Modeling and Solving Constraints

Erin Catto
Blizzard Entertainment

Basic Idea

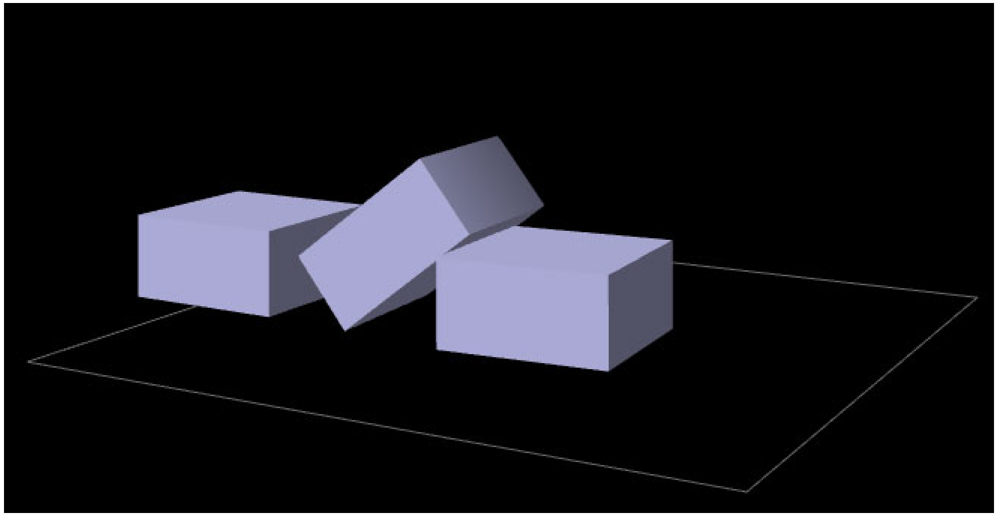
- Constraints are used to simulate joints, contact, and collision.
- We need to *solve* the constraints to stack boxes and to keep ragdoll limbs attached.
- Constraint solvers do this by calculating impulse or forces, and applying them to the constrained bodies.

Overview

- Constraint Formulas
  - Jacobians, Lagrange Multipliers
- Modeling Constraints
  - Joints, Motors, Contact
- Building a Constraint Solver
  - Sequential Impulses

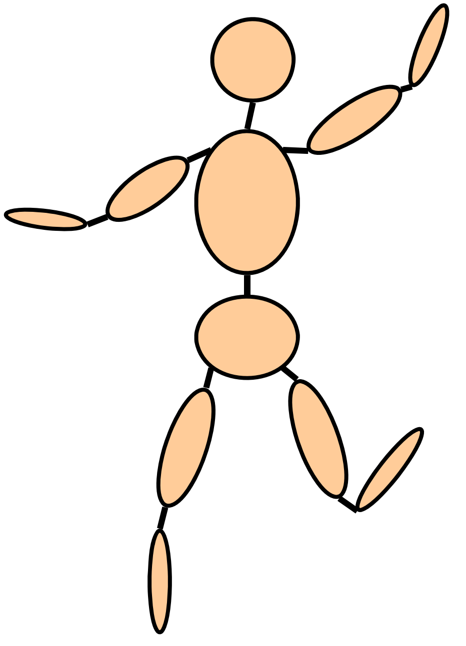
Constraint Types

Contact and Friction



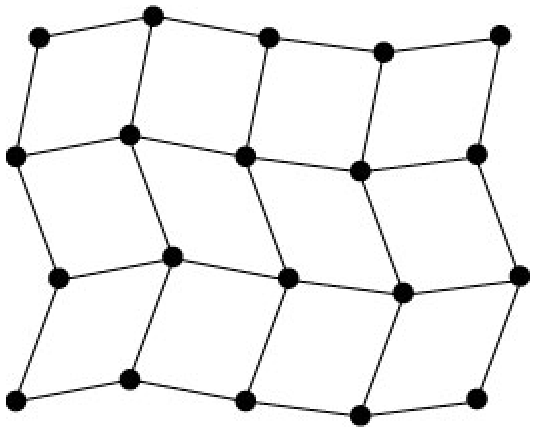
Constraint Types

Ragdolls

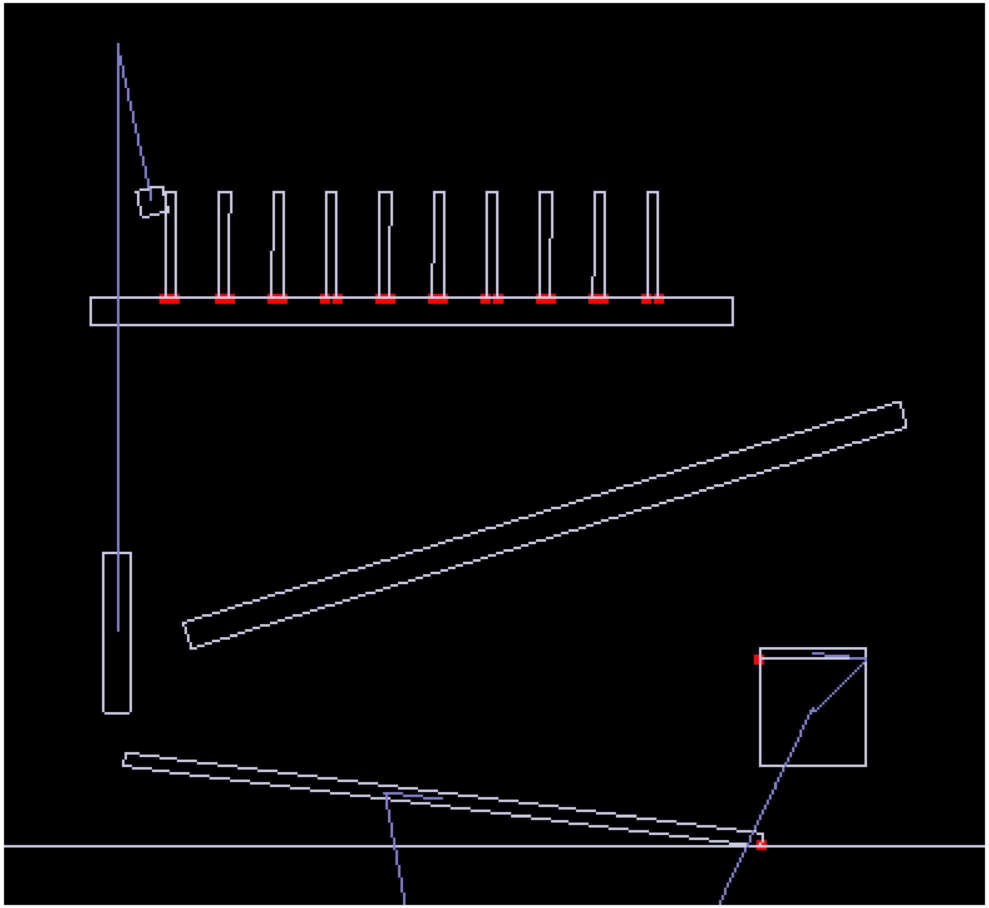


Constraint Types

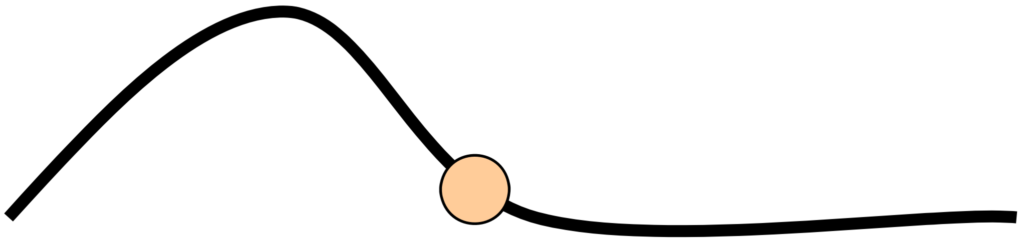
Particles and Cloth



Show Me the Demo!



Bead on a 2D Rigid Wire

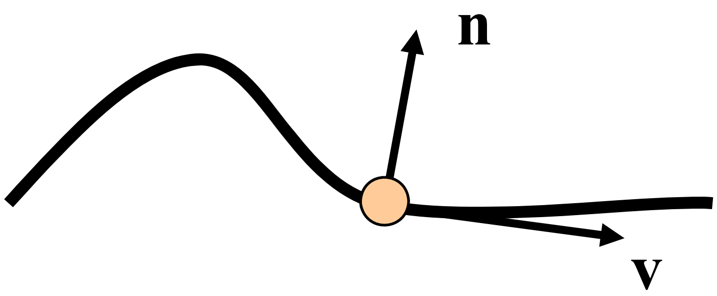


Implicit Curve Equation: $C(x, y) = 0$

This is the position constraint.

How does it move?

The normal vector is perpendicular to the velocity.



$$\text{dot}(\mathbf{n}, \mathbf{v}) = 0$$

Enter The Calculus

Position Constraint:

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

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$$

If C is zero, then its time derivative is zero.

Velocity Constraint: $\dot{C} = 0$

Velocity Constraint

$$\dot{C} = 0$$

- Velocity constraints define the allowed motion.
- Next we'll show that velocity constraints depend linearly on velocity.

The Jacobian

Due to the chain rule the velocity constraint has a special structure:

$$\dot{C} = \mathbf{J}\mathbf{v}$$

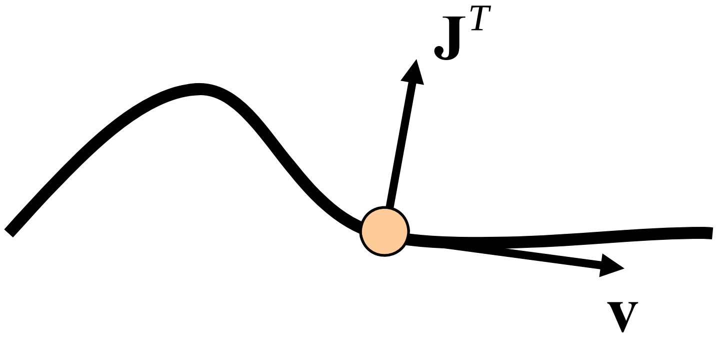
$$\mathbf{v} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix}$$

$\mathbf{J}$ is a row vector called the *Jacobian.*
$\mathbf{J}$ depends on position.

The velocity constraint is **linear**.

The Jacobian

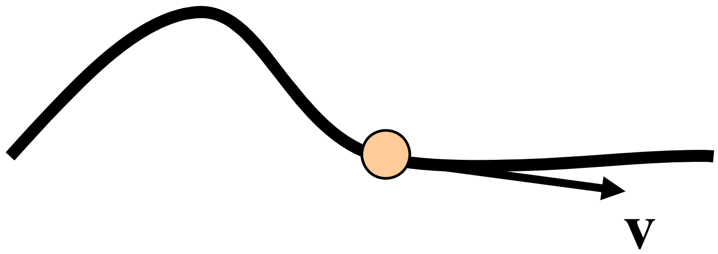
The Jacobian is perpendicular to the velocity.



$$\dot{C} = \mathbf{J}\mathbf{v} = 0$$

Constraint Force

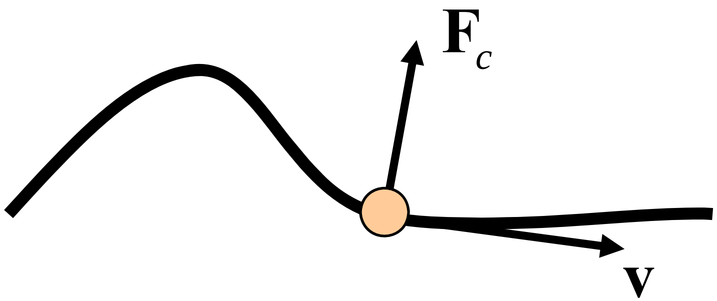
Assume the wire is frictionless.



What is the force between the wire and the bead?

Lagrange Multiplier

Intuitively the constraint force $\mathbf{F}_c$ is parallel to the normal vector.



Direction *known*.
Magnitude *unknown*.

implies

$$\mathbf{F}_c = \mathbf{J}^T \lambda$$

Lagrange Multiplier

- The Lagrange Multiplier (lambda) is the constraint force signed magnitude.
- We use a constraint solver to compute lambda.
- More on this later.

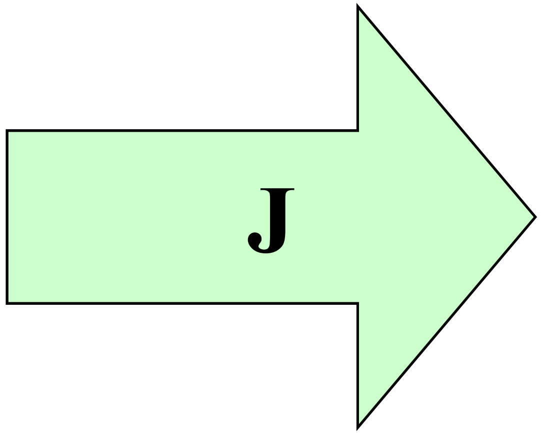
Jacobian as a CoordinateTransform

- Similar to a rotation matrix.
- Except it is missing a couple rows.
- So it projects some dimensions to zero.
- The transpose is missing some columns, so some dimensions get added.

Velocity Transform

$\mathbf{v}$

Cartesian
Space
Velocity



$\dot{C}$

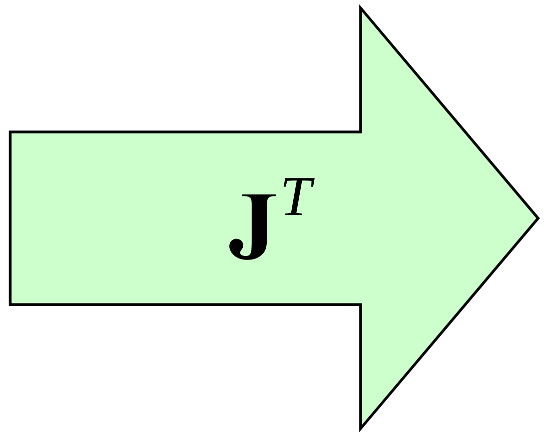
Constraint
Space
Velocity

$$\dot{C} = \mathbf{J}\mathbf{v}$$

Force Transform

λ

Constraint Space Force



$\mathbf{F}_c$

Cartesian Space Force

$$\mathbf{F}_c = \mathbf{J}^T \lambda$$

Refresher: Work and Power

Work = Force times Distance

Work has units of Energy (Joules)

Power = Force times Velocity (Watts)

$$P = \mathrm{dot}\left(\mathbf{F}, \mathbf{V}\right)$$

Principle of Virtual Work

Principle: constraint forces do **no** work.

We can ensure this by using: $$\mathbf{F}_c = \mathbf{J}^T \lambda$$

Proof (compute the power):

$$P_c = \mathbf{F}_c^T \mathbf{v} = \left(\mathbf{J}^T \lambda\right)^T \mathbf{v} = \lambda \mathbf{J} \mathbf{v} = 0$$

The power is zero, so the constraint does no work.

Constraint Quantities

Position Constraint	C
Velocity Constraint	$\dot{C}$
Jacobian	$\mathbf{J}$
Lagrange Multiplier	λ

Why all the Painful Abstraction?

- We want to put all constraints into a common form for the solver.
- This allows us to efficiently try different solution techniques.

Addendum: Modeling Time Dependence

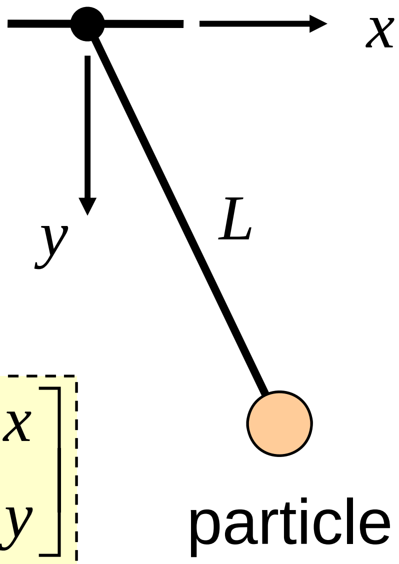
- Some constraints, like motors, have prescribed motion.
- This is represented by time dependence.

Position: $$C\left(\mathbf{x}, t\right) = 0$$

Velocity: $$\dot{C} = \mathbf{J}\mathbf{v} + b(t) = 0$$

velocity bias

Example: Distance Constraint



$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$$

λ is the tension

Position: $C = \|\mathbf{x}\| - L$

Velocity: $\dot{C} = \dfrac{\mathbf{x}^T}{\|\mathbf{x}\|}\mathbf{v}$

Jacobian: $\mathbf{J} = \dfrac{\mathbf{x}^T}{\|\mathbf{x}\|}$

Velocity Bias: $b = 0$

Gory Details

$$\frac{dC}{dt} = \frac{d}{dt}\left(\sqrt{x^2 + y^2} - L\right)$$

$$= \frac{1}{2\sqrt{x^2 + y^2}} \frac{d}{dt}\left(x^2 + y^2\right) - \frac{dL}{dt}$$

$$= \frac{2\left(xv_x + yv_y\right)}{2\sqrt{x^2 + y^2}} - 0$$

$$= \frac{1}{\sqrt{x^2 + y^2}} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} v_x \\ v_y \end{bmatrix} = \frac{\mathbf{x}^T}{\|\mathbf{x}\|} \mathbf{v}$$

Computing the Jacobian

- At first, it is not easy to compute the Jacobian.
- It gets easier with practice.
- If you can define a position constraint, you can find its Jacobian.
- Here’s how …

A Recipe for J

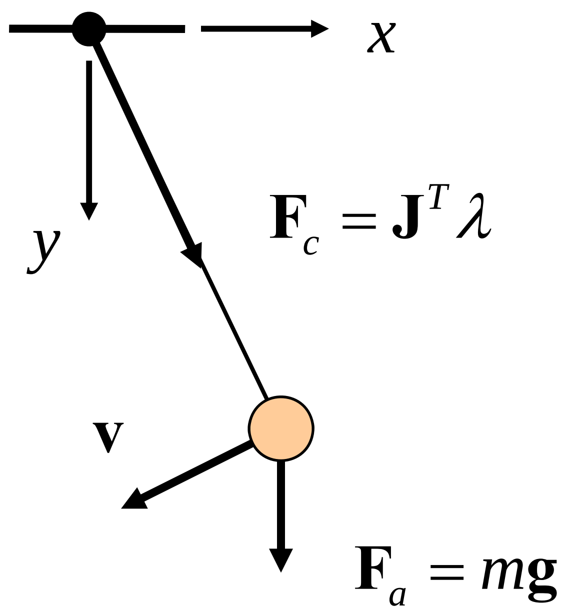
- Use geometry to write C.
- Differentiate C with respect to time.
- Isolate $\mathbf{v}$.
- *Identify* $\mathbf{J}$ and b by inspection.

$$\dot{C} = \mathbf{J}\mathbf{v} + b$$

Constraint Potpourri

- Joints
- Motors
- Contact
- Restitution
- Friction

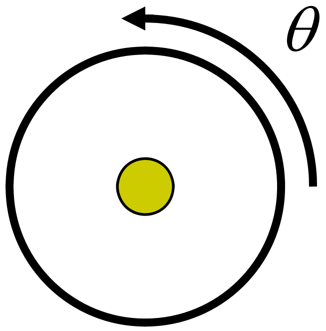
Joint: Distance Constraint



$$\mathbf{J} = \frac{\mathbf{x}^T}{\|\mathbf{x}\|}$$

Motors

A motor is a constraint with limited force (torque).



A Wheel

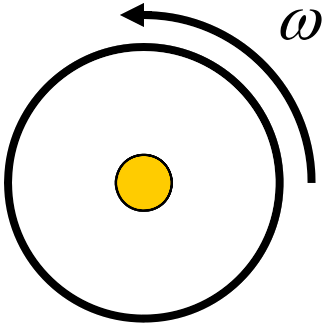
Example

$$C = \theta - \sin t$$

$$-10 \le \lambda \le 10$$

Note: this constraint does work.

Velocity Only Motors



Example

$$\dot{C} = \omega - 2$$

$$-5 \le \lambda \le 5$$

Usage: A wheel that spins at a constant rate.
We don't care about the angle.

Inequality Constraints

- So far we've looked at *equality* constraints (because they are simpler).
- Inequality constraints are needed for contact and joint limits.
- We put all inequality position constraints into this form:

$$C(\mathbf{x},t) \geq 0$$

Inequality Constraints

The corresponding velocity constraint:

If $C \leq 0$

enforce: $\dot{C} \geq 0$

Else

skip constraint

Inequality Constraints

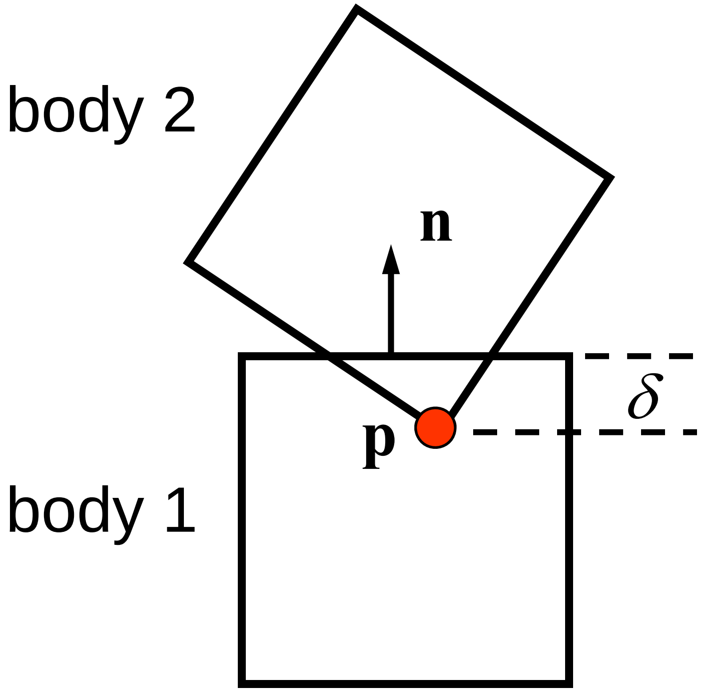
Force Limits: $$0 \leq \lambda \leq \infty$$

Inequality constraints don't *suck*.

Contact Constraint

- Non-penetration.
- Restitution: bounce
- Friction: sliding, sticking, and rolling

Non-Penetration Constraint



$$C = \delta$$

(separation)

Non-Penetration Constraint

$$\dot{C} = (\mathbf{v}_{p2} - \mathbf{v}_{p1}) \cdot \mathbf{n}$$

$$= \left[\mathbf{v}_2 + \boldsymbol{\omega}_2 \times \left(\mathbf{p} - \mathbf{x}_2 \right) - \mathbf{v}_1 - \boldsymbol{\omega}_1 \times \left(\mathbf{p} - \mathbf{x}_1 \right) \right] \cdot \mathbf{n}$$

$$= \underbrace{\begin{bmatrix} -\mathbf{n} \\ -\left(\mathbf{p} - \mathbf{x}_1 \right) \times \mathbf{n} \\ \mathbf{n} \\ \left(\mathbf{p} - \mathbf{x}_2 \right) \times \mathbf{n} \end{bmatrix}^T}_{\mathbf{J}} \begin{bmatrix} \mathbf{v}_1 \\ \boldsymbol{\omega}_1 \\ \mathbf{v}_2 \\ \boldsymbol{\omega}_2 \end{bmatrix}$$

Handy Identities

$$\mathbf{A} \cdot \left(\mathbf{B} \times \mathbf{C} \right) =$$
$$\mathbf{C} \cdot \left(\mathbf{A} \times \mathbf{B} \right) =$$
$$\mathbf{B} \cdot \left(\mathbf{C} \times \mathbf{A} \right)$$

Restitution

Relative normal velocity

$$v_n \square (\mathbf{v}_{p2} - \mathbf{v}_{p1}) \cdot \mathbf{n}$$

Velocity Reflection

$$v_n^+ \geq -e v_n^-$$

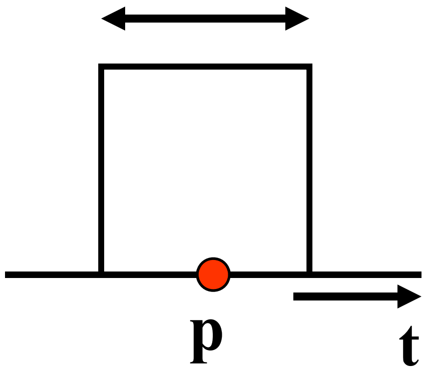
Adding bounce as a velocity bias

$$\dot{C} = v_n^+ + e v_n^- \geq 0 \quad \Longrightarrow \quad b = e v_n^-$$

Friction Constraint

Friction is like a velocity-only motor.

The target velocity is *zero.*



$$\dot{C} = \mathbf{v}_p \cdot \mathbf{t}$$

$$= \left[\mathbf{v} + \boldsymbol{\omega} \times (\mathbf{p} - \mathbf{x})\right] \cdot \mathbf{t}$$

$$= \underbrace{\begin{bmatrix} \mathbf{t} \\ (\mathbf{p} - \mathbf{x}) \times \mathbf{t} \end{bmatrix}^T}_{\mathbf{J}} \begin{bmatrix} \mathbf{v} \\ \boldsymbol{\omega} \end{bmatrix}$$

Friction Constraint

The friction force is limited by the normal force.

Coulomb's Law: $$\left|\lambda_t\right| \le \mu\lambda_n$$

In 2D: $$-\mu\lambda_n \le \lambda_t \le \mu\lambda_n$$

3D is a bit more complicated. See the references.

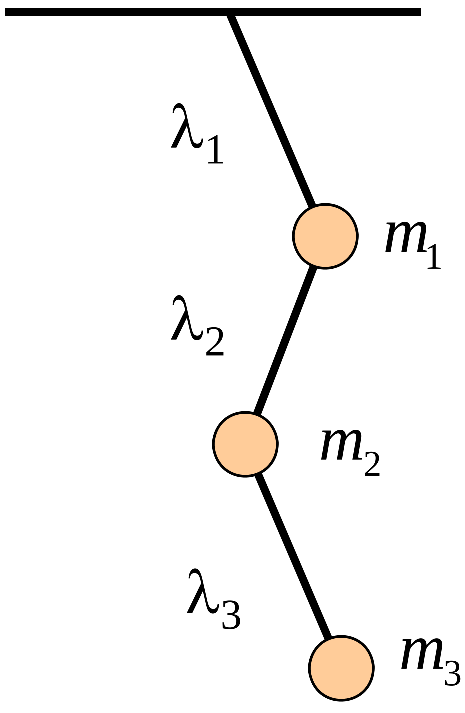
Constraints Solvers

- We have a bunch of constraints.
- We have unknown constraint forces.
- We need to solve for these constraint forces.
- There are many ways different ways to compute constraint forces.

Constraint Solver Types

- Global Solvers (slow)
- Iterative Solvers (fast)

Solving a Chain



Global:
solve for λ1, λ2, and λ3 simultaneously.

Iterative:
while !done
solve for λ1
solve for λ2
solve for λ3

Sequential Impulses (SI)

- An iterative solver.
- SI applies impulses at each constraint to correct the velocity error.
- SI is fast and stable.
- Converges to a global solution.

Why Impulses?

- Easier to deal with friction and collision.
- Lets us work with velocity rather than acceleration.
- Given the time step, impulse and force are interchangeable.

$$\mathbf{P} = h\mathbf{F}$$

Sequential Impulses

Step1:

Integrate applied forces, yielding tentative velocities.

Step2:

Apply impulses sequentially for all constraints, to correct the velocity errors.

Step3:

Use the new velocities to update the positions.

Step 1: Newton's Law

We separate *applied* forces and *constraint* forces.

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{F}_a + \mathbf{F}_c$$

mass matrix

Step 1: Mass Matrix

Particle

$$\mathbf{M} = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix}$$

Rigid Body

$$\mathbf{M} = \begin{bmatrix} m\mathbf{E} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

May involve multiple particles/bodies.

Step 1: Applied Forces

- Applied forces are computed according to some law.
- Gravity: $F = mg$
- Spring: $F = -kx$
- Air resistance: $F = -cv^2$

Step 1 :
Integrate Applied Forces

Euler's Method for all bodies.

$$\bar{\mathbf{v}}_2 = \mathbf{v}_1 + h\mathbf{M}^{-1}\mathbf{F}_a$$

This new velocity tends to violate the velocity constraints.

Step 2: Constraint Impulse

The constraint impulse is just the time step times the constraint force.

$$\mathbf{P}_c = h\mathbf{F}_c$$

Step 2: Impulse-Momentum

Newton's Law for impulses:

$$\mathbf{M}\Delta\mathbf{v} = \mathbf{P}_c$$

In other words:

$$\mathbf{v}_2 = \overline{\mathbf{v}}_2 + \mathbf{M}^{-1}\mathbf{P}_c$$

Step 2:
Computing Lambda

For each constraint, solve these for λ:

Newton's Law: $$\mathbf{v}_2 = \bar{\mathbf{v}}_2 + \mathbf{M}^{-1}\mathbf{P}_c$$

Virtual Work: $$\mathbf{P}_c = \mathbf{J}^T \lambda$$

Velocity Constraint: $$\mathbf{J}\mathbf{v}_2 + b = 0$$

Note: this usually involves one or two bodies.

Step 2: Impulse Solution

$$\lambda = -m_C \left(\mathbf{J}\overline{\mathbf{v}}_2 + b \right)$$

$$m_C = \frac{1}{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}$$

The scalar m_C is the *effective mass* seen by the constraint impulse:

$$m_C \Delta\dot{C} = \lambda$$

Step 2: Velocity Update

Now that we solved for lambda, we can use it to update the velocity.

$$\mathbf{P}_c = \mathbf{J}^T \lambda$$

$$\mathbf{v}_2 = \overline{\mathbf{v}}_2 + \mathbf{M}^{-1}\mathbf{P}_c$$

Remember: this usually involves one or two bodies.

Step 2: Iteration

- Loop over all constraints until you are *done*:
  - - Fixed number of iterations.
  - - Corrective impulses become small.
  - - Velocity errors become small.

Step 3: Integrate Positions

Use the **new** velocity to integrate all body positions (and orientations):

$$\mathbf{x}_2 = \mathbf{x}_1 + h\mathbf{v}_2$$

This is the symplectic Euler integrator.

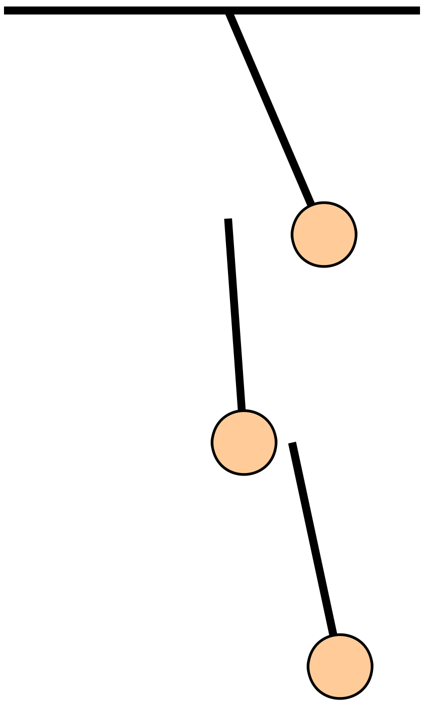
Extensions to Step 2

- Handle position drift.
- Handle force limits.
- Handle inequality constraints.
- Warm starting.

Handling Position Drift

Velocity constraints are not obeyed precisely.

Joints will fall apart.



Baumgarte Stabilization

Feed the position error back into the velocity constraint.

New velocity constraint: $$\dot{C}_B = \mathbf{Jv} + \frac{\beta}{h} C = 0$$

Bias factor: $$0 \le \beta \le 1$$

Baumgarte Stabilization

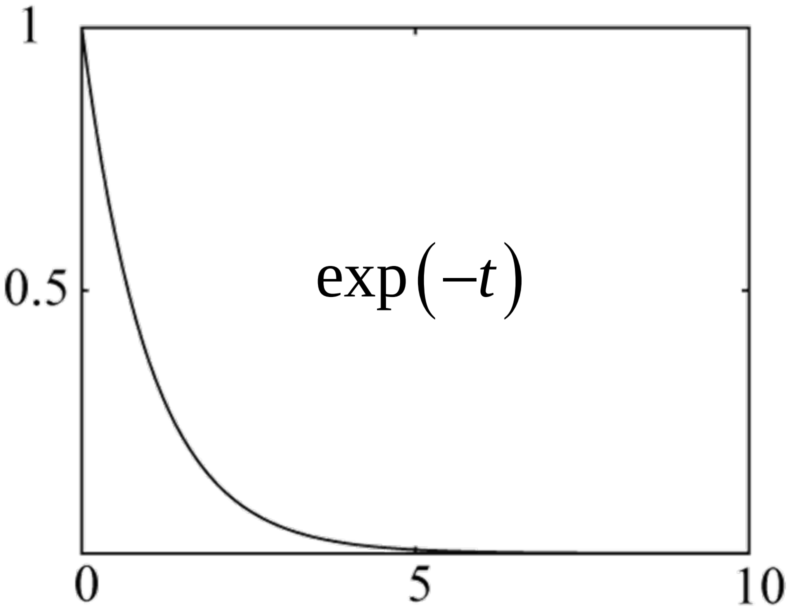
What is the solution to this?

$$\dot{C} + \frac{\beta}{h} C = 0$$

First-order differential equation …

Answer

$$C = C_0 \exp\left(-\frac{\beta t}{h}\right)$$



Tuning the Bias Factor

- If your simulation has instabilities, set the bias factor to zero and check the stability.
- Increase the bias factor slowly until the simulation becomes unstable.
- Use half of that value.

Handling Force Limits

First, convert force limits to impulse limits.

$$\lambda_{impulse} = h\lambda_{force}$$

Handling Impulse Limits

Clamping corrective impulses:

$$\lambda = \text{clamp}\left(\lambda, \lambda_{\text{min}}, \lambda_{\text{max}}\right)$$

Is it really that simple?

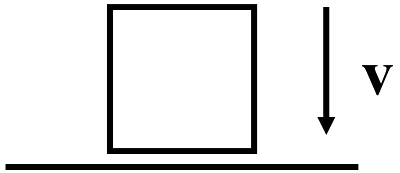
Hint: no.

How to Clamp

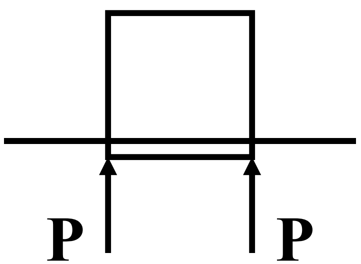
- Each iteration computes *corrective impulses*.
- Clamping corrective impulses is *wrong*!
- You should clamp the **total impulse** applied over the time step.
- The following example shows why.

Example: 2D Inelastic Collision

A Falling Box

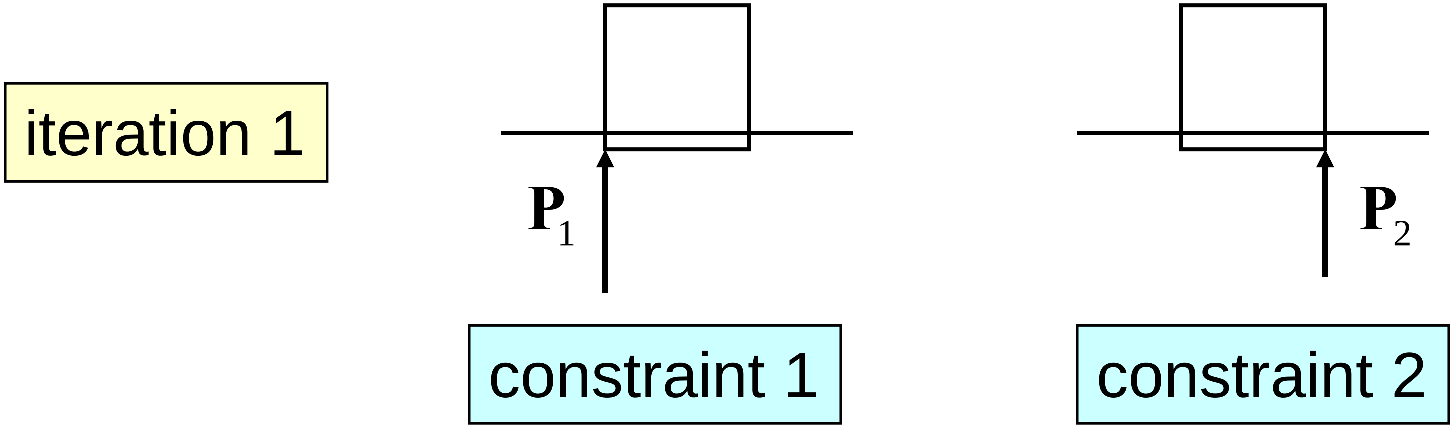


Global Solution



$$\mathbf{P} = \frac{1}{2} m\mathbf{v}$$

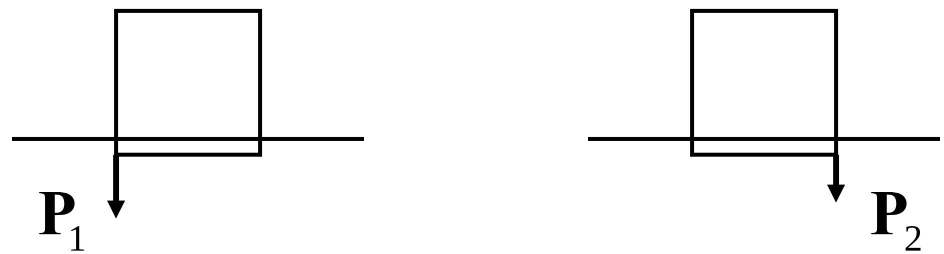
Iterative Solution



Suppose the corrective impulses are **too strong**.
What should the second iteration look like?

Iterative Solution

iteration 2



To keep the box from bouncing, we need downward corrective impulses.

In other words, the corrective impulses are **negative**!

Iterative Solution

But clamping the negative corrective impulses wipes them out:

$$\lambda = \text{clamp}(\lambda, 0, \infty)$$
$$= 0$$

This is one way to introduce jitter into your simulation. ☺

Accumulated Impulses

- For each constraint, keep track of the total impulse applied.
- This is the *accumulated impulse*.
- Clamp the accumulated impulse.
- This allows the corrective impulse to be **negative** yet the accumulated impulse is still positive.

New Clamping Procedure

1. Compute the corrective impulse, but don’t apply it.
2. Make a copy of the old accumulated impulse.
3. Add the corrective impulse to the accumulated impulse.
4. Clamp the accumulated impulse.
5. Compute the change in the accumulated impulse using the copy from step 2.
6. Apply the impulse delta found in Step 5.

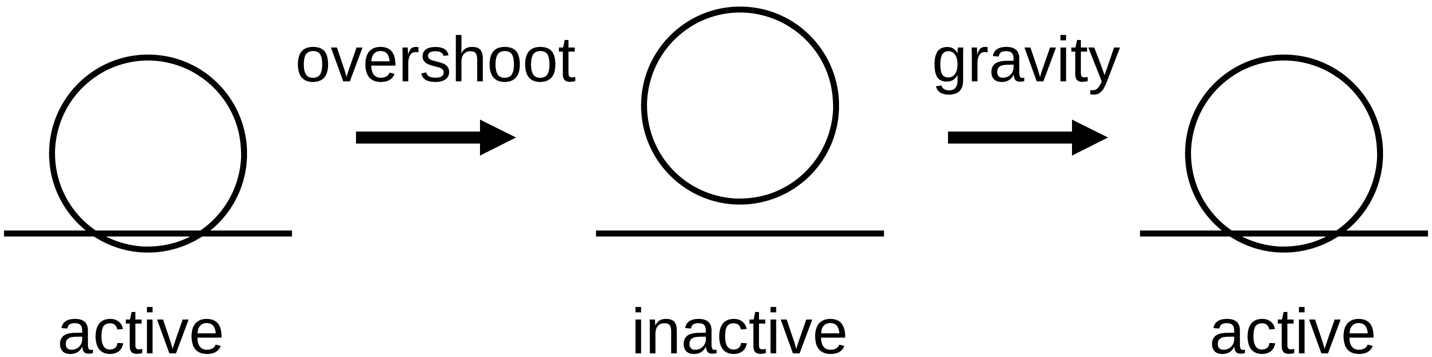
Handling Inequality Constraints

- Before iterations, determine if the inequality constraint is active.
- If it is inactive, then ignore it.
- Clamp accumulated impulses:

$$0 \le \lambda_{acc} \le \infty$$

Inequality Constraints

A problem:



Aiming for zero overlap leads to *JITTER*!

Preventing Overshoot

Allow a little bit of penetration (slop).

If separation < slop

$$\dot{C} = \mathbf{J}\mathbf{v} + \frac{\beta}{h}\left(\delta - \delta_{slop}\right)$$

Else

$$\dot{C} = \mathbf{J}\mathbf{v}$$

Note: the slop will be negative (separation).

Warm Starting

- Iterative solvers use an initial guess for the lambdas.
- So save the lambdas from the previous time step.
- Use the stored lambdas as the initial guess for the new step.
- Benefit: improved stacking.

Step 1.5

- Apply the stored impulses.
- Use the stored impulses to initialize the accumulated impulses.

Step 2.5

- Store the accumulated impulses.

Further Reading & Sample Code

- http://www.gphysics.com/downloads/

Box2D

- An open source 2D physics engine.
- http://www.box2d.org
- Written in C++.