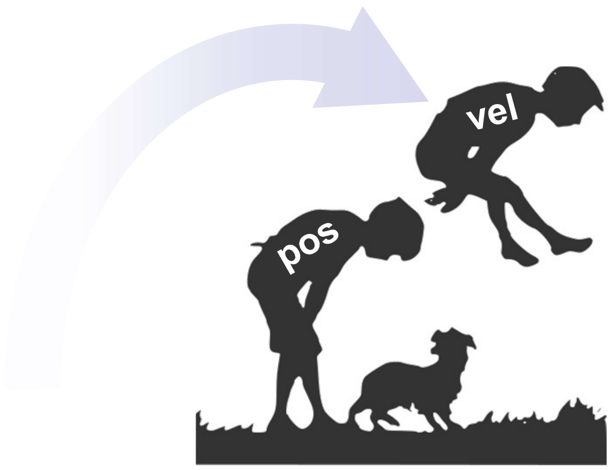
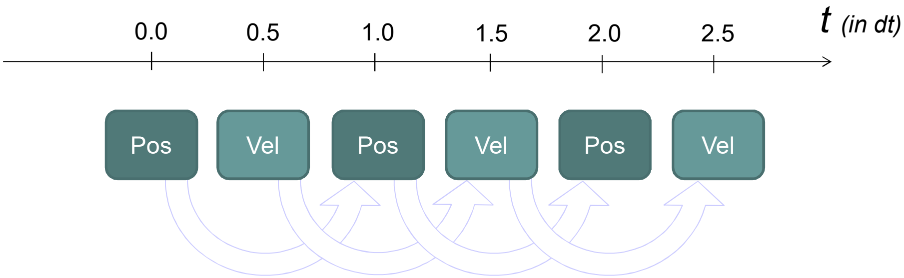
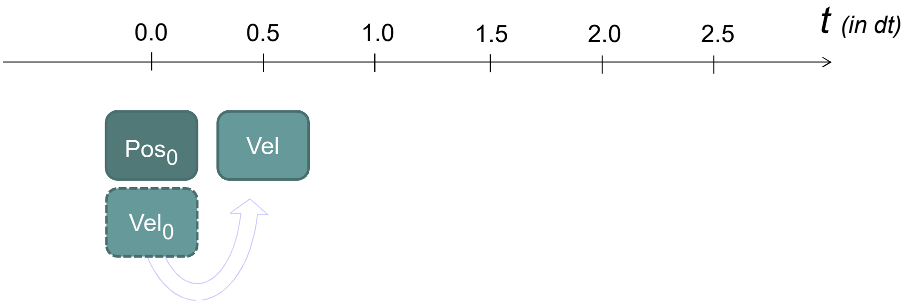
Leapfrog Integration



Leapfrog Integration

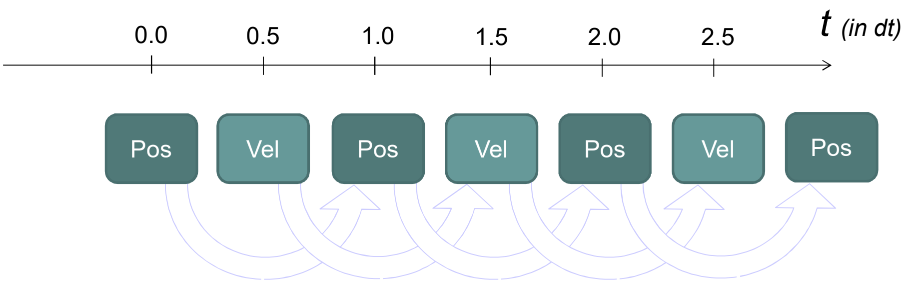


Leapfrog Integration
first step



$$a = f(p_0\ ,\ \ldots)$$
$$\vec{v}_{0.5} = \vec{v}_0 + \vec{a} \cdot dt / 2$$

Leapfrog Integration



$p_1 = p_0 + \vec{v}_{0.5} \cdot dt$ $\quad p_2 = p_1 + \vec{v}_{1.5} \cdot dt$ $\quad p_3 = p_2 + \vec{v}_{2.5} \cdot dt$

$a = f(p_1\ ,\ \ldots)$ $\quad a = f(p_2\ ,\ \ldots)$

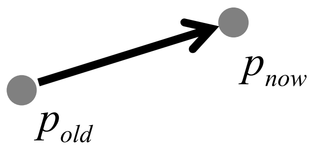
$\vec{v}_{1.5} = \vec{v}_{0.5} + a \cdot dt$ $\quad \vec{v}_{2.5} = \vec{v}_{1.5} + a \cdot dt$

Leapfrog method: advantages

- Better accuracy for same dt
  - (better asymptotic behavior)
  - "second order instead of first"
  - (residual error dt^3 instead of dt^2)
- Same cost as Euler
- bonus: fully reversible! ☺

Verlet method
("position based dynamics")

- Idea: remove velocity from state
- Current velocity: implicit
- Computed by:
  delta between
  - Current pos
  - Last pos (which is recorded)



$$\vec{v} = (p_{now} - p_{old}) / dt$$

Verlet method
(position based dynamics)

init state

$$p_{now} = \ldots$$
$$p_{old} = \ldots$$

one step

$$\vec{f} = fun(p_{now})$$
$$\vec{a} = \vec{f} / m$$
$$\vec{v} = (p_{now} - p_{old}) / dt$$
$$\vec{v} = \vec{v} + \vec{a} \cdot dt$$
$$p_{next} = p_{now} + \vec{v} \cdot dt$$

expanding this…

Verlet method
(position based dynamics)

init state

$$p_{now} = \ldots$$
$$p_{old} = \ldots$$

one step

$$\vec{f} = fun(p_{now})$$
$$\vec{a} = \vec{f} / m$$
$$p_{next} = 2 p_{now} - p_{old} + \vec{a} \cdot dt^2$$

$$p_{old} = p_{now}$$
$$p_{now} = p_{next}$$

The real advantage of position based

init state

$$p_{now} = \dots$$
$$p_{old} = \dots$$

one step

$$\vec{f} = fun(p_{now})$$
$$\vec{a} = \vec{f} \,/\, m$$
$$p_{next} = 2p_{now} - p_{old} + \vec{a} \cdot dt^2$$

Enforce constraints on (p_{next})

$$p_{old} = p_{now}$$
$$p_{now} = p_{next}$$

Verlet: characteristics

- Implicit velocity!
- Good efficiency / accuracy ratio
  - accumulated error: order of dt^2
- Extra bonus: reversibility of the system
  - (it's possible to travel the evolution backward in *t* and go back to the correct initial state)
  - (being careful with implementation details)
- Principal advantage: flexibility
  - possible to impose constraints directly on positions!
    - and get "automatic" velocity adjustment (not the correct ones, but plausible ones)

Verlet: *caveats*

- it assumes d*t* (time-step) to be constant
  - if it varies: corrections needed! (which ones?)
- how to act on velocity (which is implicit) ?
  - e.g., for: damping
  - e.g., for: impulses
  - A: change p_{old} instead
- how to change position without impacting velocity?
  - A: change both p_{now} and p_{old}

dt updates in Verlet

(if they are not constant)

Problem:

if dt now changes to a new dt'

then, all p_{old} must be updated to some p'_{old}

Find p'_{old} :

$$\vec{v} = (p_{now} - p_{old}) / dt$$

$$\vec{v} = (p_{now} - p'_{old}) / dt'$$

current velocity and position must not change

$$p'_{old} = p_{now} \cdot (dt - dt') / dt + p_{old} \cdot dt' / dt$$

Damp (drag) in Verlet

e.g. 0.99

Problem: we want to damp velocities
i.e. mult them $\vec{v}' = \vec{v} \cdot c_{DAMP}$
by updating p_{old} to some p'_{old}

Find p'_{old} :

$$\vec{v} = (p_{now} - p_{old}) / dt$$
$$\vec{v}' = \vec{v} \cdot c_{DAMP} = (p_{now} - p'_{old}) / dt$$

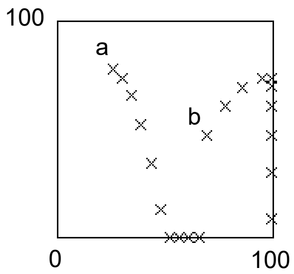
$$p'_{old} = p_{now}(1 - c_{DAMP}) + p_{old} \cdot c_{DAMP}$$

Positional constraints

- Generic and expressive
  - Lots of possible phenomena
  - for instance: “no interpenetration”
- Easily defined
- Easy to impose
  - Imposing a constraint (positional) =
  - = find the positions similar to the current ones satisfying it
  - = project the current state in the allowed state space
- Verlet benefit:
  - update velocity: *automatic !*
  - without using forces / impulses
    - (the ones that in reality impose the constraints)
  - → approximation, but plausible results!

Example of positional constraint

«Particles must stay within [0 – 100] x [0 – 100] *»*



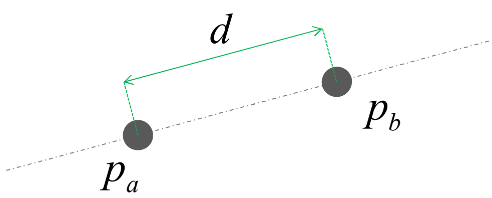
Imposing constraint: simple clamp !
ex:

```
for(int i=0; i<NUM_PARTICLES; i++)
{
  p[i].x = clamp( p[i].x, 0, 100 );
  p[i].y = clamp( p[i].y, 0, 100 );
}
```

Imposing constraints like this is a first **collision response**.
But: for bounces (impact impulses) must be added.

Ex: Equidistance constraints

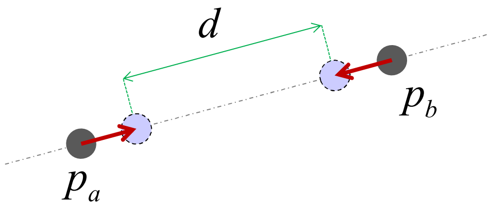
«Particles ***a*** *and* ***b*** *must be at distance* ***d*** *»*



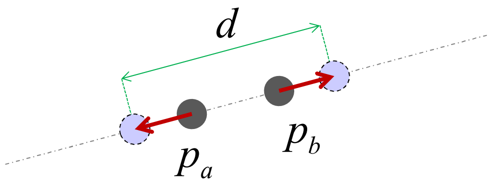
$$| p_a - p_b | = d$$

Impose equidistance constraints

if $|p_a - p_b| > d$



if $|p_a - p_b| < d$



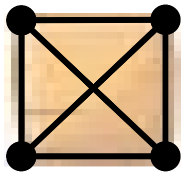
Equidistance constraints: pseudo code

```
Vector3 pa, pb; // curr positions of a,b
float d;        // distance (to enforce)

Vector3 v = pa - pb;
float currDist = v.length;

v /= currDist;  // normalization of v

float delta = currDist - d ;

pa += ( 0.5 * delta) * v;
pb -= ( 0.5 * delta) * v;
```

assuming equal mass, each particle moves half the way
(see later for the general case)

Combinations of equidistance constraints

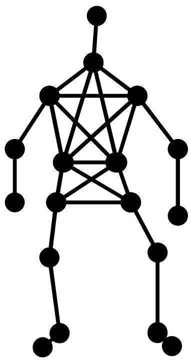
- For obtaining:
  - Rigid bodies
  - note:
    - only positions / vel / acc!
    - no: angles,
      angular vel
      angular acc

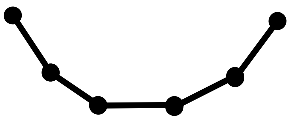


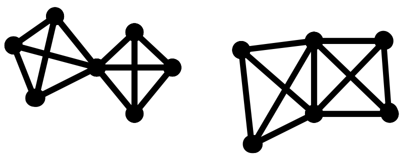
A box?
(rigid object)
configuration of:
- 4 particles
- 6 equidistance constraints

Combinations of equidistance constraints

- For obtaining:
  - Rigid bodies
  - Ragdolls
  - Cloth
  - Non-elastic ropes
  - …







Spring-like behaviour, but for rigid bodies!

Equidistance constraints VS springs

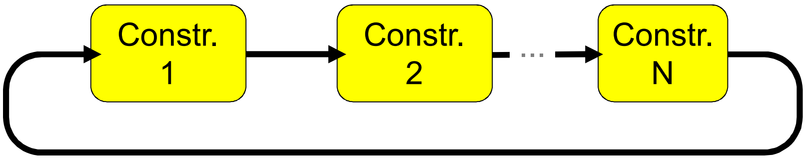
- Aren't they similar?
  - they both mean:
    these two particles "want to be" at this distance
- But:
  - equidistance constraint:
    - applied during constraint enforcement
      - directly affecting positions
    - models a rigid rod between the two particles
      - of a given length
    - sometimes called an "HARD" constraint
  - spring:
    - applied during force evaluation step
      - affecting forces, therefore accelerations
    - models a deformable spring between the two particles
      - of a given length
    - sometimes called a "SOFT" constraint
- They can be combined in one object!

More examples of positional constraints

- Preserving areas / volumes: *«Volume is* v_C *»*
  - How to impose it:
    1. Estimation of current total volume v
    2. uniform scaling of the object of $\sqrt[3]{v_C / v}$
- Fixed positions: *«particle* a *stays in* p_a *»*
  - particles «pinned in position»
  - trivial, but useful!
- Angle constraints
  - ex, on joints: *«elbows cannot bend backward»*
- Coplanarity / colinearity
- Non interpenetration
  - (part of collision handling – see later)

Enforcing sets of constraints

- Many constraints to impose:
  solve one → break another one!
- Simultaneous enforement: computationally expensive
- Practical solution: enforce them in cascade
  (*a-la* Gauss-Seidel):



Repeat until convergence (= max error below threshold)
…but at most for *N* times! (remember: *soft* real time)

Enforcing sets of constraints

- Note:
  - The whole loop for imposing the constraints happen in just one physics step!
- Convergence:
  - if constraints are not contradictory
  - if convergence not reached (or solution doesn't exist): never mind, next frames will fix it (fairly robust)
  - needed iterations (typically): 1 ~ 10 (efficient!).
  - Optimization (to decrease number of needed iterations): solve the most unsatisfied constraints first
- ⚠ Problem: it's a sequential approach! ☹
  - but parallelized versions (similar to Jacobi) have a worse convergence in practice

Enforcing a positional constraint: the general case.

Check constraint (on position)

- It holds? Nothing to do
- It doesn't?
  - All positions must be changed so that it does
  - Conceptual problem:
    infinite ways to achieve this. Which one to pick?
  - Answer:
    minimize the sum of all *squared* displacements
    (with respect to current position)
    weighted by particle masses
  - Find it by analytically solving simple problems
    ("analytically" = "on paper")

Enforcing a positional constraint the general case: formally

To enforce a constraint $\mathcal{C}$ on particles a , b , c,…
which are currently in position $\mathbf{p}_a, \mathbf{p}_b, \mathbf{p}_c, \dots$
and have masses $m_a, m_b, m_c \dots$:
we must apply the displacements $\vec{d}_a, \vec{d}_b, \vec{d}_c$
that defined by minimizing:

$$\underset{\vec{d}_a, \vec{d}_b, \vec{d}_c, \dots}{\operatorname{argmin}} \left(m_a \|\vec{d}_a\|^2 + m_b \|\vec{d}_b\|^2 + m_c \|\vec{d}_c\|^2 + \cdots \right)$$

$$\text{such that } \mathcal{C}\left(\mathbf{p}_a + \vec{d}_a, \mathbf{p}_b + \vec{d}_b, \mathbf{p}_c + \vec{d}_c, \dots\right)$$

among all the choices that satisfy this,
we want the one which minimizes this

Enforcing positional constraint Example: equidistance

- To enforce the constraint
  "particles a and b must stay at distance D"
  - Given: current positions $\mathbf{p}_a$, $\mathbf{p}_b$
  - and masses m_a, m_b
- We need to apply the displacements $\overrightarrow{d_a}$, $\overrightarrow{d_b}$ found by minimizing:

$$\underset{\overrightarrow{d_a}, \overrightarrow{d_b}}{\operatorname{argmin}} \left(m_a \left\| \overrightarrow{d_a} \right\|^2 + m_b \left\| \overrightarrow{d_b} \right\|^2 \right)$$

$$\text{such that } \left\| \left(\mathbf{p}_a + \overrightarrow{d_a} \right) - \left(\mathbf{p}_b + \overrightarrow{d_b} \right) \right\| = D$$

- And the solution (in closed form) is…

Equidistance constraints: solution for non-equal masses

```
Vector3 pa, pb; // curr positions of a,b
float ma, mb;   // masses of a,b
float D;        // distance (to enforce)

Vector3 v = pa - pb;
float currDist = v.length;

v /= currDist;  // normalization of v

float delta = currDist - D ;

/* solution of the minimization: */
pa += ( mb/(ma+mb) * delta) * v;
pb -= ( ma/(ma+mb) * delta) * v;
```

Enforcing positional constraint Example: don't sink in a plane

- To enforce the constraint
  "particle a must be over (not below) a plane q "
  - Given: position of the particle $\mathbf{p}_a$ and its mass m_a
  - Point on a plane $\mathbf{p}_q$ and its normal (unit vec) $\hat{n}_q$
- We need to apply the displacement $\overrightarrow{d_a}$ found by minimizing:

$$\underset{\overrightarrow{d_a},\,\overrightarrow{d_b}}{\operatorname{argmin}} \left(m_a \left\| \overrightarrow{d_a} \right\|^2 \right)$$

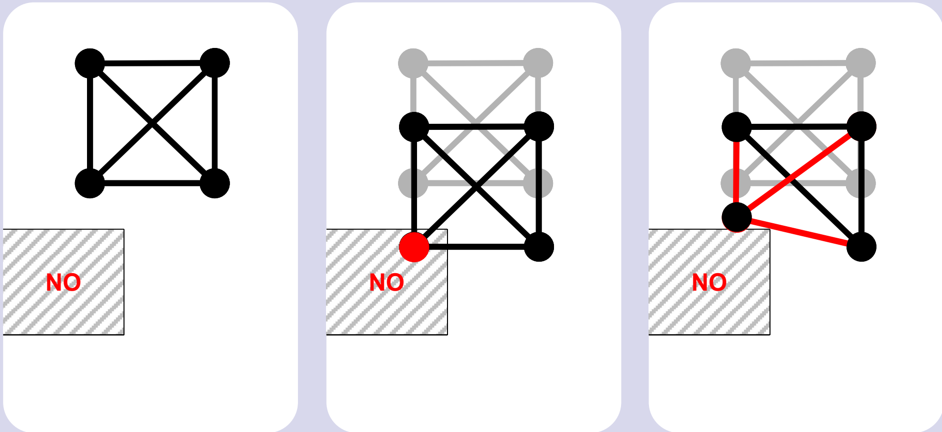
$$\text{such that } \left\| (\mathbf{p}_a - \mathbf{p}_q) \cdot \hat{n}_q \right\| > 0$$

- And the solution (in closed form) is, trivially…

In pseudocode

```
Vector3 pa; // curr positions of a,b
float ma;   // masses (no effect here)
Vector3 pq; // point on the plane
Vector3 nq; // normal of the plane (unit)

Vector3 v = pa - pq;
float currDist = Vector3.dot( v , n );

if (currDist < 0.0)
   pa -= currDist * n; // just project!
else {} // constrain holds, do nothing
```

Example

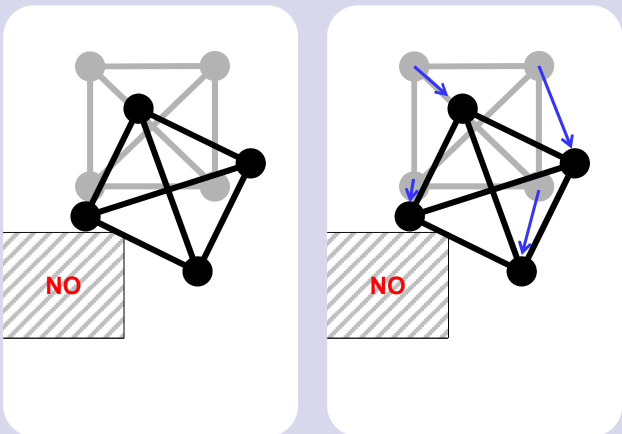


STEP 0

STEP 1
before constraints

STEP 1
after 1st constraint

Example



STEP 1
after all constraints
multiple times

STEP 1
(implicit) velocities

So, in total:
the “box”, under gravity + impact

- had rotated
- gained angular velocity (will keep on rotating by inertia)

even the system does not (explicitly) handle rotations or angular velocities

(works in 3D as well!)

Position Based Dynamics: Advantages

- Interestingly, rich/useful set of "emerging behaviors" (i.e. effects with "just automatically happens") including:
  - rigid, deformable, jointed objects
    - made of particles + hard constraints
  - their angular velocities
    - automatically around…
  - their barycenter
  - their momentum of inertia
    - angular velocity is maintained
  - somewhat believable bounces on "impacts"
    - but, out of designer control: impact impulses can be added
- Simulation is intrinsically fairly robust
  - sensible constraints explicitly re-enforced every frame:
    - e.g. the ball won't be (permanently) out of the box containing it

don't need to be computed (or stored)

simply, enforcement of non-compenetration

Position Based Dynamics: Challenges

- Simulation is only approximate
- Satisfying many constraints can be demanding
  - especially collision constraints, not know a priori!
  - a large number low-level constraints are needed
- Order of constraint enforcement is crucial
  - and so is the need to do them *in parallel*
- Much of the data which is kept and dealt with implicitly can be needed by the rest of the engine, and therefore it must be extracted ☹
  - e.g. current orientation (rotation) of a compound rigid object made of connected particles:
    - (needed for rendering!)
  - its angular speed, barycenter pos, (average) speed…

In total, Particle-Based PBD is one good solution, but by no means an easy, or universal, one.