Physically-based Animation

- Introduction to animation
- Particle systems
- Mass spring systems
- Solving differential equations

Main reference: Witkin & Baraff's SIGGRAPH course notes on Physically-Based Modelling: http://www-2.cs.cmu.edu/~baraff/sigcourse/index.html

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Introduction to animation

- Technically, it's fairly trivial to make an animation
  - Have a 3D model
  - Positions/orientations of some components and/or camera are a function of time
  - "Just" render frames at regular intervals
  - Assemble frames into an AVI/MPEG/whatever
- Hard bit is all the non-technical stuff
  - The story
  - The modelling ("3D art")
  - The sound track
  - etc
- Except for ... the physically-based modelling and animation ☺

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Methods of defining animations

- Hard code it! [e.g. ARL]
  - Rare
- Scripts e.g. POVRay
  - Cumbersome, obsolete (?)
- Interactive control (games, motion capture)
- "Manual" control, e.g. 3D Studio Max, Maya
  - The vast majority of animation is prepared this way
- Physically-based animation (i.e. *simulation*)
  - Used to control components of an animation, e.g. fluids, fire, trees in wind, collapsing structures, ...
  - Typically supported by plug-ins for programs like Maya.

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Types of Physically-based Animation

- Particle systems, e.g.
  - Mass-spring systems ← All we do
  - Fire
  - Fluids
- Finite Element Methods
  - For modelling deformable materials, and fracture.
- Fluid mechanics
- Rigid body animations (incl. jointed structures)
  - Classical mechanics + robotics

COMPSCI 715 Notes. ©Richard Lobb, 2003.

1. Particle Systems

- Examples
- Definitions
- Mass-spring systems
- Constraints and penalty forces

COMPSCI 715 Notes. ©Richard Lobb, 2003.

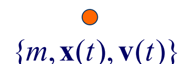
Examples

- Lobby-blobby model (Nixon & Lobb, CG&A August 2002)
- Turning the page (work in progress)
- Lexus Ad (from Siggraph “Animation Tricks” course)
- http://runevision.com/3d/anims/anims.asp
- http://www.siggraph.org/education/materials/HyperGraph/animation/movies/Eric.mov
- http://www.siggraph.org/education/materials/HyperGraph/animation/movies/SecretSerpent320.mov
- http://www.id8mediagallery.com/RealFlow_gallery.htm

COMPSCI 715 Notes. ©Richard Lobb, 2003.

A Particle

- *Particle*: “(Physics) A body whose spatial extent and internal motion and structure, if any, are irrelevant in a specific problem.”
  - American Heritage Dictionary
- Has *position* plus other attributes (e.g. mass)
  - Position varies with time
- It *represents* the behaviour of a bit of some (usually continuous) material.
  - A “super-atom”
  - Or a *sample*

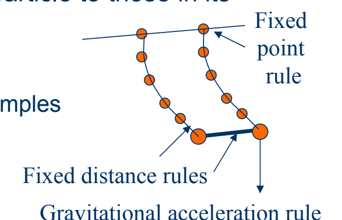


$$\{m, \mathbf{x}(t), \mathbf{v}(t)\}$$

COMPSCI 715 Notes. ©Richard Lobb, 2003.

A Particle System

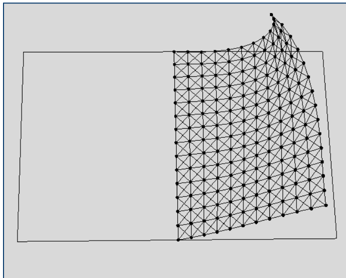
- *Particle* System: A set of particles, plus a set of rules governing their motion, which models the spatio-temporal behaviour of a (usually continuous) physical entity or set of entities
  - Rules usually relate the motion of a particle to those in its vicinity plus some global rules.
  - May also have birth/death rules
    - e.g. to maintain uniform density of samples



COMPSCI 715 Notes. ©Richard Lobb, 2003.

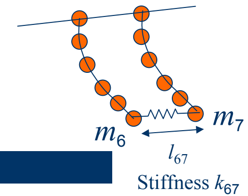
A Particle System (cont'd)

– In graphics, also have *rendering rules* which control how the system is displayed so it looks like the entity it represents.



COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mass-Spring System



- Is a simple particle system comprising:
  1. $\{m_i, \mathbf{x}_i(t), \mathbf{v}_i(t)\} = \{m_i, \mathbf{x}_i(t), \dot{\mathbf{x}}_i(t)\}$
     - A set of particles, each with fixed mass and time-varying position and velocity
  2. $\{k_{ij}, l_{ij}\}$
     - A set of springs connecting pairs of particles.
     - $k_{ij} = k_{ji}$ is the spring constant of the spring between particles *i* and *j*
       – Zero implies no spring present
     - $l_{ij} = l_{ji}$ is the spring's rest length

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mass-Spring System (cont'd)

3. $\{\mathbf{F}_g(\mathbf{x},t)\}$
   – A set of (possibly time-varying) global forces applying to all particles, e.g. gravity, wind forces
4. Other behaviour rules (e.g. particles can't penetrate ground)

Important note – I define:

$\{\mathbf{x}_i\}$ to be the *configuration* of the system

– How it looks at an instant in time

$\{\mathbf{x}_i, \dot{\mathbf{x}}_i\}$ to be the *state* of the system

– How it looks and how it is moving (necessary to predict its behaviour)

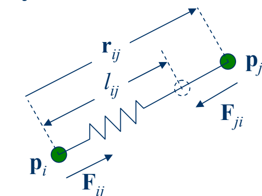
COMPSCI 715 Notes. ©Richard Lobb, 2003.

Spring Force

- Restoring force is proportional to the stretch of the spring beyond its rest length. k_{ij} is the constant of proportionality.

$$\mathbf{F}_{ij} = k_{ij}\left(|\mathbf{r}_{ij}| - l_{ij}\right)\frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|}$$

where $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$



- $\mathbf{F}_{ij}$ is the force on particle *i* due to particle *j*
  – $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$

COMPSCI 715 Notes. ©Richard Lobb, 2003.

A Simple Particle System Algorithm

```
T = 0
for each step in time, Δt {
    T += Δt
    for each particle {
        compute total force F_i on particle
        a_i = F_i/m_i
        v_i += a_i Δt
        x_i += v_i Δt
    }
    display  [or write frame to file]
}
```


COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mass Spring Demo

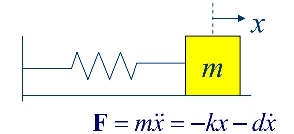
COMPSCI 715 Notes. ©Richard Lobb, 2003.

Damping

- Need to dissipate energy – “damping”
- Physically, energy loss depends on system being modelled
  - Fire particles – damped by motion through air
  - Elastic materials – heating of molecular lattice (imperfect elasticity)
    - Plus maybe air damping if motion large
  - Fluids – internal viscous damping
  - Physical mass-spring systems
    - Damping by air plus elastic losses in springs
    - Plus maybe extra dampers (e.g. car suspension)

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Damping (cont’d)



$$\mathbf{F} = m\ddot{x} = -kx - d\dot{x}$$

- Common engineering and physics model is *viscous damping*
  - Damping force proportional to velocity
- This is very poor approximation
  - See http://www.csiberkeley.com/Tech_Info/19.pdf
- Precise form of energy loss not normally important in graphics
- Typically *stability* is the key issue

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Damping (cont’d)

- Usual hack is to include two forms of viscous damping
  - particle-air damping – force opposes particle motion, proportional to particle velocity:
  - spring damping – force opposes spring compression/extension proportional to rate of change of length
- Ad Hoc!

Damped Spring Force

- Spring force equation is now

$$\mathbf{F}_{ij} = \left[k_{ij}\left(\left|\mathbf{r}_{ij}\right| - l_{ij}\right) + k_d \frac{d}{dt}\left|\mathbf{r}_{ij}\right| \right] \frac{\mathbf{r}_{ij}}{\left|\mathbf{r}_{ij}\right|}$$

$$= \left[k_{ij}\left(\left|\mathbf{r}_{ij}\right| - l_{ij}\right) + k_d \frac{\mathbf{r}_{ij} \cdot \dot{\mathbf{r}}_{ij}}{\left|\mathbf{r}_{ij}\right|} \right] \frac{\mathbf{r}_{ij}}{\left|\mathbf{r}_{ij}\right|}$$

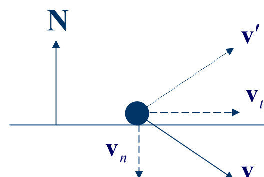
where $\dot{\mathbf{r}}_{ij} = \mathbf{v}_j - \mathbf{v}_i = \dot{\mathbf{x}}_j - \dot{\mathbf{x}}_i$

Particle/Plane Collisions

Biped animation

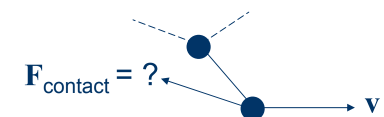
- Trivial to test if particle is inside a half-space (e.g. below ground)
- Should back-off simulation until it’s just on the plane
  - But usually don’t bother!
- Particle impacting plane generally bounces.
  - Decompose velocity into normal and tangential components
  - Multiply normal component by $-r$, where r is *coefficient of restitution*

$$\mathbf{v} = \mathbf{v}_n + \mathbf{v}_t = (\mathbf{N} \cdot \mathbf{v})\mathbf{N} + \mathbf{v}_t$$
$$\mathbf{v}'_n = -r\,\mathbf{v}_n$$
$$\mathbf{v}'_t = \mathbf{v}_t$$



Particle/Plane Contact

- Inter-bounce time gets smaller and smaller
- At some point have to define particle as being in contact with plane ($\mathbf{v}_n$ < threshold)
- Then must apply a *contact force* to the particle to prevent penetration and to incorporate (sliding) friction

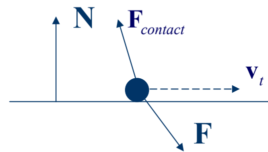


Particle/Plane Contact

- First compute total force **F** on particle. Then add:

$$\mathbf{F}_{contact} = \begin{cases} \mathbf{0} & \mathbf{N}\cdot\mathbf{F} >= 0 \\ -(\mathbf{N}\cdot\mathbf{F})(\mathbf{N} + k_f \mathbf{v}_t) & otherwise \end{cases}$$

- k_f is friction coefficient
  - Very simplistic model
  - Should at least switch to a static friction model if $\mathbf{v}_t$ < some threshold



COMPSCI 715 Notes. ©Richard Lobb, 2003.

2. Differential Equation Solving

- Refces:
  - Witkin/Baraff Siggraph Tutorial
  - mathworld.wolfram.com/OrdinaryDifferentialEquation.html
    - But we are concerned only with *numerical* solution methods
    - Assume no analytic solution
  - Press et al “Numerical Recipes in <x>”, Chapter 16 (http://www.library.cornell.edu/nr/bookcpdf.html)

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Initial Value Problems (IVPs)

- Particle system problem is to track the state of the system S(t) =({$\mathbf{x}_i(t)$,$\mathbf{v}_i(t)$}) over time, given S(0) and a way to compute S’(t).
- This is an *initial value problem*, characterised by a linear first-order ordinary differential equation (ODE) of form

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

  - **x** is the system state (a vector in $\mathbf{R}^n$)
  - f is a known function
    - In mass-spring systems, often no t dependence but might have e.g. time-varying wind field

COMPSCI 715 Notes. ©Richard Lobb, 2003.

An Aside: Order of an ODE

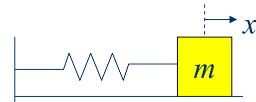
- Our particle system is actually a *second-order* ODE of form

$$\ddot{\mathbf{x}} = \frac{\mathbf{F}}{m} = f(\dot{\mathbf{x}}, \mathbf{x}, t)$$

- However, by defining “state” as $(\mathbf{x}, \dot{\mathbf{x}})$ we reduce this to a first-order ODE (or two coupled first-order ODEs)
  - *Order reduction*
  - Standard technique in numerical soln of ODEs.
  - When using e.g. *Mathematica*, we can just provide the original ODE and let it do the order reduction.

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Example IVP



- Equation of motion (undamped) is:

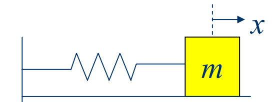
$$\ddot{\mathbf{x}} = \frac{\mathbf{F}}{m} = -\frac{k}{m}x = -cx$$
$$\mathbf{s} = (x, \dot{x})$$
$$\dot{\mathbf{s}} = (\dot{x}, -cx) = (\mathrm{Snd}(\mathbf{s}), -c\,\mathrm{Fst}(\mathbf{s}))$$

Fst and *Snd* are first and second element operators. [This is just to emphasise that $\dot{\mathbf{s}}$ is a function of $\mathbf{s}$.]

- If map **s** onto (x,y) plane, with y for velocity, the d.e. is

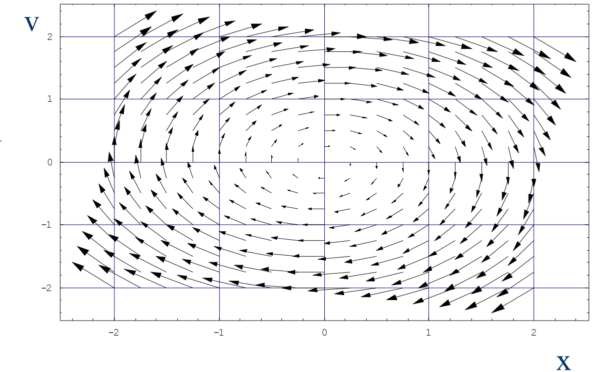
$$\dot{\mathbf{s}} = (\dot{x}, \dot{y}) = (y, -cx)$$

Example IVP (cont'd)

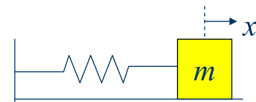


Vectors show the *vector field* $\dot{\mathbf{s}} = (y, -cx)$ over the state space {(x,y)}

Soln for any given I.V. is an ellipse in state space

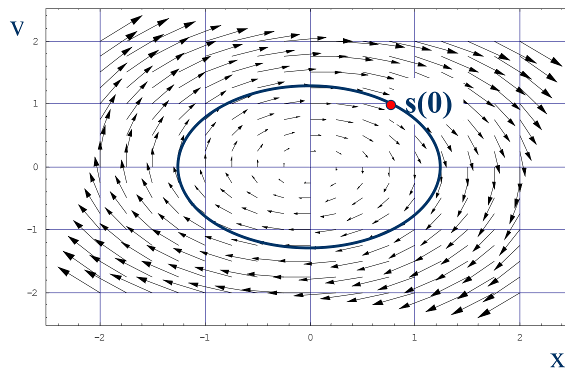


Example IVP (cont'd)



Job of ODE solver is to track the path of a particle in this vector field.

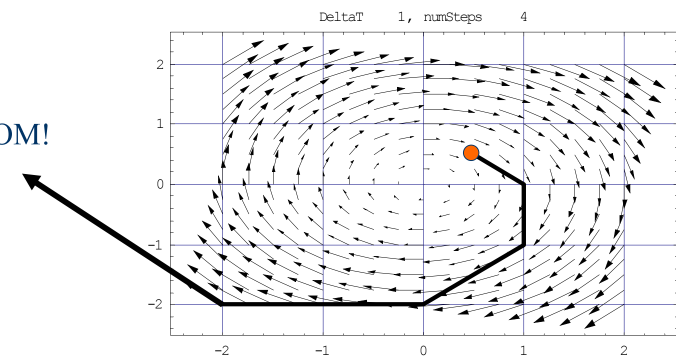
If $\dot{\mathbf{s}}$ depends explicitly on t, vector field is changing with t, i.e. is $\dot{\mathbf{s}}(\mathbf{s}(t), t)$.



The Euler Method

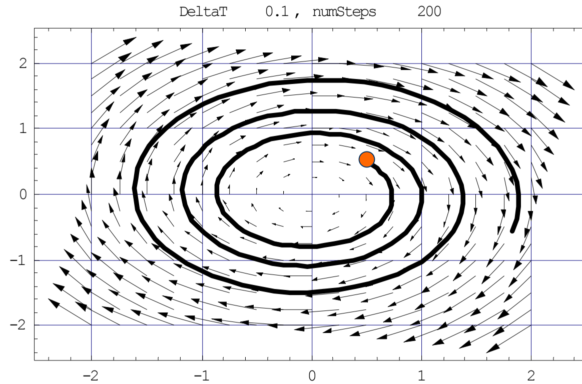
Simple linear steps in state space: $\mathbf{s}(t + \Delta t) = \mathbf{s}(t) + \Delta t\, \dot{\mathbf{s}}(\mathbf{s}(t))$

BOOM!



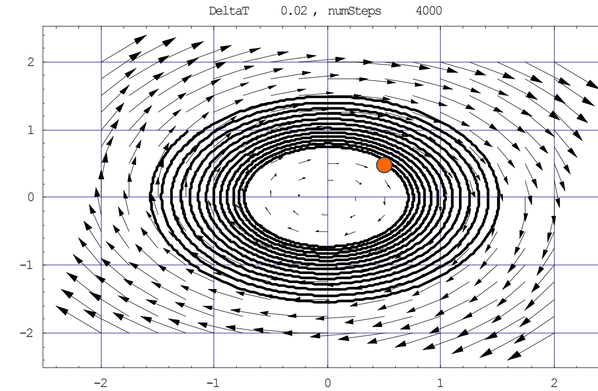
The Euler Method (cont'd)

After only 3 oscillations have 3 times the velocity, 9 times the energy!!



COMPSCI 715 Notes. ©Richard Lobb, 2003.

The Euler Method (cont'd)



CONCLUSION: need very small time steps and/or large damping to avoid energy "blow up"

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Runge-Kutta Methods

- To get a more-accurate step, need a higher-order estimate of local derivative field
- Runge-Kutta methods are based on truncated Taylor series extrapolation.

$$\underbrace{\mathbf{s}(t+\Delta t) = \mathbf{s}(t) + \Delta t\,\dot{\mathbf{s}}(t)}_{} + \frac{\Delta t^2}{2!}\ddot{\mathbf{s}}(t) + \ldots + \frac{\Delta t^n}{n!}\frac{\partial^n \mathbf{s}(t)}{\partial t^n} + \ldots$$

$[\dot{\mathbf{s}}(t),\ \ddot{\mathbf{s}}(t)$ are short for $\dot{\mathbf{s}}(\mathbf{s}(t)),\ \ddot{\mathbf{s}}(\mathbf{s}(t))]$

Euler method is first-order Runge-Kutta
Error is $O(\Delta t^2)$ per step.
Error decreases linearly with Δt over *a given time interval.*

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point Method

- Try second-order Runge-Kutta.
- Take next term in Taylor expansion. Write as:

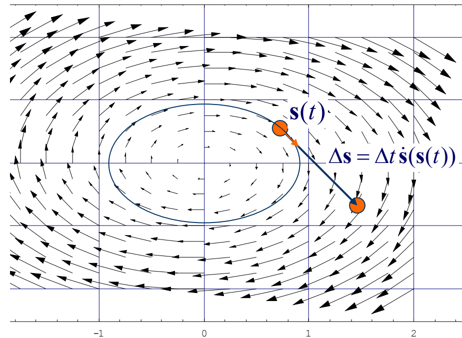
$$\mathbf{s}(t+\Delta t) = \mathbf{s}(t) + \Delta t\left(\dot{\mathbf{s}}(t) + \frac{\Delta t}{2}\ddot{\mathbf{s}}(t)\right)$$

- Term in parentheses is the first order estimate of the state derivative half way through the next step.
- Hence (with zero rigour) ....

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point method: Algorithm

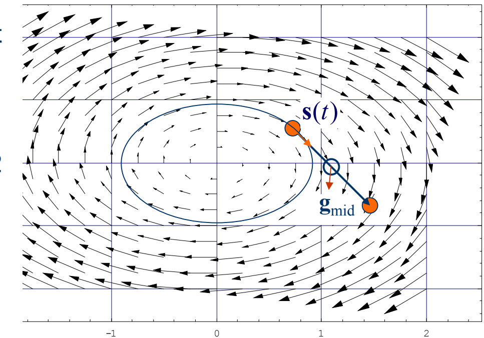
- Compute Euler step $\Delta\mathbf{s}$ using derivative at start
- Evaluate derivative $\mathbf{g}_{mid}$ at mid-point of step, i.e. at $\mathbf{s}(t)+\Delta\mathbf{s}/2$
- Go back to start and take a step using derivative $\mathbf{g}_{mid}$



COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point method: Algorithm (cont'd)

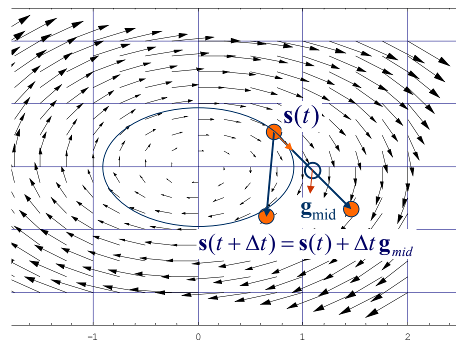
- Compute Euler step $\Delta\mathbf{s}$ using derivative at start
- Evaluate derivative $\mathbf{g}_{mid}$ at mid-point of step, i.e. at $\mathbf{s}(t)+\Delta\mathbf{s}/2$
- Go back to start and take a step using derivative $\mathbf{g}_{mid}$



COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point method: Algorithm (cont'd)

- Compute Euler step $\Delta\mathbf{s}$ using derivative at start
- Evaluate derivative $\mathbf{g}_{mid}$ at mid-point of step, i.e. at $\mathbf{s}(t)+\Delta\mathbf{s}/2$
- Go back to start and take a step using derivative $\mathbf{g}_{mid}$



COMPSCI 715 Notes. ©Richard Lobb, 2003.

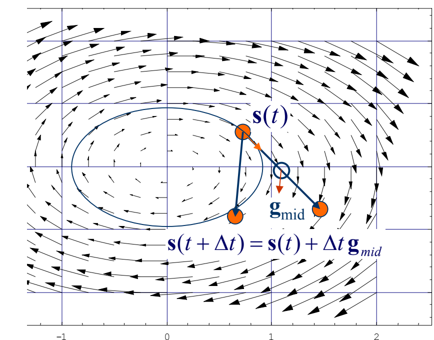
Mid-point method: Algorithm (cont'd)

Allowing for time-varying derivative field, algorithm is:

$\Delta\mathbf{s} = \Delta t\,\dot{\mathbf{s}}(\mathbf{s}(t),t)$ (* Euler step *)

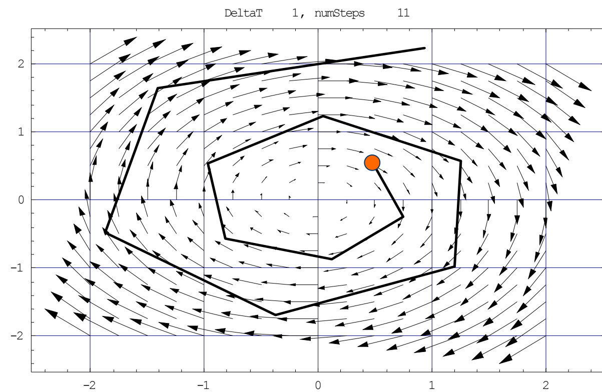
$\mathbf{g}_{mid} = \dot{\mathbf{s}}\left(\mathbf{s}(t)+\frac{\Delta\mathbf{s}}{2}, t+\frac{\Delta t}{2}\right)$

$\mathbf{s}(t+\Delta t) = \mathbf{s}(t) + \Delta t\ \mathbf{g}_{mid}$



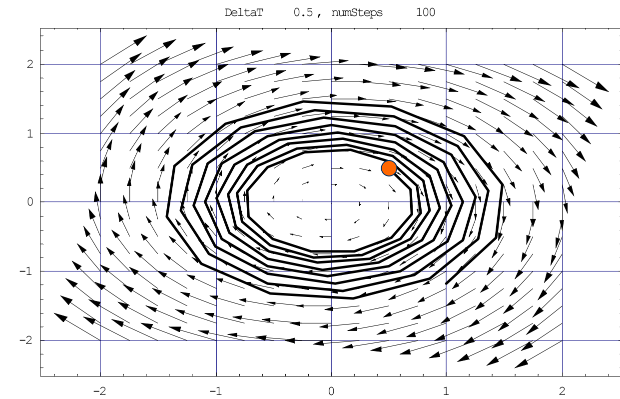
COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point Method, dt = 1 sec, 11 steps



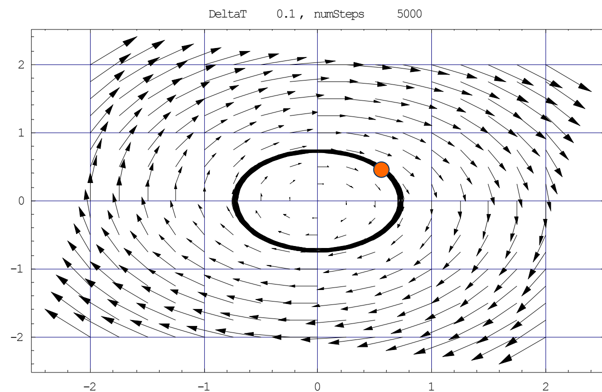
COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point Method, dt = 0.5 sec, 100 steps



COMPSCI 715 Notes. ©Richard Lobb, 2003.

Mid-point Method, dt = 0.1 secs, 5000 steps



COMPSCI 715 Notes. ©Richard Lobb, 2003.

4th-Order Runge-Kutta

"For many scientific users, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm. ... Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields."

– Press et al, "Numerical Recipes"

COMPSCI 715 Notes. ©Richard Lobb, 2003.

4th-Order Runge-Kutta Algorithm

- Quote without derivation

$$\mathbf{k}_1 = \Delta t\, \dot{\mathbf{s}}(\mathbf{s}(t), t) \quad \text{(* Euler step } \Delta\mathbf{s} \text{ *)}$$

$$\mathbf{k}_2 = \Delta t\, \dot{\mathbf{s}}\left(\mathbf{s}(t) + \frac{\mathbf{k}_1}{2}, t + \frac{\Delta t}{2}\right) \quad \text{(* Midpoint step *)}$$

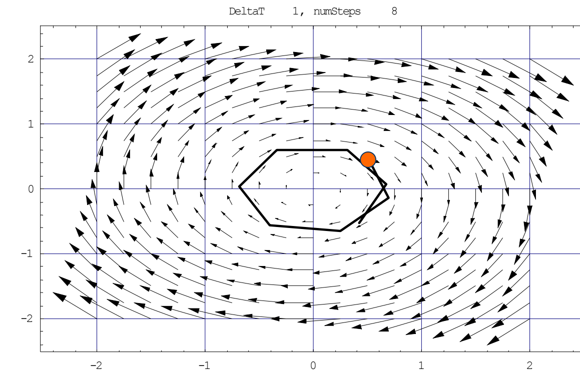
$$\mathbf{k}_3 = \Delta t\, \dot{\mathbf{s}}\left(\mathbf{s}(t) + \frac{\mathbf{k}_2}{2}, t + \frac{\Delta t}{2}\right) \quad \text{(* Refined midpoint step *)}$$

$$\mathbf{k}_4 = \Delta t\, \dot{\mathbf{s}}\left(\mathbf{s}(t) + \mathbf{k}_3, t + \Delta t\right) \quad \text{(* Refined refined step *)}$$

$$\mathbf{s}(t + \Delta t) = \mathbf{s}(t) + \frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4 \quad \text{(* 4th order R-K result *)}$$

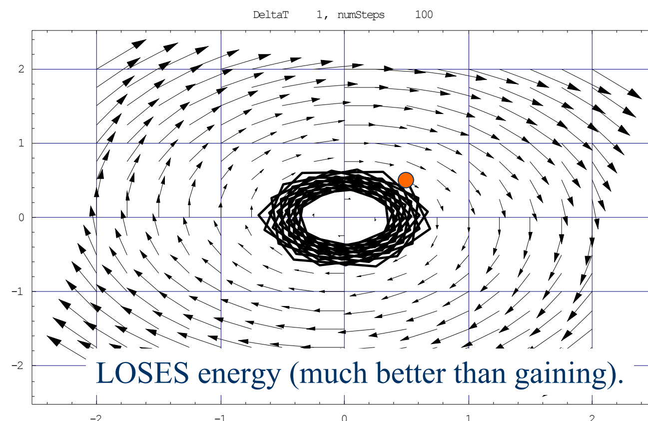
COMPSCI 715 Notes. ©Richard Lobb, 2003.

RK4 Results, *dt* = 1 sec, 8 steps



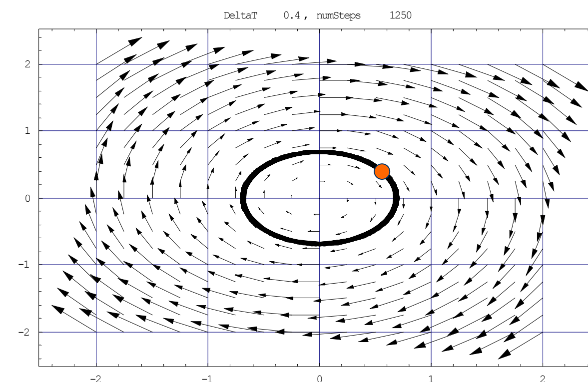
COMPSCI 715 Notes. ©Richard Lobb, 2003.

RK4 Results, *dt* = 1 sec, 100 steps



COMPSCI 715 Notes. ©Richard Lobb, 2003.

RK4 Results, *dt* = 0.4 secs, 1250 steps



- UDOO: Estimate cost of RK4 versus midpoint for similar accuracy

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Adaptive Stepsizes

- Simple idea:
  - "Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside." – Numerical Recipes.
- Simple implementation:
  - Take two steps of Δt, one step of 2 Δt, compare results
  - Adjust step size to keep error within pre-specified bounds
- See notes + Numerical Recipes
- I haven't found this very useful in PBM – "terrain" usually same roughness everywhere.

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Idle Thoughts of an Idle Fellow

- An Euler solver is too horrible. NEVER use it.
  - Yeah I know – you will anyway.
- Accuracy is rarely the issue in CG – stability is everything.
- Benefit of RK4 over midpoint is moot.
- Often have discontinuities (e.g. collisions) – have to stop d.e. solver and back-up to time of collision.
  - Further reduces benefit of RK4
- Stiff springs → high frequencies → small steps reqd
- If overall behaviour is low frequency (non-stiff springs) but have some stiff springs, small steps are still necessary for stability
  - Such "stiff d.e.s" best handled by implicit methods – next section.

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Restructuring the Particle System

- Should isolate the d.e. solver from the physical simulation code
- *State* of *n*-particle system is a vector in $\mathbf{R}^{6n}$:

$$\mathbf{s} = \left(\mathbf{x}_1, \dot{\mathbf{x}}_1, \mathbf{x}_2, \dot{\mathbf{x}}_2, \ldots, \mathbf{x}_n, \dot{\mathbf{x}}_n\right)$$

- State derivative is

$$\dot{\mathbf{s}} = \left(\dot{\mathbf{x}}_1, \frac{\mathbf{F}_1}{m_1}, \dot{\mathbf{x}}_2, \frac{\mathbf{F}_2}{m_2}, \ldots, \dot{\mathbf{x}}_n, \frac{\mathbf{F}_n}{m_n}\right)$$

COMPSCI 715 Notes. ©Richard Lobb, 2003.

Interface to DE Solver

```
// DESolver.java -- the abstract class from which are subclassed all
// the real DESolvers.

package deSolvers;

public abstract class DESolver {
    public abstract void stepState(DEStateSource source, double deltaT);
    // Called to advance the state of the given state source by
    // a time step of deltaT.
}
```


COMPSCI 715 Notes. ©Richard Lobb, 2003.

DEStateSource – the interface a client provides to the DESolver

```
package deSolvers;

// All the methods in this interface are called by the DESolver while processing
// the stepState call.

public interface DEStateSource {
    public int stateLength();                // Number of doubles in the state vector
    public void getState(double[] state);    // Get the current state
    public void getStateDerivative(double[] state);
                                             //  Get the current state derivative
    public void setState(double[] state);    // Set a new state
    public void incTime(double deltaT);      // Set a time increment of deltaT
}
```


COMPSCI 715 Notes. ©Richard Lobb, 2003.

Euler Solver Code

```
package deSolvers;
import java.util.*;

public class EulerDESolver extends DESolver {
    double[] state = null;     // Buffer for copy of state
    double[] dSdt = null;      // Buffer for copy of state derivative

    public void stepState(DEStateSource source, double deltaT) {
        int n = source.stateLength();
        if (state == null || state.length != n) {    // On first call, allocate buffer space.
            state = new double[n];   // [This is ugly compared to having getState etc ...
            dSdt = new double[n];   // ... return an array, but much more efficient.]
        }
```


COMPSCI 715 Notes. ©Richard Lobb, 2003.

Euler Solver Code (cont'd)

```
        // Now the real gutz of stepState ...

        source.getState(state);
        source.getStateDerivative(dSdt);
        for (int i = 0; i < n; i++)
            state[i] += dSdt[i] * deltaT;
        source.setState(state);
        source.incTime(deltaT);
    }
}
```


COMPSCI 715 Notes. ©Richard Lobb, 2003.

The Particle System

```
public abstract class ParticleSystem implements DEStateSource {
    public Particle[] particles = null;      // The particles in the system
    public Force[] forces = null;            // The forces acting on them
    public double t = 0;                     //  current time

    public int stateLength() { return 6 * particles.length; }

    public void getState(double[] state) { .. copy state from particles into caller's buffer ..}

    public void getStateDerivative(double[] sd) { ... clear forces on each particle, apply ...
            ... all forces, compute all particle accelerations, return all v's and a's ... }

    public void incTime(double dt) {  this.t += dt;    }
}
```


COMPSCI 715 Notes. ©Richard Lobb, 2003.