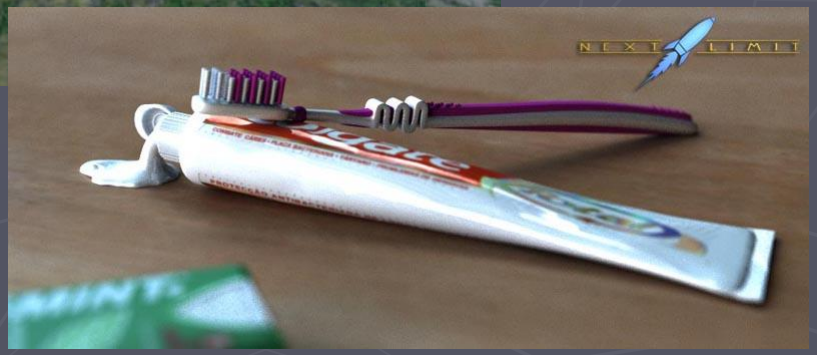
COMP_SCI 351-2: Intermediate Graphics



Jack Tumblin
COMP_SCI 351-2

CS 351-2 Intermediate Graphics

- **TWO big projects:** (2 x 45% of grade)
  - A System of Particle Systems (Feb 01 Demo)
    (or 'soft amorphous things that move')
  - Recursive Ray Tracer (Mar 10 Demo)
    (or 'light done right')

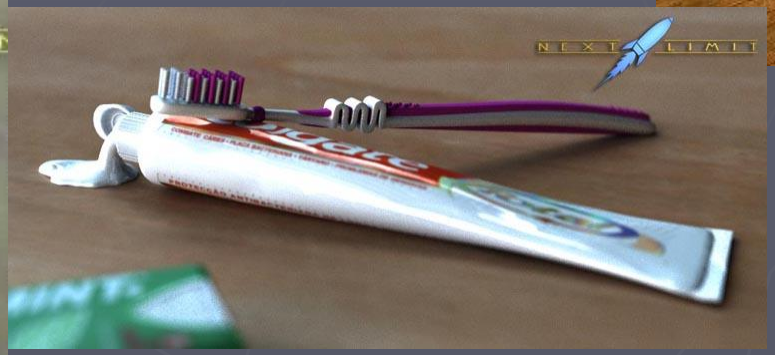
- **SEVERAL small 'Activities':** (10% of grade)
  - Informal; meant to help you keep up;
  - Some start in-class, ad-hoc
  - Some are 'check-points' on projects

Log in to CANVAS and…

- Sign up for CampusWire
- Review Syllabus
- Start Readings

- (after a few minutes) try Starter code…

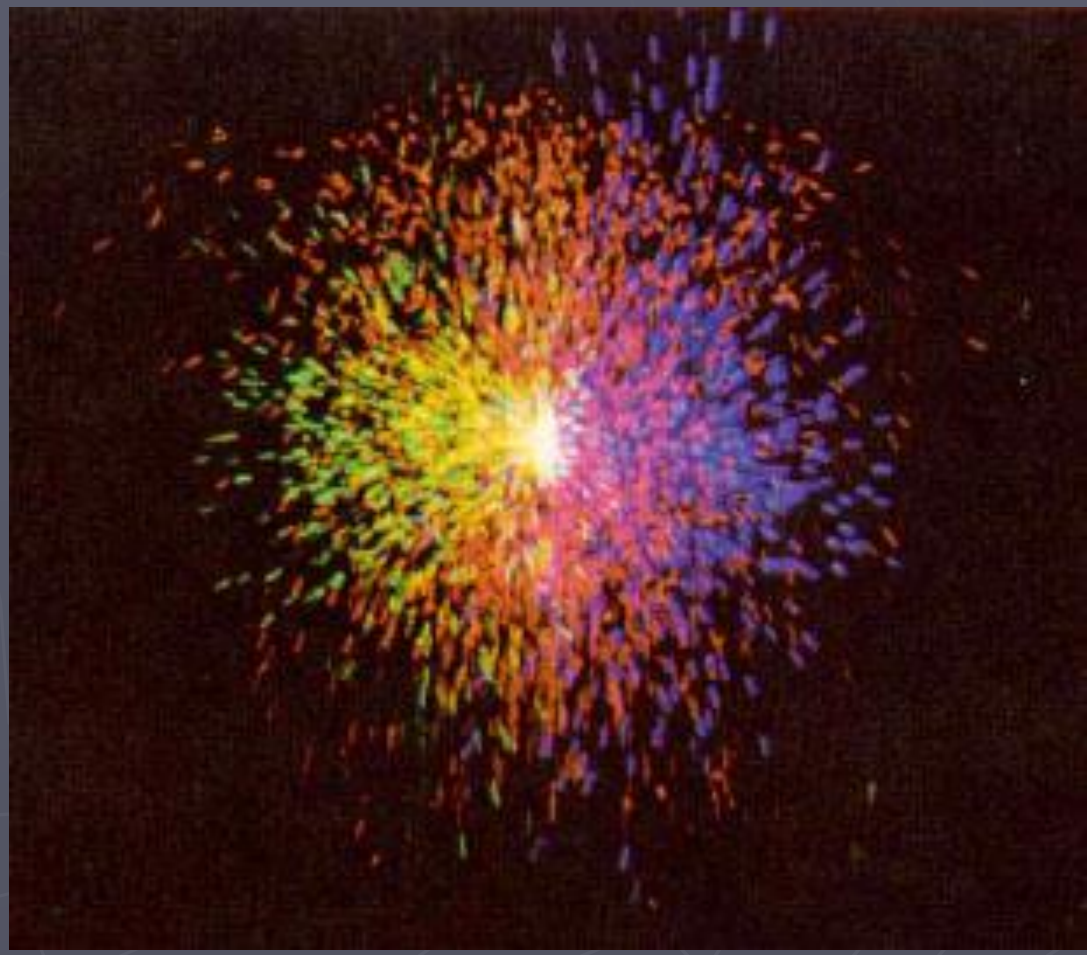
Writing Code for Particle Systems--A



Jack Tumblin
COMP_SCI 351-2

What is a Particle System?

- Uses *points* to define movement, shapes, mass, color, behavior & movement *rather than polygons*

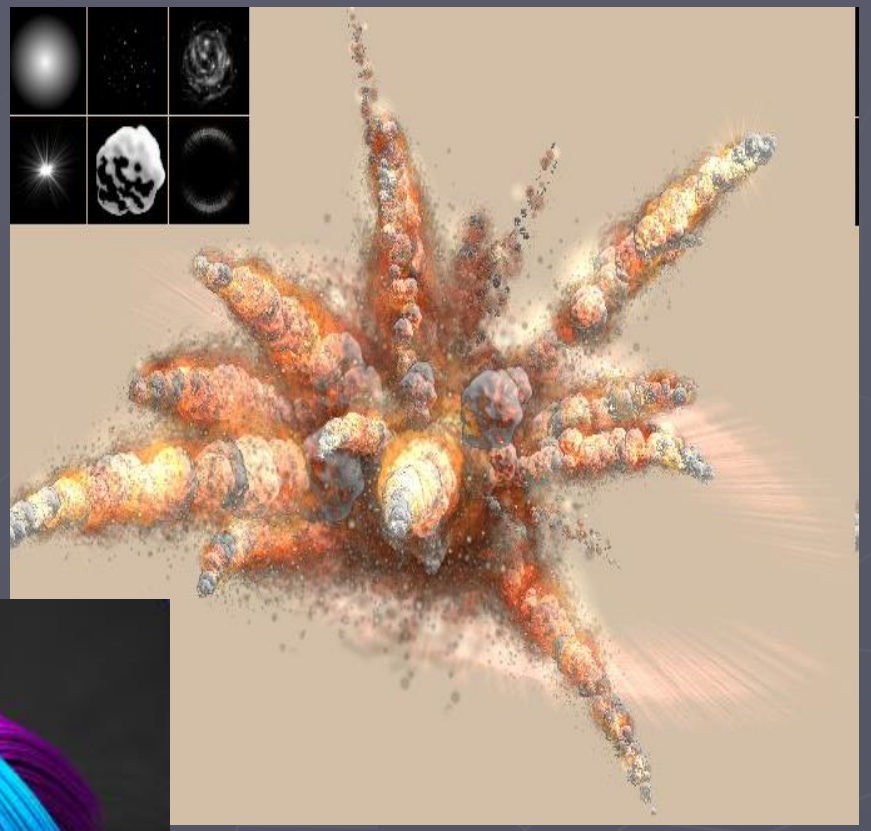


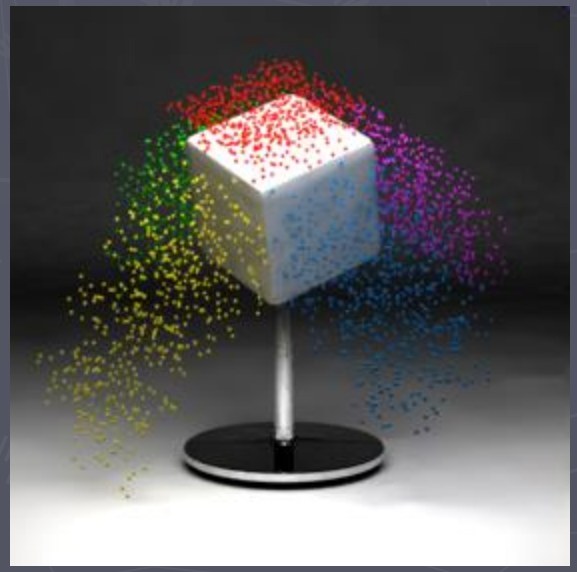
- SIMPLER than 3D solid dynamics, yet suitable for physical simulation of many complex & high-DOF phenomena

BUT they don't have to be ***RENDERED*** as points: ('fuzzy bunny' particles)
http://www.youtube.com/watch?v=XVTo4y5e08k

Many Ways to **Render** Particles

- Simple points or balls;
- Tumbling textures or movies (e.g. smoke puffs)
- 'Trails' that hold ***N*** previous locations (fireworks, hair)





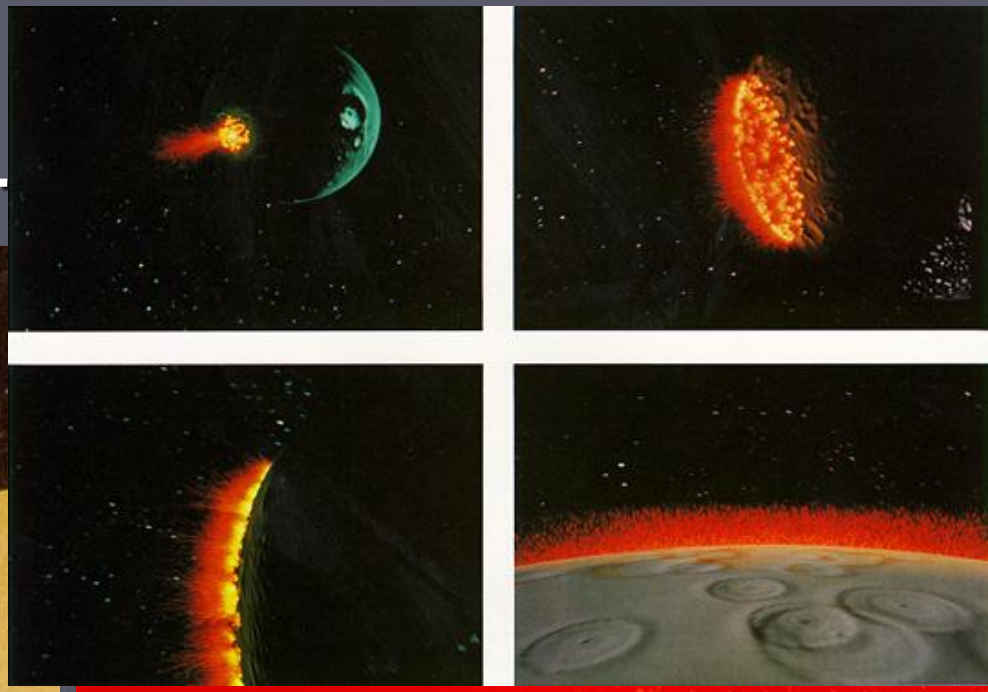
Particle Objects approximate many dynamic, tedious, aggregate behaviors

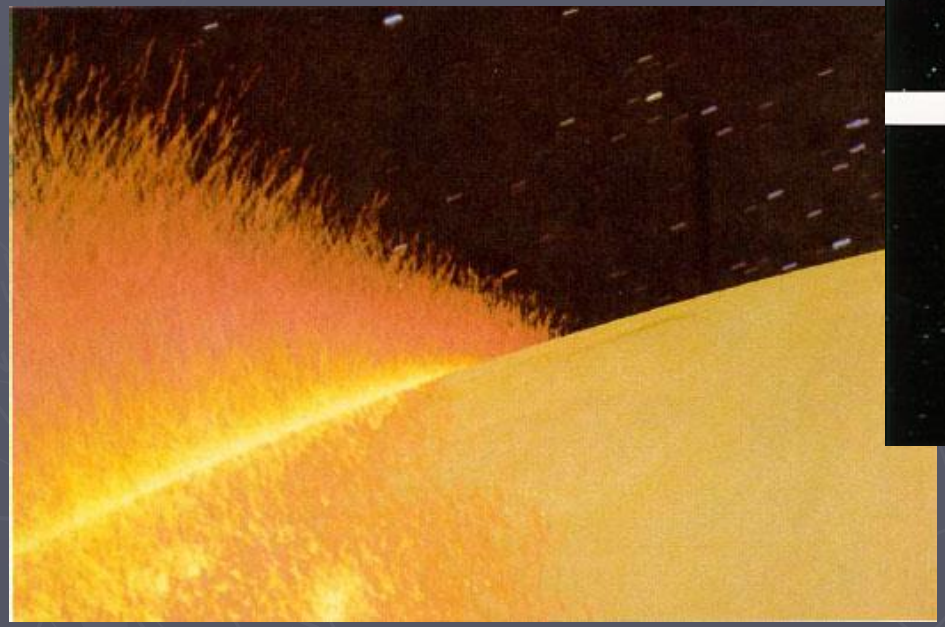
- Grass, Smoke, fire, clouds, water, …
- Fireworks, explosions
- Fluid flow
- Physical simulations
- Flocking:
  Bird migration, ants, schools of fish, riots…
- **a nice online tour:**

https://gpfault.net/posts/webgl2-particles.txt.html (@page end); then https://youtu.be/teN-52YIRNs?t=14

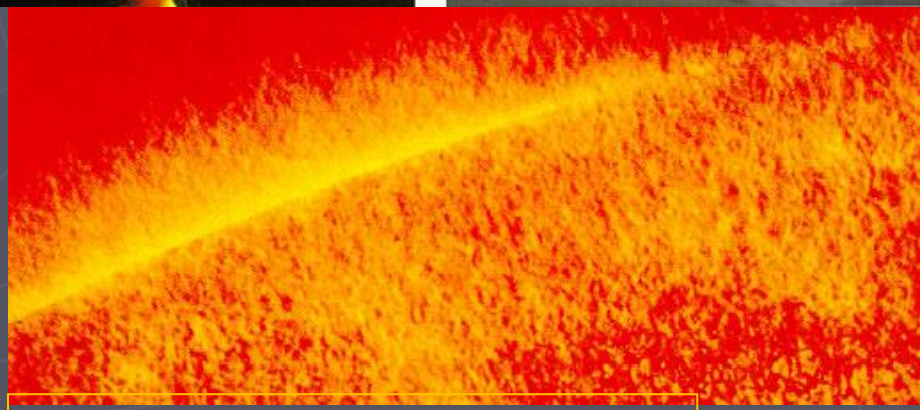
1981: ? Simulate Planet on Fire ?

- "A particle has no volume:
- "No occlusion: particles only *add* light on-screen…





Reeves, W.T. "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects", Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83)



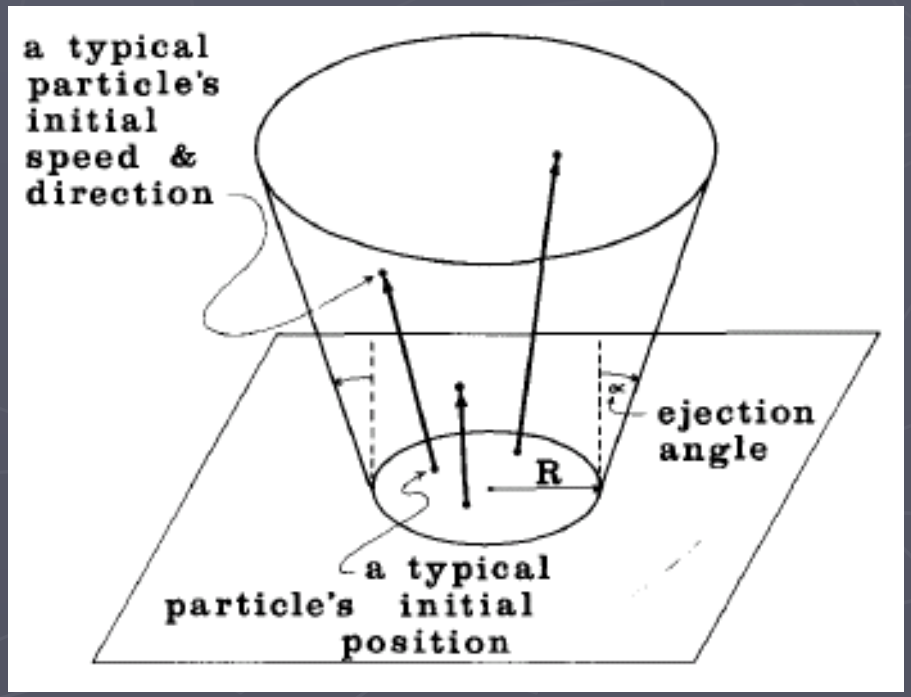
Star Trek II: The Wrath of Khan (1982)

Reeve's Particle-Fire Model

- Born: set initial position, mass, velocity; set random lifetime for each particle
- Age: add to each particle's age every frame
- Die: "delete"(re-use) any particles that are:
  - Older than their assigned lifetime, or
  - Too dim (below a visibility threshold value), or
  - Permanently outside camera field of view, or
  - Permanently occluded (in movies: behind you)

Reeves: 'Ballistic' Fire Particles: Randomized Launch Params

- Randomized initial Position, velocity, color,…
- Uniform distribution usually OK; (Gaussian is costly)
  = (mean – variance) + 2.0*variance*math.Random();



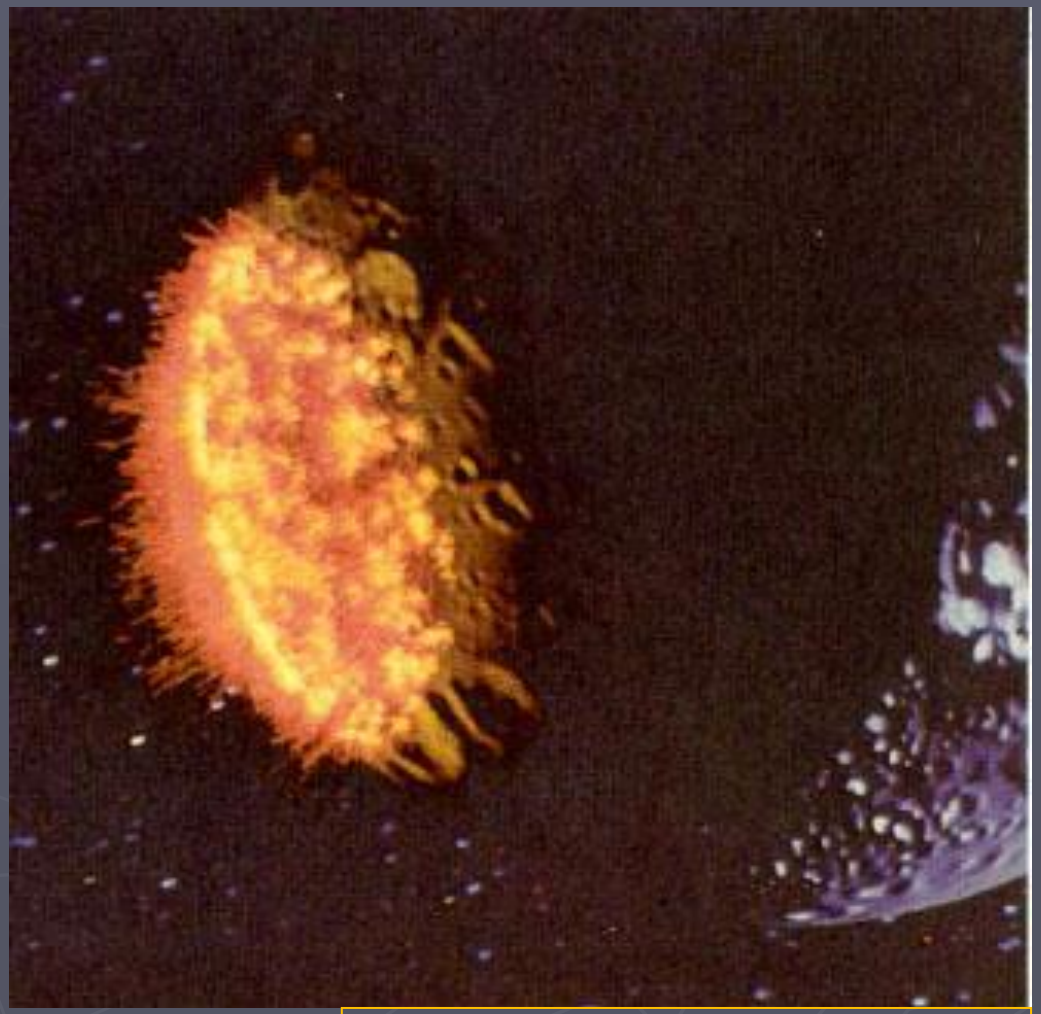
Meet Bill Reeves! Pixar In A Box (Khan Academy) https://youtu.be/ovlVh-QgVao?t=59)

Fire: Particle Rendering

- Rendering Difficulties
  - Particles occlude particles?
  - Particle-Particle collisions? Shadows? Transparency?
  - How do they interact with Polygonal Shapes?

EASY:

- Assume NO collisions, NO occlusions, NO shadows.
- Assume particles act as blurred point-light sources (infinitesimal)
- https://youtu.be/Qe9qSLYK5q4?t=2m19s



Star Trek II: The Wrath of Khan (1982)

The 'Life' of a Fire Particle

- At each frame,
  Add 3D velocity vector to 3D position vector *AND*
  Add 3D accelerations (e.g. gravity) to 3D velocity
  - Particles then move in parabolic arcs over time
  - Colors change – they depend on 'age' vs. 'lifetime'
  - Render all as 'streaks' that connect it's last few positions

- What else can we do? What other behaviors?
- How can we make particles interact?
- How can we *organize & unify* all particle systems?

Particle System Objects
(a core idea for Project A:)

Don't write code for just one set of particles, encapsulate it **(JS: object or prototype)**

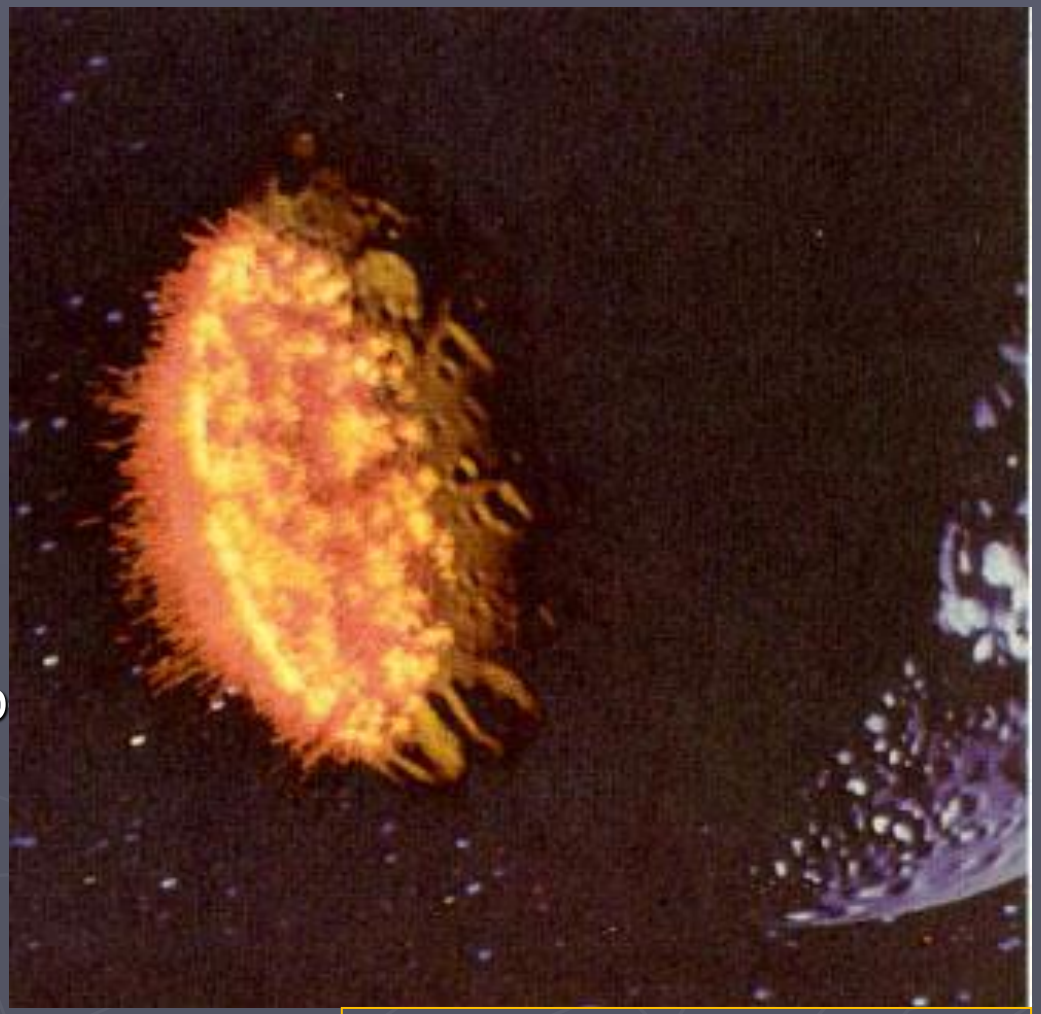
(http://www.phpied.com/3-ways-to-define-a-javascript-class/)
(https://www.w3schools.com/js/js_classes.asp)

to create a particle-system 'type'

- Each variable of that type contains ==one complete particle system
- Permits **multiple simultaneous systems** of particles, each with its own behaviors, forces, constraints, rendering methods, and user interactions. Proj A: make 4 of them!

1: Make your own Reeves Fire

- Write your own Reeves Fire and flame system:
  - Randomized launch
  - Gravity + ballistic travel
  - Aging affects color, mass
  - streak rendering
  - Interactive? Can you make user-controlled 'wind' too? Can your fire travel & leave b burned ground-plane?

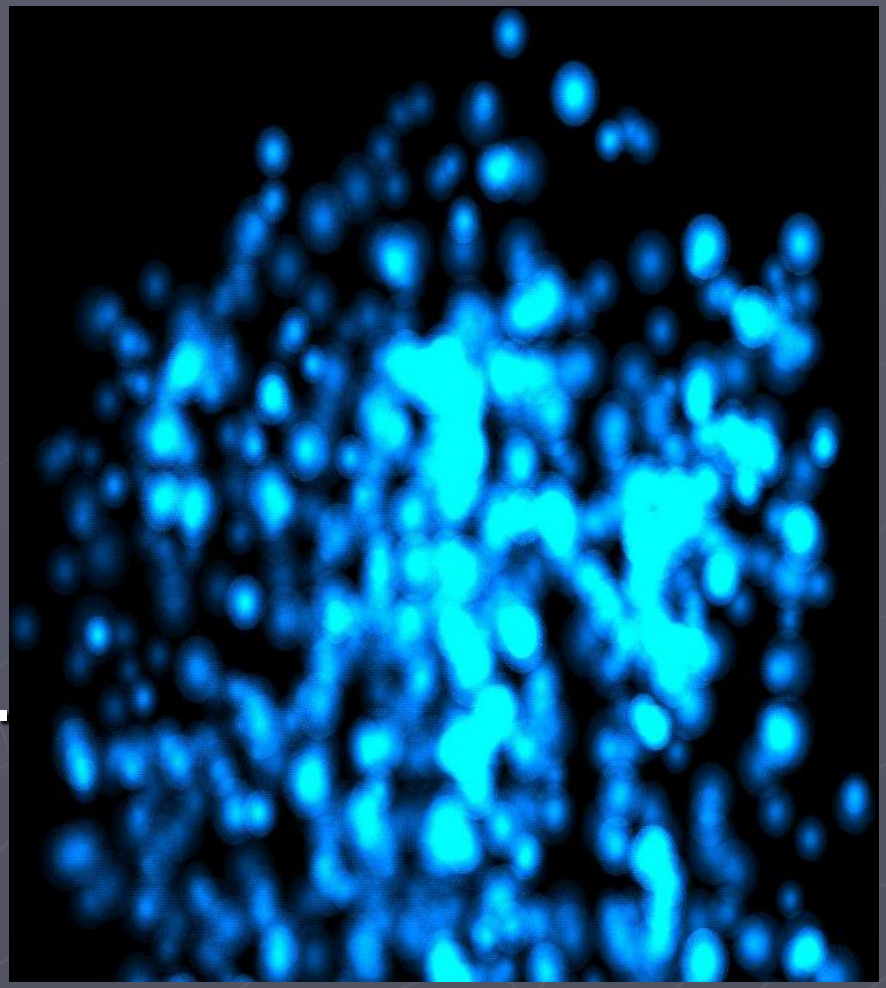


Star Trek II: The Wrath of Khan (1982)

- https://youtu.be/Qe9qSLYK5q4?t=2m19s
  (Reeves and others explain / demo it)

1: Generalize to 'Dumb' Particles

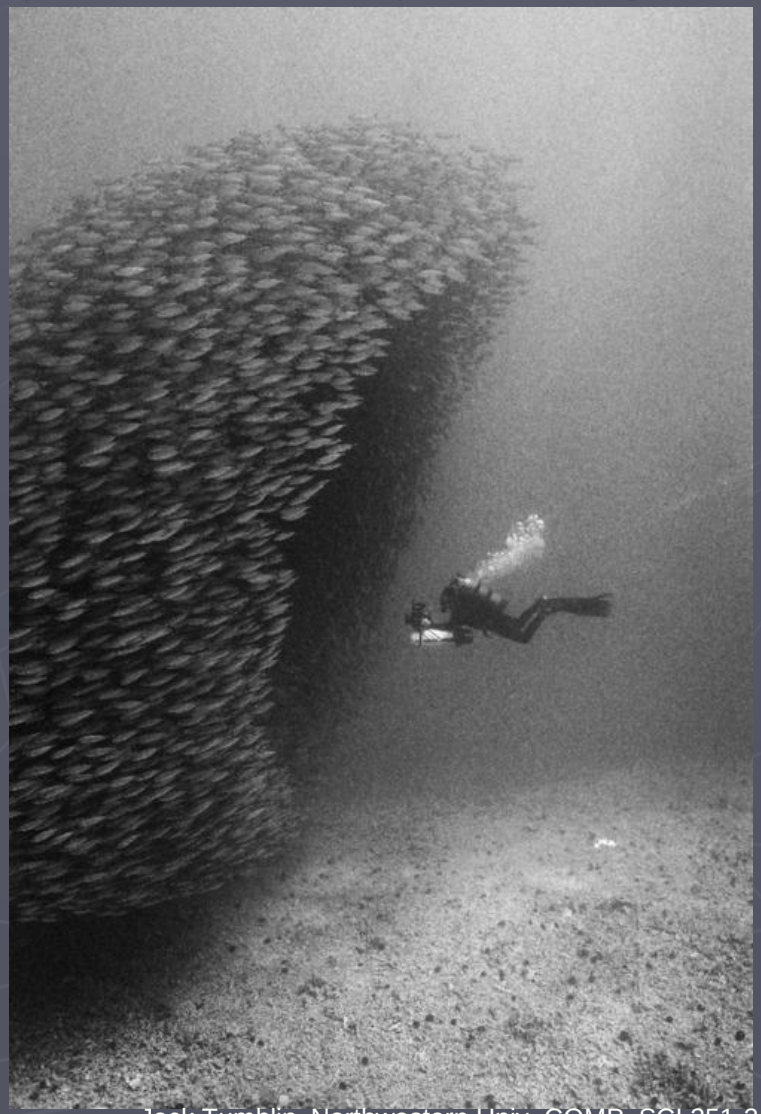
- Particles that ignore each other, but ***respond to 3D force fields ...*** gravity, heat, wind, temp,
- INCLUDES: Bouncy balls, fireworks, rain, hail, tornados & other vortices..
- ...but what could we do if particles can *respond* to their neighbors? → boids!



Demo

2: Flocking Particles ('boids' 1987)

- http://www.red3d.com/cwr/boids/ SIGGRAPH 1987, Craig Reynolds, 'Artificial Life'
- http://harry.me/blog/2011/02/17/neat-algorithms-flocking/ Will You Harry Me? (Harry Brundage)

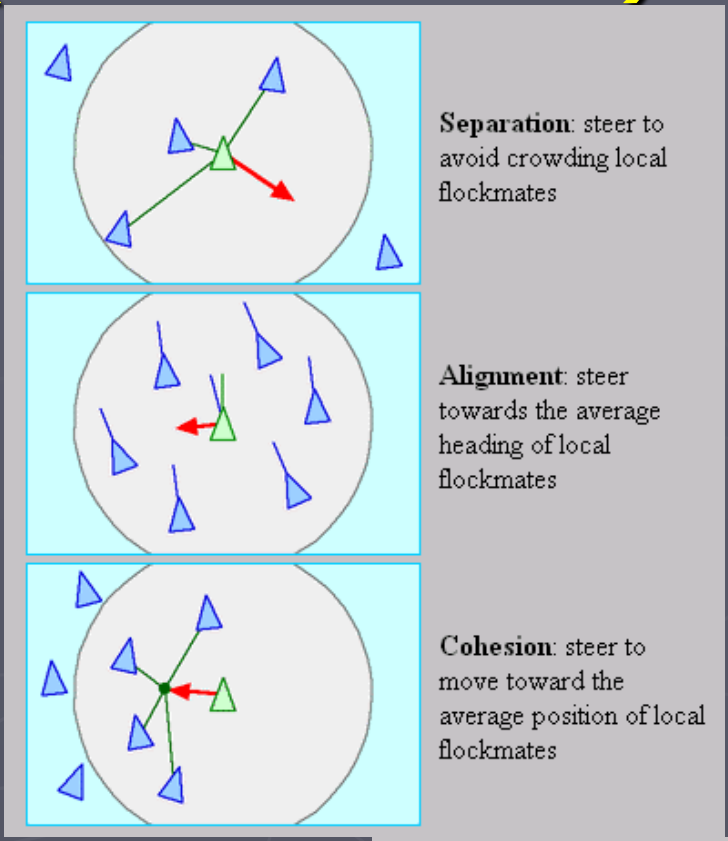


: Flocking Patterns ('boids' 1987)

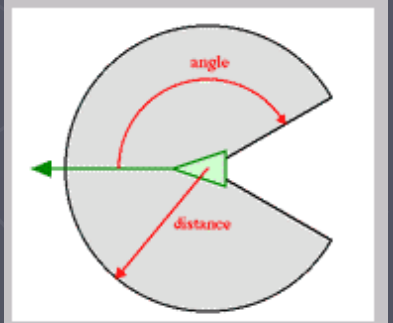
Main Biomimetic Goal: Survive!

Make mearby-particle forces for

- **Separation:** Don't collide with your neighbors!
- **Alignment:** Point in the same direction as neighbors
- **Cohesion:** Move toward average position of neighbors
- **Evasion:**
  Don't hit obstacles;
  Run from predators



http://www.red3d.com/cwr/boids/
SIGGRAPH 1987, Craig Reynolds
"Boids and the development of 'Artificial Life'

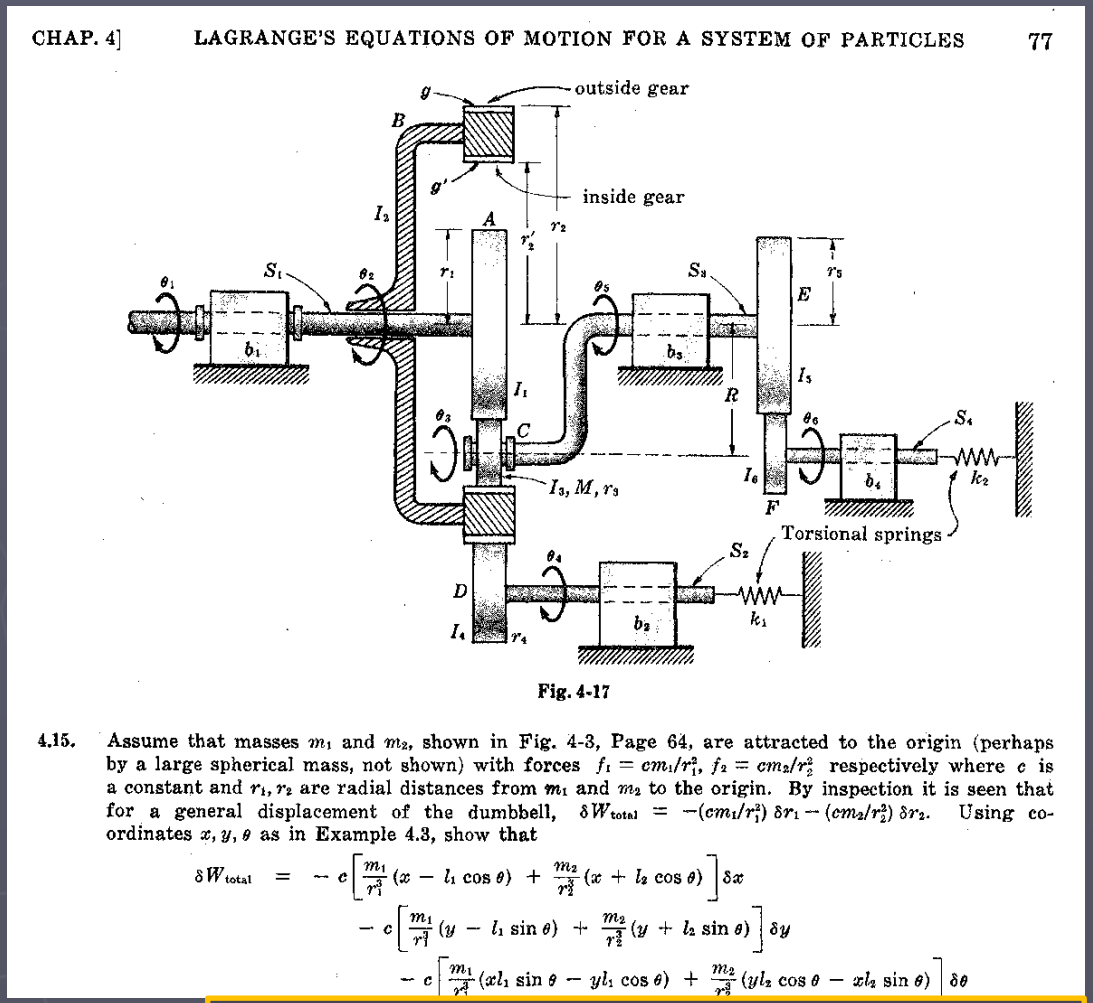


a boid's neighborhood

Classical Dynamics? → Tougher!

- Systems of 1D, 2D, 3D rigid bodies & forces
- Custom-tailored coord. systems, DOFs & algebraic forms
- Numerical Solutions often 'stiff' & unstable
- Lagrangian Dynamics yield SUPERB results, but
- **Difficult to generalize,** especially in code:

  new configurations→
  →new equations→
  →new code to compile?



CHAP. 4] LAGRANGE'S EQUATIONS OF MOTION FOR A SYSTEM OF PARTICLES 77

Fig. 4-17

4.15. Assume that masses m_1 and m_2, shown in Fig. 4-3, Page 64, are attracted to the origin (perhaps by a large spherical mass, not shown) with forces $f_1 = cm_1/r_1^2$, $f_2 = cm_2/r_2^2$ respectively where c is a constant and r_1, r_2 are radial distances from m_1 and m_2 to the origin. By inspection it is seen that for a general displacement of the dumbbell, $\delta W_{\text{total}} = -(cm_1/r_1^2)\,\delta r_1 - (cm_2/r_2^2)\,\delta r_2$. Using coordinates x, y, θ as in Example 4.3, show that

$$\delta W_{\text{total}} = -c\left[\frac{m_1}{r_1^3}(x - l_1\cos\theta) + \frac{m_2}{r_2^3}(x + l_2\cos\theta)\right]\delta x$$
$$- c\left[\frac{m_1}{r_1^3}(y - l_1\sin\theta) + \frac{m_2}{r_2^3}(y + l_2\sin\theta)\right]\delta y$$
$$- c\left[\frac{m_1}{r_1^3}(xl_1\sin\theta - yl_1\cos\theta) + \frac{m_2}{r_2^3}(yl_2\cos\theta - xl_2\sin\theta)\right]\delta\theta$$

from 'Shaum's Outline of Theory and Problems of Lagrangian Dynamics' D. A. Wells,(1967)

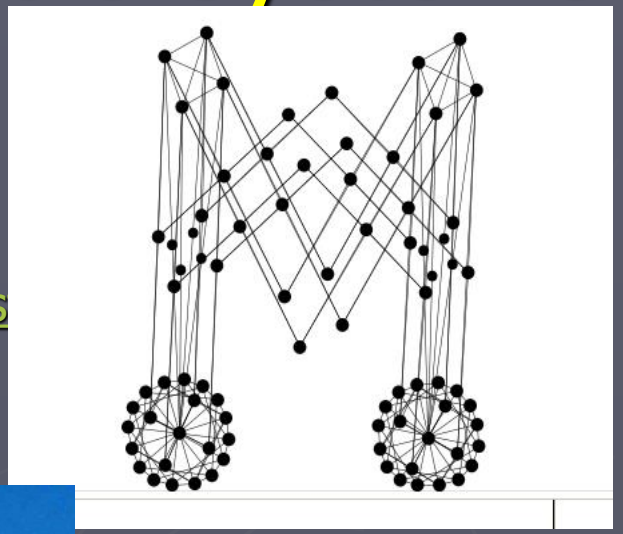
Easier: Mass-Aggregate Particle Systems

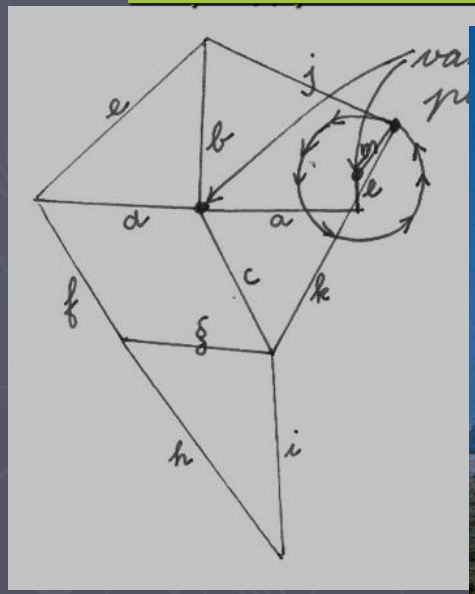
- Linked by **springs**, **constraints**, & **3D dynamics**
- Particles that interact with
  - Neighboring particles and barriers
  - Follow rules, behaviors and conditions (constraints)
  - Subject to sets of interdependent forces that vary unpredictably over time

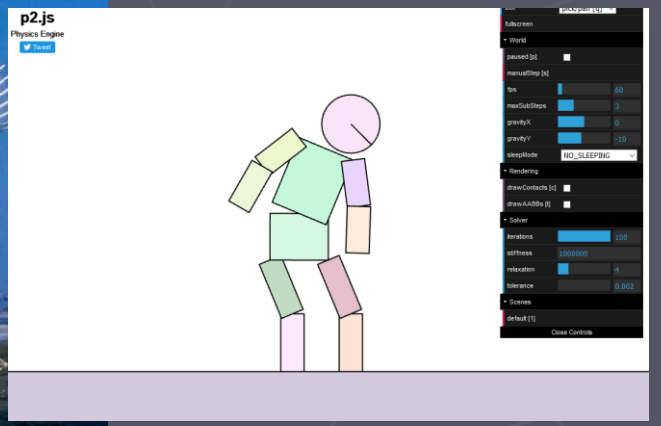
- Suitable for simulation of:
  - Flexible, High DOF objects:
    rope, cloth, Jell-o, fluids, gasses
  - Semi-Rigid, lower DOF objects (rubbery shapes)
  - object interactions + turbulence, and more…

4: Mass-Aggregate Particle Systems

- 2D constrained springs: (try double-pendulum) https://www.myphysicslab.com/
- Powered Rod/Spring walkers (revisited): http://maciejmatyka.blogspot.com/2018/02/soda-cons
- Theo Jansen's wind-driven 'Strandbeests' https://youtu.be/hHTuXe1rZrQ?t=167



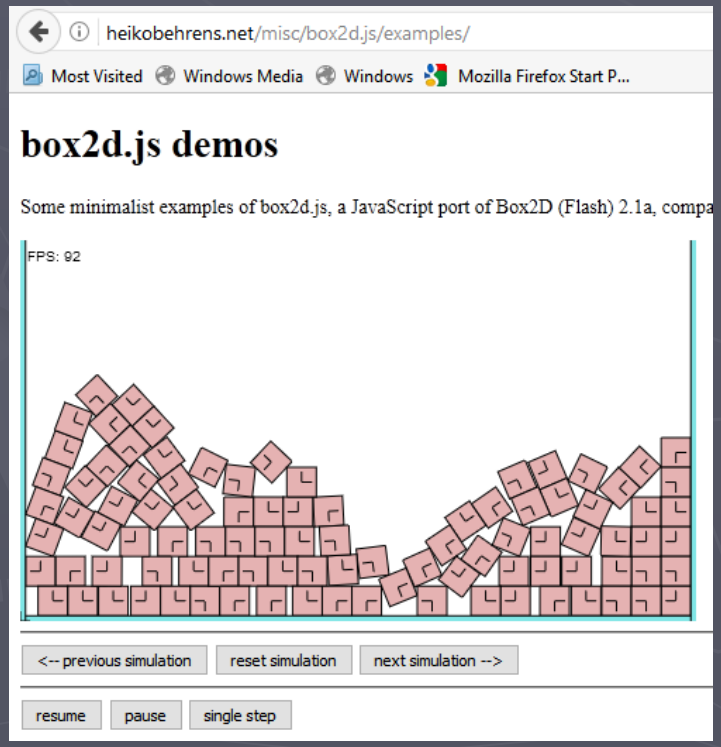




- "Ragdoll Physics" https://ragdollphysicsplayground.com/index.html (extends to rigid-body dynamics) https://playcanv.as/p/wxnL032C/ (try this one: https://schteppe.github.io/p2.js/demos/ragdoll.html

4:Mass-Aggregate Particle Systems

- Particles greatly **SIMPLIFY** Collision Detection, and enable stable solutions to stiff systems with Verlet Integration & others (we'll explain soon…)
- http://heikobehrens.net/misc/box2d.js/examples/
- http://kripken.github.io/box2d.js/demo/webgl/box2d.html



Fluid flows: (Advanced Topic: limited time)

Smoothed Particle Hydrodynamics (SPH)

- Volume-like Particles have mass and inertia (and 3D rotational inertia too!)
- Density, pressure, viscosity per particle
- Particles == centers of space-filling 'blobs'
  - Conserves momentum during collisions
  - Responds to gravitational forces
  - Responds to local changes or differences in velocity, pressure, temperature, etc.
- 2D demo: http://www.youtube.com/watch?v=2415K_0AL8o
  https://nerget.com/fluidSim/
- 3D demo: https://youtu.be/DhNt_A3k4B4?t=11s (http://david.li/fluid/)
- What about Heat transfer? and
- Surface tension? Tricky, but particles can do it!

Got it: Particles are Versatile!

But a ***SYSTEM***

for ***all kinds*** of particle systems?

(e.g. Unity 'Shuriken'; Unreal 'Cascade';

Blender Particles https://youtu.be/6p74pEH-k8s)

*HOW? *

HOW? Start with a Particle class.

One object→*all* attribs of one Particle

Ballistic elements:

- Position, Velocity,
- Acceleration, mass,
- Forces, Constraints,…

Non-Ballistic elements:

- Size, color, transparency
- Render Method, ID#,
- Age, lifetime, temperature,
- Charge, flux, magnetism,
- pressure, density, etc., etc., etc…

Then SOMEHOW…

Design a methodical, general system to:

- CREATE a suitably large **set** of particles,
- INITIALIZE all particles realistically, then:
- MOVE particles slightly, but realistically:
  - Find forces on each particle (from sets of force-makers')
  - Find correct new positions, velocities, mass, & all other attribs from old one(s)
- Suitably CONSTRAIN the new movements, to:
  - Avoid obstacles, stay inside containers, etc.
  - Collide/Bounce/Stick/Slide-along barriers, etc.
  - Satisfy or break particle-to-particle links (rods, threads, hinges, etc)
- DRAW particles to depict visually interesting phenomena

TRY IT! (1)

'Bouncy-Ball' Starter Code, Part 1:

1) Animate a 2-D 'screen saver' particle that:

- Appears on-screen at position (xpos,ypos)
- after each frame, moves by (xvel,yvel)
  xpos += xvel; ypos += yvel;
- 'bounce' off sides of the screen:

```
if(xpos>xmax && xvel > 0) xvel = -xvel;
if(xpos<0    && xvel < 0) xvel = -xvel;
if(ypos>ymax && yvel > 0) yvel = -yvel;
if(ypos<0    && yvel < 0) yvel = -yvel;
```

TRY IT! (1)

'Bouncy-Ball' Starter Code, Part 1:

1) Animate a 2-D 'screen saver' particle that:

- Appears on-screen at position (xpos,ypos)
- after each frame, moves by (xvel,yvel)
  xpos += xvel; ypos += yvel;
- 'bounce' off sides of the screen:

```
if(xpos>xmax && xvel > 0) xvel = -xvel;
if(xpos<0    && xvel < 0) xvel = -xvel;
if(ypos>ymax && yvel > 0) yvel = -yvel;
if(ypos<0    && yvel < 0) yvel = -yvel;
```

why test velocity like this?
we only want ONE sign reversal for any off-screen particle.
Decelerating particles may need >1 timestep to return on-screen!

TRY IT! (2)

'Bouncy-Ball' Starter Code, Part 2:

2) Add gravity & drag: for each re-drawing…

- apply downward acceleration to emulate gravity:
  yvel += -0.01;
- reduce velocity by a little bit to emulate drag:
  xvel *= 0.985; yvel *=0.985;

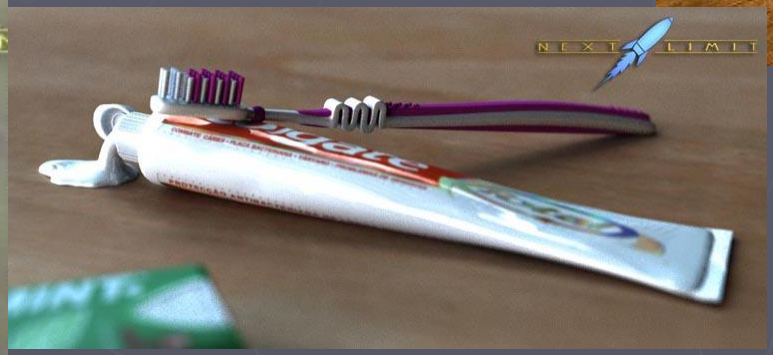
3) Add user controls that:

- Enable users to run/pause/ single-step (try p, space,…)
- Enable users to reset to init. velocity &position (R)
- Enable users to 'push' ball: add to xvel,yvel) (r)

END

END

Writing Code for Particle Systems--B



Jack Tumblin
COMP_SCI 351-2

OK! – Particle Systems!

- 1. Ballistic Living-Dying Particles: bouncy-balls…
- 2. 3D Varying Force Fields: Wind, Reeves Fire, etc.
- 3. Responsive, Interactive Particles:
  Boids & Flocking; user steering; obstacles
- 4. Coupled Particles: spring-mass; rope, cloth, Jell-O
- 5. Fluids Particles: water, smoke, SPH, etc

Cool!

But must we customize each & every one?

What do these systems have in common?

How can we unify *ALL* of these?

PREVIEW – Just 5 Big Parts

----------Description----------

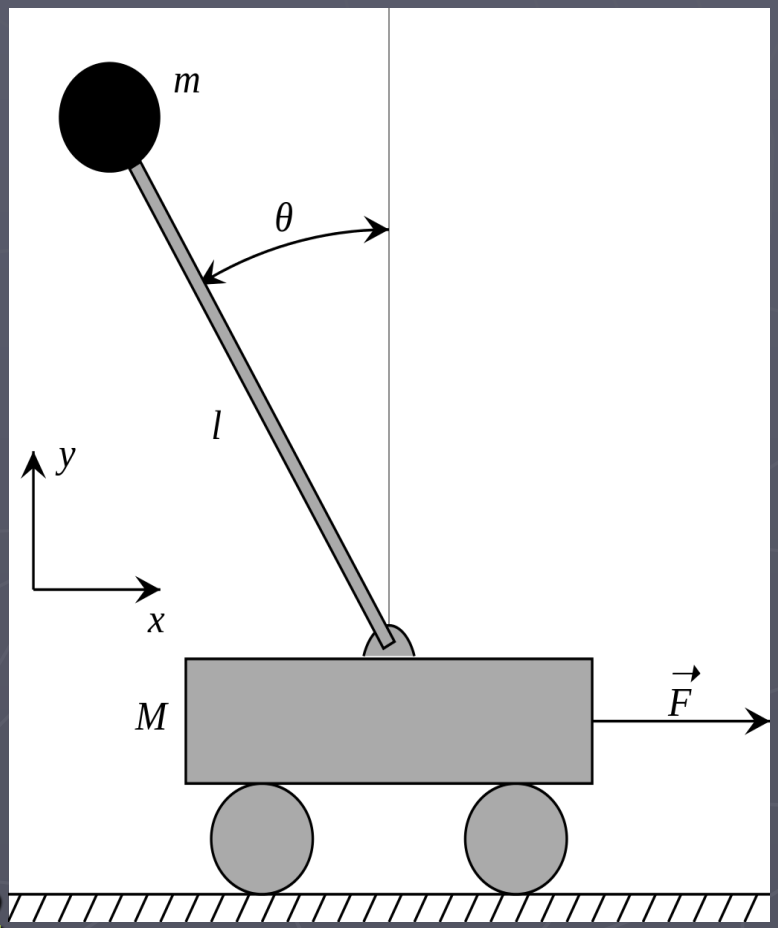
- State-space Vector(s):
  one array that holds all parameters for all particles for one (and only one) instant of time
  s1:now s2: future (just 1 timestep later)
- Force-Creating Object(s):
- Constraint Object(s):

----------Computing----------

- Solver(s):
- Constraint-Enforcer(s):

How Can we **UNIFY** this mess?

- GOOD ANSWER: **state space representation**
- Elegant mathematical model for automatic self-adjusting systems:
  - Stable, reliable governors, regulators, servo-mechanisms of all kinds:
  - Linear math for modern responsive robotics; Kalman filters, etc.



- Full Vector/Matrix Model of ANY 'controlled' physical system (!)
  EX1: 1-D inverted pendulum controller:
  https://youtu.be/XWhGjxdug0o?t=35
  EX2: 3-D cube moved by intertial rotors:
  'Cubli': https://youtu.be/n_6p-1J551Y?t=52

Fig. By Krishnavedala - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=20347385 (http://en.wikipedia.org/wiki/State_space_representation)

How Can we **UNIFY** this mess?

- GOOD ANSWER: **state space representation**

ESSENTIALS?

- Describe ***ALL internal params*** in just one 'state' vector:
  - *Instantaneous* values – (no 'lag', no smoothing)
  - Includes any & all useful time-derivatives, too! *e.g.*

$$\boldsymbol{x}(\mathrm{t}) == (\text{pos, vel, acc, mass, } \theta, \ldots)$$

- **Drive *ALL changes*** to state indirectly --
  by making changes to its time-derivative "**x-dot**":

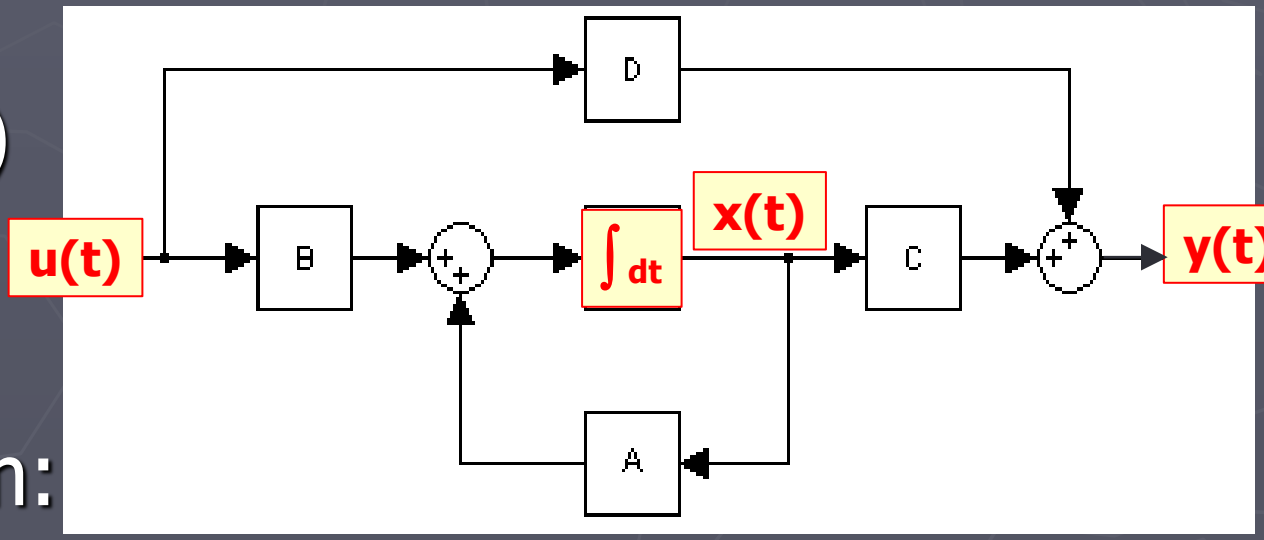
$$\dot{\boldsymbol{x}}(\mathrm{t}) == \frac{d\boldsymbol{x}(\boldsymbol{t})}{dt}$$

How Can we **UNIFY** this mess?

- GOOD ANSWER: **state space representation**
  == A *linear* model of any 'controlled' physical system as
  - a set of inputs(or 'controls'), that modify rate-of-change and output
  - all expressed by chains of **first-order differential equations.**
    (http://en.wikipedia.org/wiki/State_space_representation)

$\boldsymbol{x}(t)$== state(t)

$\dot{\boldsymbol{x}}(t) == \frac{d\boldsymbol{x}(t)}{dt}$



Canonical Form:

$$\dot{\boldsymbol{x}}(t) = [\boldsymbol{A}]\boldsymbol{x}(t) + [\boldsymbol{B}]\boldsymbol{u}(t)$$
$$\boldsymbol{y}(t) = [\boldsymbol{C}]\boldsymbol{x}(t) + [\boldsymbol{D}]\boldsymbol{u}(t)$$

Where **[A]**, **[B]**, **[C]**, **[D]**, are matrices (could be ***time-varying!***)

How Can we **UNIFY** this mess?

- LINEAR **state space representation**

$$\dot{\boldsymbol{x}}(t) = [\boldsymbol{A}]\boldsymbol{x}(t) + [\boldsymbol{B}]\boldsymbol{u}(t)$$
$$\boldsymbol{y}(t) = [\boldsymbol{C}]\boldsymbol{x}(t) + [\boldsymbol{D}]\boldsymbol{u}(t)$$

- Definitions:
  - ***Input:*** (or `External Controls & Influences')
    $\boldsymbol{u}(t)$ == forces, constraints, and all user inputs
  - ***State:*** all **internal** params (some might be hidden, 'un-observable')
    $\boldsymbol{x}(t)$== Params of ALL particles; mass, pos, vel, acc, …
    $\dot{\boldsymbol{x}}(t)$== $\frac{d\boldsymbol{x}(t)}{dt}$ == **time-derivative** of 'state' vector x
  - ***Output:*** (or 'Observables' of the system)
    $\boldsymbol{y}(t)$ == vector of all on-screen results, etc.

How Can we **UNIFY** this mess?

- LINEAR **state space representation**

$$\dot{x}(t) = [A]x(t) + [B]u(t)$$
$$y(t) = [C]x(t) + [D]u(t)$$

- Definitions:
  - ***Input:*** (or `External Controls & Influences')
    $u(t)$ == forces, constraints, and all user inputs
  - ***State:*** all **internal** params (some might be hidden, 'un-observable')
    $x(t)$== Params of ALL particles; mass, pos, vel, acc, …
    $\dot{x}(t)$== $\frac{dx(t)}{}$ == **time-derivative** of 'state' vector x
  - ***Output:***
    $y(t)$ == vector of all on-screen results, etc.

RARELY NEEDED FOR PARTICLE SYSTEMS:
y==displayed result; a *transformed* depiction of state **x(t)**

How Can we **UNIFY** this mess?

At first,

particle systems don't seem to fit this model…

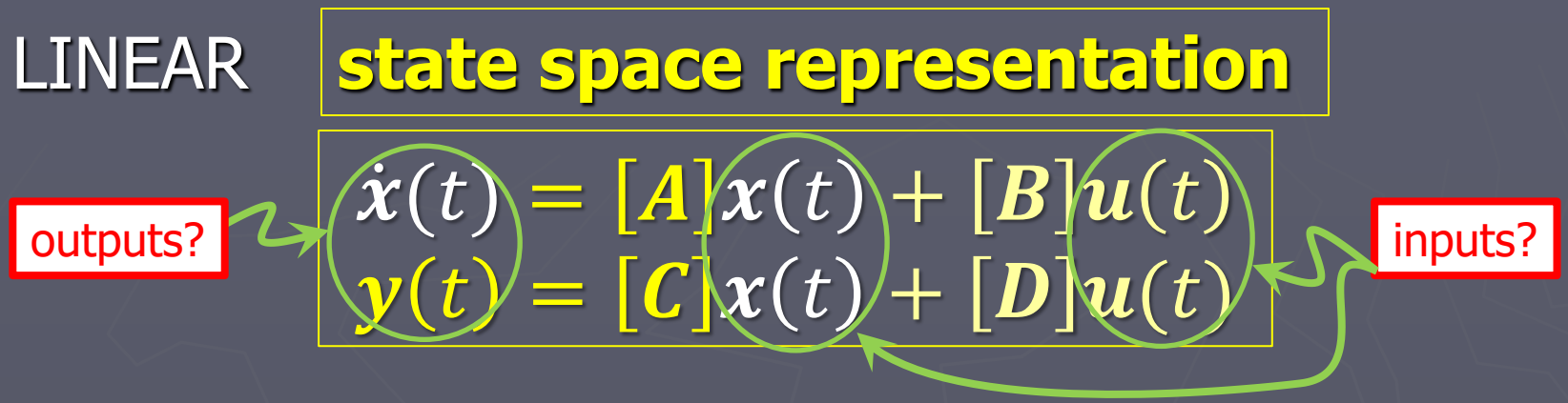
What's input, state, output for a particle sys?

everything can change on *every* timestep!

- Changeable particles, forces, constraints,
- Many time-varying attributes for each one, (mass, position, velocity, accel, force, age...)
- Continuous-time differential eqn. problem -- but solved in discrete-time steps? Really?

How Can we **UNIFY** this mess?

- LINEAR **state space representation**



- AHA! -- SIMPLIFY to a single procedure:
  - **Input:** u == forces, constraints + current state
  - **Output:** $\dot{x}(t)$ == time-derivative of current state
  - Rename state $x(t)$ and $\dot{x}(t)$ as **s** and **sdot**
  - Simplify, Generalize:
    use *ONE* 1st-order (but nonlinear) diff. equation:

    sdot = dotFinder(**s**, **u**)

What's Particle System 'State' **s** ?

ANSWER: One GIANT Vector of particles

That holds ***all*** changeable params that describe ***all*** particles
--all 'particles' consist of the **same** list of the **same** parameters

--all 'particles' keep all their own parameters grouped together

--all 'particles' keep all their parameters in the same order
(recall 'stride' and 'offset' in WebGL's **V**ertex **B**uffer **O**bjects)

--write state-vector software that permits easy parallel vector calculations for these state-vectors & repeated elements:
(mass, position, velocity, accel, color, lifetime, etc.)

ADVICE:

Remember, you're writing a SYSTEM of Particle **SYSTEMS**, not just one!

Make a 'particle system' object-type (prototype) **CPartSys**; then each particle system instance holds one giant state vector: variable '**s**'

What's Particle System 'input' **u** ?

ANSWER: One Collection of

Force Descriptors 'CForcer'

that hold ***all*** changeable params of ***all*** **causes** of motion

AND: One Collection of

Constraint Descriptors 'CLimit'

that hold ***all*** changeable params of ***all*** **limits** of motion

How can these input 'forces' and 'constraints' be applied to our system?

--A 'force' consists of a set of parameters *and* force-making rules

--For each 'force', visit each 'particle', one by one:

--compute the 'force vector' it applies to that particle

--find each particle's vector sum of all force vectors: $\mathbf{F}_{tot}$

--Newton's 2nd Law finds each particle's acceleration (F=mA), which we use to find current state's time-derivative **sdot**

--A 'constraint' limits how we find the NEXT state from **s** and **sdot**

What's Particle System 'Output'?

Valid (but slightly strange) 'System' Definition:

- **Inputs:** Time, All forces F, All constraints C for *all* particles
- **State Vector 's':** all changeable params of *all* particles stacked into one gigantic high-dimensional column vector(mass,position,velocity,accel,color,lifetime, etc.)
- **Output 'sdot':** 1st time-derivative of state: **sdot** = d**s**/dt
- Just *one* differential equation, done as one procedure:

sdot = dotFinder(**s**, **F**(**s**), **C**(**s**))

- Time-derivative of state vector,
- Current state vector,
- List of all forces (depends on state & user inputs),
- List of all constraints (depends on state & user inputs).

No,No,No - What's the ***OUTPUT***?

Valid (but slightly strange) 'System' Definition:

- **Inputs:** Time, All forces F, All constraints C for *all* particles
- **State Vector 's':** all changeable params of *all* particles stacked into one gigantic high-dimensional column vector(mass,position,velocity,accel,color,lifetime, etc.)
- **Output 'sdot'**: 1st time-derivative of state: **sdot** = d**s**/dt
- Just *one* differential equation, done as one procedure:

sdot = dotFinder(**s**, **F**(**s**), **C**(**s**))

- Time-derivative of state vector,
- Current state vector,
- List of all forces (depends on state & user inputs),
- List of all constraints (depends on state & user inputs.

Acch!!**NoooOO!!** what's the *Next* State!?

How can I change current state **s1**
into next state **s2**?!?!?

Answer: solve an ugly 1st-order differential equation for a slightly-later time, advanced by △t

sdot = dotFinder(**s**, **F**(**s**), **C**(**s**));

We have state s(t0) and DERIVATIVE of state s(t);
from these, solve for state(t+ △t) somehow:

$$s(t) = \int_0^t dotFinder(s(t), F(s(t)), C(s(t))dt \qquad \Delta t \equiv h$$

Uh oh. How could we *ever* do that? → approximate!

Acch!!NoooOO!! what's the *Next State!?*

$$s1 \equiv s(t) = \int_0^t dotMaker(s(t), F(s(t)), C(s(t))dt$$

$$\Delta t \equiv h$$

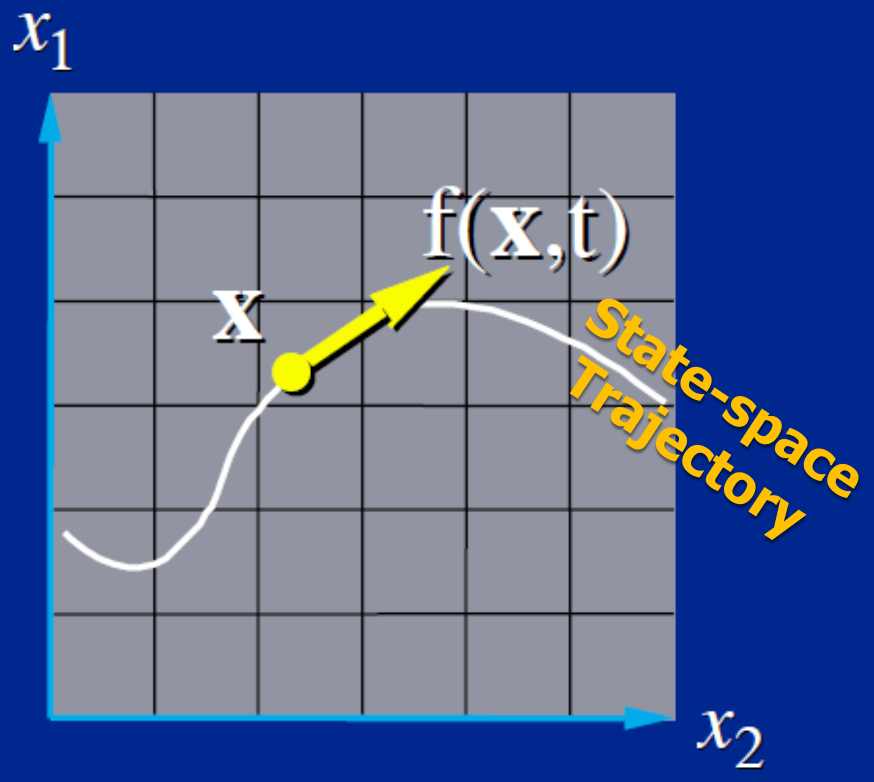
$$s2 \equiv s(t+h) = s(t) + \int_t^{t+h} dotMaker(s(t), F(s(t)), C(s(t))dt$$

We must somehow **integrate** s1dot (!?!)

- Recall infinitesimal time-step *dt* from calculus:
- Assume **s**(t) is smooth, differentiable (it will be)
- In the limit as *dt*→0, calculus guarantees that:

  s(t + dt) = s(t) + sdot*dt

A Canonical Differential Equation



$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

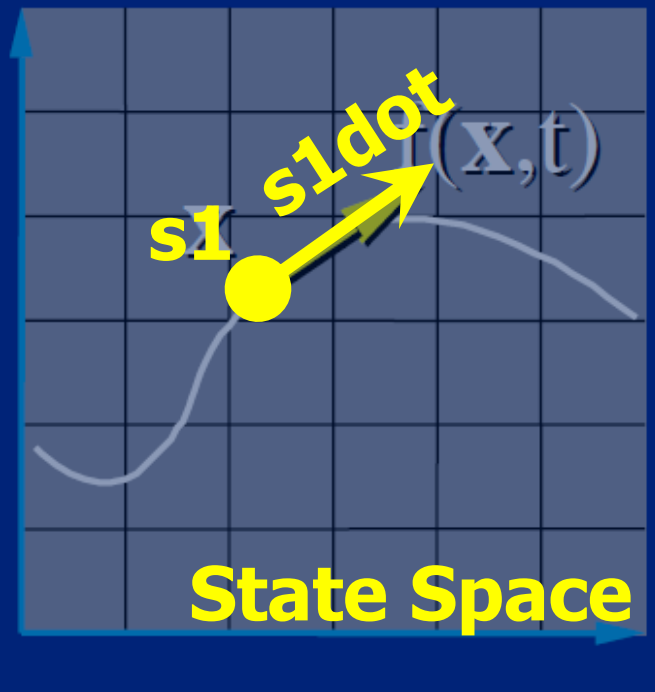
- $\mathbf{x}(t)$: a moving point.
- $\mathbf{f}(\mathbf{x},t)$: x's velocity.

From Witkin/Baraff/Pixar 2001 Particles Tutorial Notes

EXAMPLE: Moon orbiting the earth. Position vector x? time-deriv xDot?

A Canonical Differential Equation

integrate s1dot to trace out the 'trajectory' of s1 in state space



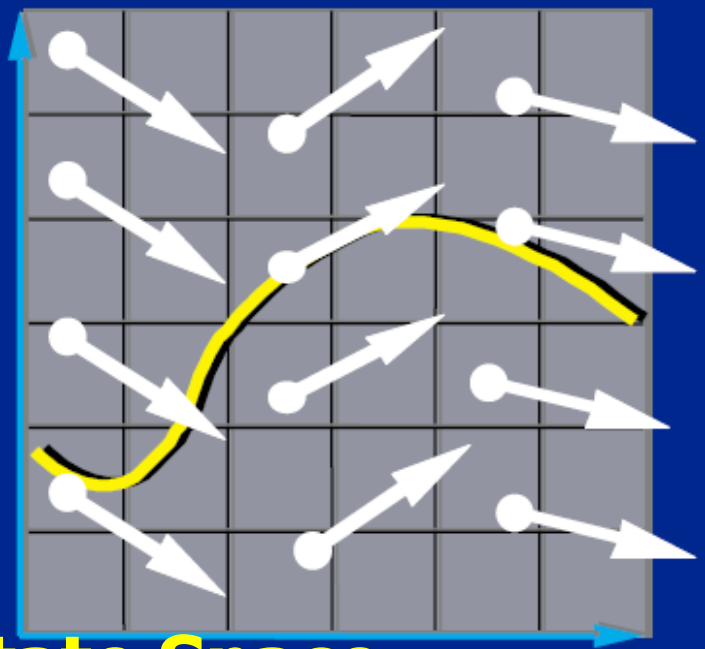
$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

- $\mathbf{x}(t)$: a moving point.
- $\mathbf{f}(\mathbf{x},t)$: x's velocity.

From Witkin/Baraff/Pixar 2001 Particles Tutorial Notes

EXAMPLE: Moon orbiting the earth. Position vector x? time-deriv xDot?

Vector Field



State Space

The differential equation

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

defines a vector field over x.

From Witkin/Baraff/Pixar 2001 Particles Tutorial Notes

Vector Field

State Space

The differential equation

sdot = dotFinder(t,**s**,**F**,**C**)

defines a vector field over x.

"From any given state **s1**, trace along a trajectory through state-space
In the direction given by **s1dot**"

From Witkin/Baraff/Pixar 2001 Particles Tutorial Notes

A '**solver()**' finds next state **s2**

sdot = dotFinder(t, **s**, **F**(**s**), **C**(**s**))

solver(Δt, s, F(s), C(s)) tries find next state of **s**

- Continuous-time differential eqn. problem, but solved in discrete-time steps: How?

- **A Simple solver():**
  Let s1 == current state
  s2 == next state
  h == 'time-step'

- **Euler Integration:** **s2** = **s1** + **s1dot***h

Theory + Practice ('Thractice'?)

Particle System:

- Current State **s1**, derivative **s1dot**, & Next State **s2**

  Each one is a *set* of Particle Objects. To drive them we need:

- A *set* of Force-Applying Objects
- A *set* of Constraint-Applying Objects
- DotFinder():given state $\boldsymbol{s}$, find time deriv $\dot{\boldsymbol{s}}$ or '**sdot**'
- Solver() find next state s2 from s1 and s1dot
- doConstraints(); adjust s1 and s2 to satisfy rules of collisions, friction, sliding, etc.
- Render() : depict new state s2 on-screen

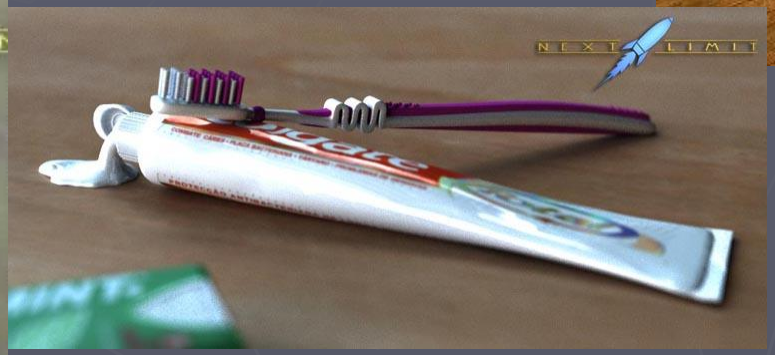
  (NOTE: use s1 & s2 together to draw blur, streaks, etc.)

- Swap: s1 ⬅ s2

END

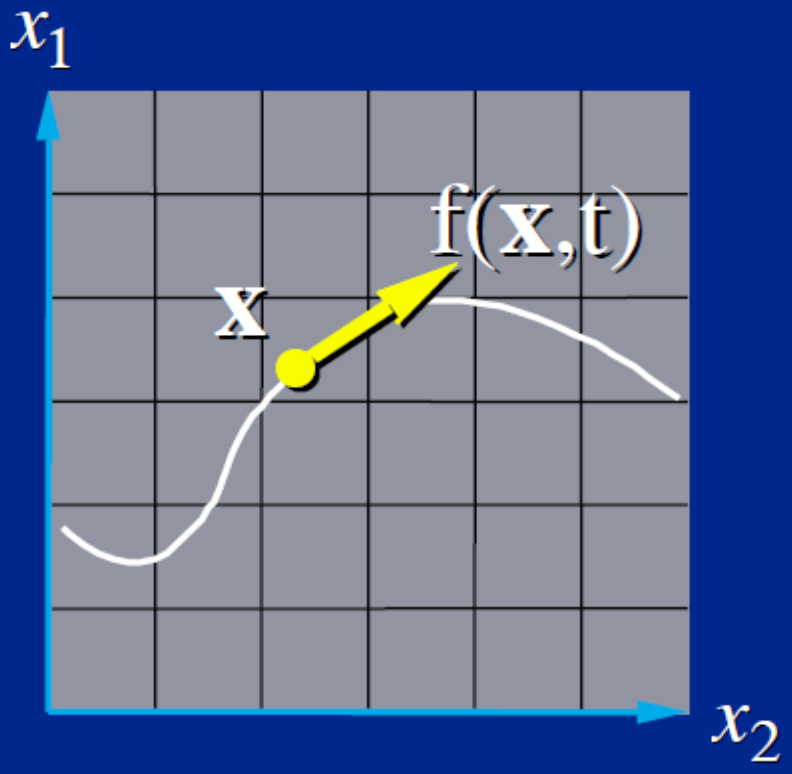
END

Structure of Unified Particle Systems--C



Jack Tumblin
COMP_SCI 351-2

A Canonical Differential Equation



$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

- $\mathbf{x}(t)$: a moving point.
- $\mathbf{f}(\mathbf{x},t)$: x's velocity.

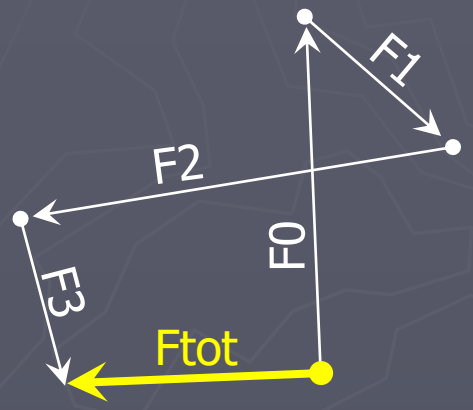
From Witkin/Baraff/Pixar 2001 Particles Tutorial Notes

Key Idea: **Forces** Drive Particles

- Combine all Forces by simple vector addition:

  $Ftot = F_0 + F_1 + F_2 + \ldots + F_n$

- Force Total Ftot causes
  all movements, all changes:



- Newton's laws convert particle's Force Total to acceleration – just rewrite 'F = mA' as
  acc := Ftot * (1/mass)

Key Idea: Force == Cause
Movement == Effect

- **IDEA 1:** define separate 'force-creating' objects
  Each applies its own forces to its selected particles
  - Gravity object: applies downward forces to ***all*** particles
  - Wind-Field object: applies position-dep. forces to ***all…***
    Spring object: applies forces to ***2*** picked particles
  - Charge object: applies force TO ***all*** particles
    FROM ***one*** particle
  - 'Flocking' object: applies force TO ***one*** particle
    FROM '***nearby***' particles
  - Others you design: planetary, magnet, repulsion, etc.
- Make your 'force object' class VERY FLEXIBLE;
  just one class that can describe every possible kind of force!
  (e.g. we'll make a collection: a gravity object, 6 spring objects, 7 charge objects, 5 flocking objects, etc.)

'CForcer' Ideas (1) : F(x,y,z)

- **Simple Gravity?** **F**(x,y,z) = (0,0,-mass*g_0)
  g_0 == gravity at planet's surface (earth: -9.8 m/s^2)
  System 1? Needs more: it's mass-dependent,
  but it's not a 'position dependent' force-field

- **Wind-Sink/Source?** strength '**a**', aimed at origin.
  (**N** == unit-length vector in x,y,z direction;
  ||**F**||==a; **N**==(x,y,z) / sqrt(x^2 +y^2 +z^2);
  F = a**N** System 1? Yes: 'position-dependent'
  What gives endless motion? Tornado? Firehose?

- **Planetary Gravity?** ||F|| = mg /$dist^2$ refined:
  ||**F**|| = f == 2·m·g_0 / (r_0^2 + x^2 + y^2 + z^2);
  F = f**N** System 1? Yes: 'orbital movement'
  --m is mass, and
  --planet radius r_0 term prevents divide-by-zero.

'CForcer' Ideas (2) : F(x,y,z)

- **Air Drag?** Rayleigh's elastic fluid model (google it!)

  $\mathbf{F} = -\rho_d \cdot \mathbf{V} \cdot ||\mathbf{V}|| \cdot A \cdot K_d$ where:

  ρ_d == fluid density (kg/m^3)

  $\mathbf{V} \cdot ||\mathbf{V}||$ == velocity vector with squared strength

  A == Cross-sectional area of particle (m^2)

  K_d== Drag coefficient (unitless)

- **'Traveling Waves' Force?**

  $\mathbf{F}(x,y,z) = (0, 0, A*\sin(2\pi(Bx + Ct))$ where:

  A== Wave half-height

  $1/B$== Wave width in x direction

  C== Wave speed(cycles per unit time t)

- **'Tornado' Forces?**

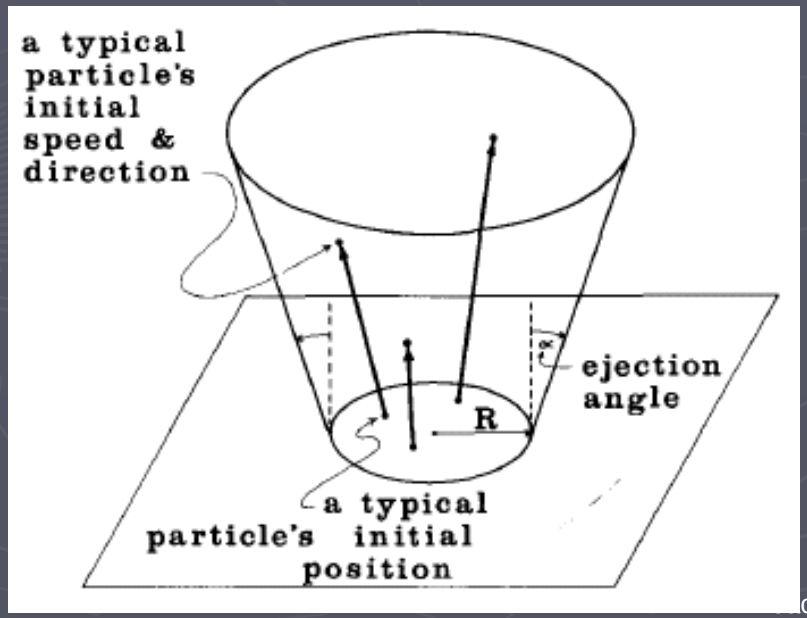
  YOU design it!

‘Force Field’ Ideas (3) : F(x,y,z)

- Forces along **Magnetic Flux** Lines?
- Forces due to **Charge**?
- Fanciful ‘**Invisible Planet**’ force-field?
  - spherical region at (movable) center position (x_0, y_0, z_0),
  - radius (r_0) defines sphere around the surface
  - Zero force at sphere surface:
  - Force **vector** always radial to/from sphere center, but
  - Force **magnitude** always proportional to distance from sphere surface;
  - ‘**Invisible Bubble**’ force-field:
    --Inside the sphere: zero force.
    --Outside the sphere: attractive force; towards center
  - ‘**Forbidden Zone**’ force-field:
    --Inside the sphere: repelling force; away from center
    --Outside the sphere: zero force.

‘Force Field’ Ideas (4) : F(x,y,z)

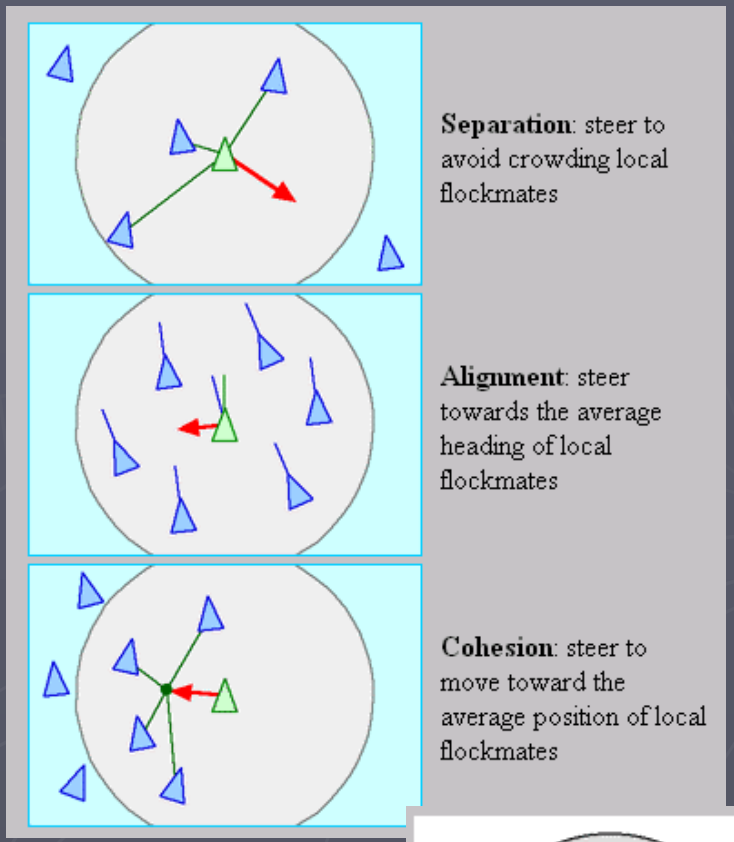
- **‘Repeller’, ‘Attractor’ or ‘Firehose’ force-field**? Construct it from others:
  --’sink’ force-field
  attracts all particles towards one point
  --place Reeve-like force-field that forms a narrow strong-force ‘spray’ of particles near that point:

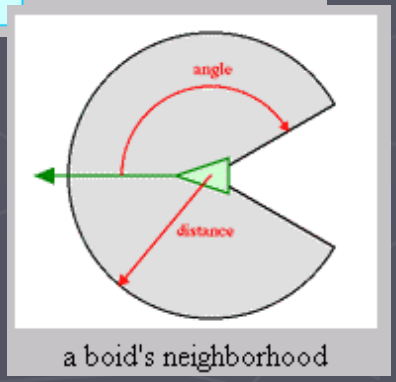


‘Force Field’ Ideas (5) : F(x,y,z)

‘Boids’ Behaviors:

- Avoid hitting nearby neighbors!
- Point in ~ same direction as neighbors!
- Move toward average position of neighbors!
- Avoid hitting obstacles!
- Details? Read the paper…





Real+Sim: https://youtu.be/4LWmRuB-uNU?t=75 and at 3:07 (BOIDs intro)
SIGGRAPH 1987, Craig Reynolds http://www.red3d.com/cwr/boids/
Boids and the development of ‘Artificial Life’

'Constraints'

Key idea: Given **s1** AND **s2**, repair **s2**

A 'correction' to the next state **s2**

that applies all discontinuities we missed by just extrapolating from state **s1**.

- Solid Barriers: 'bounce'-- what, when, and how?
- Distance Limits: particle 'shell' radius, particles on fixed-length ropes, rigid rods, plates, solids…
- Joint Limits: pivots, sliding joints stay in-range
- Destruction: over-stretched or broken springs, expired particles, fractured shapes, etc.

'Constraint' Ideas (1)

- Axis-Aligned Rectangular Volume (IN / OUT)
- Spherical Volume (particles kept IN or OUT)
- Planar, rectangular 'Plates'
  of any size and orientation;
  with/without circular or rectangular 'Holes'
- fixed-length cables/Ropes, fixed-length Rods,
- Selected particle(s) positions in fixed-size
  1-D 'Slots' or 2D 'Apertures' or 3D 'Berths'
- Rigid-Object Aggregates:
  Assembled Spheres, Boxes, Cylinders, etc.

Good idea: Create "Container" Objects ∀Particles, ∀Forces, ∀Constraints

- **Idea 3:** Just as we created particle objects (CPart) all kept in a particle-collection object type (CState), we will create force-causing objects (CForcer) all kept in a force-object array (forceList[]), and create constraint-causing objects (CLimit) all kept in a constraint-object array (limitList[]).

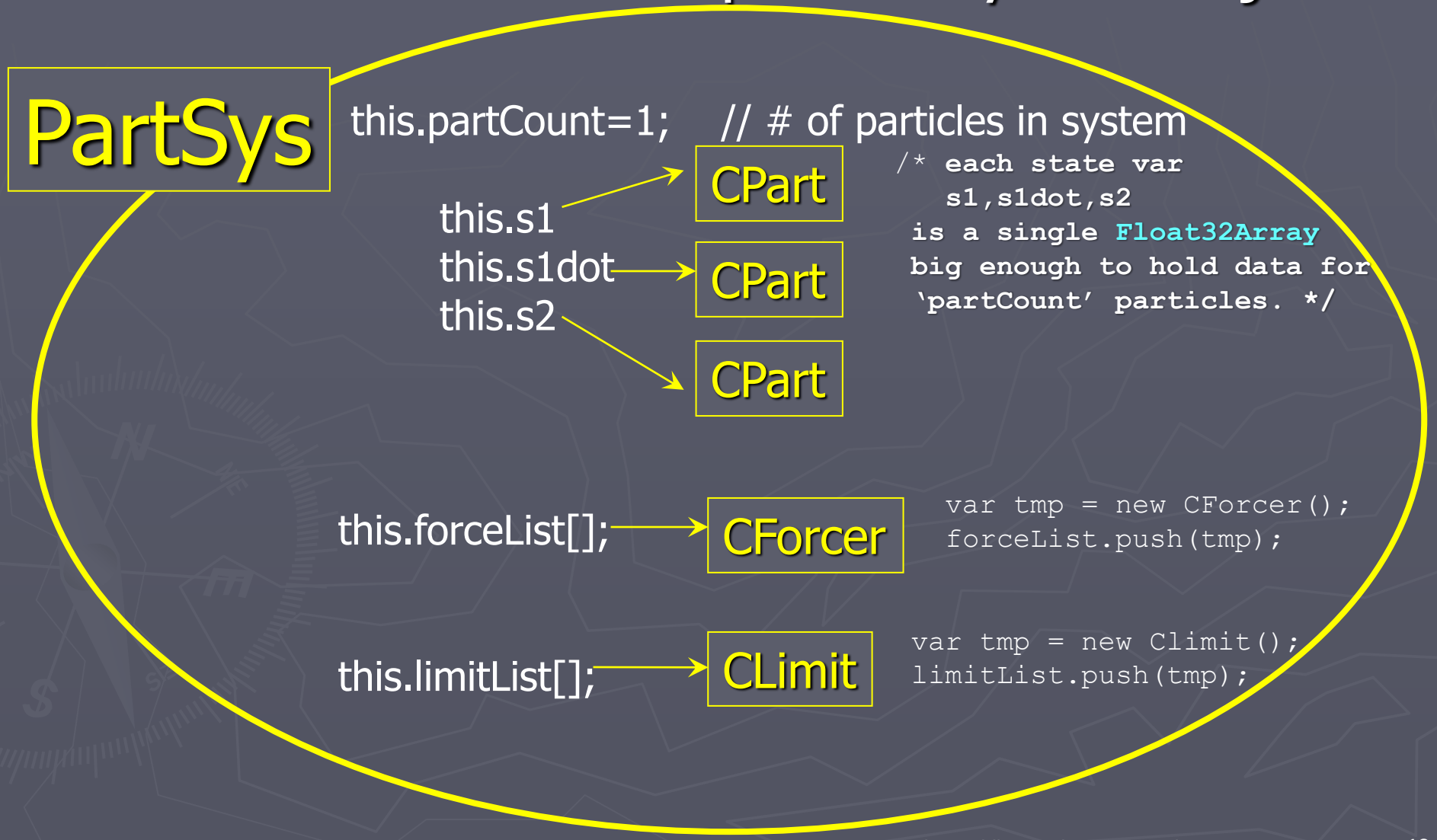
ALL within a particle-system class (CPartSys)

- **Idea 3a:** Refer to each collection object by **REFERENCES** to enable easy swapping: in C/C++? Pointers! In Javascript? Be careful – tricky!

```
CState  *pS1,*pS1dot,*pS2;// ptr-to-state vars
CForcer *pF1;             // ptr-to-all-force-objects
CLimit  *pC1;             // ptr-to-all-constraint-objects
```

Particle System: Master Plan

- Essential data in one particle system object:



'Ball-in-a-Box' PartSys (1)

- One simple particle:
  (state variable **s1** can hold many of these)

CPart

```
/* Instead of an array of individual 'CPart' objects,
   we store the i-th particle in each state-variable 's' (a Float32Array)
   in one contiguous span of array indices defined by consts:  */
var j = i*PART_MAXVAR;          // j==array index where particle 'i' begins.
 // position                    // velocity                    // force accum., …
 S[j+PART_XPOS]=0;      S[j+PART_XVEL]=1;      S[j+PART_X_FTOT]=0;
 S[j+PART_YPOS]=0;      S[j+PART_YVEL]=1;      S[j+PART_Y_FTOT]=0;
 S[j+PART_ZPOS]=0;      S[j+PART_ZVEL]=1;      S[j+PART_Z_FTOT]=0;
          S[j+PART_MASS]=1.0;    // mass (or 1/mass)
```

```
const PART_XPOS = 0;        const PART_XVEL = 3;        const PART_X_FTOT = 6;
const PART_YPOS = 1;        const PART_YVEL = 4;        const PART_Y_FTOT = 7;
const PART_ZPOS = 2;        const PART_ZVEL = 5;        const PART_Z_FTOT = 8;
        const PART_MASS = 9;      … dotSize, age, R,G,B color…   const PART_MAXVAR = 10;
```

'Ball-in-a-Box' PartSys (2)

- One single force-causing object:
  (ForceList[] array can hold many CForcers)

CForcer

```
this.forceType = F_GRAV_E;      // kind of force implemented by this object

// all params needed by all possible force causes (use onlyl the ones you need)
this.grav_e = 9.832;                    // earth's gravity g: (causes force m*g)
this.grav_p = 1.0;                      // planetary gravitational constant
this.downDir =new Vec4(0,0,-1,1);       // earth's 'down' direction
this.Ks = 0.5; this.len_s = 1.0;        // spring constant; spring rest-length
this.e0; this.e1; this.K_drag = 0.15;   // part#, viscous drag (vel*m*K_drag)
...
```

```
const F_NONE     =0;   // disabled; inactive force-making object;
const F_GRAV_E   =1;   // global earth gravity: force is –grav_e*mass in 'downDir' direction
const F_GRAV_P   =2;   // planetary gravity due to mass of particle e0
const F_SPRING   =3;   // applies spring force between particle e0 and e1
const F_MAXVAR =4;     // number of types of force-causing objects possible
```

'Ball-in-a-Box' PartSys (3)

- One simple constraint-causing object: (LimitList[] array may hold many of these)

CLimit

```
this.limitType = LIM_VOL;          // kind of constraint implemented by this object

// all params needed by all possible constraints (use as needed)
this.K_resti=0.85;                 // coeff. of restitution for 'bounce' on a limit surface
this.xmin = -1;   this.xmax = +1;  // axis-aligned rectangular box
this.ymin = -1;   this.ymax = +1;
this.zmin = -1;   this.zmax = +1;
this.px=0; this.py=0; this.pz=0; this.radius=1;
```

```
const LIM_NONE     = 0;   // disabled; inactive constraint-causing object;
const LIM_VOL      = 1;   // Axis-aligned rectangular volume.
const LIM_BALL_OUT = 2;   // spherical solid barrier centered at px,py,pz; radius
const LIM_MAXVAR = 3;    // number of types of limit-causing objects possible
```

Simple* 'Full-Feature' Particle System

*(as simple as I can make it)

- CPartSys object: HOLDS EVERYTHING: one complete particle system!

Members:

- State vectors s1, s1dot, s2, etc.
  Be sure you can 'swap' them: s1 ⟲ s2 (see starter code)
  Each state is one big Float32Array; holds sets of CPart.
  - CPart: set of Float32Array elements for one *particle*.
- Force List: One big array (?Float32Array?);
  to hold an indexed set of Cforcer Objects.
  - Cforcer: a set of elements to hold one force-applying object.
- Limit List: One big array (?Float32Array?);
  to hold an indexed set of CLimit Objects.
  - CLimit: a set of elements to hold one constraint-imposing object.

Theory + Practice ('Thractice'?)

- Create, Initialize Particle System:
  - Current State s1, derivative s1dot, & Next State s2

    Each one is a *set* of Particle Objects. To drive them we need:
  - A *set* of Force-Applying Objects
  - A *set* of Constraint-Applying Objects
- DotFinder(): given state s, find time deriv $\dot{s}$ or 'sdot'
- Solver(): find next state s2 from s1 and s1dot (Euler)
- doConstraints(); adjust s1 and s2 to satisfy rules of collisions, friction, sliding, etc.
- Render() : depict new state s2 on-screen

  (NOTE: use s1 & s2 together to draw blur, streaks, etc.)
- Swap: s1 ← s2

PartSys Program:

- INIT State Vectors, Forcer Sets, Limit Sets
  - State Vector s1 describes the (array of) ball(s) at the current time t; s2 is 'next' state at next time instant t+h.
  - State Vector s1dot is the s1 time-derivative; the time-rate-of-change for s1 during an *infinitesimal* time (dt). At (t+dt), state s=s1 + dt*s1dot
  - Forcer Set describes the (array of) force-applying object(s) (e.g. gravity object, wind object, spring) applied at time instant t.
  - Limit Set describes (array of) constraint-applying object(s) (e.g. walls, ropes, hinges, rods) enforced at current time t.
- ApplyForces(): use Forcer Set to find net forces Ftot on each particle in s1. (Store Ftot in each particle obj)
- dotFinder(): find s1dot by applying Newton's laws to Ftot
- Solver(): find next state (Euler/Explicit: s2 = s1+ h*s1dot)
- doConstraint(): apply LimitSet to s2: 'bounce' off walls, etc
- Render(): depict s2 (& maybe Forcers & Limits) on-screen
- Swap(); Transfer contents of state vector s1 ← s2.

Why doesn't **s1** *contain the* **forcers** and **limits**?

- Current 'state' s1 – holds all current particles
- Current 'motive set' f1 – holds all current forcers
- Current 'limit set' c1 – holds all current constraints

Wait wait wait – wasn't the current 'state' variable s1 meant to describe ***ALL*** changeable system elements?

Well, yes but … think of state as **only the particles**, and f1,c1 as the holder for all its influences

BAD IDEA! **Because** 1) they won't easily allow ***changeable*** sets of force-applying objects and constraint objects.

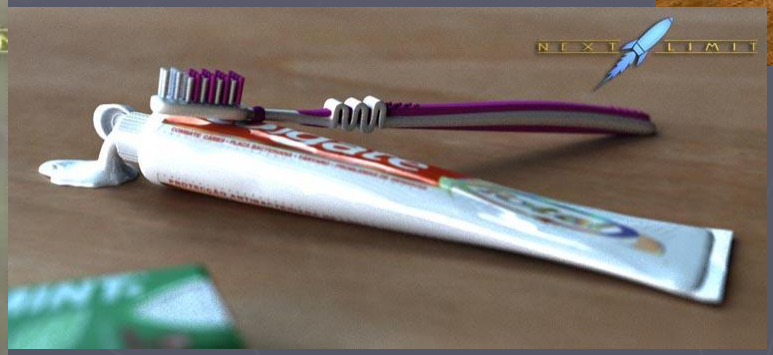
(e.g. boids that weaken with age; fabric that can tear; springs that get 'sprung' if stretched too far)

and 2) if 'state' s1 holds all of them, they get tangled up inside the solver; we'll need well-defined F2, C2, F2dot and C2dot?!?! Must we write new 'solvers' for them too ?!? !YIKES!

END

END

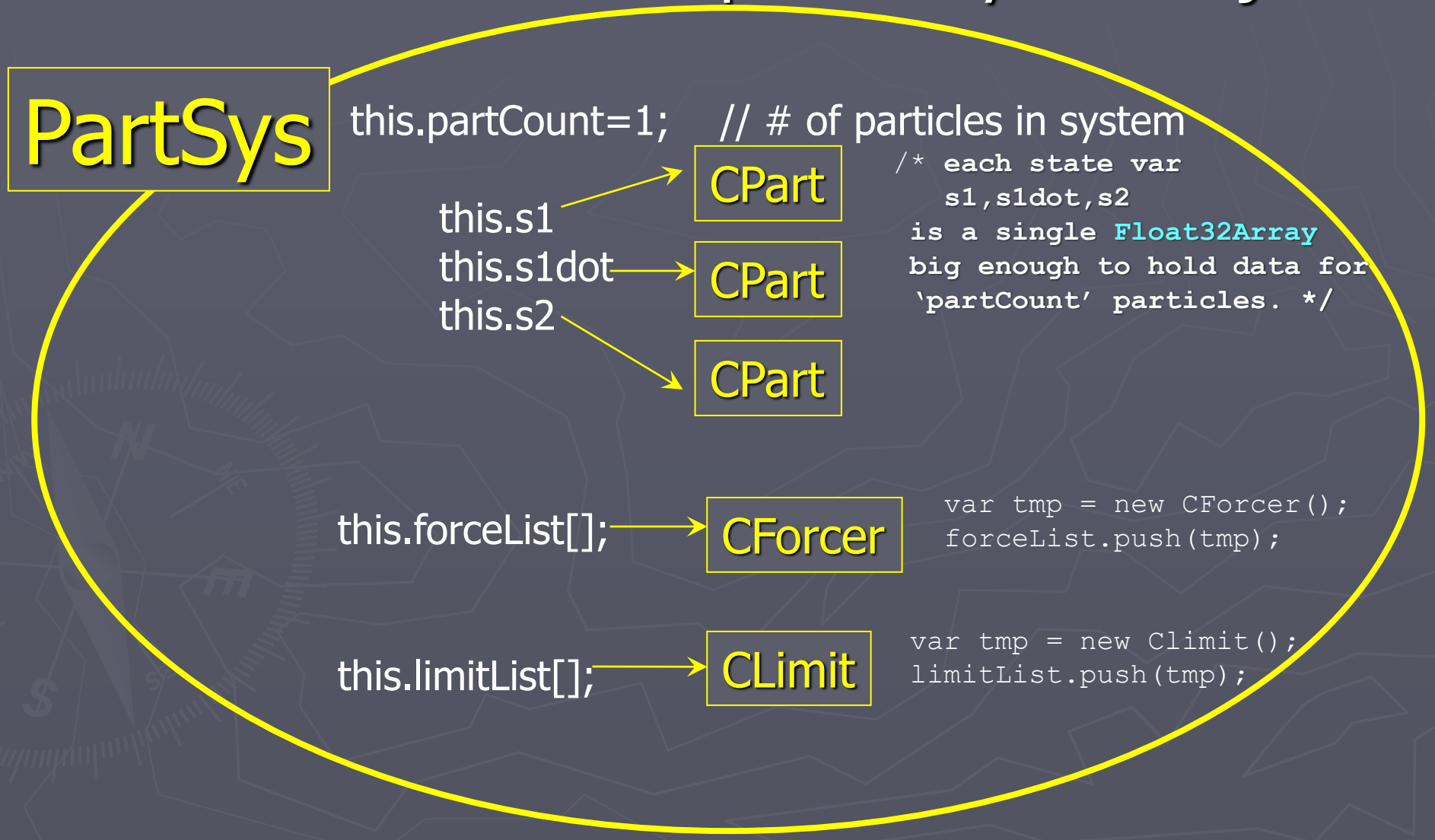
Extending Code for Particle Systems—D



Jack Tumblin
COMP_SCI 351-2

Recall: Master Plan

- Essential data in one particle system object:



Recall: PartSys Program

- INIT State Vectors, Forcer Sets, Limit Sets
  - State Vector s1 describes the (array of) ball(s) at the current time t; s2 is 'next' state at next time instant t+h.
  - State Vector s1dot is the s1 time-derivative; the time-rate-of-change for s1 during an *infinitesimal* time (dt). At (t+dt), state s=s1 + dt*s1dot
  - Forcer Set describes the (array of) force-applying object(s) (e.g. gravity object, wind object, spring) applied at time instant t.
  - Limit Set describes (array of) constraint-applying object(s) (e.g. walls, ropes, hinges, rods) enforced at current time t.
- ApplyForces(): use Forcer Set to find net forces Ftot on each particle in s1. (Store Ftot in each particle obj)
- dotFinder(): find s1dot by applying Newton's laws to Ftot
- Solver(): find next state (Euler/Explicit: s2 = s1+ h*s1dot)
- doConstraint(): apply LimitSet to s2: 'bounce' off walls, etc
- Render(): depict s2 (& maybe Forcers & Limits) on-screen
- Swap(); Transfer contents of state vector s1 ← s2.

Newtonian or 'Classical' Physics

- Newton's Laws of Motion:
  - 1st Law: Only force can change the velocity

    "I. Every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it."
  - 2nd Law: Force sets velocity-change rate: F = (d/dt)(mv)
    and ***if mass is constant:*** F = ma

    "II. The relationship between an object's mass **m**, its acceleration **a**, and the applied force **F** is **F = ma**. Acceleration and force are vectors. In this law, the direction of the force vector is the same as the direction of the acceleration vector."

    (MAJOR IMPROVEMENT– replaced Aristotle's claim that F=mv)
  - 3rd Law: Apply all forces bi-directionally:

    "III. For every action there is an equal and opposite reaction."

Physics in Particle Systems

- Newton's Laws of Motion:
  - **1st Law:** Only force can change the velocity
    - →→ Force-applying objects **cause** **all** state changes over time.
  - **2nd Law:** Force sets velocity change rate: F = (d/dt)(mv)
    and ***if mass is constant:*** F = ma
    - →→ Tells us how to write dotFinder()…
    - Aristotle: "F=mv": → dotFinder() could offer this too…
  - **3rd Law:** Apply all forces bi-directionally:
    - ***All*** force-applying objects create force ***pairs*** !

    Not just springs—be sure to account for *all* force pairs!
    (e.g. 'Earth Gravity' ignores force applied to Earth…)

Physics, Notation, and Calculus

- **2nd Law:** F = (d/dt)(mv) (and *not* F=ma)!
  Recall that in vector notation:
  - Position == p p
  - velocity == v = (dp/dt) or just write v = p′
  - acceleration == a = (dv/dt) = (d^2p/dt^2)or a = v′ = p″
- Recall from calculus:

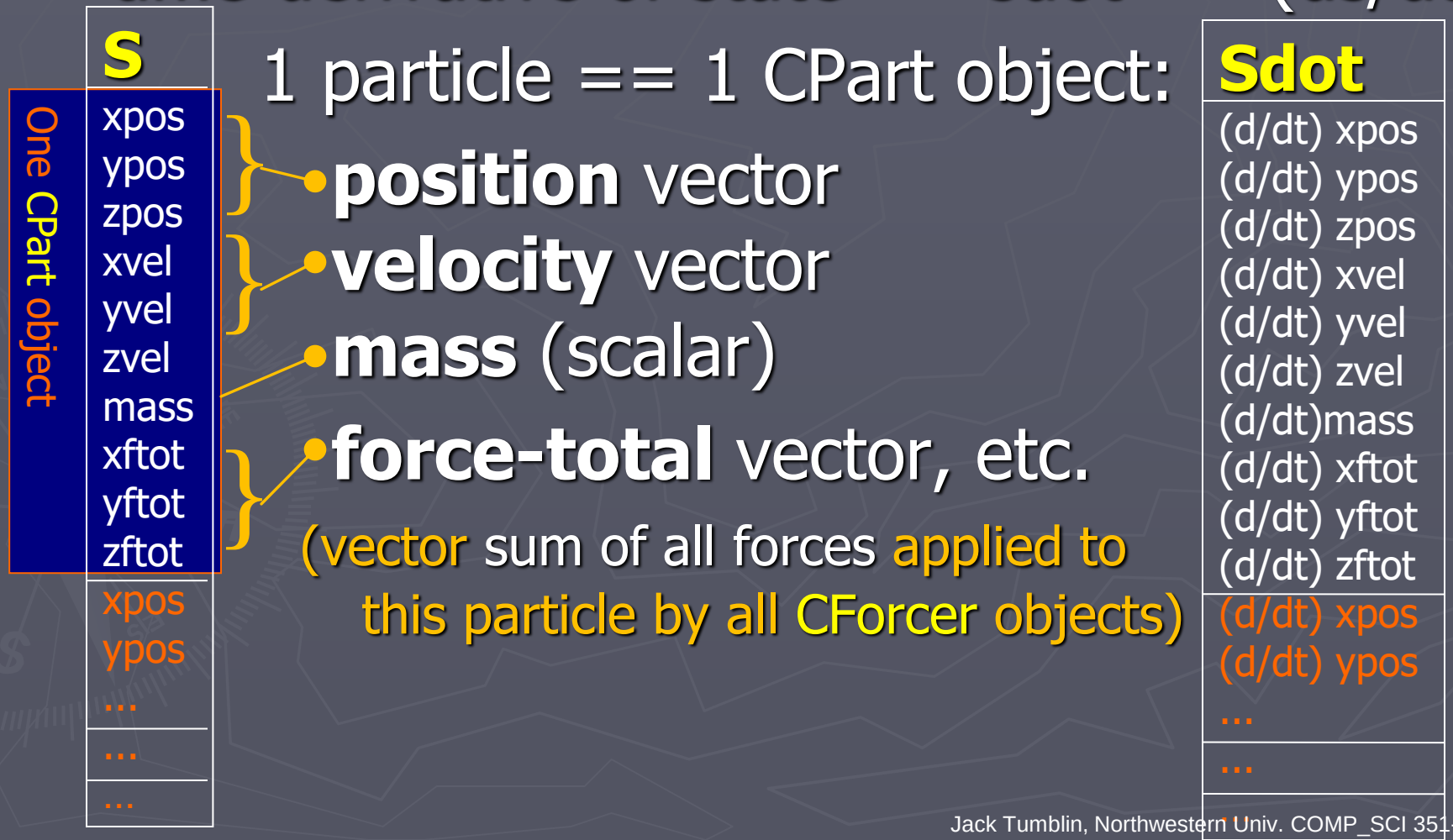
  (uv)′ = uv′ + u′v, so
  - 2nd Law: F = (m v)′ = mv′ + m′v by chain rule;

    = m(dv/dt) + (dm/dt)v

    F = ma + (dm/dt)v
  - Constant mass → (dm/dt)=0; reduces to F=ma.
  - (What happens in Aristotlean particle physics: F=mv?)

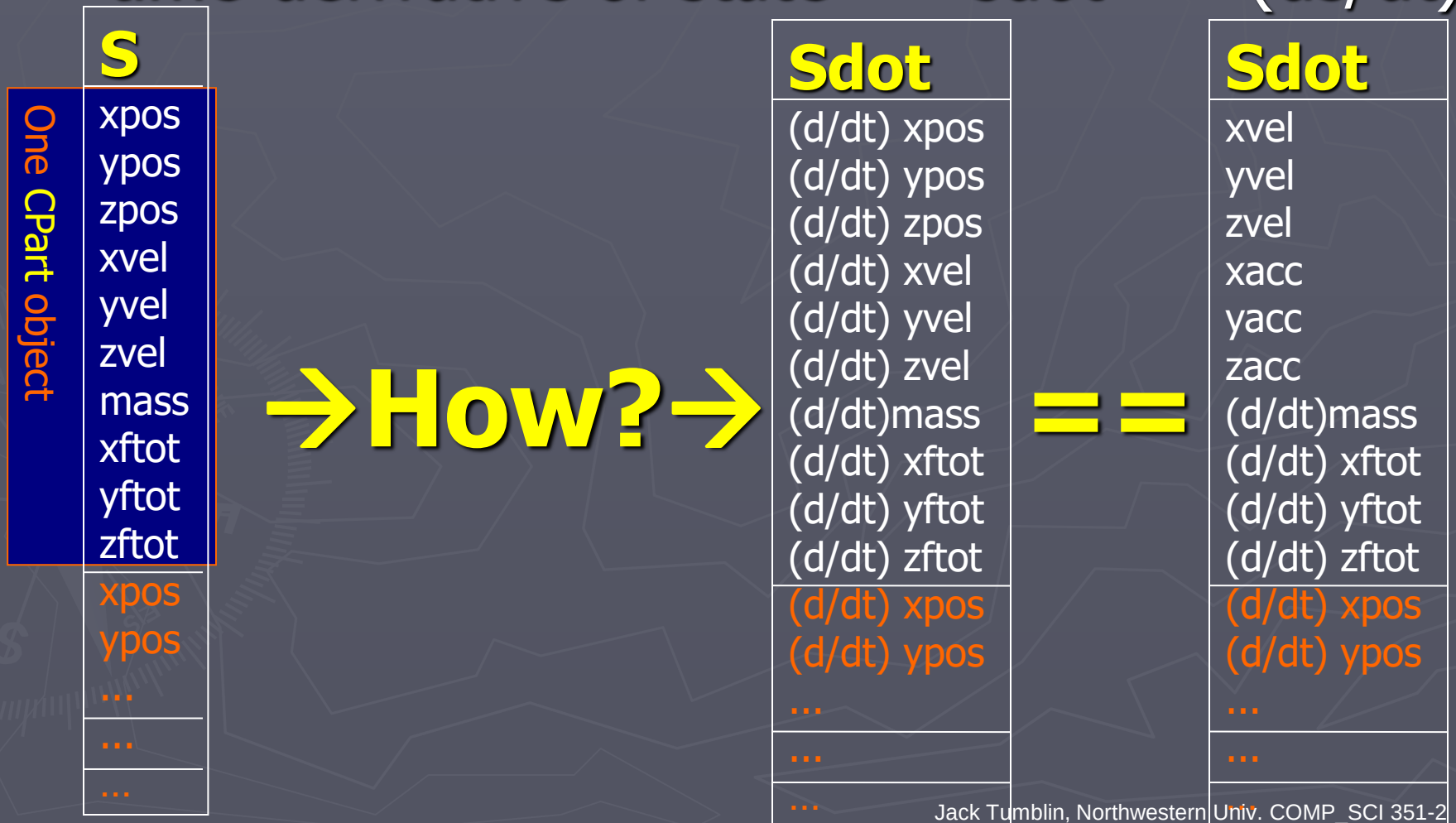
Physics in state-vector form

- Recall: 'state' == s
  'time-derivative of state' ==sdot == (ds/dt)



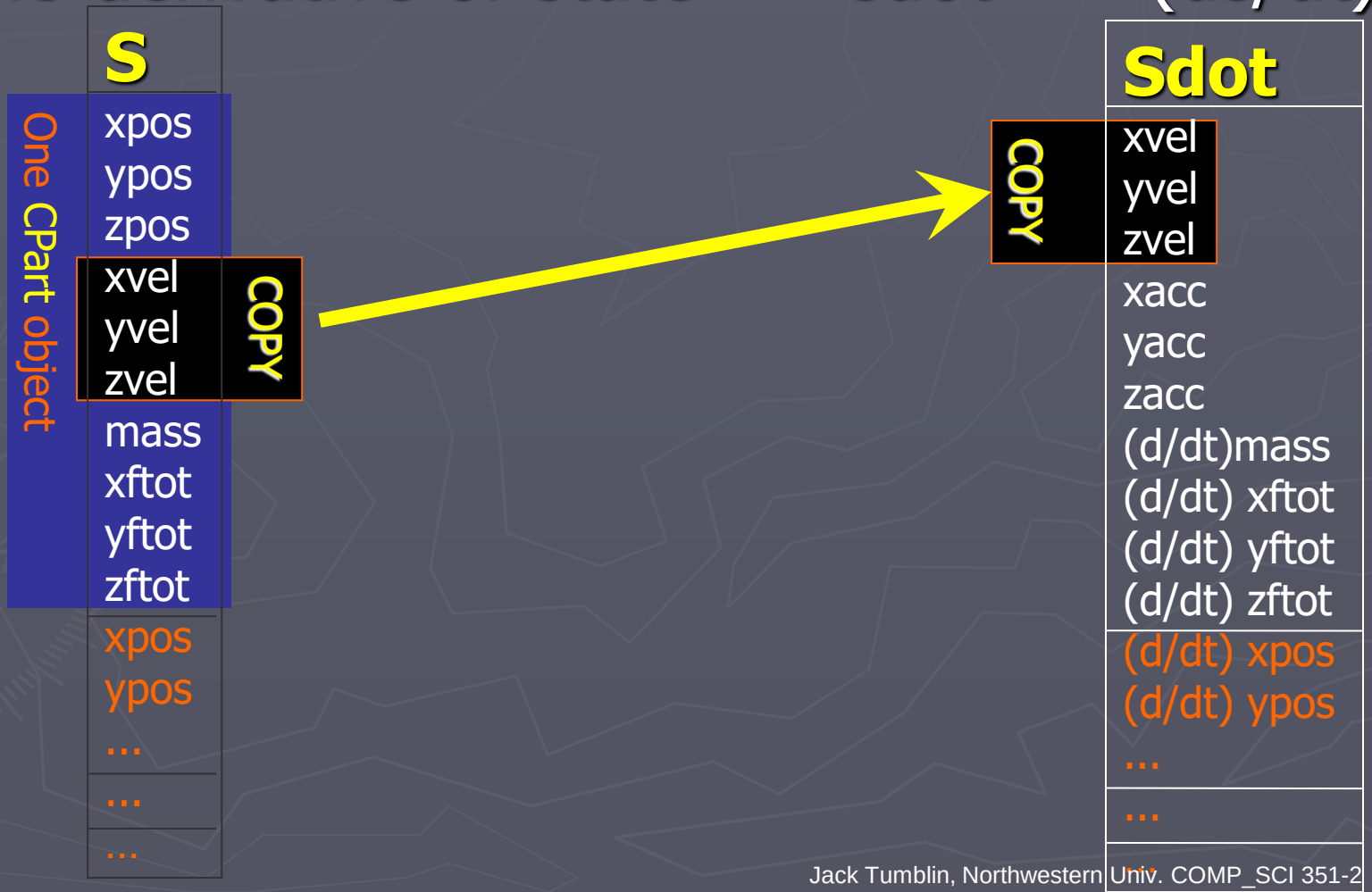
Physics in `dotMaker(pS, pSdot);`

- Recall: 'state' == s
  `time-derivative of state' ==sdot == (ds/dt)



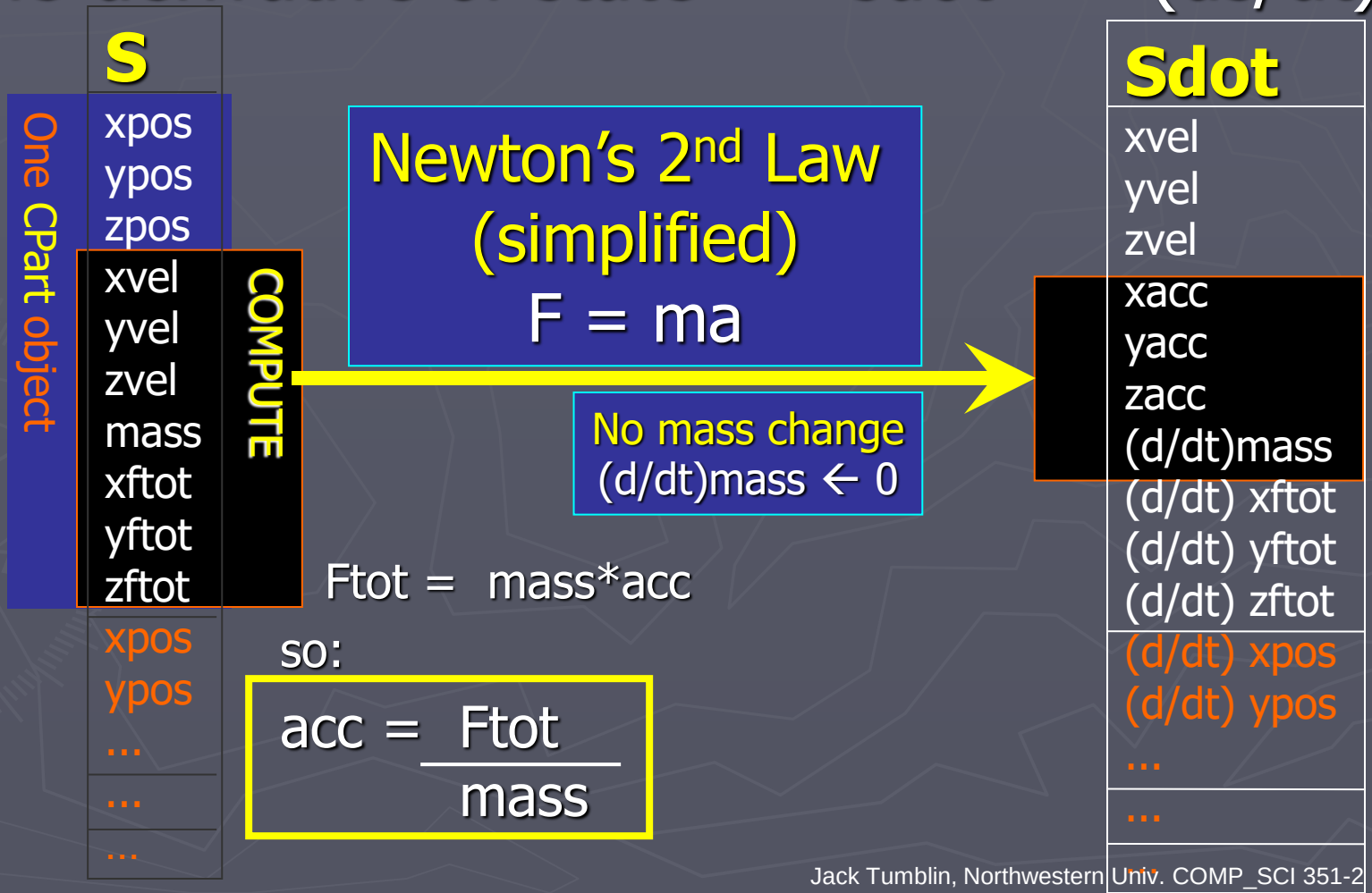
Physics in `dotMaker(pS, pSdot);`

- Recall: 'state' == s
  `time-derivative of state' ==sdot == (ds/dt)



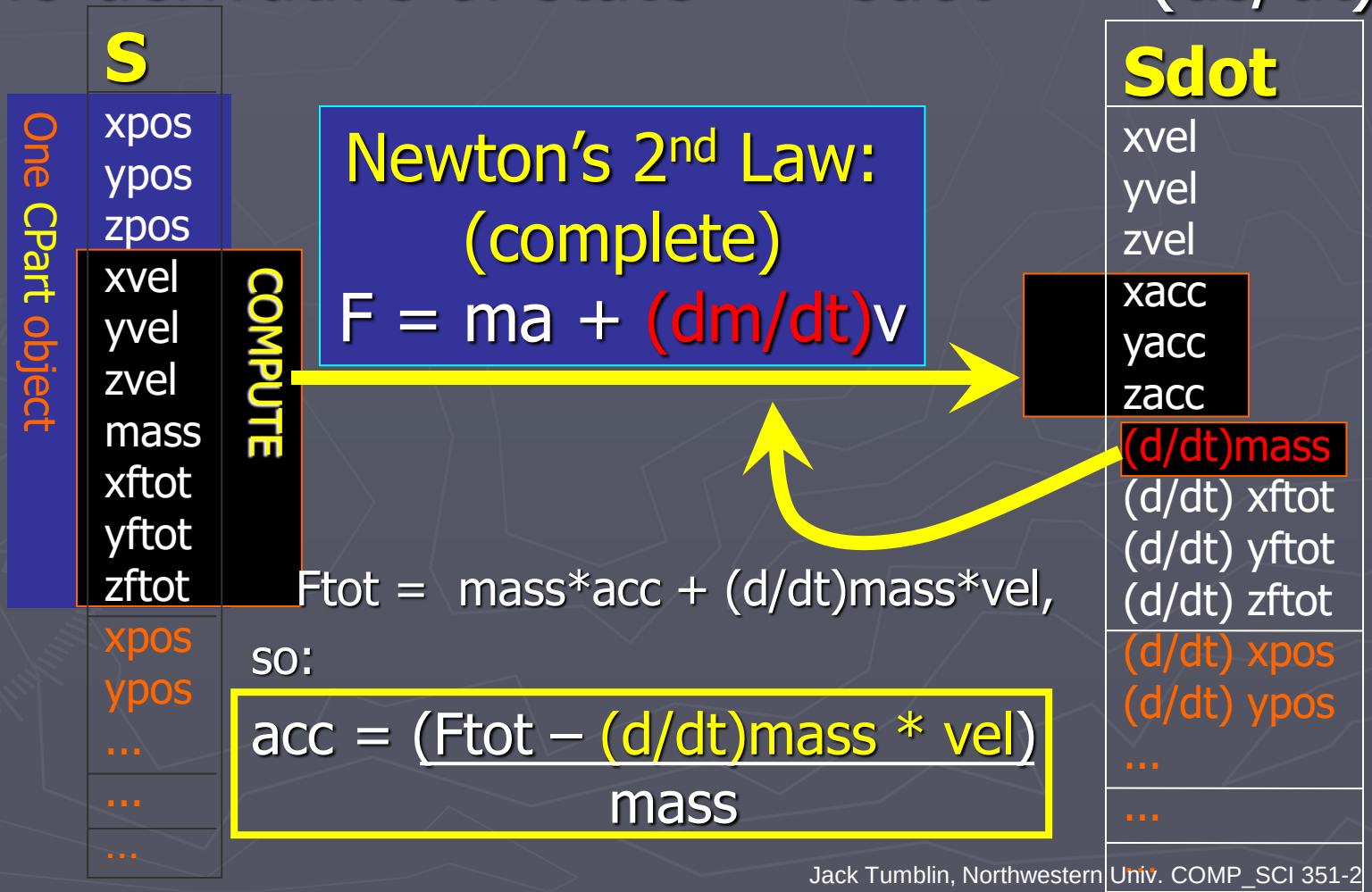
Physics in `dotMaker(pS, pSdot);`

- Recall: 'state' == s
  `time-derivative of state' ==sdot == (ds/dt)



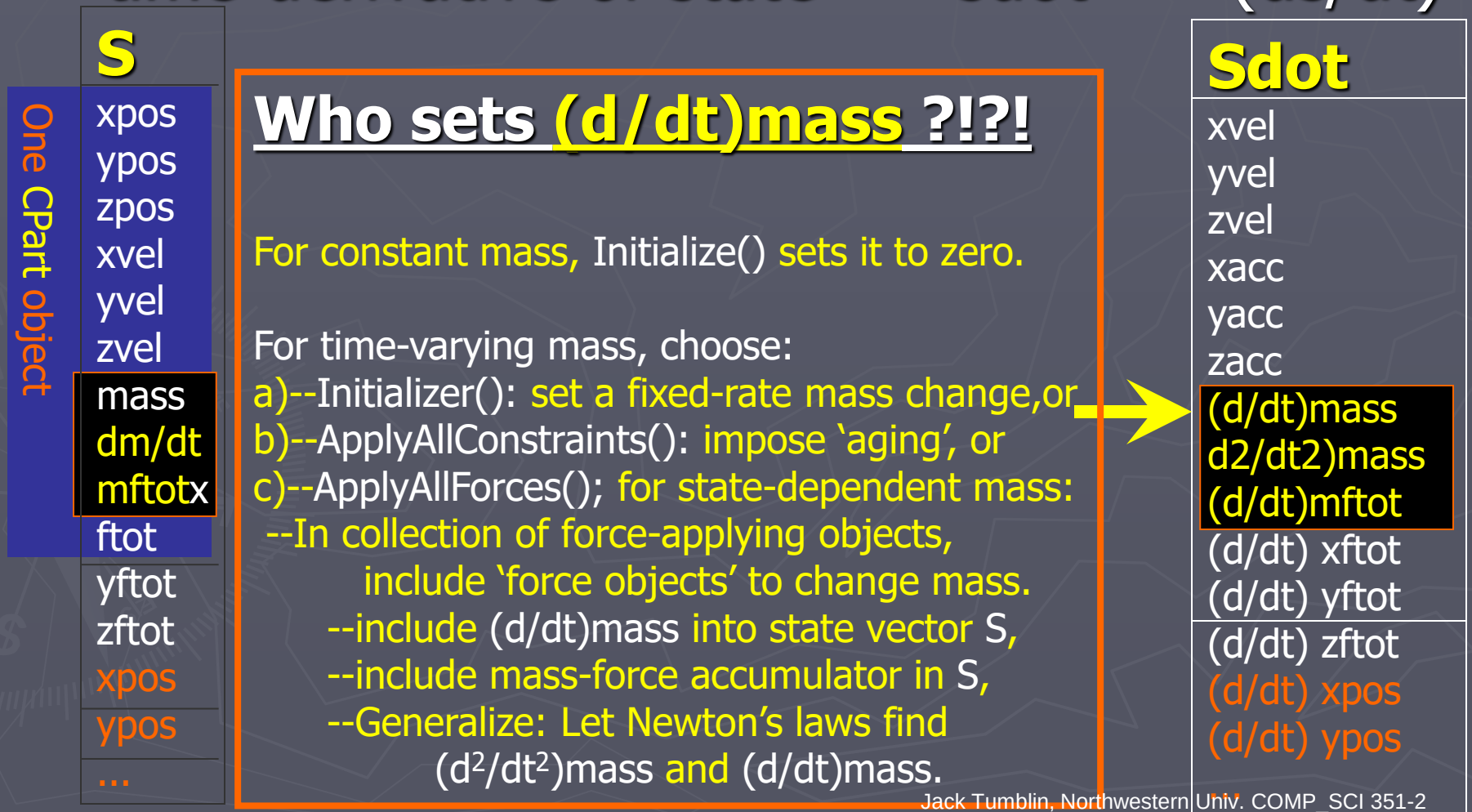
Physics in `dotMaker(pS, pSdot);`

- Recall: 'state' == s
  `time-derivative of state' ==sdot == (ds/dt)



Physics in `dotMaker(pS, pSdot);`

- Recall: 'state' == s
  'time-derivative of state' ==sdot == (ds/dt)



Ballistics → *Generalized* Force Fields:

- Generalized particle state s1 can include MANY MORE parameters than just (inverse) mass, position, velocity, and its force-accumulator:
  - Age, Color, Pressure, Temperature, Friction, Diameter, Viewing Direction, crowding, etc.
- Why not '*generalize*' a few to change over time using Newtonian Dynamics (F=ma+...)?
- HOW? Expand s, sdot to include derivatives of ***all*** particle params:

e.g.

$$s \leftarrow \text{diam}, (d/dt)\text{diam}, \ldots$$
$$\dot{s} \leftarrow (d/dt)\,\text{diam}, (d^2/dt^2)\text{diam}, \ldots$$

Ballistics → *Generalized* Force Fields:

- Generalized particle state s0 can include MANY MORE parameters than just (inverse) mass, position, velocity, and its force-accumulator:
  - Age, Color, Pressure, Temperature, Friction
- Whoa! Wait! Hold on there! –
- ?we should store 'inverse' mass? **Why?** (textbook explains)
- It's faster! Save 1/m in particle, not **m**, to replace divide (slow!) with multiply (faster):

  acc = Ftot*(**1/m**)

sdot ⇐ (d/dt) diam, (d²/dt²)diam, …

More CForcer Objects: **Springs**

- Newton's 3rd Law: Spring imposes equal and opposite **forces on TWO particles P0, P1**
- Simple Spring Law: Force $\propto$ -Displacement:

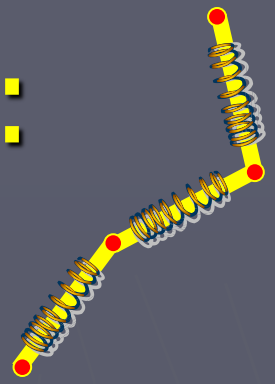
$$\|F\| = K_s * L$$

  - Spring Constant: K_s
  - Spring 'rest' length== L_r (zero-force length)
  - Spring displacement length == L

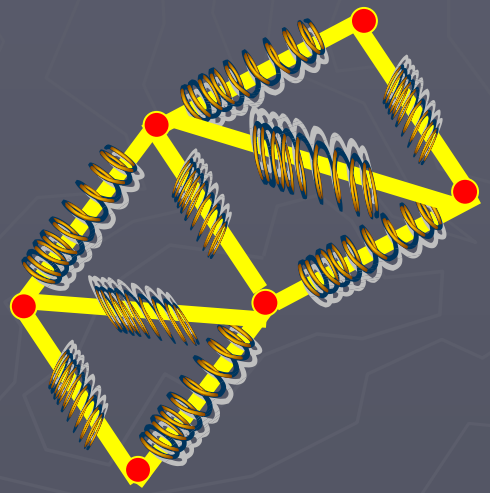
$$L = \|P1-P0\| - L_r$$

  - L>0? **F0** pulls P0 *towards* P1 and **F1** pulls P1 *towards* P0. Find F0, then **F1 = -F0**

Spring-Mass **Structures:**

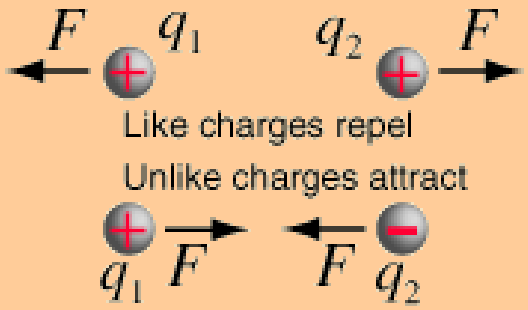


- Assemble 1D, 2D, 3D 'soft' structures from:
  - point-mass particles +
  - Massless springs with
  - non-zero rest-lengths



- 1D: Rope, knots, tendrils…
- 2D: Bubbles, Cloth, image textures…
- 3D: Jello, Living Tissues (surgery?)
- ? How should you render these hybrid CForcer + CPart objects ?
- How would you simulate 'tearing' and 'breaking' with these force-applying objects?

Coulomb's Law: Force from **Charge**



$$F = \frac{kq_1q_2}{r^2} = \frac{q_1q_2}{4\pi\varepsilon_0 r^2}$$ *Coulomb's Law*

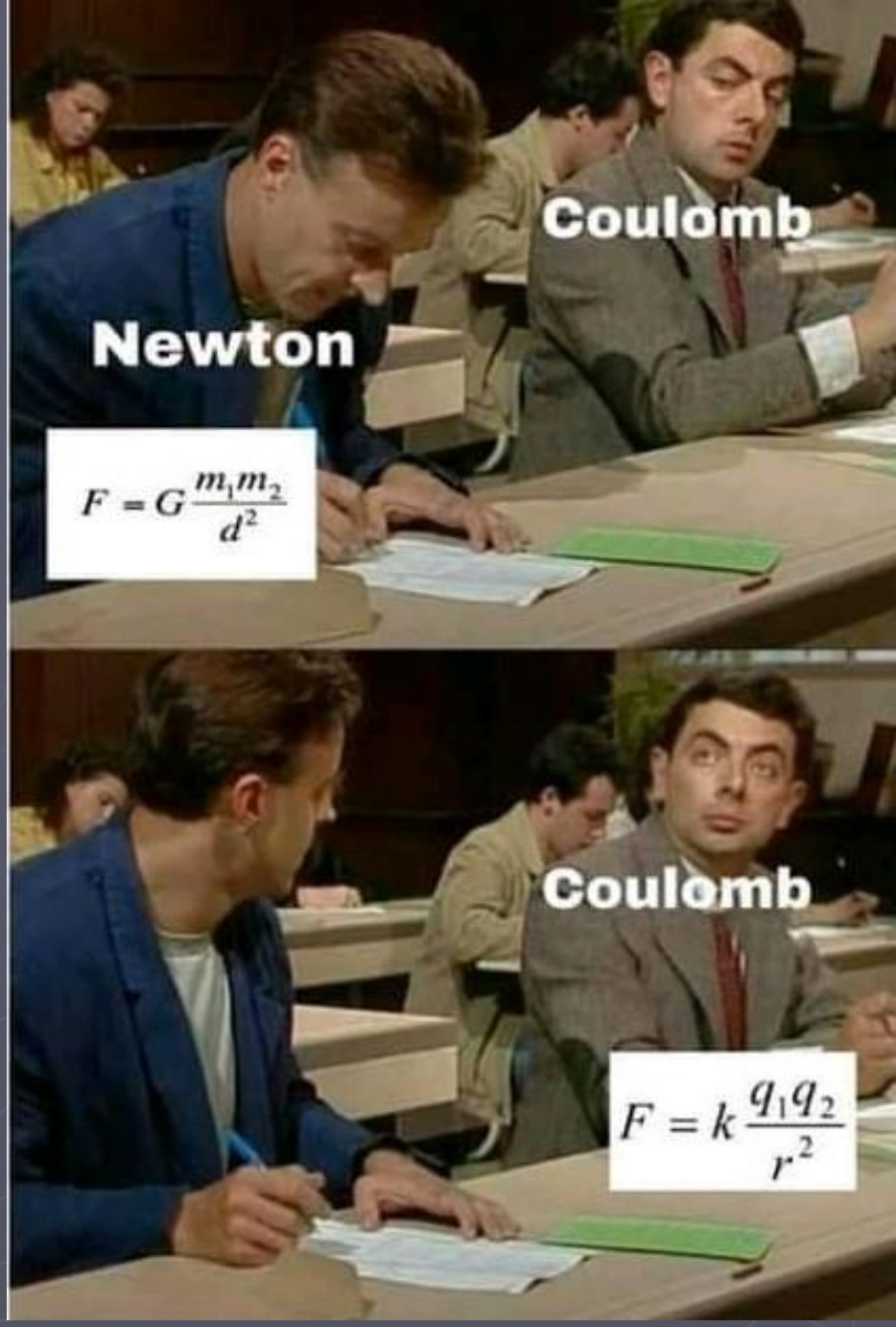
- Electrostatics:
  see http://hyperphysics.phy-astr.gsu.edu/hbase/electric/elefor.html

  Isolated Charges; assume no nearby conductors
  (Why no conductors? Their charges all MOVE FREELY; much trickier!)
- Coulomb forces act as a '**generalized spring**':
  - Each charge q trades force with all other charges
  - Particle P0 has charge q0, … PN has charge qN
  - Sum forces from *all pairs* of charges in Ftot

Force from **Gravity** (Newton)

Force from **Charge** (Coulomb)

- Hmmm … similar …
- Inverse-square laws,
- But mass is always >0;
- And charge is >0 or <0; can attract **or** repel…



Other Force-Making Objects

- 'Boids' – The 4 forces the paper describes…
- Drag
- Viscosity
- Etc.

(see textbook, Chap 6.5 'flocking systems')

If you wish,

Particles let you re-create all of classical physics through particles-as-molecules… (!!!)

Constraint Objects: Walls

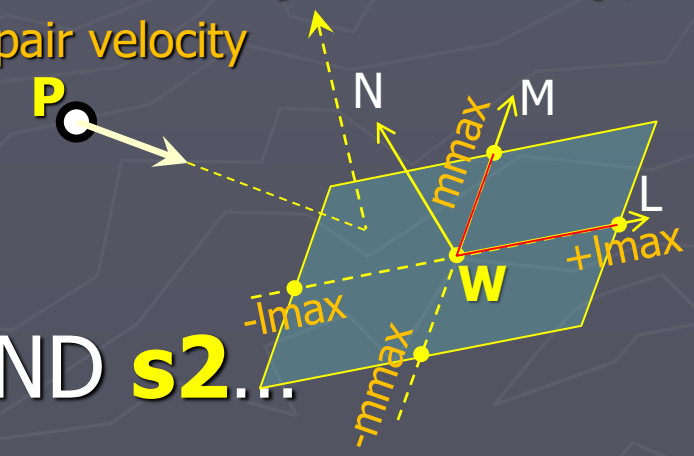
Axis-Aligned walls can be simple: (see Week1 starter code)

Adjust 'next' state s2: "did particle P hit the wall, AND is it moving further into the wall? if both true → reverse velocity"

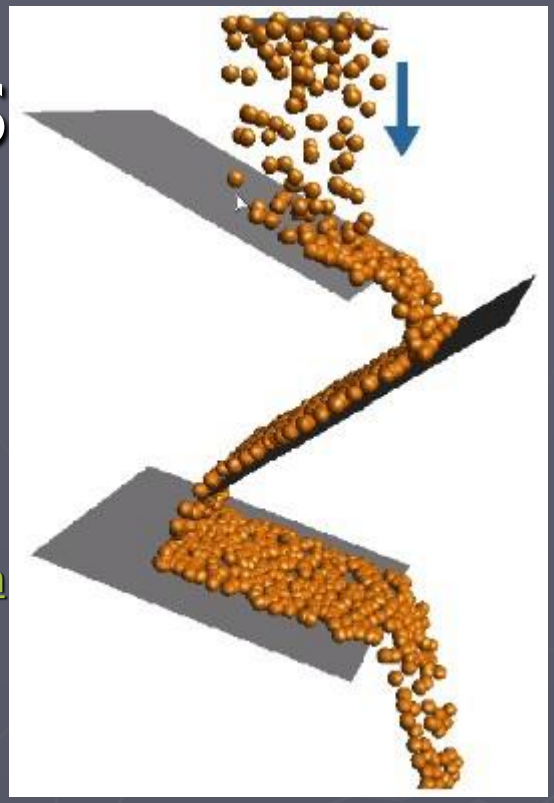
- if(xpos > xmax) && xvel > 0) bounce(right wall);
  if(ypos > ymax && yvel > 0) bounce(top wall);
- if(xpox < xmin && xvel < 0) bounce(left wall);
  if(ypos < ymin && yvel < 0) bounce(floor wall);
- bounce(): Week 1 code--resolve contact & repair velocity

? How can we make *arbitrary* 3-D bouncy plates?

AH! Easier given *BOTH* **s1** AND **s2**...



Constraint Objects: Walls



How could we do this?

(illustration from SoftImage tutorial)

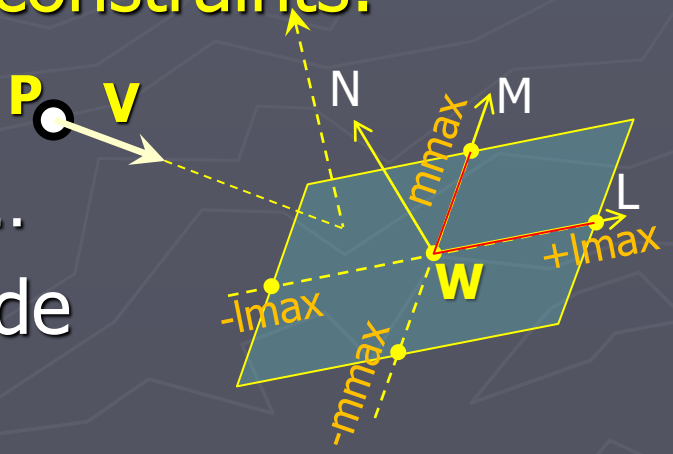
http://softimage.wiki.softimage.com/xsidocs/ICE_behavior_ICEParticlesSlidingonObstacles.htm

To make arbitrary 3-D wall plate constraints:

a) Define terms → → →

b) easier with BOTH **s1** AND **s2**…

c) review Week 1 bouncy ball code that 'solves floors' too…



Constraint Objects: Plates (1)

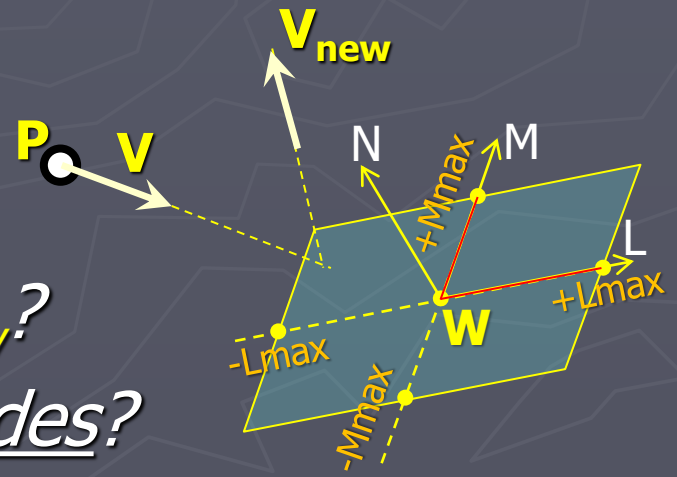
Arbitrary 3D Wall Rectangle as a "bouncy plate" ?

- *Define:*
  - plate or wall-surface point **W** (at wall center)
  - plate-surface normal vector **N**
  - plate-length vector **L** (wall length: +/- Lmax)
  - plate-width vector **M** (wall width: +/- Mmax)
  - particle **P** with velocity **V**

Solve:

1) *Did particle **P** hit the plate?*
2) *What its its new **velocity** $\mathbf{V}_{new}$?*
3) *Is the plate bouncy on both sides?*

(e.g. when **P** travels below **W** and **V**·**N** <0)



Constraint Objects: Plates (2)

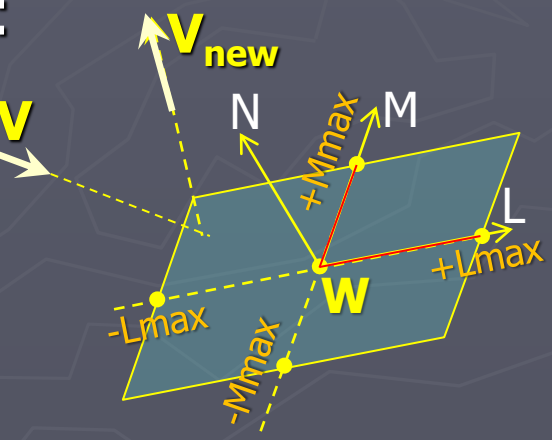
*1) Did particle **P hit** (pass through) the plate as s1 → s2?*

- 'yes' if P is ≤ the LM plane: **(P-W)·N** ≤0 ,
- AND if P is inside the rectangle: **((P-W)·L)**^{2} ≤ lmax^2
  AND **((P-W)·M)**^{2} ≤ mmax^2

BETTER SOLUTION: Convert point P from world to 'wall' coordinates! How? make a world2wall matrix:

Caution! requires unit-length L,M,N vectors!

$$[world\,2wall][P] = \begin{bmatrix} L_x & L_y & L_z & 0 \\ M_x & M_y & M_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -W_x \\ 0 & 1 & 0 & -W_y \\ 0 & 0 & 1 & -W_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} P_L \\ P_M \\ P_N \\ 1 \end{bmatrix}$$

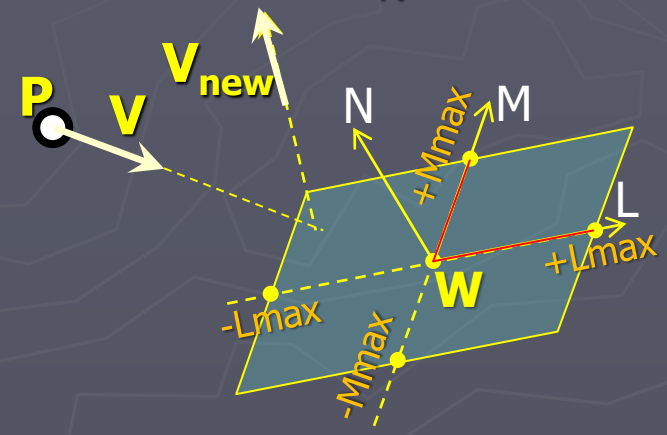


IF((P$_N$ > 0) in **s0** but (P$_N$ ≤ 0) in **s1**) then P touched the wall plane,
AND IF((P$_L$^{2} > Lmax^2) && (P$_M$^{2} ≤ Mmax^2) in **s1**) then P HIT the plate.
(good! Now how shall we 'bounce'?)

Constraint Objects: Plates(3)

2) What is new ***velocity*** $\mathbf{V}_{new}$*? To 'bounce' we must:*

- Reverse the **N**-direction portion of state **s1** velocity **V** ,
  which is $\mathbf{V}_N = (\mathbf{V}\cdot\mathbf{N})\mathbf{N}$
  Caution! requires unit-length N vector!
  $\mathbf{V} = \mathbf{V}_L + \mathbf{V}_M + \mathbf{V}_N$; reverse sign of *only* $\mathbf{V}_N$ to get
  $\mathbf{V}_{new} = \mathbf{V}_L + \mathbf{V}_M - \mathbf{V}_N$; thus we just subtract V_N twice:
- $\mathbf{V}_{new} = \mathbf{V} - 2\mathbf{V}_N$
- **!Careful!** don't 'bounce' unless **V** aims ***INTO*** **the plate**; bounce *ONLY* when $(\mathbf{V}\cdot\mathbf{N}) < 0$



REVISED plate-bounce code:

```
if(in s1{P_N > 0} && in s2 {(P_N ≤ 0) && (P_L² ≤ Lmax² ) && (P_M² ≤ Mmax²)} )
{       //Yes, we HIT the wall. Shall we 'bounce()'?
        if( in s1 {V·N < 0}) then in s2{V_new = V -2V_N;}   // yes! bounce()!
}
```

Constraint Objects: Plates(4)

Caution! requires unit-length N vector!

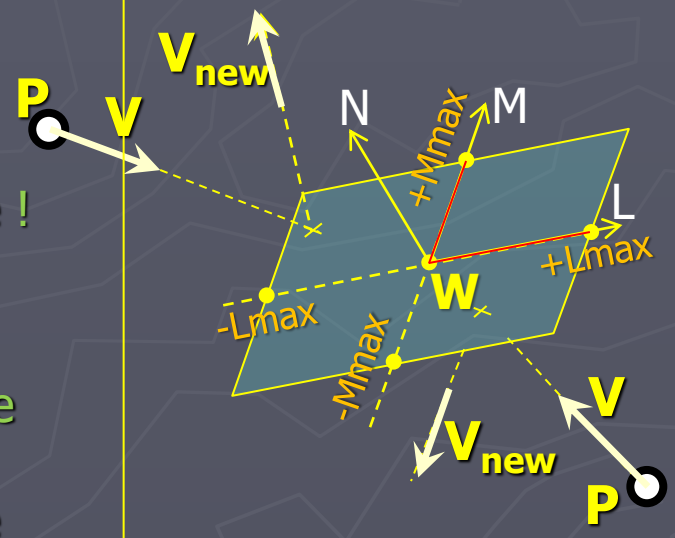
3) Is the plate bouncy on both sides?

We defined 'bouncing' on the **N>0** side, the 'top' of the wall.

How can we make it 'bounce' when hit from below?

*ANSWER: REVERSE the rules for the **N** axis:*

```
if((P_L^2 ≤ Lmax^2 ) && (P_M^2 ≤ Mmax^2)} )
{          // Yes; particle is within the plate's limits
   if( (in s1 {P_N > 0} && in s2 {P_N ≤ 0} )
   {    // particle moved down, thru top of the plate
      if( in s2 {V·N < 0))  // still moving down?
         THEN: set s2: { V_new = V -2V_N;}     // bounce !
   }
   else if (in s1 {P_N ≤ 0} && in s2 {P_N > 0} )
   {       // particle moved up, thru bottom of the plate
      if( in s2 {V·N > 0))  // still moving up?
         THEN set s2: { V_new = V -2V_N;}     // bounce
   }
}
```



Constraint Objects: Plate Hole (1)

- Make a circular 'hole' in the plate that permits particles to flow through it:

(How would YOU do this?)

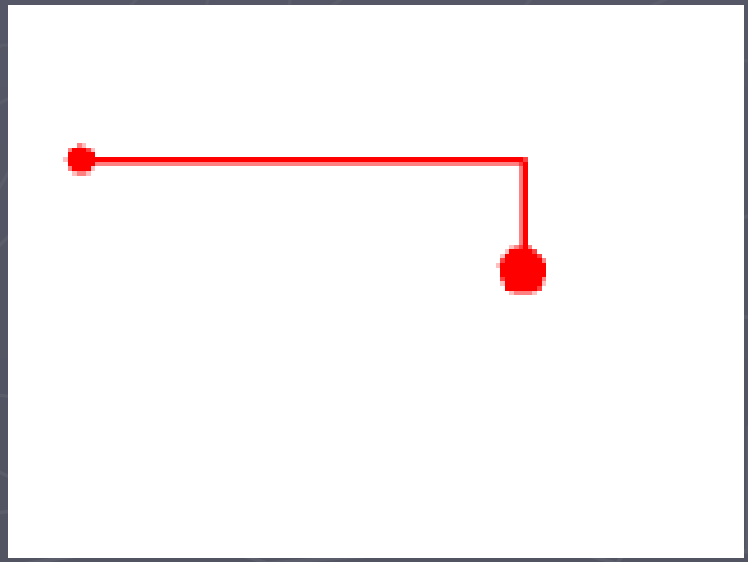
Constraint Objects: Ropes (1)

- 'Free' rope constraint (it touches nothing):
  - Define rope length as fixed scalar L_r, and $L2 = L_r^2$
  - Rope ties particle at $\mathbf{P}_A$ to particle at $\mathbf{P}_B$

How do we ENFORCE limit ($||\mathbf{P}_A - \mathbf{P}_B|| \leq L_r$) ?

- An ideal 'dead' rope (zero elasticity):
  - $dist^2$ from dot-product:

    $d2 = (\mathbf{P}_A - \mathbf{P}_B) \cdot (\mathbf{P}_A - \mathbf{P}_B)$;
  - IF (in s1{d2 >L2})
    - Find unit vector ($\mathbf{P}_A - \mathbf{P}_B$);
    - YOU figure it out…
      - (HINT: remove relative velocity in that direction from both particles…)

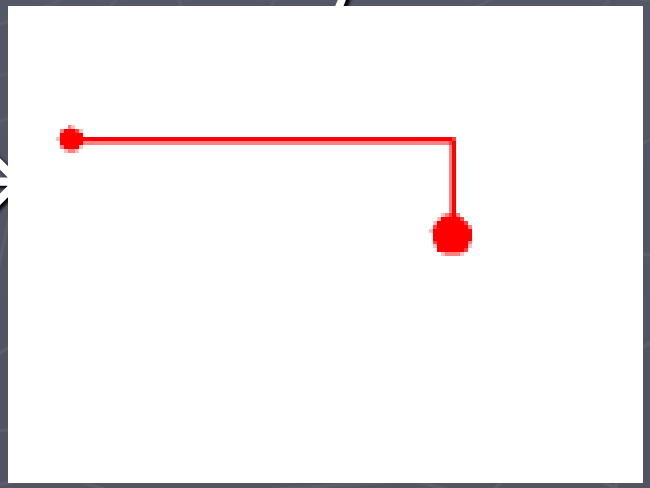


Constraint Objects: Ropes (2)

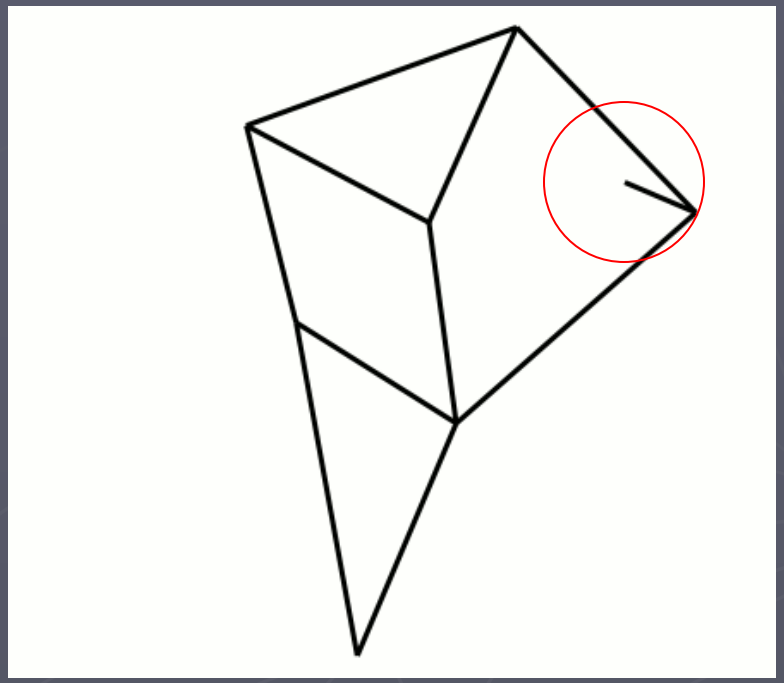
- 'Free' rope constraint (it touches nothing):
  - Define rope length as fixed scalar L_r, and $L2 = L_r^2$
  - Rope ties particle at $\mathbf{P}_A$ to particle at $\mathbf{P}_B$

How do we ENFORCE limit ($||\mathbf{P}_A - \mathbf{P}_B|| \leq L_r$) ?

- Elastic rope? non-zero restitution coefficient! Bounce-like reversal of 'stretch' velocity…
- Non-Free Rope?
  How would you do this? → →



Constraint Objects: Rods (1)



(How would YOU do this?)

Strandbeest bike: https://giphy.com/gifs/bike-strandbeest-C8DgeBFxDDDaw/download

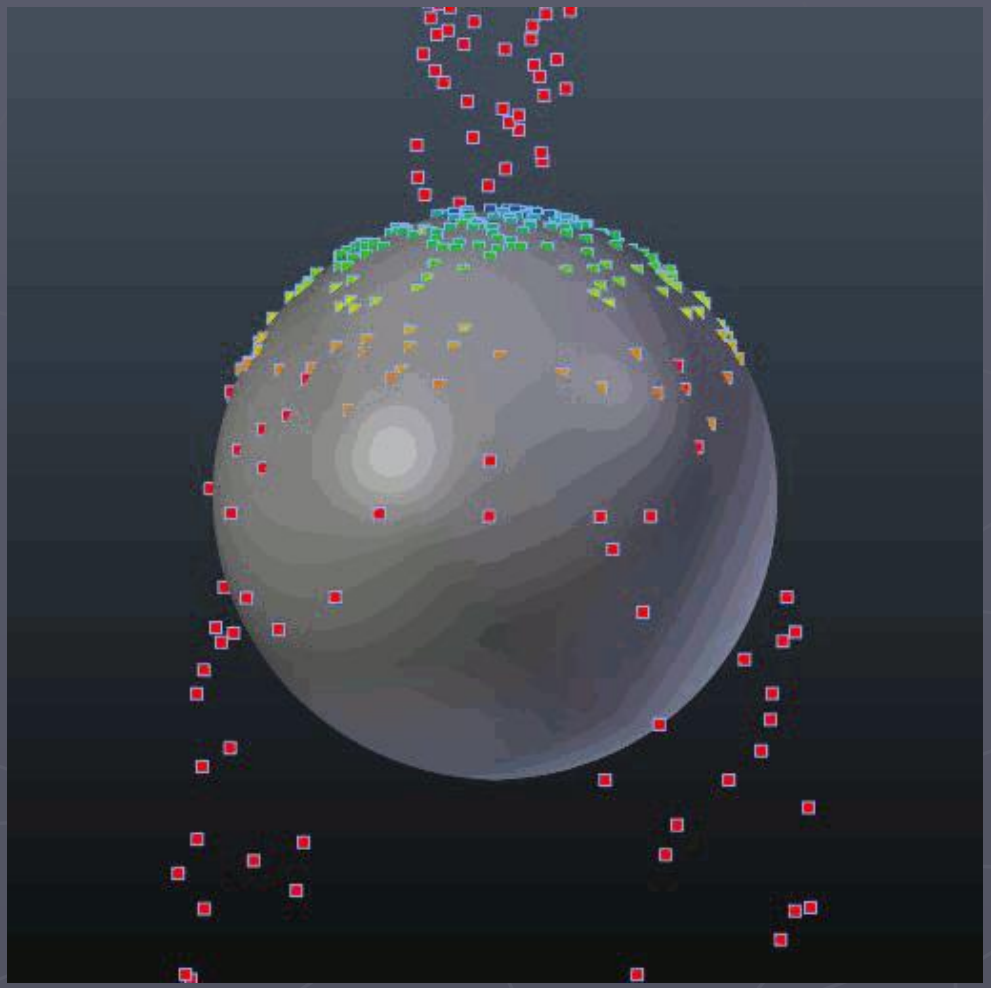
Strandbeest leg-joint:
https://commons.wikimedia.org/wiki/File:Strandbeest-Walking-Animation.gif

Constraint Objects: Slots (1)

Challenge: two-slot pivoting, or:
'Secrets of the Nothing Grinder' --
worth watching! https://youtu.be/7Fn-26Jmi5E

(How would you implement this?)

Constraint Objects: Spheres (1)



by Arek Kasprzyk :

(How would you do this?)

https://blenderartists.org/forum/showthread.php?383547-Particles-as-water-drops

REVIEW 1:

- (2) BIG IDEAS of state space:
  - Even if the causes of change (motive forces, etc) affect and respond to *many* derivatives, we can can put them inside **s**. Describe them ALL with
    - just ONE state vector **s**, and
    - Just ONE derivative, **sDot**, the time derivative
  - Numerical Integration == 'solver()' function: **find next state from current and previous states and their derivatives** (with timestep **h**)

    e.g. Euler: s2 = s1 + s1dot*h

    Midpoint: sM = s1 + s1dot*(h/2)

    s2 = s1 + sMdot*h (more? next class)

REVIEW 2: Physics & Calculus

- Other helpful quantities:
  - 'Work' W == $\int$(force*dist) d(dist)
  - 'Kinetic Energy':
    work req'd to accelerate mass to its current velocity
    $E_k = \frac{1}{2}mv^2$ == ½ mass * velocity^2
  - 'Potential Energy':
    work done if we released all constraints on movement
    --From gravity: $E_g = mg_0 \Delta y$ == mass * grav * Δheight
    --From spring: $E_s = \frac{1}{2} K_s \Delta L^2$ == ½ K_{spring} * (Δlength)^2
    --From charge: E_q = … see 'Coulomb's law…
    --From planetary gravity? Similar to charge…
    --What others can you find?

REVIEW 3: F=mA isn't enough!

- Write position vector as **x** or **x(t)** where t==time,
  velocity vector as **v** or **v(t)** = (d/dt)x,
  acceleration vector as **a** or **a(t)** = (d/dt)v= (d^2/dt^2)x

- Write mass as **m** or **m(t)**, and
  momentum as = m **v** = m (d/dt)**x**

- THEN Newton's 2nd Law is: **F** = (d/dt)(m**v**)

  "force == time-derivative of MOMENTUM mv"

- For time-varying mass m(t) and time-varying velocity:

  F(t) = m(t) · (d/dt)v(t) +
  (d/dt)m(t) · v(t)

  (by product rule: (d/dt)(uv) = u(dv/dt) + v(du/dt))

- *BUT* if mass is constant, then (d/dt)m(t) = 0; reduces to

  F = m · (d/dt)v(t) = m (d^2/dt^2)x, or F = ma

- Surprise! F=ma is a special case (constant mass);
  when mass may change, use F = ma + (dm/dt)v instead!

Physics, Notation, and Calculus

- HINT: Euler Method is an approximation, and its errors often add energy quickly!
  - 'Blow-ups'; energy→∞. What can we do?
  - Simple, 'hacky' solutions:
    - Smaller time-steps
    - Conservation of Energy:
      Find E_{tot}, the system's total energy(kinetic energy + potential energy + heat (energy lost to friction)): scale all velocities to keep E_{tot} constant, or limit its growth
    - Apply extra velocity damping forces to springs
    - Reduce step-size to reduce error. Adaptive step size: complex!

- **BETTER SOLUTION: use a better solver!**

END

END

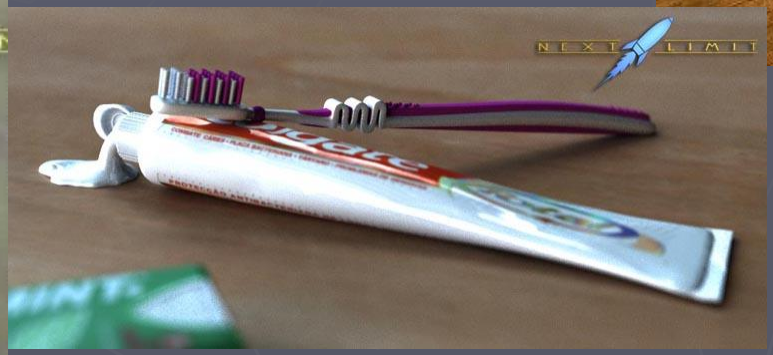
Partial Derivatives VS Full Derivatives

GREAT illustrated website:
http://www.math.umn.edu/~rogness/multivar/partialderivs.shtml

Higher ORDER Partial Derivatives:
http://tutorial.math.lamar.edu/Classes/CalcIII/HighOrderPartialDerivs.aspx

http://apollo.lsc.vsc.edu/classes/met380/Fingerhuts_notes/driv.pdf

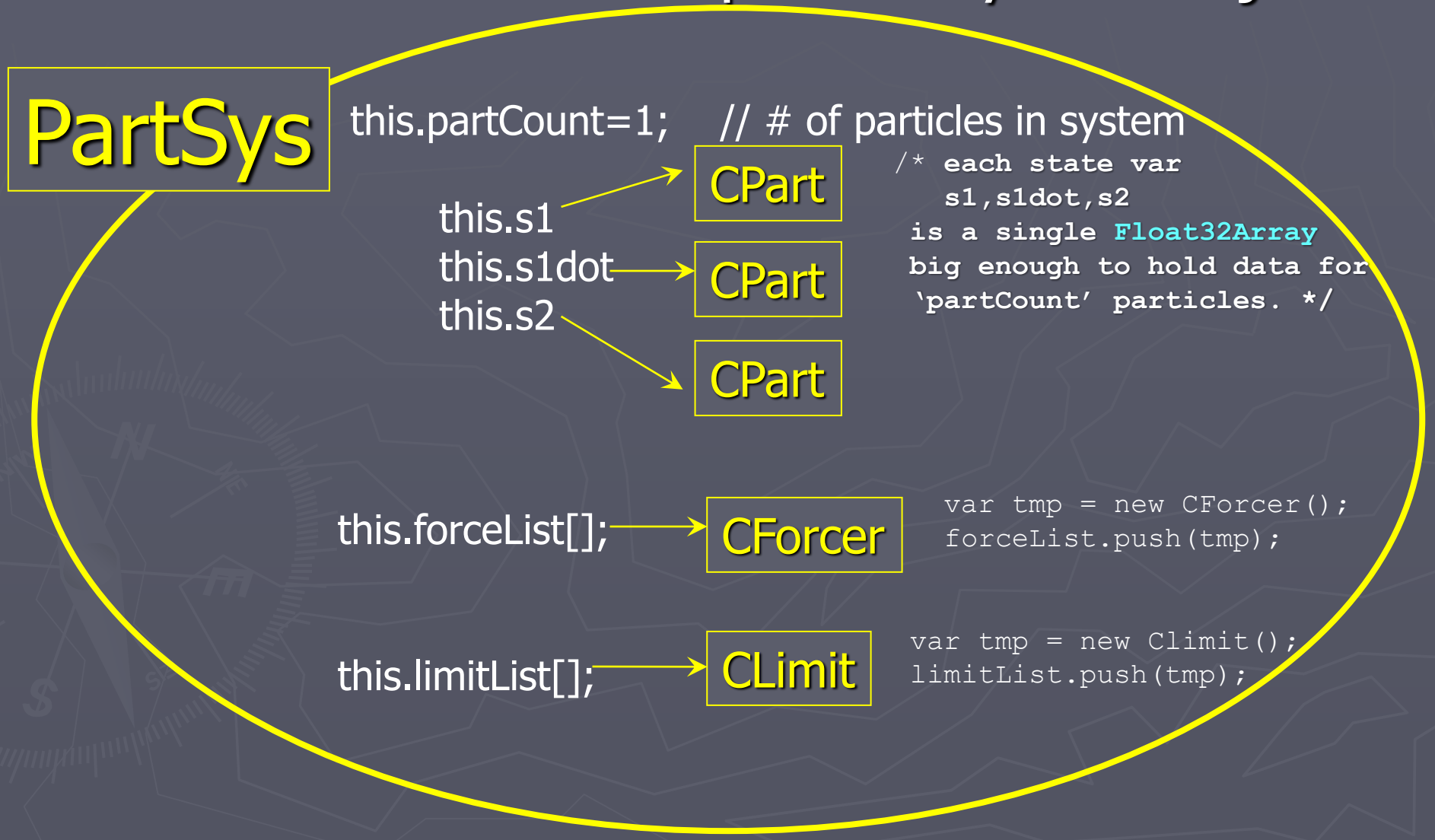
Solver Code for Particle Systems—E



Jack Tumblin
COMP_SCI 351-2

Recall: Master Plan

- Essential data in one particle system object:

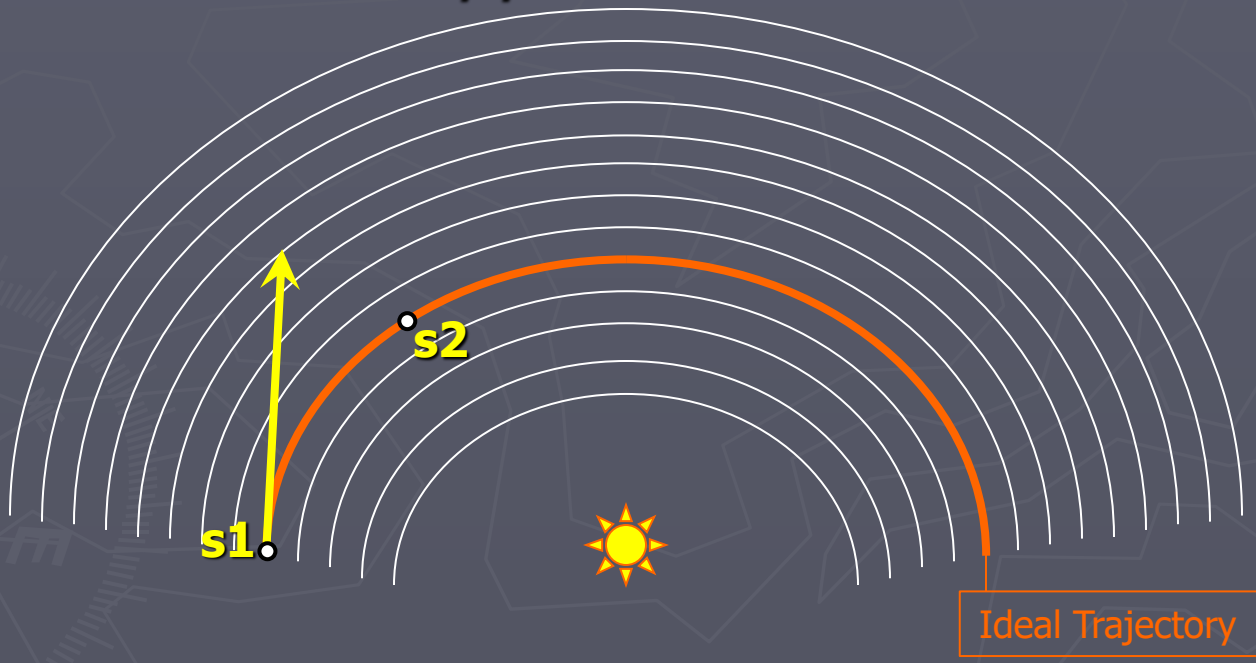


Recall: PartSys Program

- INIT State Vectors, Forcer Sets, Limit Sets
  - State Vector s1 describes the (array of) ball(s) at the current time t; s2 is 'next' state at next time instant t+h.
  - State Vector s1dot is the s1 time-derivative; the time-rate-of-change for s1 during an *infinitesimal* time (dt). At (t+dt), state s=s1 + dt*s1dot
  - Forcer Set describes the (array of) force-applying object(s) (e.g. gravity object, wind object, spring) applied at time instant t.
  - Limit Set describes (array of) constraint-applying object(s) (e.g. walls, ropes, hinges, rods) enforced at current time t.
- ApplyForces(): use Forcer Set to find net forces Ftot on each particle in s1. (Store Ftot in each particle obj)
- dotFinder(): find s1dot by applying Newton's laws to Ftot
- Solver(): find next state (Euler/Explicit: s2 = s1+ h*s1dot)
- doConstraint(): apply LimitSet to s2: 'bounce' off walls, etc
- Render(): depict s2 (& maybe Forcers & Limits) on-screen
- Swap(); Transfer contents of state vector s1 ← s2.

Solvers and State Space Trajectories

- ALL WE KNOW OF STATE-SPACE: given a state **s**, dotFinder() can find its exact time derivative **sdot**. *All the rest is an approximation!*



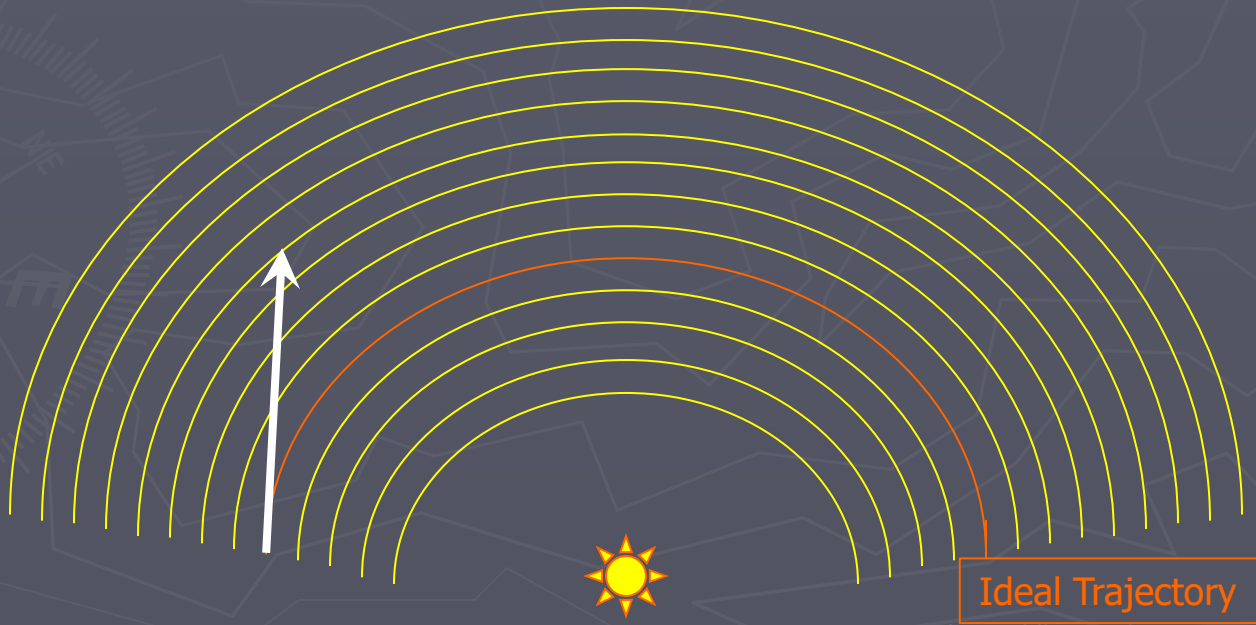
State Space
for an
Orbiting planet

- GOAL: Find **s2** on the same trajectory as given **s1**

Solvers and State Space Trajectories

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:
  s2 = s1 + h*s1dot
  - Simple fast, easy, but
    a) errors add energy($E = \frac{1}{2}(m\,||\mathbf{v}||)^2$) and it won't conserve momentum ($\mathbf{p} = m\,\mathbf{v}$) as we wanted;
    b) smaller errors will require smaller timesteps **h**



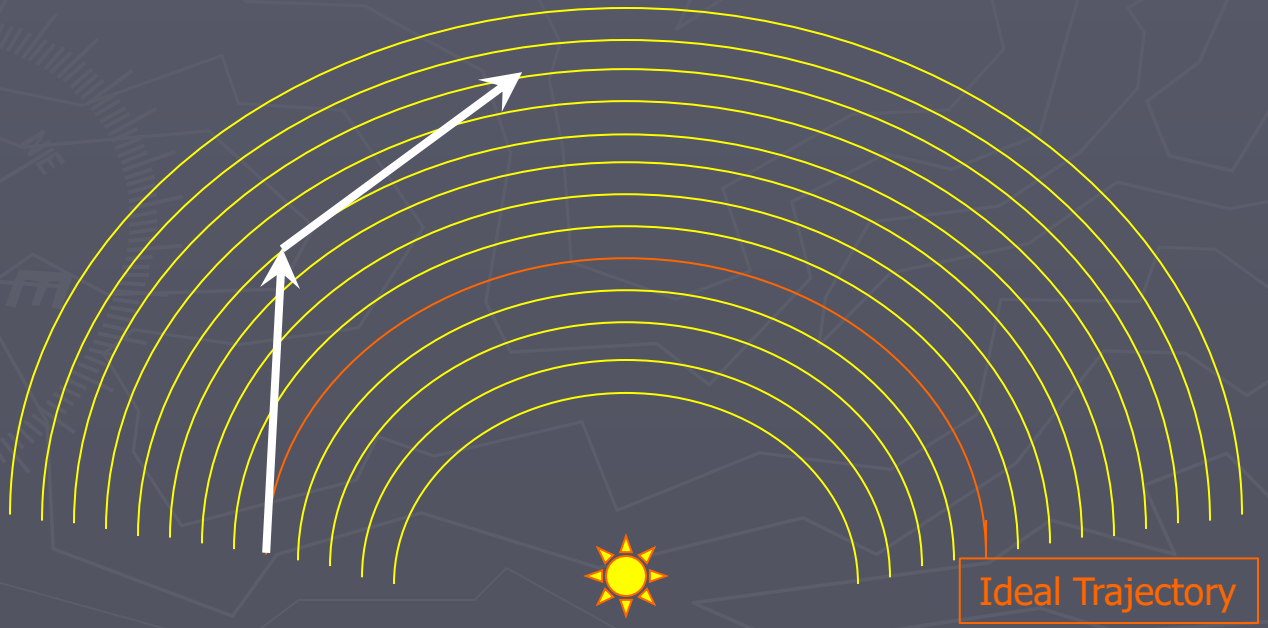
Solvers and State Space Trajectories

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:

  s2 = s1 + h*s1dot

  - Simple fast, easy, but
    a) errors add energy($E = \frac{1}{2}(m\,||\mathbf{v}||)^2$) and it won't conserve momentum ($\mathbf{p} = m\,\mathbf{v}$) as we wanted;
    b) smaller errors will require smaller timesteps **h**



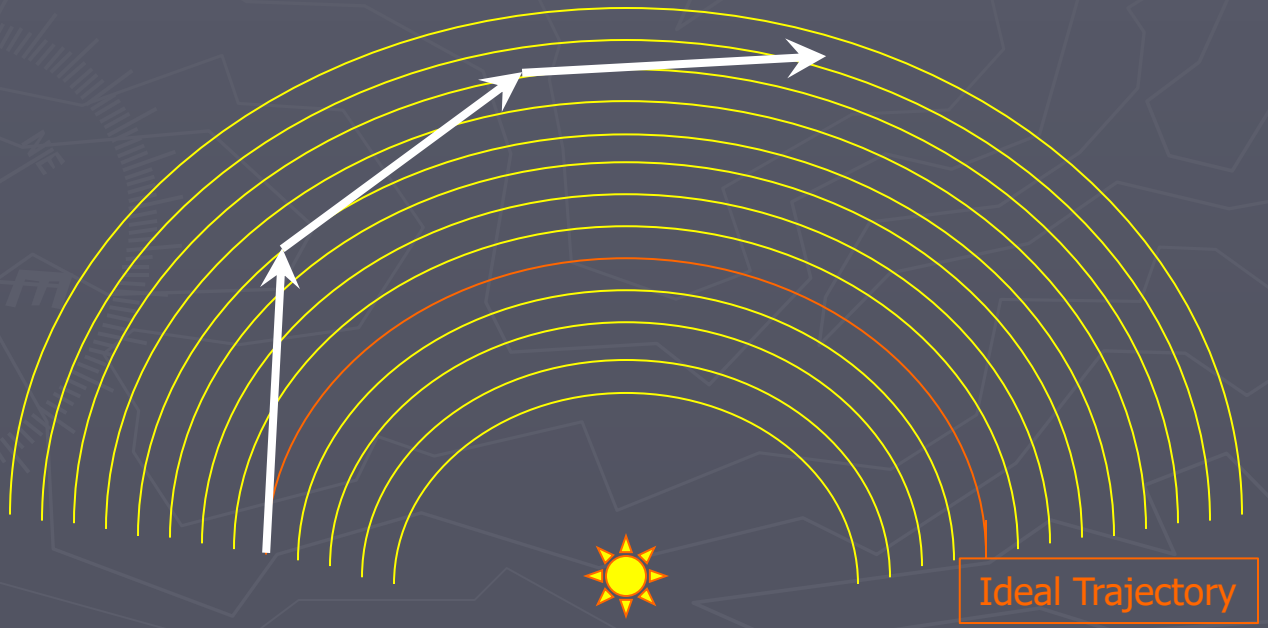
Solvers and State Space Trajectories

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:

  s2 = s1 + h*s1dot

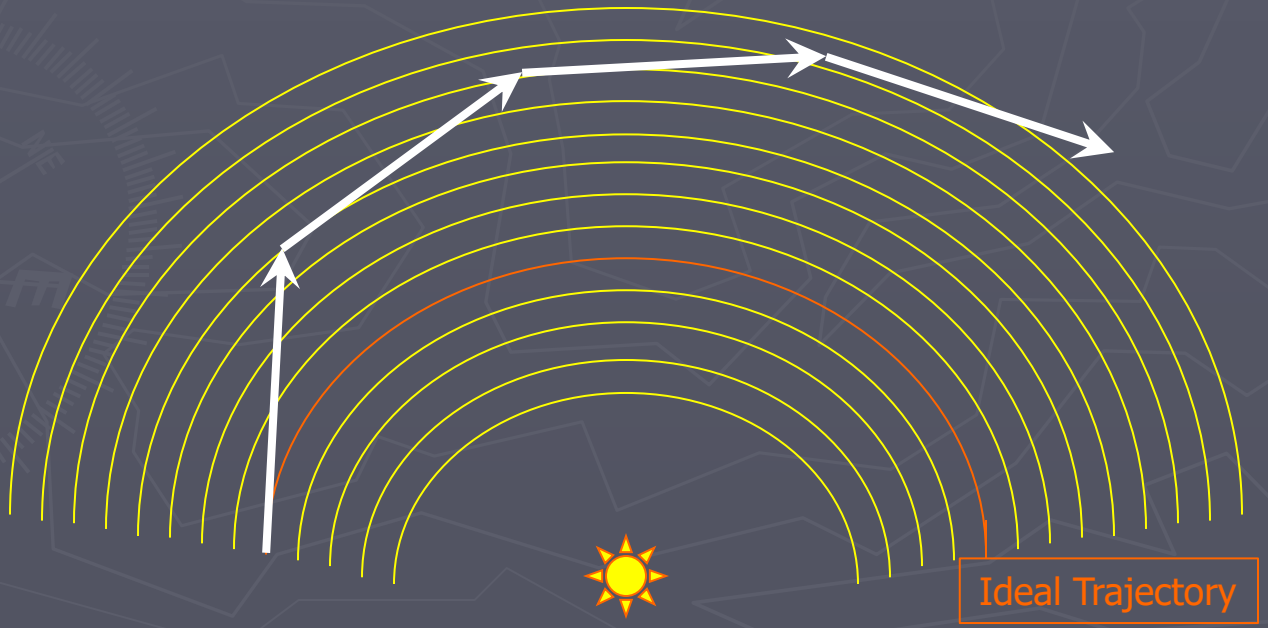
  - Simple fast, easy, but
    a) errors add energy($E = \frac{1}{2}(m\,||\mathbf{v}||)^2$) and it won't conserve momentum ($\mathbf{p} = m\,\mathbf{v}$) as we wanted;
    b) smaller errors will require smaller timesteps **h**



Solvers and State Space Trajectories

How can we find **s2**, the state at next timestep?

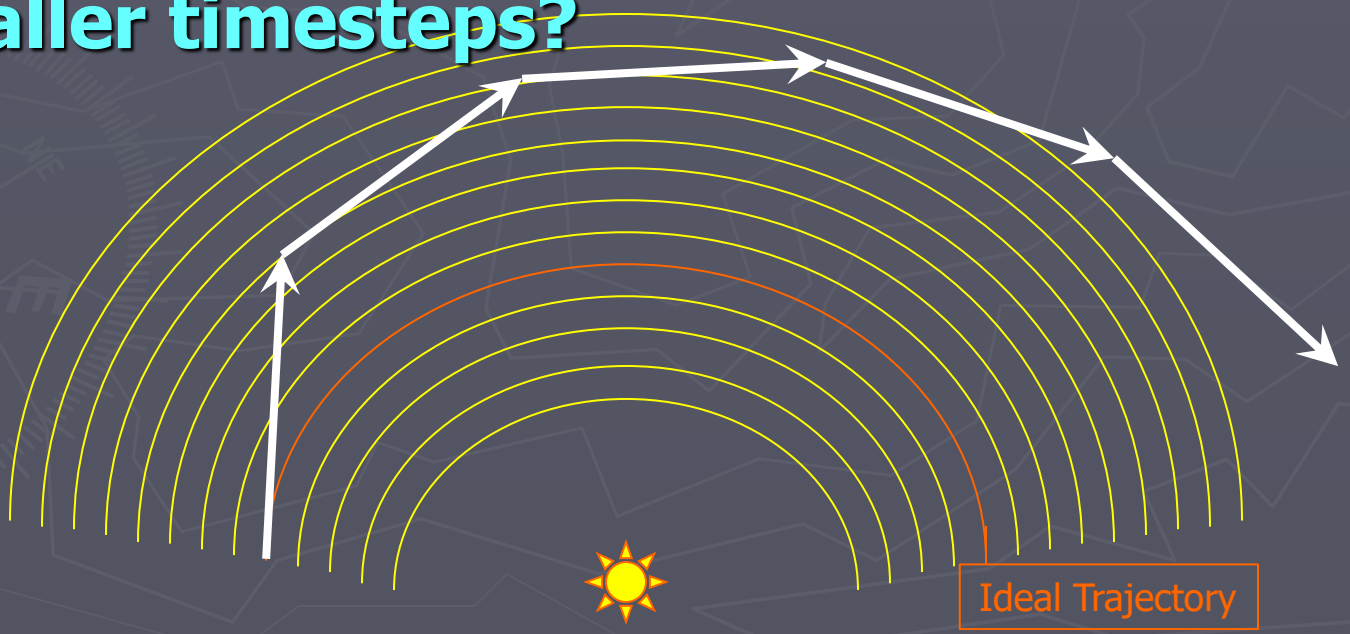
- Method 1: 'Euler' or 'Explicit' Solver:
  s2 = s1 + h*s1dot
  - Simple fast, easy, but
    a) errors add energy($E = \frac{1}{2}(m\,||\mathbf{v}||)^2$) and it won't conserve momentum ($\mathbf{p} = m\,\mathbf{v}$) as we wanted;
    b) smaller errors will require smaller timesteps **h**



Solvers and State Space Trajectories

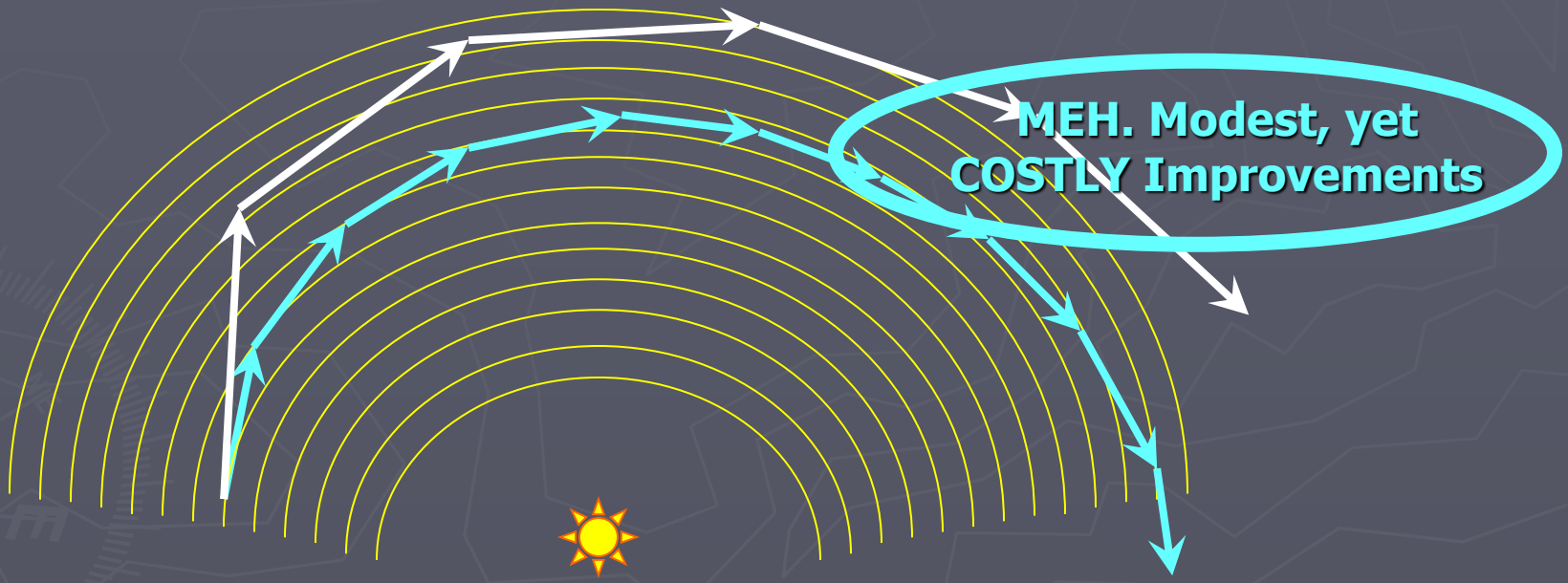
How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:
  s2 = s1 + h*s1dot
  - Simple fast, easy, but
    a) errors add energy($E = \frac{1}{2}(m\,||\mathbf{v}||)^2$) and it won't conserve momentum ($\mathbf{p} = m\,\mathbf{v}$) as we wanted;
    b) smaller errors will require smaller timesteps **h**
- **Smaller timesteps?**



Euler + Tiny Timesteps? → Tiny Help

- Method 1: 'Euler' or 'Explicit' Solver: **?TINY TIMESTEPS?**
  s2 = s1 + h*s1dot
  - Simple fast, easy, but a) errors add energy or momentum (mv)
    b) smaller errors require smaller steps



- Remember: All we can know in STATE SPACE: given a state **s**, dotFinder() can find its *exact* time derivative **sdot**. All the rest is an approximation!

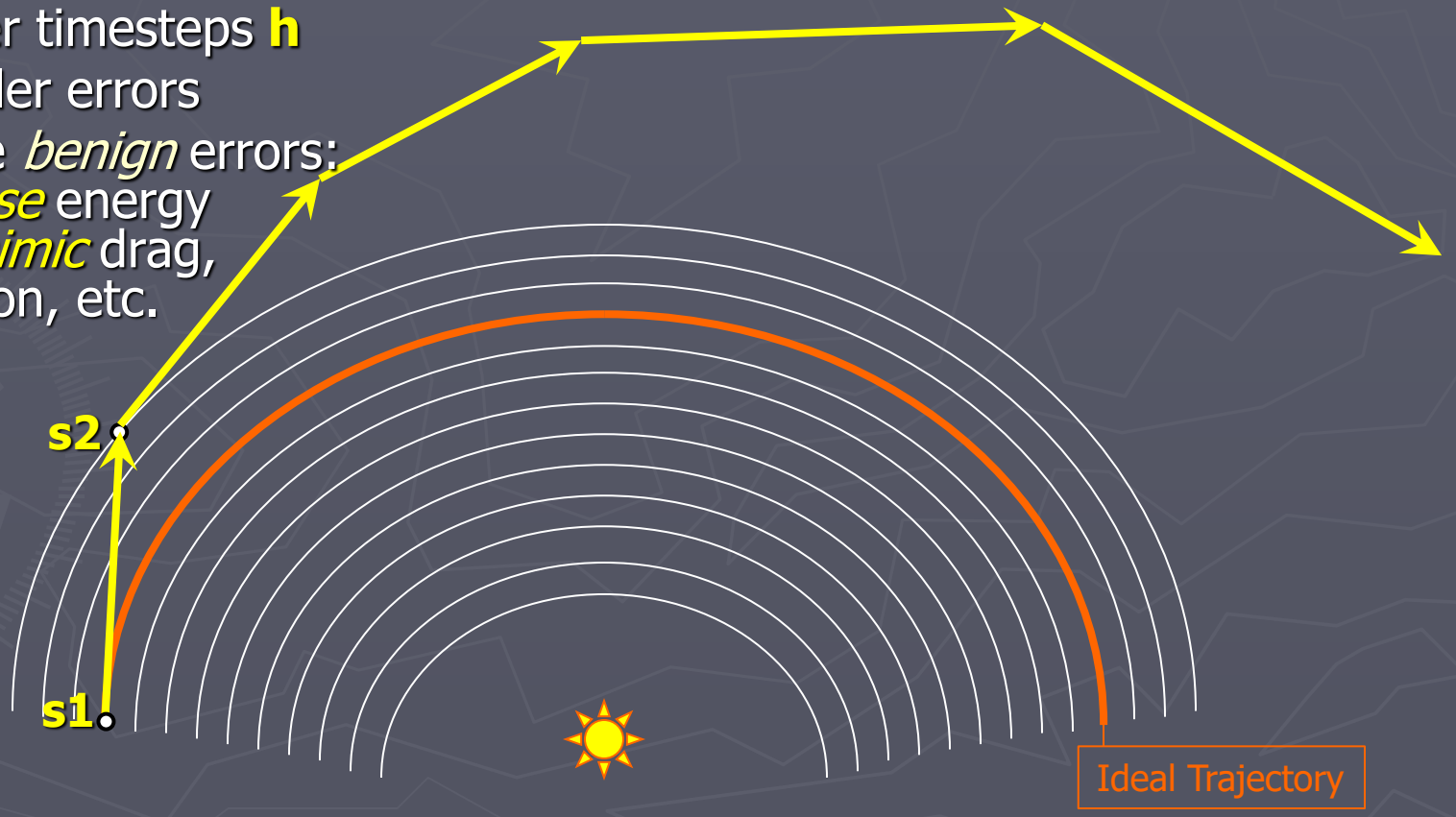
Better Goal : Better Solvers

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:

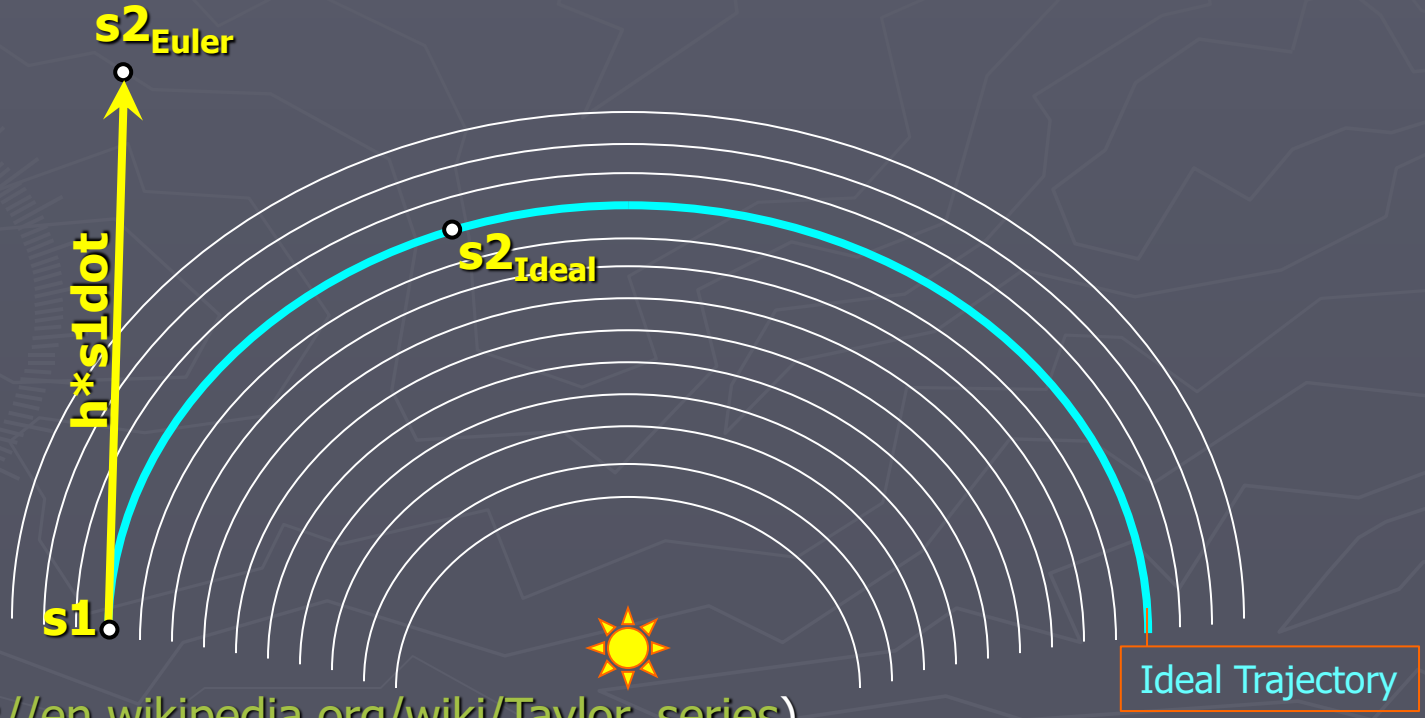
  s2 = s1 + h*s1dot

- WANTED:
  - larger timesteps **h**
  - smaller errors
  - more *benign* errors:
    to *lose* energy
    to *mimic* drag,
    friction, etc.



Solver Error in State Space

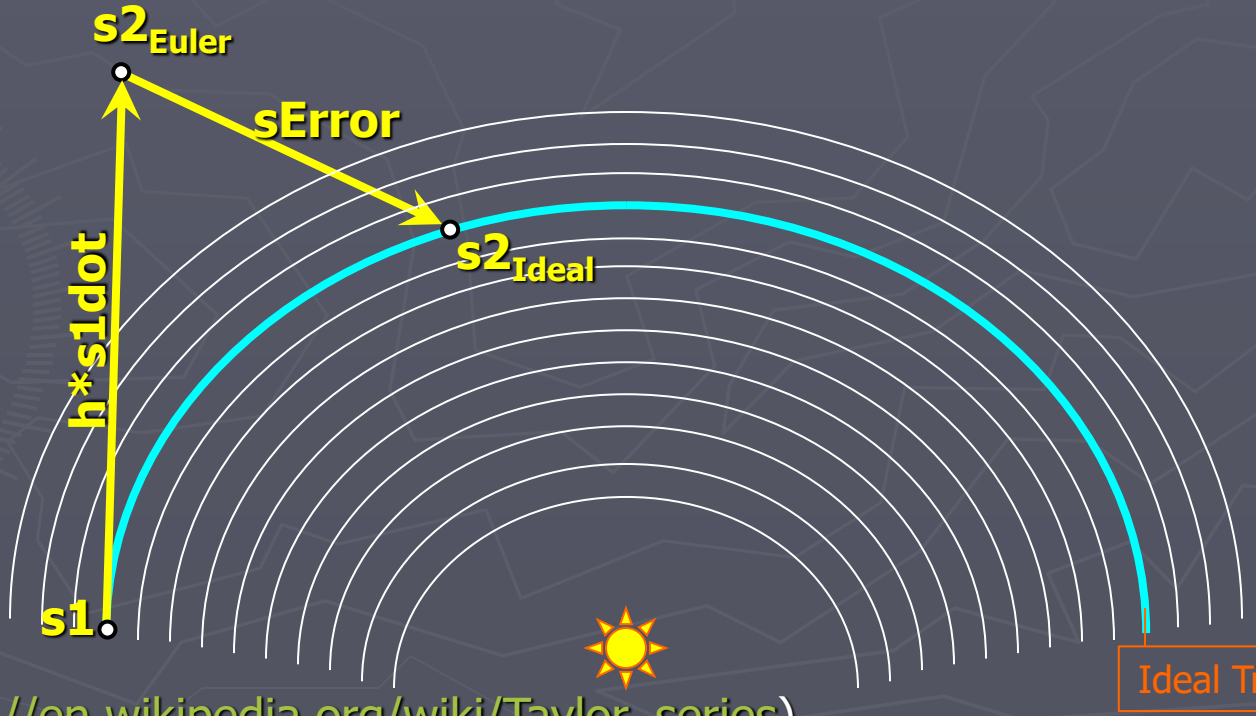
- State-space 'ideal 'trajectory' T() plots **s** over time:
  - Usually **CAN'T** compute it analytically(too interdependent)
  - But we **CAN always** find it's 1st time-derivative (**s1dot**)
- "Solver Error?" (ideal – computed)…. or → → →



(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Solver Error: a State-Space vector

- State-space 'ideal 'trajectory' T() plots **s** over time:
  - Usually **CAN'T** compute it analytically(too interdependent)
  - But we **CAN always** find it's 1st time-derivative (**s1dot**)
- Solver error vector: sError = ($s2_{ideal}$ – $s2_{solver}$)



(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

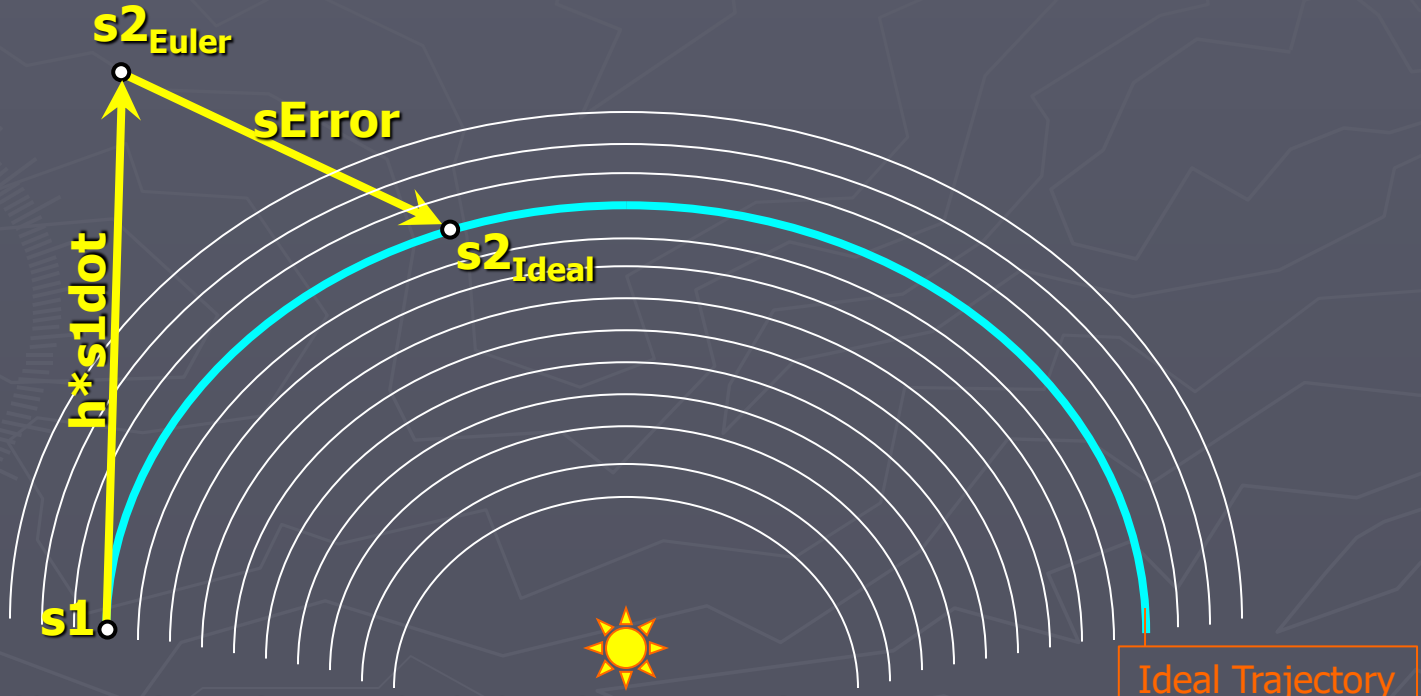
Solver Error

- State-space 'idea
  - Usually **CAN'T** c
  - But we **CAN** alw
- Solver error vector: $sError = (s2_{ideal} - s2_{solver})$

!?! WHAT ?!?

we don't know $s2_{ideal}$!

How could we ever find sError?



(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Answer: Taylor Series Expansion

1-D Taylor: for any smooth, single-variable function f(x),

- **if** we know: the fcn's value & all derivatives at just one point 'a' :
  f(a), f'(a), f''(a) f'''(a), etc.
- **Then** the function's value at any (nearby) point (a+x) is:

$$f(a+x) = f(a) + f'(a)x + \frac{f''(a)}{2!}x^2 + \frac{f'''(a)}{3!}x^3 + \cdots$$

- Higher Derivatives usually smaller, AND contribute less -- use only the first derivative?
  Or only the first 2 derivatives?
  ➔ *residual error* == 'Higher-Order Terms' or 'HOT()':

- Replace 'a' with current state (s1) and 'x' with timestep h:

f(a+h) = f(a) + f'(a)*h + f''(a) (h^2/2) + HOT3(a+h)

(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Recall: Taylor Series Expansion

- Taylor Series for multi-variable scalar functions? Yikes!
  Uses *ALL* the partial derivatives: time, pos, vel, force, etc

$$f(x_1, \cdots, x_d) =$$

$$= \sum_{n_1=0}^{\infty} \cdots \sum_{n_d=0}^{\infty} \frac{(x_1 - a_1)^{n_1} \cdots (x_d - a_d)^{n_d}}{n_1! \cdots n_d!} \left(\frac{\partial^{n_1+\cdots+n_d} f}{\partial x_1^{n_1} \cdots \partial x_d^{n_d}} \right) (a_1, \ldots, a_d)$$

- State **s**: multi-variable **VECTOR** fcn. An *exact* Taylor Series?
  - **a nearly-hopeless mess!** …lots of nonsensical partials…
    - higher derivatives become matrices and tensors
    - We don't know them exactly and (almost always) we can't find them exactly
    - AH! But we can still describe it well as **vector time-series**, a 1-D case:

$$f(a+x) = f(a) + f'(a)x + \frac{f''(a)}{2!}x^2 + \frac{f'''(a)}{3!}x^3 + \cdots$$

s2 = s1 + (h)*s1dot +
(h²/2)* s1dot2 + (h³/6)*s1dot3 + … + (h^N/N!)*s1dotN

(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Recall: Taylor Series Expansion

- Taylor Series for state **s** after a timestep h:
  a sum of vectors, quite similar to 1-D case:

$$f(a+x) = f(a) + f'(a)x + \frac{f''(a)}{2!}x^2 + \frac{f'''(a)}{3!}x^3 + \cdots$$

s2 = s1 + (h)*s1dot +
(h^2/2)* s1dot2 + (h^3/6)*s1dot3 + … + (h^N/N!)*s1dotN

- You only have the first derivative? the s1dot?
  Or only the first 2 derivatives, the s1dot and s1dot2 ?
  →then call the rest the 'Higher-Order Terms' or 'HOT():

s2 = s1 + h*s1dot + HOT2(s1,h)

(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Recall: Taylor Series Expansion

- Taylor Series for State **s** after a timestep h:
  a sum of vectors, similar to 1-D case:

$$f(a+x) = f(a) + f'(a)x + \frac{f''(a)}{2!}x^2 + \frac{f'''(a)}{3!}x^3 + \cdots$$

s2 = s1 + (h)*s1dot +
(h²/2)* s1dot2 + (h³/6)*s1dot3 + … + (h^N/N!)*s1dotN

- Have only the first derivative s1dot?
  Or only the first 2 derivatives s1dot, s1dot2 ?
  → call the error the 'Higher-Order Terms' or 'HOT():

s2 = s1 + h*s1dot + HOT2(s1,h)

Euler Solver (or 'explicit' solver)

Solver Error

Midpoint Method: Quadratic fit!

- Taylor series with the 2nd time-deriv. vector fits a *parabola* to our state-space trajectory

$$s2 = s1 + h*s1dot + (h^2/2)s1dot2 + HOT3(s1,h)$$

- *We don't KNOW* **s1dot2**, but we can approximate it using 1st derivative at 'nearby' point sM:
- By Euler method, find 'midpoint' sM a half-step past s1:

$$sM = s1 + (h/2)*s1dot$$

- Use dotFinder(sM) to find sMdot, and then approximate:

$$s1dot2 = \lim_{h\to 0} \frac{sMdot - s1dot}{h/2} \approx \frac{sMdot - s1dot}{h/2}$$

(Why 'h/2'?
We found sM by moving a half-step forward from **s1**. We find the limit as we shrink movement to zero…)

(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Midpoint Method: Quadratic fit!

Again:

$$s1dot2 = \lim_{h\to 0} \frac{sMdot - s1dot}{h/2} \approx \boxed{\frac{sMdot - s1dot}{h/2}}$$

- Substitute into Taylor Series and simplify:

  s2 = s1 + h*s1dot +(h²/2)s1dot2 + HOT3(s1,h)

  = s1 + h*s1dot +(h²/2)(sMdot-s1dot)(2/h)+HOT3(s1,h)

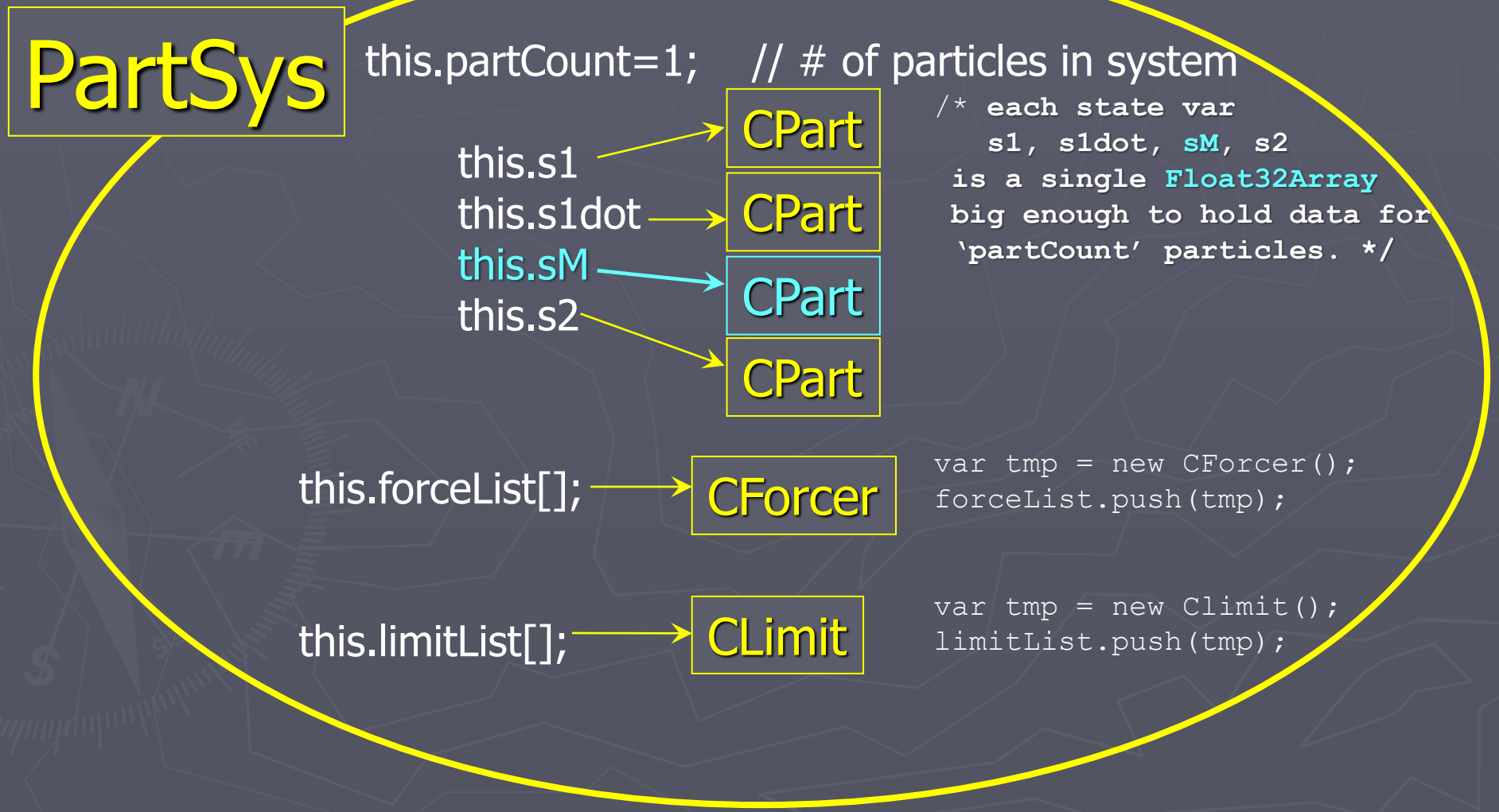
  = s1 + h*s1dot + h*sMdot – h*s1dot + HOT3(s1,h)

s2 = s1 + h*sMdot + HOT3(s1,h)

- Simple!
  0) (create new state 'sM' in your particle system)
  1) Half-step with Euler method finds midpoint sM
  2) DotFinder() finds the sM first derivative sMdot
  3) Euler Method with sMdot instead of s1dot

UPDATED: Master Plan

- Add a 'midpoint' state vector sM (and put sMdot in s1dot)



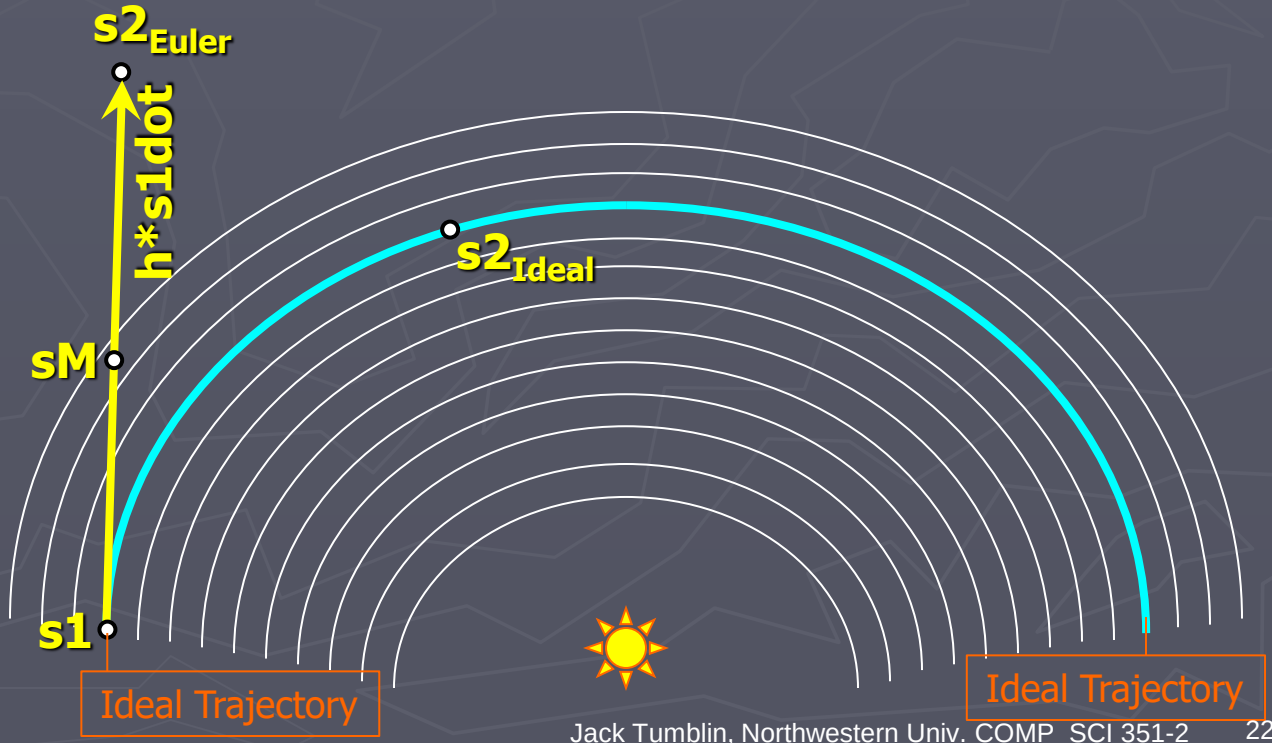
Look Again : Better Solvers

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:

  s2 = s1 + h*s1dot

- Method 2: 'Midpoint' (aka Runge-Kutta RK2); quadratic fit for less error;
- START: Find 'midpoint' of Euler method: **sM = s1 + (h/2)*s1dot**



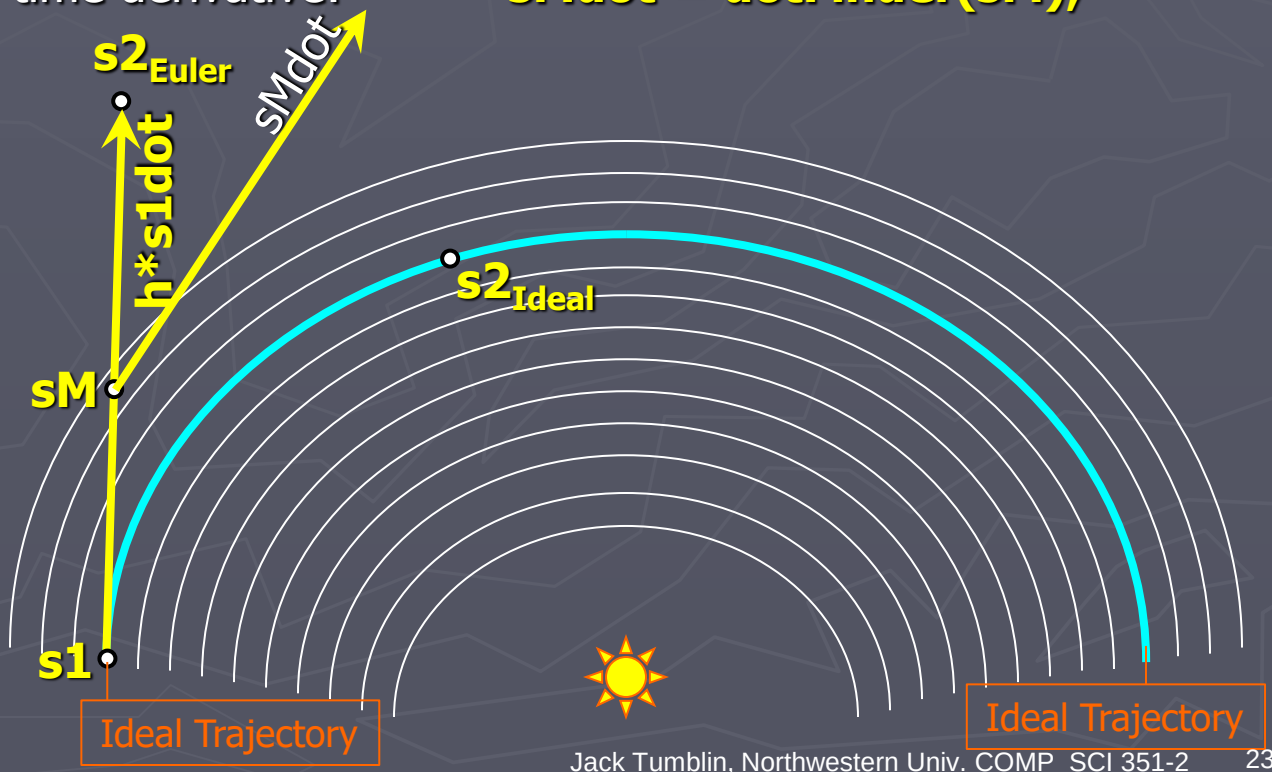
Look Again : Better Solvers

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:

  s2 = s1 + h*s1dot

- Method 2: 'Midpoint' (aka Runge-Kutta RK2); quadratic fit for less error;
- START: Find 'midpoint' of Euler method: **sM = s1 + (h/2)*s1dot**
- NEXT: Find 'midpoint' time derivative: **sMdot = dotFinder(sM);**



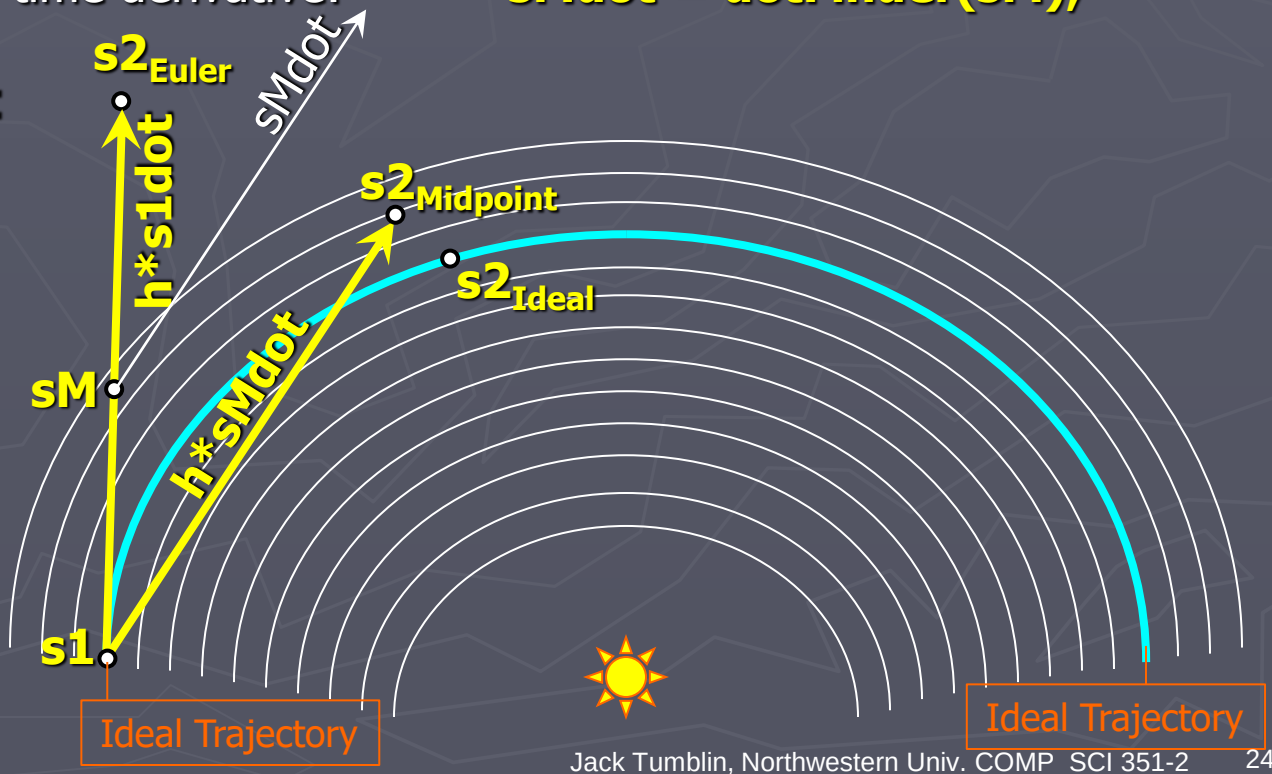
Look Again : Better Solvers

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver:
  s2 = s1 + h*s1dot

- Method 2: 'Midpoint' (aka Runge-Kutta RK2); quadratic fit for less error;
- START: Find 'midpoint' of Euler method: **sM = s1 + (h/2)*s1dot**
- NEXT: Find 'midpoint' time derivative: **sMdot = dotFinder(sM);**
- FINISH:
  Add to s1 to find s2:

s2 = s1 + h*sMdot



Explicit Methods use ***Forward*** Timesteps

How can we find **s2**, the state at next timestep?

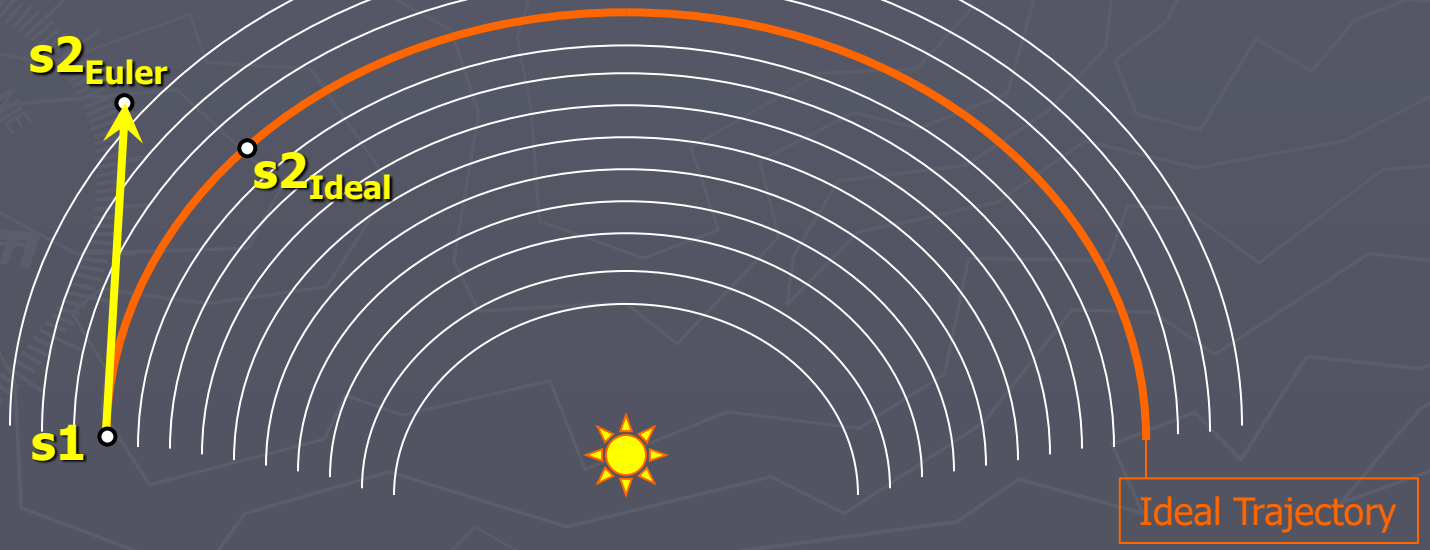
- 'Euler' and 'Midpoint' methods are 'Explicit' Solvers:

$$s2_{ideal} = s1 + h*s1dot + \ldots + HOT()$$

$$s2_{euler} = s1 + h*s1dot; \qquad s_{err} = HOT2(s1,h)$$

$$s2_{midpt} = s1 + h*sMdot; \qquad s_{err} = HOT3(s1,h)$$

- **START** at current state **s1**, go forward in time (timestep **h**) to find the next state **s2**.

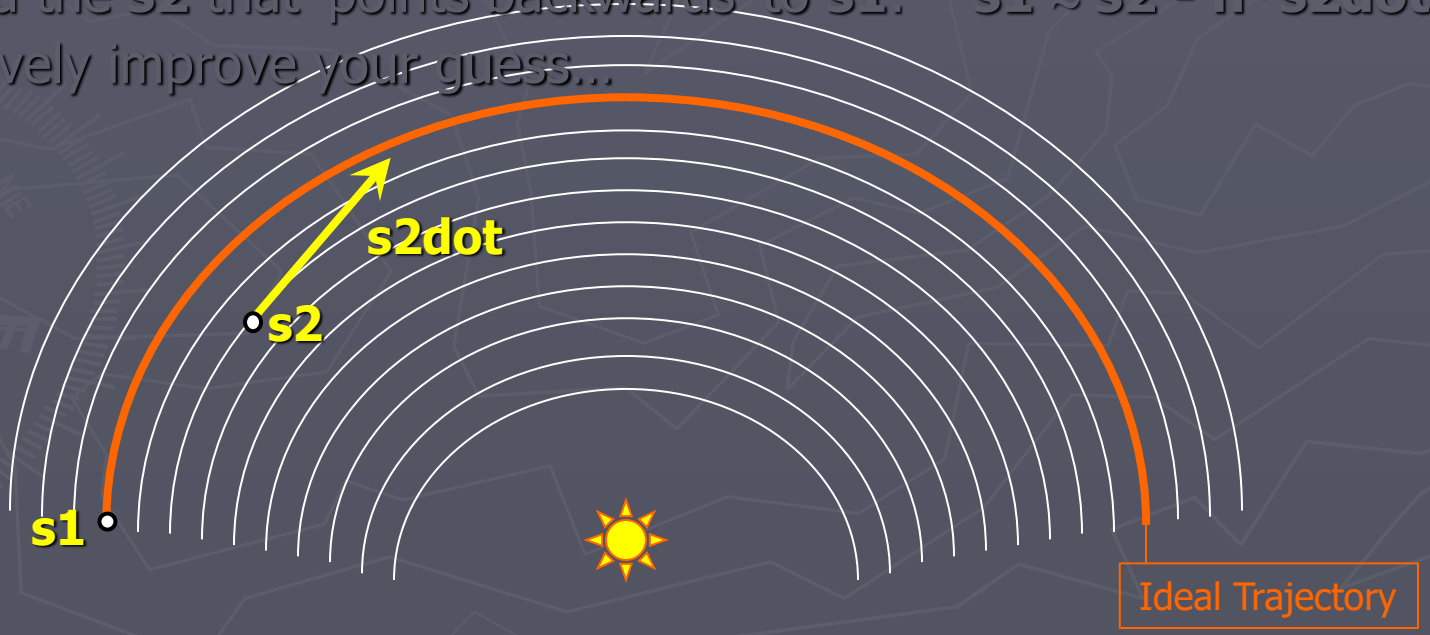


New Idea: *Time-Reversed* Methods

How can we find **s2**, the state at next timestep?

- 'Euler' and 'Midpoint' methods are 'Explicit' Solvers:
  (start from s1, go forward in time to find s2)
- New idea: 'Implicit' or 'Inverse' or 'Backwind' methods:
  1) Presume magic: somehow ***guessed*** a good **s2**, & computed **s2dot**
  2) Go 'backwards in time' from s2: find s2 - h*s2dot
  3) Find the s2 that 'points backwards' to s1: s1 ≈ s2 - h*s2dot

and iteratively improve your guess...

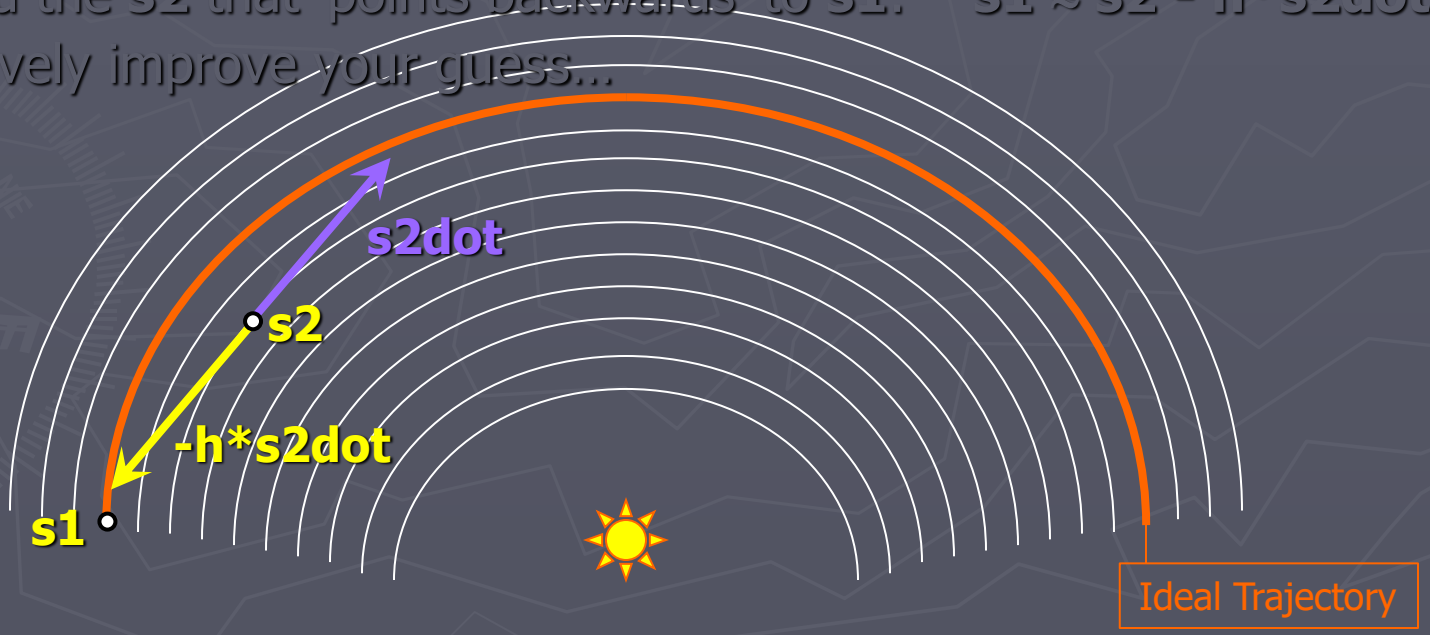


New Idea: *Time-Reversed* Methods

How can we find **s2**, the state at next timestep?

- 'Euler' and 'Midpoint' methods are 'Explicit' Solvers:
  (start from s1, go forward in time to find s2)
- New idea: 'Implicit' or 'Inverse' or 'Backwind' methods:
  1) Presume magic: somehow ***guessed*** a good **s2**, & computed **s2dot**
  2) Go 'backwards in time' from s2: find **s2 - h*s2dot**
  3) Find the s2 that 'points backwards' to s1: $s1 \approx s2 - h*s2dot$

and iteratively improve your guess...



New Idea: *Time-Reversed* Methods

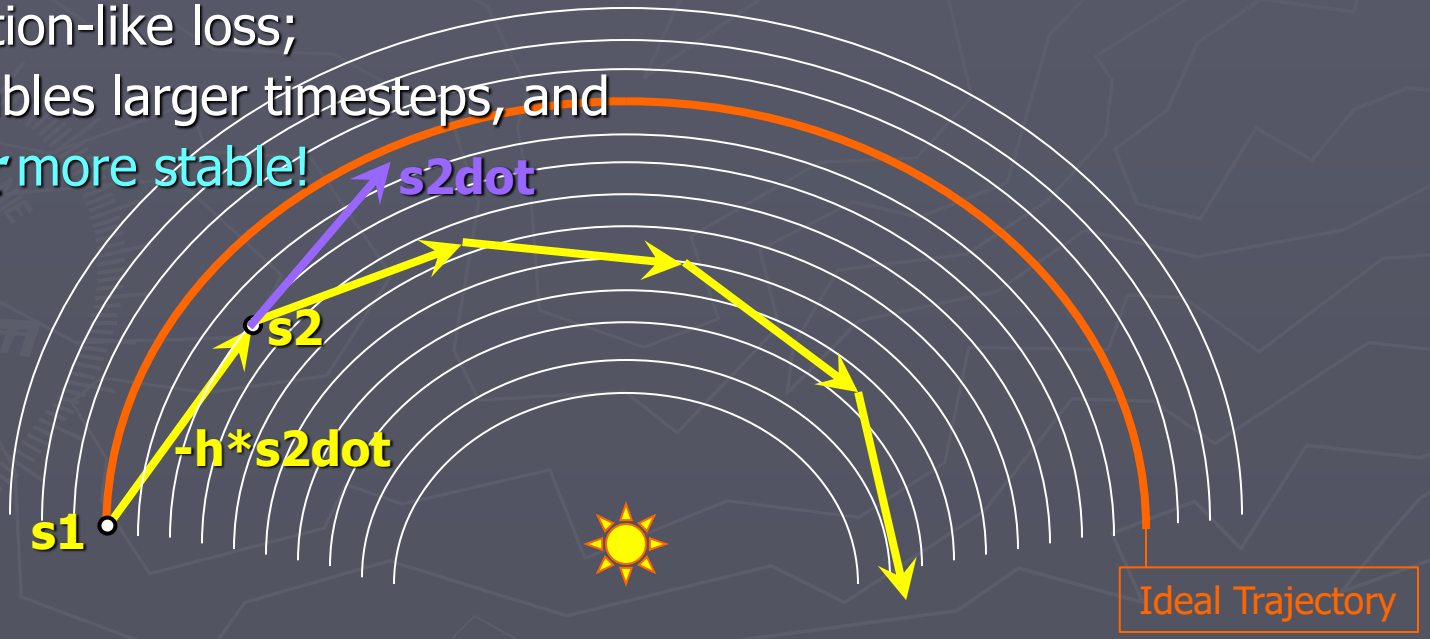
How can we find **s2**, the state at next timestep?

- 'Euler' and 'Midpoint' methods are 'Explicit' Solvers:
  (start from s1, go forward in time to find s2)
- New idea: 'Implicit' or 'Inverse' or 'Backwind' methods:
  1) Presume magic: somehow ***guessed*** a good **s2**, & computed **s2dot**
  2) Go 'backwards in time' from s2: find **s2 - h*s2dot**
  3) Find the **s2** that 'points backwards' to **s1**: **s1 ≈ s2 - h*s2dot**

and iteratively improve your guess…

New Idea: *Time-Reversed* Methods

- New idea: 'Implicit' or Inverse, 'Backwind' methods:
  1) Attempt magic: somehow ***guessed*** a good **s2**, & computed **s2dot**
  2) Go 'backwards in time' from s2: find **s2 - h*s2dot**
  3) Find best **s2** value whose previous state is **s1**: **s1 ≅ s2 - h*s2dot**
- **RESULT:** Benign, Lossy Errors & ***MUCH Better stability!***
  - errors ***lose*** energy & momentum ($mv^2/2$ & mv) from 'reversed time';
  - friction-like loss;
  - Enables larger timesteps, and
  - ***Far*** more stable!



'Implicit' or 'Backwind' Methods

- **s2 = s1 + h*s2dot** !?! 'Attempt Magic' ?!?!?
  How could we ever *find* **s2** if
  we must *compute* it from its own derivative **s2dot**?
  ("What change-in-state can we apply to s1 (e.g. add h*s2dot ?) that will give us a new state (s2), but an s2 whose time-derivative s2dot is the change-in-state that we want?" → ANS: Root finding!
  Algebraic solutions exist ONLY for boringly simple systems: e.g. <= quadratic (HOT3==0)
  ANSWER: iterative-refinement strategies
  - a vector-valued form of *root-finding* *thus* the fcn name '**solver()**'

  - Notation: let SUPERSCRIPTS denote each iteration, *e.g.* **s2$^{(0)}$, s2$^{(1)}$, s2$^{(2)}$, ... s2$^{(N)}$** == 0th $_{(initial)}$, 1st, 2nd, … N-th estimates of the vector **s2**.
  - As always:
    - h==timestep, **s1**==current state, **s2**== next state;
    - s1dot, s1dot2, s1dot3…s1dotN are 1st,2nd,3rd,…Nth derivs of **s1**,

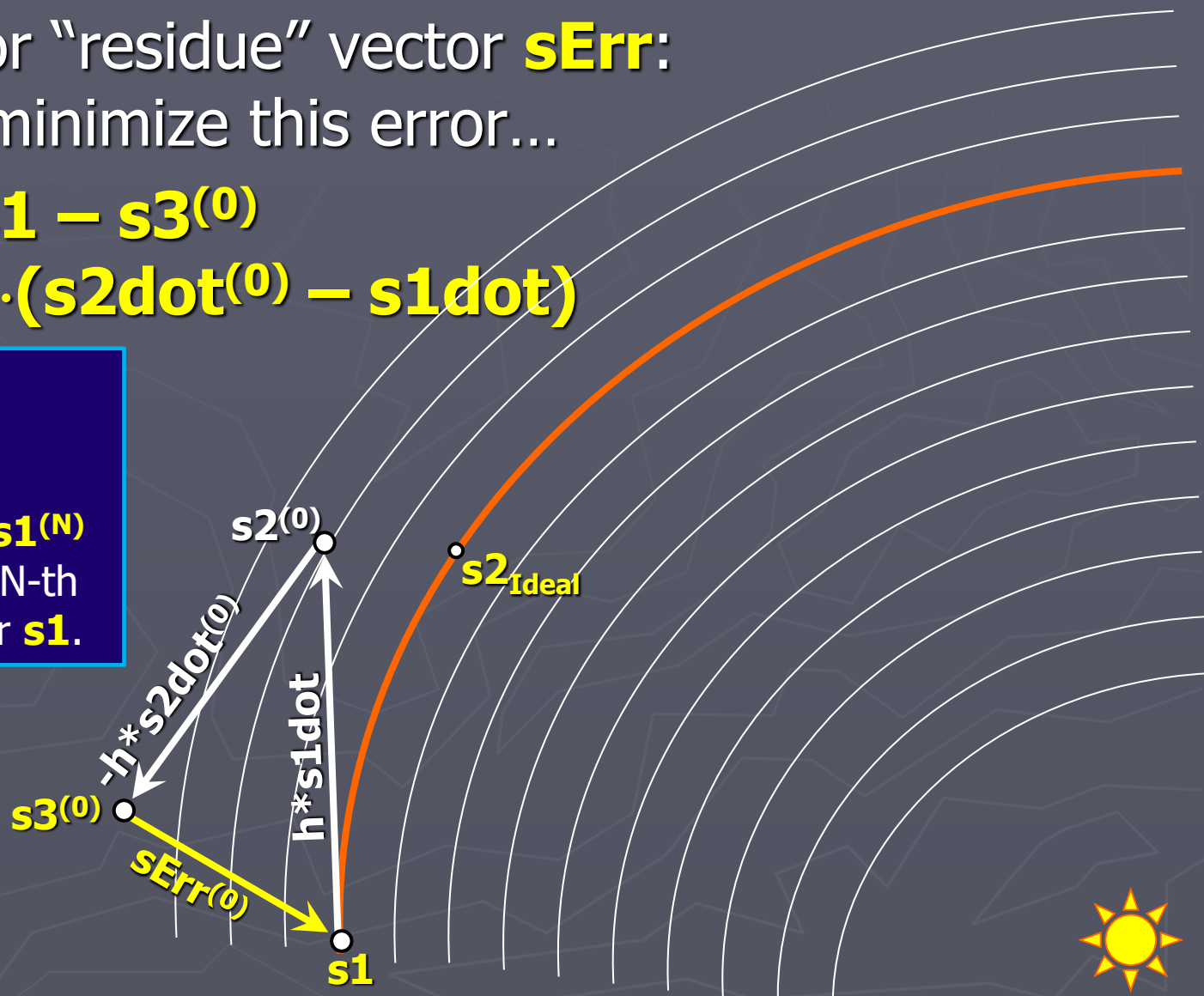
---

  - Example: **s2dot3$^{(2)}$** is the 2nd estimate of s2's 3rd time-derivative)

Iterative Backwind Method

- 3) Find error "residue" vector **sErr**: Iteratively minimize this error…
- **$sErr^{(0)} = s1 - s3^{(0)}$**
  $= h \cdot (s2dot^{(0)} - s1dot)$

Notation:
superscripts
denote each iteration:
$s1^{(0)}$, $s1^{(1)}$, $s1^{(2)}$, … $s1^{(N)}$
== 0^{th} (initial), 1^{st}, 2^{nd}, … N-th
estimates of the vector **s1**.

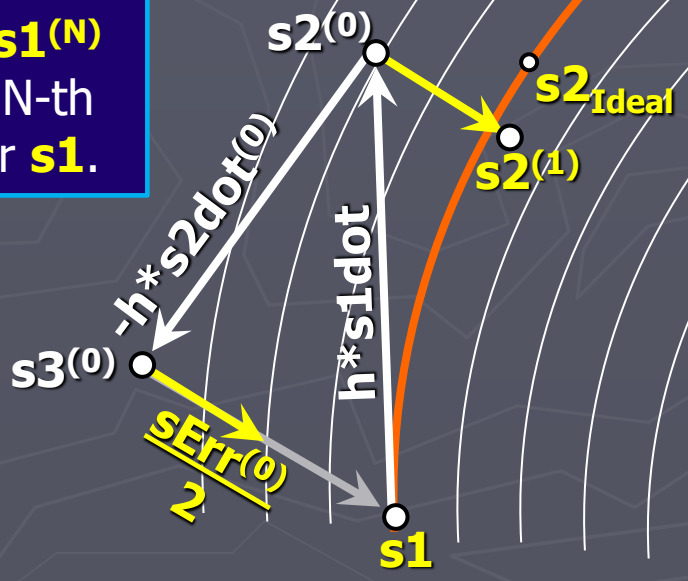


Iterative Backwind Method

- 4) 'Correct' the error. How?
  Try this: Correct HALF the error
  to make a new, better **s2** estimate:

$$s2^{(1)} = s2^{(0)} + 0.5*sErr^{(0)}$$

Notation:
superscripts
denote each iteration:
$s1^{(0)}, s1^{(1)}, s1^{(2)}, \ldots s1^{(N)}$
== 0^{th} (initial), 1^{st}, 2^{nd}, … N-th
estimates of the vector **s1**.



Iterative Backwind Method

- 5) STOP NOW? Pick one…
  - YES: use estimate **$s2^{(1)}$** as our next state **s2**
  - NO: further refine, starting from **$s2^{(1)}$:**
- 2) Find derivative $s2dot^{(1)}$
  - $s2dot^{(1)} \leftarrow dotFinder(s2^{(1)})$
    $s3^{(1)} = s2^{(1)} - h*s2dot^{(1)}$
- 3) Find residue $sErr^{(1)}$
  - $sErr^{(1)}=(s1 - s3^{(1)})$
- 4) Correct half the error:
  - $s2^{(2)} = s2^{(1)} + 0.5*sErr^{(1)}$

Iterative Backwind Method

Surprise!

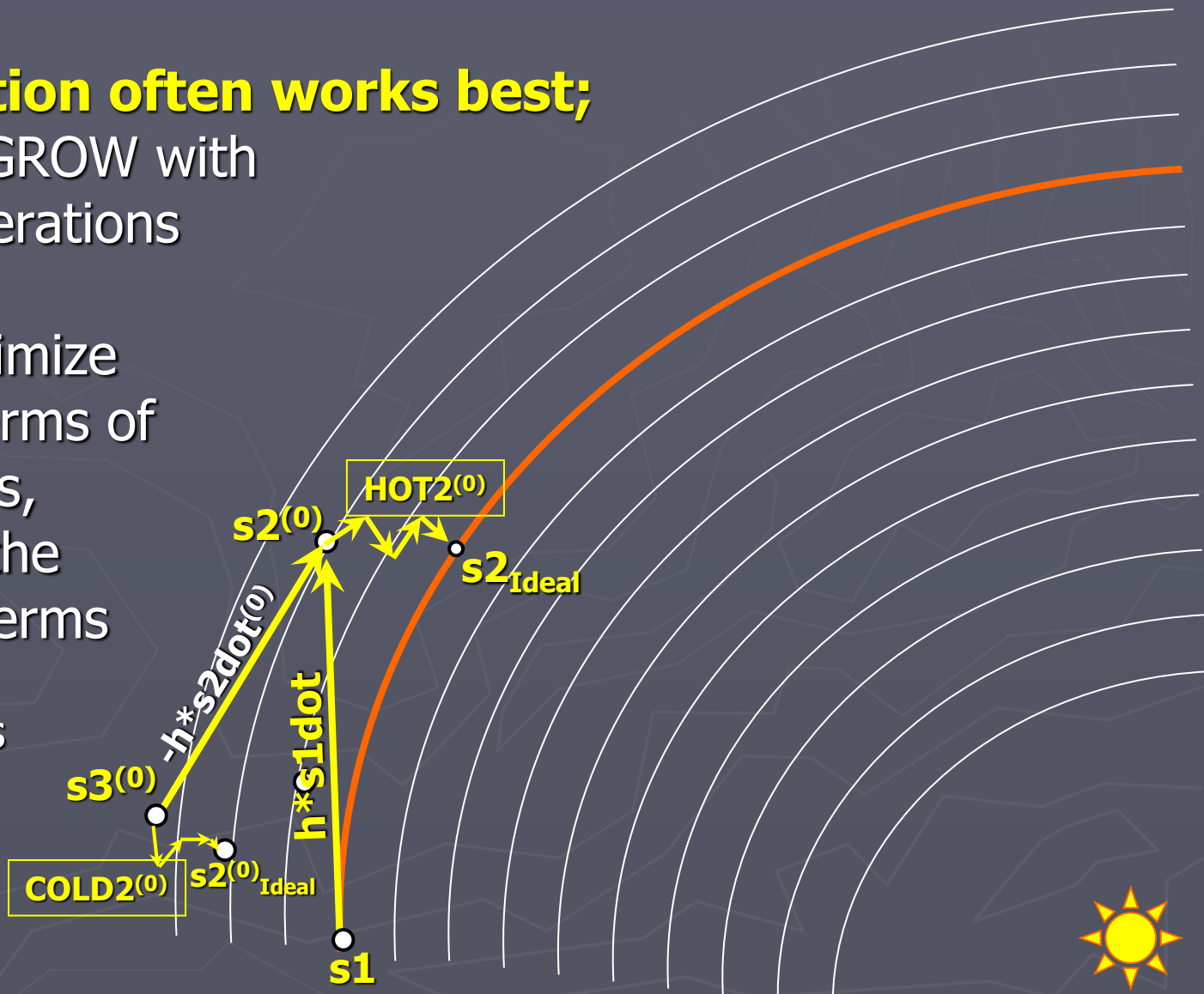
--**First iteration often works best;**

--Errors can GROW with excess iterations

BECAUSE:

GOAL is to minimize Error for ***ALL*** terms of the Taylor Series, including all of the **H**igher-**O**rder **T**erms (**HOT**).

Backwind iterations ignore HOTs: uses the reversed-time error: "COLDs" ;-)



Iterative Backwind Method

Why is one iteration is usually enough? **Part 1:**

All Forward methods (Euler, Midpoint, Runge-Kutta) apply the first few Taylor Series terms:

s2 = s1 + h*s1dot + (h^2/2)s1dot2 + … + HOT(s1,h)

HOT() == all ***forward*** error. For Euler:

$$HOT2(s1,h) = \left(\frac{h^2}{2}\right)s1dot2 + \left(\frac{h^3}{6}\right)s1dot3 + \left(\frac{h^4}{24}\right)s1dot4 + \ldots + \left(\frac{h^N}{N!}\right)s1dotN$$

Forward: h>0 → h^2,h^3,h^4,h^5,h^6,h^7… ALL >0,

Reverse: h<0 → h^2, h^4, h^6, …>0, but
h^3, h^5, h^7 …<0 !!!

adding Fwd+Rev *~cancels* the odd terms.

TRY IT: Starting state sA
Euler 'forward' state sF
Euler 'backward' state sB …

NOTE forward HOT != inverse HOT
(inverse HOT: let's call it COLD)

Iterative Backwind Method

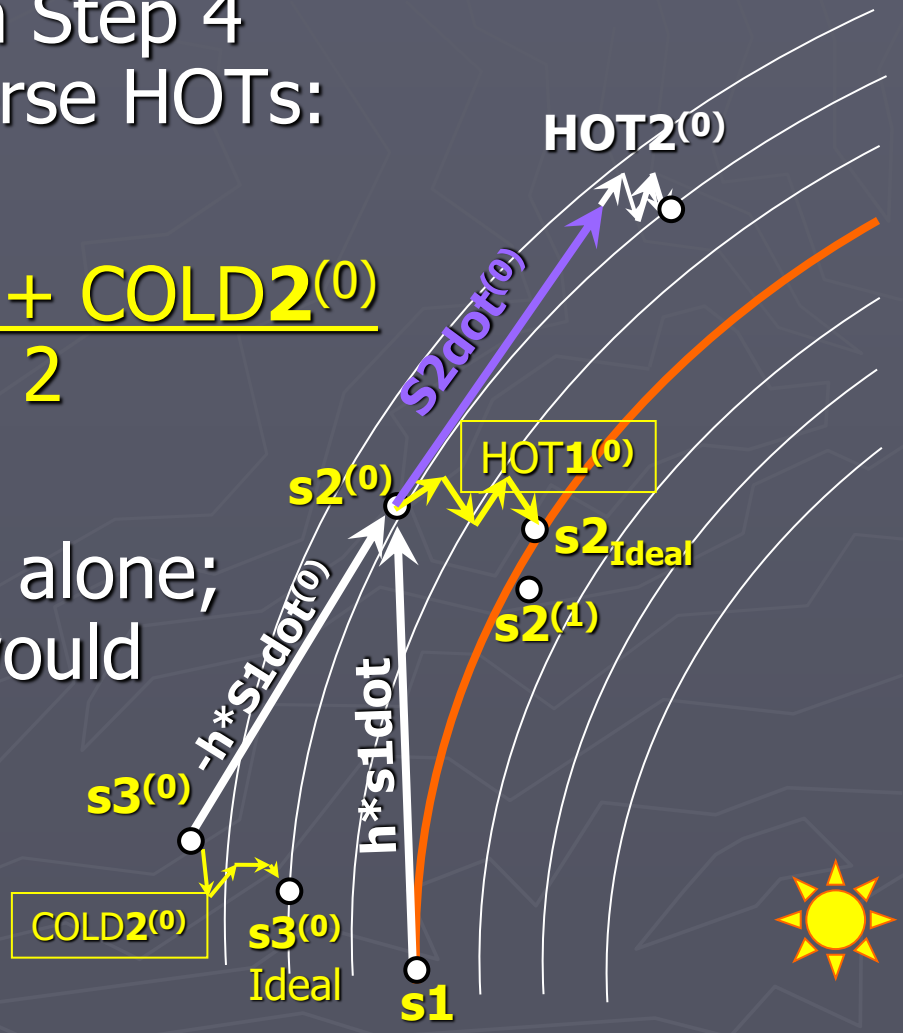
Why one iteration is usually enough? **Part II**

The half 'residue' removed in Step 4 combined both fwd & reverse HOTs:

$$\frac{sErr}{2} = \frac{s1\text{-}s3^{(0)}}{2} = \frac{HOT\mathbf{1}^{(0)} + COLD\mathbf{2}^{(0)}}{2}$$

And it approximates $HOT\mathbf{1}^{(0)}$ alone; perfect $HOT\mathbf{1}^{(0)}$ removal would reach to $\mathbf{s2}_{Ideal}$:

$$s2^{(0)} + HOT\mathbf{1}^{(0)} = s1_{Ideal}$$



Higher-Order Implicit Methods

Summarize our current iterative 'backwind' Euler method:

- Forward Euler Method to find s2 estimate $s2^{(0)}$; then
- Backward Euler Method to find s3 estimate $s3^{(0)}$; then
- Find residue: $sErr = s1 - s3^{(0)}$; then
- Remove half-residue: $s2^{(1)} = s2^{(0)} - sErr/2$.

We already learned these 'forward' methods:

- Euler Method: straight-line arcs approximate ideal state-space trajectories
- Midpoint Method: parabolic arcs approximate ideal state-space trajectories
- Runge-Kutta RK-N: N^{th} degree polynomial arcs approximate ideal state-space trajectories

Higher-Order Polynomial fittings

- **Higher-Order Backwind Method?** Just make forward, backwards steps with Better-than-Euler fittings! Only the 'backwards' part may not be obvious yet...

Midpoint (Quadratic) Inverse Method

Try this higher-order implicit or 'backwind' method:

- **Forward** Midpoint Method from s1 finds s2 estimate $s2^{(0)}$; then
- **Backward** Midpoint Method from $s2^{(0)}$ finds $s3^{(0)}$; then
- Find residue: sErr = $s3^{(0)} - s1$
- Remove half-residue: $s2^{(1)} = s2^{(0)} - sErr/2$. Done!

Details:

- **Forward** Midpoint Method for first-estimate $s2^{(0)}$:
  - Given s1, find midpoint by Euler half-step: $sM^{(0)} = s1 + (h/2)*s1dot$
  - Midpoint derivative for Euler-like full-step: $s2^{(0)} = s1 + h * sM^{(0)}dot$
- Then **Backward** Midpoint to find & fix estimated error:
  - Half-step with $s2^{(0)}$ derivative $s2^{(0)}dot$ to find BACKWARD midpoint: $sM^{(1)} = s2^{(0)} - (h/2)*s2^{(0)}dot$
  - BACKWARD midpoint derivative $sM^{(1)}dot$ for an Euler-like full-step: $s3^{(1)} = s2^{(0)} - h * sM^{(1)}dot$
  - Correct half the residue error sErr to for implicit midpoint result:
    $sErr = s3^{(1)} - s1;$
    $s2^{(1)} = s2^{(0)} - sErr/2;$

More rigorous approaches

- Widely used Implicit Method for Particle-Based Cloth:
  from "Large steps in cloth Simulation" (Witkin & Baraff, SIGGRAPH 1998).

$$\begin{bmatrix} \Delta pos \\ \Delta vel \end{bmatrix} = h \begin{bmatrix} vel_0 + \Delta vel \\ M^{-1}\left(\Sigma F_0 + \frac{\partial \Sigma F}{\partial\ pos} \Delta pos + \frac{\partial \Sigma F}{\partial\ vel} \Delta vel \right) \end{bmatrix}$$

- For these, DotMaker() is not enough:
  - need time-derivatives AND pos. AND net force derivatives
  - 2nd order sparse matrix formulation:
    Conjugate-Gradient Solver
  - Suitable for complex constraints (self-collisions, etc)
  - Not bad; IF(and only if) you're comfortable w/ these tools!
  - Concise overview: (Macri, Intel Software Network)

    http://software.intel.com/en-us/articles/simulating-cloth-for-3d-games/
    http://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1

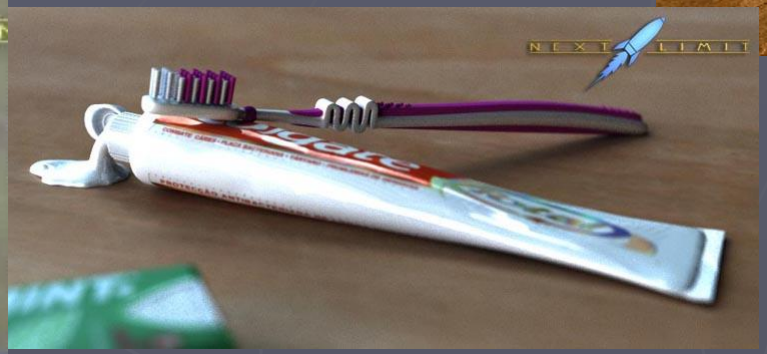
Verlet Integration: A Game Winner!

- MORE rigorous, MORE accurate, and yet
- MORE ingenious, more stable, much faster, much more compact, no iteration,
- Orderly, plausible losses ('ghost friction')
- **'Semi-implicit'** solver
  - Easy to describe,
  - Subtle/More complex to explain/understand
- HUGELY popular in games & physics engines
- See: http://en.wikipedia.org/wiki/Verlet_integration

END

END

Implicit Solvers & Verlet Method —F



Jack Tumblin
COMP_SCI 351-2

REVIEW: Explicit Methods: Make Forward Timesteps

START at current state **s1**, go forward in time (timestep **h**) to find the next state **s2**.

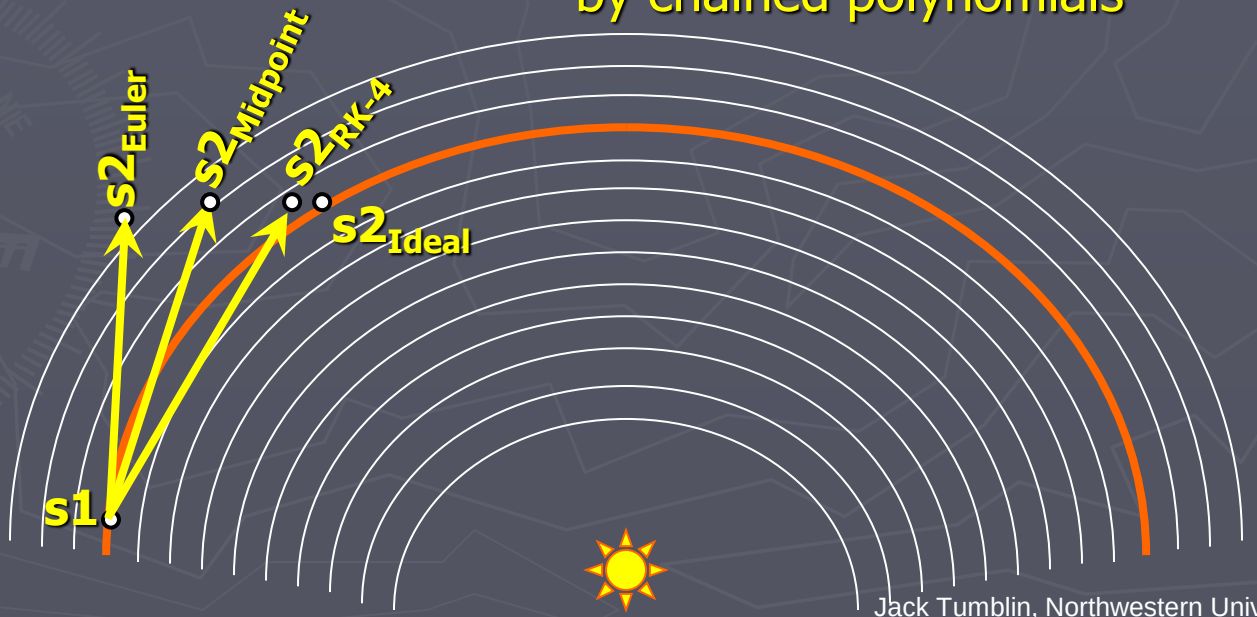
- 'Euler' and other 'Explicit' Solvers:

  Taylor Series: $\mathbf{s2_{ideal} = s1 + h*s1dot + \ldots + HOT()}$

  Euler: **s2 = s1 + h*s1dot**

  Midpoint: **s2 = s1 + h*sMdot**
  where **sM = s1 +(h/2)s1dot;**

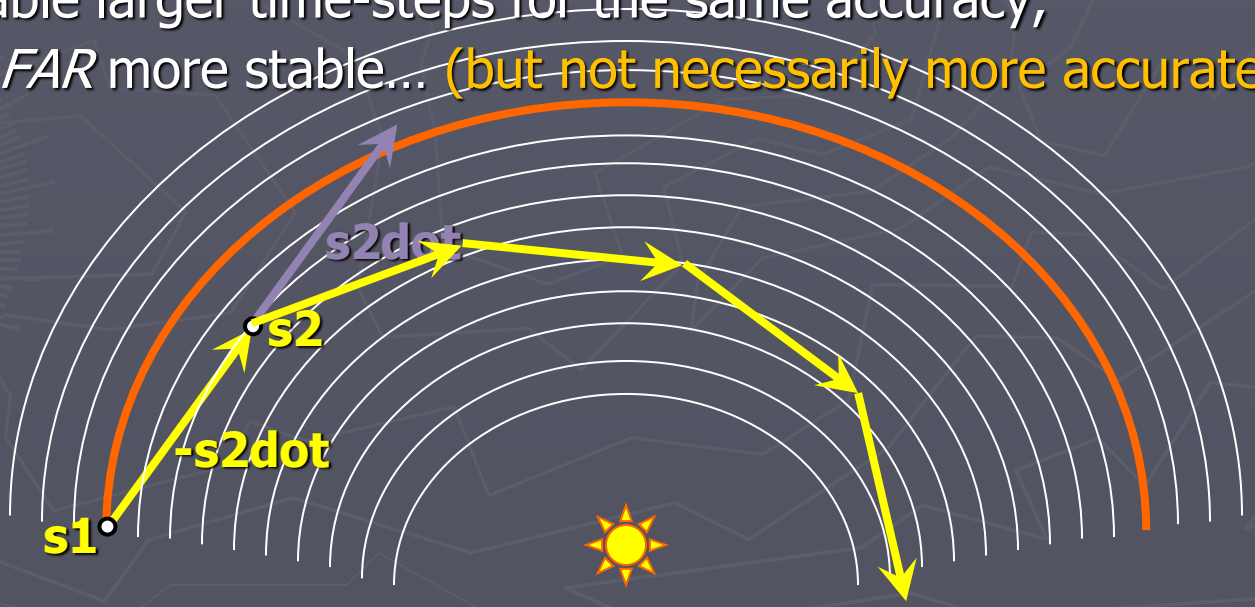
  'Runge-Kutta': **s2** = Higher-order Taylor-Series estimates by chained polynomials



REVIEW: Implicit Methods:

Time-Reversals at Each Step

- 'Implicit' or 'Inverse' or 'Backwind' methods:
  1) Attempt magic: Make a good estimate of **s2**; then find its **s2dot**.
  2) Go 'backwards in time' from s2: e.g. find **s3 = s2 - h*s2dot ≈s1**
  3) Find the **s2** value whose time-reversal fits **s1** well:
- **RESULT:** More-benign, mostly-lossy errors: better stability!
  - errors ***lose*** energy (E=||mv||) due to reversed time; friction-like…
  - Can enable larger time-steps for the same accuracy,
  - *Can be FAR* more stable… (but not necessarily more accurate)



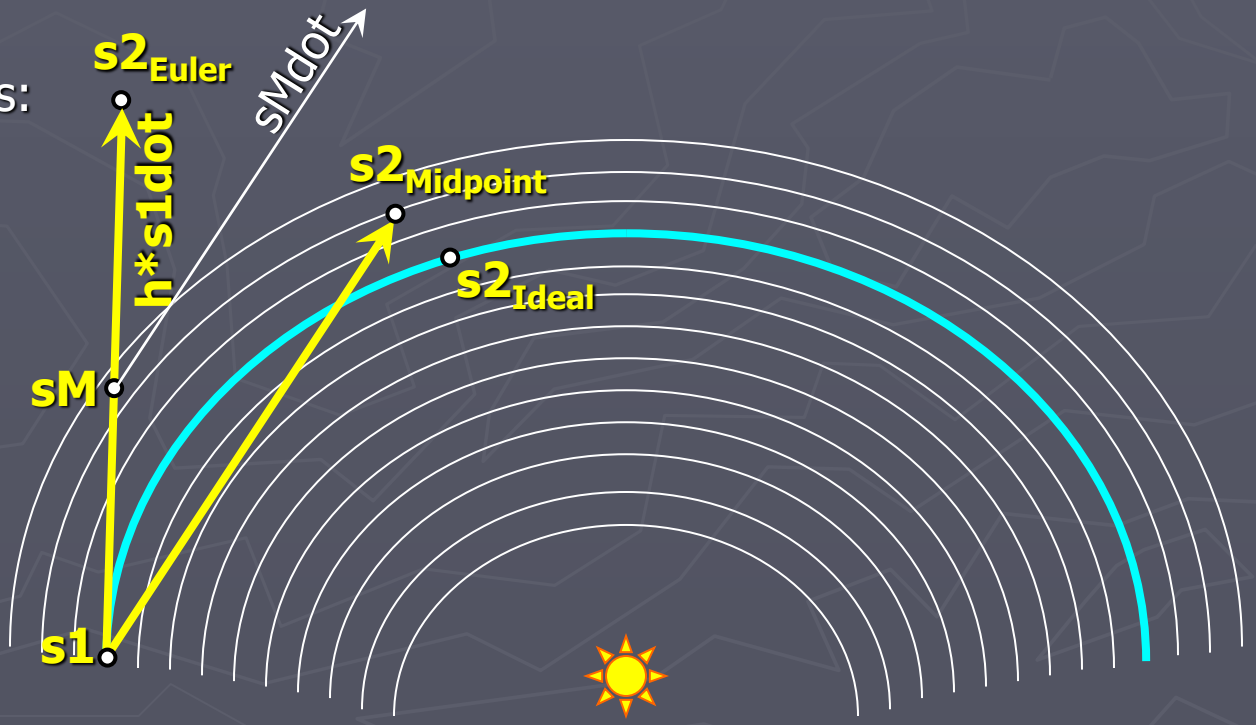
LIST of Better-&-Better Solvers(1)

How can we find **s2**, the state at next timestep?

- Method 1: 'Euler' or 'Explicit' Solver: easy, but prone to energetic errors

  s2 = s1 + h*s1dot

- Method 2: 'Midpoint' (aka Runge-Kutta RK2) less error; still gains energy
- **WANTED:**
  - larger timesteps
  - smaller errors
  - more benign errors: **lose** energy to mimic drag, friction etc.
  - More stable



Euler → Midpoint Method →More?

- Key idea: est. higher derivatives from nearby points
- (linear) Euler Method: **s2 ≅ s1 + h*s1dot**
- (quadratic) Midpoint:
  - 2nd derivative s1dot2 estimated from well-chosen nearby point: sM = s1 + ½h*s1dot

$$s1dot2 = \lim_{h \to 0} \frac{sMdot - s1dot}{h/2} \approx \frac{sMdot - s1dot}{h/2}$$

  - cancels, simplifies quadratic fit to:

s2 ≅ s1 + h*sMdot

- (more?) cleverness for higher-order derivatives?
  YES! the famous Runge-Kutta solutions create them by chained polynomials k1,k2,k3,…

(Wikipedia: http://en.wikipedia.org/wiki/Taylor_series),

Runge-Kutta Methods (RK-1,2,3,…N)

- Done ~1900 by German mathematicians Carl D. T. Runge and Martin W. Kutta
- Quick solution plots for 1st-order ODE problems of *any* dimension; perfect for state-space probs
- Higher-order answers re-use lower-order results
- GIVEN an ODE: $s' = f(t,s)$
  and its initial values t1, s1,
- ESTIMATE the solution ('trace the trajectory')
  s(t1+h) for any h

- (Note that their f() is our dotMaker() function)

More? See: **TEXTBOOK, CHAPTER 7.3** (!good!); online tutorials:
http://people.revoledu.com/kardi/tutorial/ODE/Euler%20Method.htm

You Already know Runge-Kutta 1,2:

Runge-Kutta estimates ODE solutions:

RK-N estimates the first N terms of the Taylor Series;

constructs re-usable 'k' terms from nearby points & f():

- RK-1 (Euler-Cauchy) define: k1 = f(t1,s1) = s1dot

  (Linear): $s(t1 + h) = s2_{ideal} \cong s2 = s1 + h*k1$

- RK-2 ('Midpoint' or 'Heun' or 'Improved Euler')

  define: k2 = f(t1 + ½h, s1 + ½h*k1)= sMdot ≅ s1dot2

  (Quadratic): $s(t1 + h) = s2_{ideal} \cong s2 = s1 + h*k2$

http://people.revoledu.com/kardi/tutorial/ODE/Euler%20Method.htm

New: Runge-Kutta RK-3, RK-4

- RK-3 (Cubic):

  define k3 = f(t1+h, s1 –h*k1 +2h*k2) ≅ s1dot3

  $s(t1 + h) = s2_{ideal} \cong s1 + (h/6)*(k1 + 4*k2 + k3)$

- RK-4 (Quartic) or 'Standard': SURPRISE! Revised k3 term!

  define k33 = f(t1 + ½h, s1+ ½h*k2) ≅ s33dot

  define k4 = f(t1 + h, s1 + h*k33) ≅ s4dot

  $s(t1 + h) = s2_{ideal} \cong s1 + (h/6)*(k1 + 2*k2 + 2*k33 + k4)$

For a beautiful step-by-step tutorial with MS Excel code examples, see:
http://people.revoledu.com/kardi/tutorial/ODE/WhatisODE.htm or skip ahead to:
http://people.revoledu.com/kardi/tutorial/ODE/Euler%20Method.htm

New: Runge-Kutta RK-3, RK-4

- RK-3 (Cubic):

  define k3 = f(t1+h, s1 –h*k1

  s(t1 + h) = $s2_{ideal} \cong$

- RK-4 (Ou ised k3 term!

  k2) ≅ s33dot

  h*k33) ≅ s4dot

  $_{ideal} \cong s1 + (h/6)*(k1 + 2*k2 + 2*k33 + k4)$

! OUR TEXTBOOK HAS AN EVEN BETTER EXPLANATION – SEE Chapter 7.3; pages 120 – 123

For a utiful step-by-step tutorial with MS Excel code examples, see:
http://people.revoledu.com/kardi/tutorial/ODE/WhatisODE.htm or skip ahead to:
http://people.revoledu.com/kardi/tutorial/ODE/Euler%20Method.htm

MORE! "2-step" Adams-Bashforth:

- Another (linear) Explicit Quadratic Method:
- Relies on PREVIOUS state **s0** to estimate **s1dot2**:

  s2 == **NEXT** state we wish to find,
  s1 == **CURRENT** state
  s0 == **PREVIOUS** state

- $\mathbf{s2}_{\text{Ideal}} \cong$

  s2 = **s1** + h***s1dot** + (h/2)(**s1dot-s0dot**)

Simplify: **s2 = s1 +** (3/2)h***s1dot** **-** (h/2)***s0dot**

- Details? See 'Linear Multistep' for ODE solutions: http://en.wikipedia.org/wiki/Linear_multistep_method#Adams.E2.80.93Bashforth_methods

COMPARE TO: **s2 = s1 + s1dot*h + s1dot2*(h^2/2)**

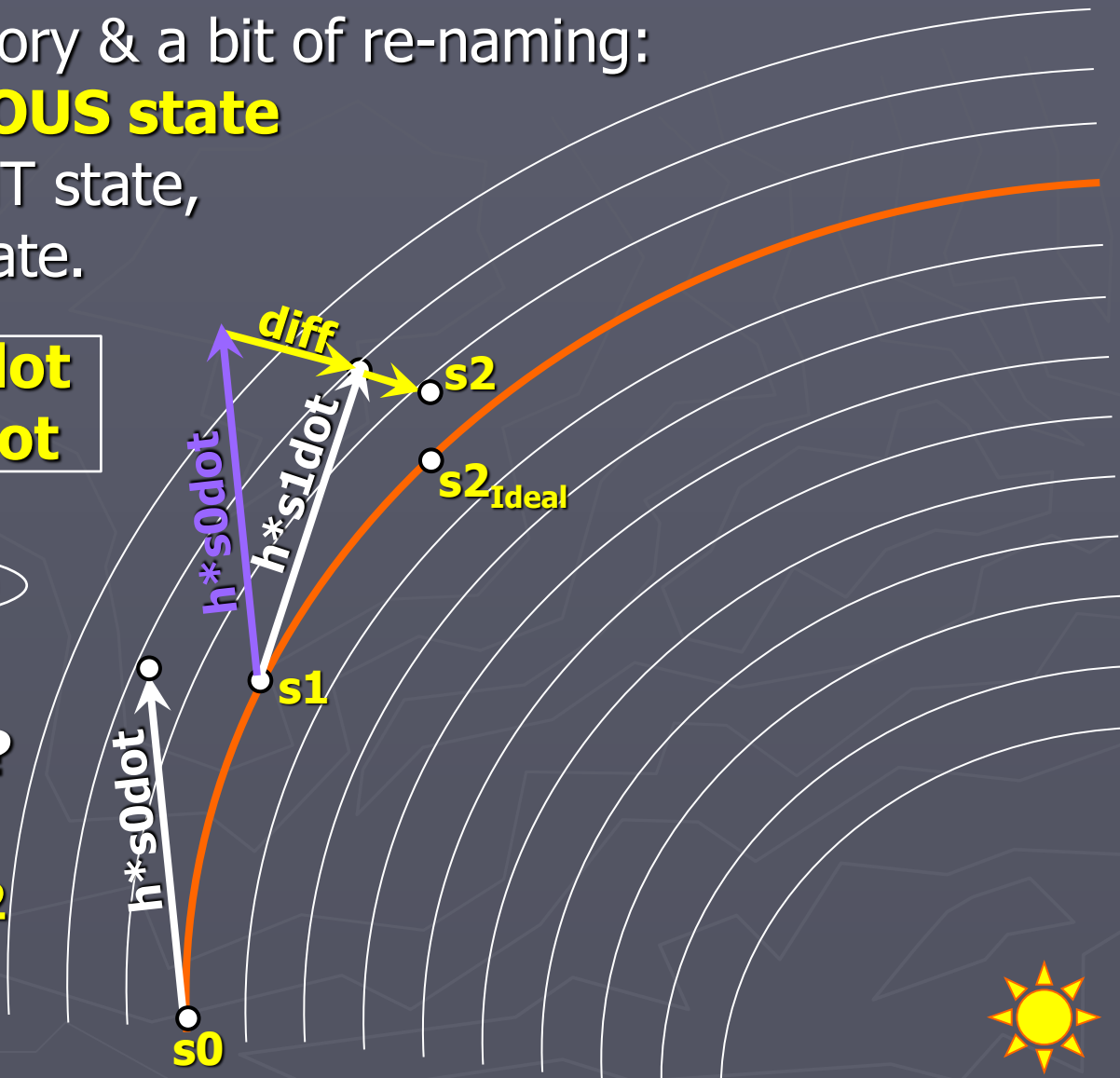
"2-step" Adams-Bashforth: (explicit)

- Requires more memory & a bit of re-naming:
  s0 == PREVIOUS state
  s1 == CURRENT state,
  s2 == NEXT state.

s2 = s1 + (3/2)h*s1dot
– (h/2)*s0dot

= s1 + h*s1dot
+ (h/2)(s1dot – s0dot)

- **See how it works?**
  =Euler: **Pos + vel*h**
  + (Acceleration*h)*h/2
  "≈ velocity-change during one timestep"



Verlet Integration: Why?

- ++Quadratic: integrates 2nd derivative **s1dot2**
- ++SEMI-Implicit – best features of both!
  - 'Presumes Magic', yet simple to compute
  - More accurate Applied Forces:
    - 3-point estimated accel. *always* matches the ideal accel.
    - Taylor series 3rd-order term cancels (due to time-symmetry)
  - Most errors are lossy & resemble ordinary drag
- ++Variants widely used in game engines

- −−BUT **ODD, STRANGE**:
  a little bit tricky to fit into state-space…
  lets see why:

Verlet Integration: How? (1)

Here's how -- https://en.wikipedia.org/wiki/Verlet_integration

- These explanations ignore state-space, & use only:

	previous:	current:	next:
position:	x_{n-1},	x_n,	x_{n+1}
acceleration:	a_{n-1},	a_n,	a_{n+1}

KEY IDEAS:

- (1) Find sum-of-forces to get the ideal, exact, desired, current acceleration: A_{ideal} = Ftot / mass
- (2) If we already *knew* x_{n+1}, two forward-differences can find a good acceleration **EST**imate(x_{n-1}, x_n, x_{n+1})
- (3) Set **EST**imate()== A_{ideal} and solve for x_{n+1}. Clever!

Verlet Integration: How (2)

- STEP (1): ideal current accel $\mathbf{A}_{ideal} = Ftot_n / mass_n$
- STEP (2): **Est**imate **acc** from *only* **positions** :
  - Time rate-of-change of PREVIOUS time-step: $(x_n - x_{n-1})/h$
  - Time rate-of-change of the NEXT time-step: $(x_{n+1} - x_n)/h$
  - How fast do these rates change? ≅ **second derivative**
  - let's measure it from the (middle of) PREVIOUS step to the (middle of) NEXT step:
  - $\dfrac{\left(\frac{x_{(n+1)} - x_n}{h}\right) - \left(\frac{x_n - x_{(n-1)}}{h}\right)}{h} = \mathbf{EST_n} = (x_{n+1} - 2{*}x_n + x_{n-1}) / h^2$
- STEP (3): Set $\mathbf{A}_{ideal}$ = **EST**imate, & **solve** for x_{n+1}:
  - Multiply h^2 on both sides: $(x_{n+1} - 2{*}x_n + x_{n-1}) = \mathbf{A}_{ideal}{*}h^2$
  - Move x_{n+1} to one side: $x_{n+1} = 2{*}x_n - x_{n-1} + \mathbf{A}_{ideal}{*}h^2$

State-Space Verlet? (1)

- **MISSING:** a clear state-space description.

  Hmmm…Looks obvious & straightforward:

  - **s0**==PREVIOUS; **s1**==CURRENT; **s2**==NEXT;
  - Create a **dot2maker()** function to find exact, current 2nd derivative **s1dot2**, and then
  - Generalize Verlet solution: positions → complete states:

$$x_{n+1} = 2*x_n - x_{n-1} + A_{ideal}*h^2$$

$$s2 = 2*s1 - s0 + s1dot2*h^2 .$$ Easy!

- But look more closely: what would **dot2maker()** *actually do?*

State-Space Verlet? (2)

- What should **dot2maker(s1)** calculate?
  - **s1dot2**.pos ← s1.acc (set by **s1** force totals **ftot**)
  - **s1dot2**.vel ← ? (?!? Zero?)
  - **s1dot2**.acc ← ? (!?! Zero?)
- And then how would **solver()** use it? If Verlet is:

$$s2 = 2*s1 - s0 + s1dot2*h^2$$

Solver would do this:

- $s2.pos = 2*s1.pos - s0.pos + s1dot2.pos*h^2$
- $s2.vel = 2*s1.vel - s0.vel + s1dot2.vel*h^2$
- $s2.acc = 2*s1.acc - s0.acc + s1dot2.acc*h^2$

- !!! SOMETHING IS WRONG HERE !!!

State-Space Verlet? (3)

- What should **dot2maker(s1)** calculate?
  - **s1dot2**.pos ← s1.acc (set by **s1** force totals **ftot**)
  - **s1dot2**.vel ← ? (?!? Zero?)
  - **s1dot2**.acc ← ? (!?! Zero?)
- And then how would **solver()** use it?

$$s2 = 2*s1 - s0 + s1dot2*h^2$$

Would mean:

- s2.pos = 2*s1.pos – s0.pos + s1dot2.pos*h^2
- s2.vel = 2*s1.vel – s0.vel + s1dot2.vel*h^2
- s2.acc = 2*s1.acc – s0.acc + s1dot2.acc*h^2

!UH OH! THIS IS WRONG!

ZERO?!?

- **s1** force totals apply **ONLY** to **s2**.positions!

State-Space Verlet? (4)

!ODD! -- Verlet Solver **won't set any velocity values** in any of our state-vectors!

!ODD! -- Verlet Solver **ignores all velocity values** stored in our state-vectors!

UH OH. Simple wall constraints 'bounce' by velocity-change!! That just won't work for our State-Space Verlet solver..

Abandon State Space Verlet? Some do…

(see Canvas→Week 4: How To Go Further)
GamesPhysicsPearlsChap11-Verlet Particles.pdf esp. pages 254-255 describes a practical but too-limited system:
Verlet-only integrator; no state vectors—instead, separate arrays for current, previous positions, for force-totals, etc., and no velocity computed except by temporary forward-differences as needed.

State-Space Verlet? (5)

Solution 1: 'Delayed velocity'

- Add another step to Verlet solver: Find s1.vel from s0.pos, s1.pos, s2.pos:

  s1.vel = s2.pos – s0.pos / 2*h

- CAUTION! Still no valid s2.vel value!
  ?perhaps compute it from (s2.pos – s1.pos)/h?
  (Swap() function should *not* copy s2.vel into s1.vel !)

- CAUTION! Verlet solver still ignores velocity in all states!
  THUS simple 'bouncy balls' walls still won't work...

State-Space Verlet? (6)

MORE TROUBLE:

Each new Verlet solution **s2** relies on both the current **s1** and the previous state **s0**.

? How do we start the system ?

Given only initial, complete current state **s1**, how do we initialize the 'previous' state **s0** ?

- Easy, first, 'Motionless' answer: set **s0** = **s1**.
- Better: quadratic extrapolation, *backwards*:
  - **s0**.pos = **s1**.pos – **s1**.vel*h + **s1**.acc*(h^2/2);
  - **s0**.vel = **s1**.vel – **s1**.acc*h

State Space Verlet? (7)

UGH! What will compute velocity for s2, and respond properly to all velocity-changing constraints?

Solution 2: Use the **'Velocity Verlet'** Solver

++ 2-Step, Fast, Stable, Simple, Semi-Implicit;

++ quadratic solver; symmetry cancels 3rd-order err

++ computes & responds properly to velocity values **s1**.vel and **s2**.vel, yet

++ does not need to keep a 'previous' state **s0**, and

++ does not require special initializations for **s0** !

Velocity Verlet in State Space (1)

'Velocity Verlet' Solver, Step-by-Step:

1) compute next-position s2.pos quadratically:

s2.pos = **s1**.pos + **s1**.vel*h + **s1**.acc*(h^2/2)

2) from **s2**.pos, find next-accelerations **s2**.acc:
call applyAllForces(**s2**) to find all **s2**.acc values

3) then compute next-velocity estimate **s2**.vel
from average of **s1** and **s2** accelerations:
s2.vel = **s1**.vel + (**s2**.acc + **s1**.acc)*(h/2)

Velocity Verlet In State Space (2)

'Velocity Verlet' Solver, Step-by-St

YES!
DO THIS!

1) compute next-position s2.pos quadratica

s2.pos = **s1**.pos + **s1**.vel*h + **s1**.acc*(h^2/2)

2) from **s2**.pos, find next-accelerations **s2**.acc:

call applyAllForces(**s2**) to find all **s2**.acc values

3) then compute next-velocity estimate **s2**.vel from average of **s1** and **s2** accelerations:

s2.vel = **s1**.vel + (**s2**.acc + **s1**.acc)*(h/2)

(***!SURPRISE!*** Not difficult at all!)

more details: https://en.wikipedia.org/wiki/Verlet_integration)

END? There's always more…

Please look into 'Leapfrog' Integration:
https://en.wikipedia.org/wiki/Leapfrog_integration

And Khan Academy's "Pixar In A Box":
https://www.khanacademy.org/partner-content/pixar/effects/particle/v/fx1-finalcut

Including Smoothed Particle Hydrodynamics:
https://www.khanacademy.org/pixar/spin-off-of-smoothed-particle-hydrodynamics/4661787965652992

END