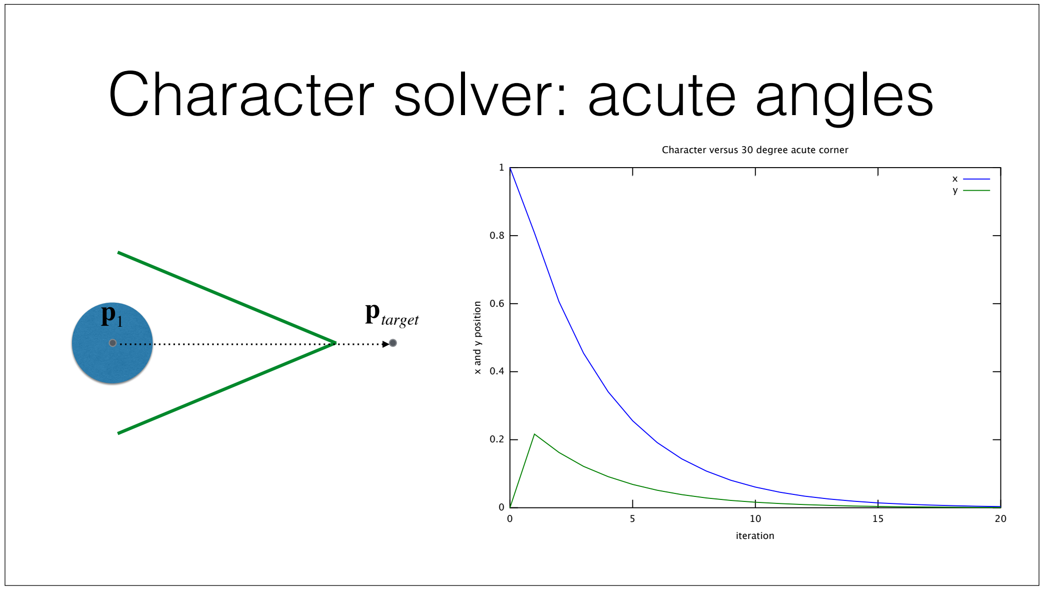
for all planes i

```
p = p_target;
for j=1 to max_iterations
    for i=1 to plane_count
        s=dot(p-c[i], n[i]);
        if (s < 0)
            p=p-s*n[i];
        end
    end
end
```

So how does the plane solver work? The input data to the character solver is the target position and the constraint planes. We do not need the current position of the character! The current position of the character is only used to gather contact planes.

I can now state the job of the character solver: find the position closest to the target position that satisfies the contact plane constraints. We can use a simple NGS solver to solve this. It is incredibly simple because the mass doesn't matter when there is only one movable object that doesn't rotate. Also, the constraints are all linear.

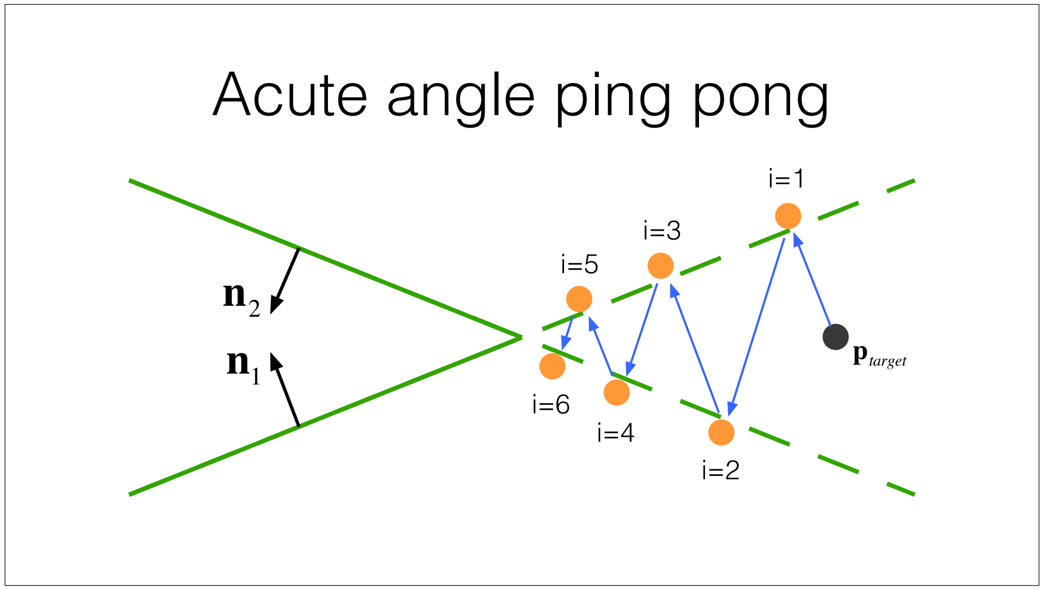
With just a couple planes we can compute an exact solution. But as we get many contact planes this becomes more difficult, so I use an iterative solve to get an approximate solution. Also, we don't need an exact solution, we just need a valid solution.



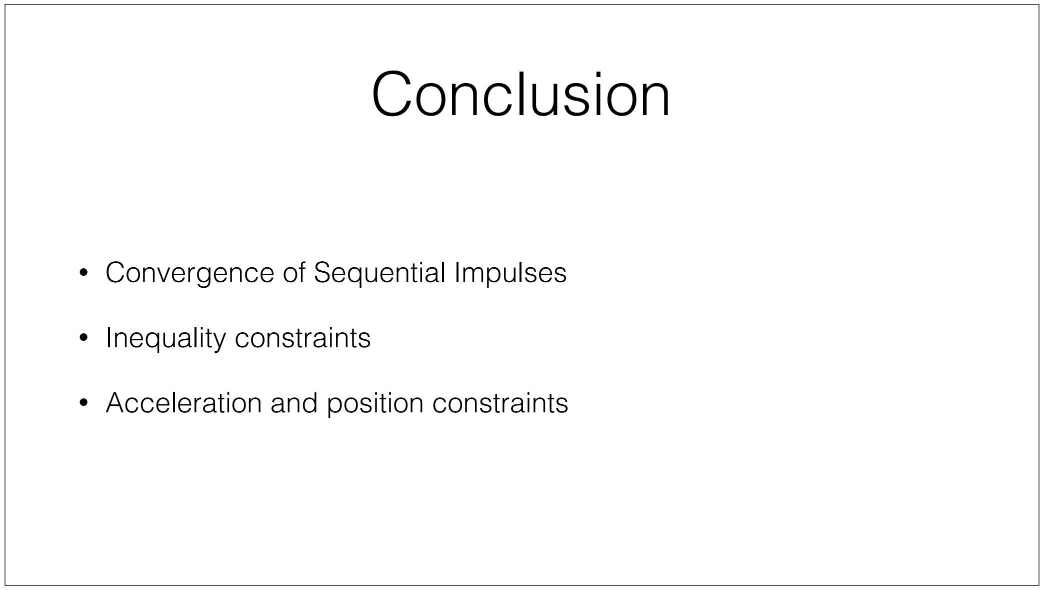
Character solver: acute angles

Acute angles can cause some convergence trouble for the character solver. As you can see it takes quite many iterations to approach the vertex of the corner. Fortunately the solver is fast and the iteration cost is usually much smaller than the collision detection.

See the Matlab script Character.m for code that generates this graph.

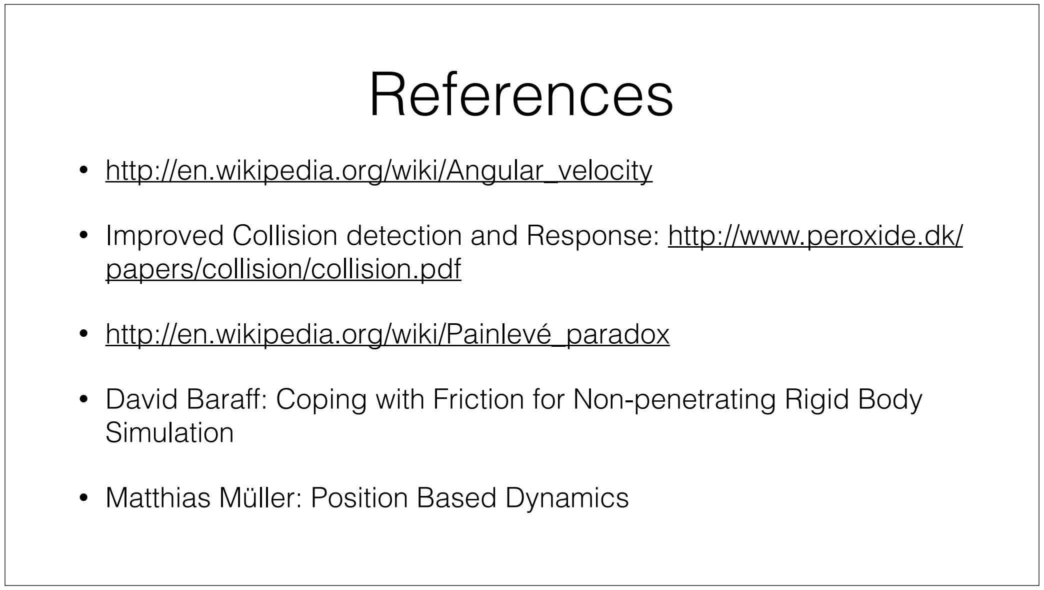

Acute angle ping pong

Here’s how the iterations look for the an acute angle. It seems like every iterative solver has a weakness. Global solvers have their weaknesses too. For example, if we have redundant collision planes the global solver will have trouble inverting the matrix. Local solvers have no problem with redundant constraints.



Conclusion

- Convergence of Sequential Impulses
- Inequality constraints
- Acceleration and position constraints



References

- http://en.wikipedia.org/wiki/Angular_velocity
- Improved Collision detection and Response: http://www.peroxide.dk/papers/collision/collision.pdf
- http://en.wikipedia.org/wiki/Painlevé_paradox
- David Baraff: Coping with Friction for Non-penetrating Rigid Body Simulation
- Matthias Müller: Position Based Dynamics

Questions?

@erin_catto
box2d.org