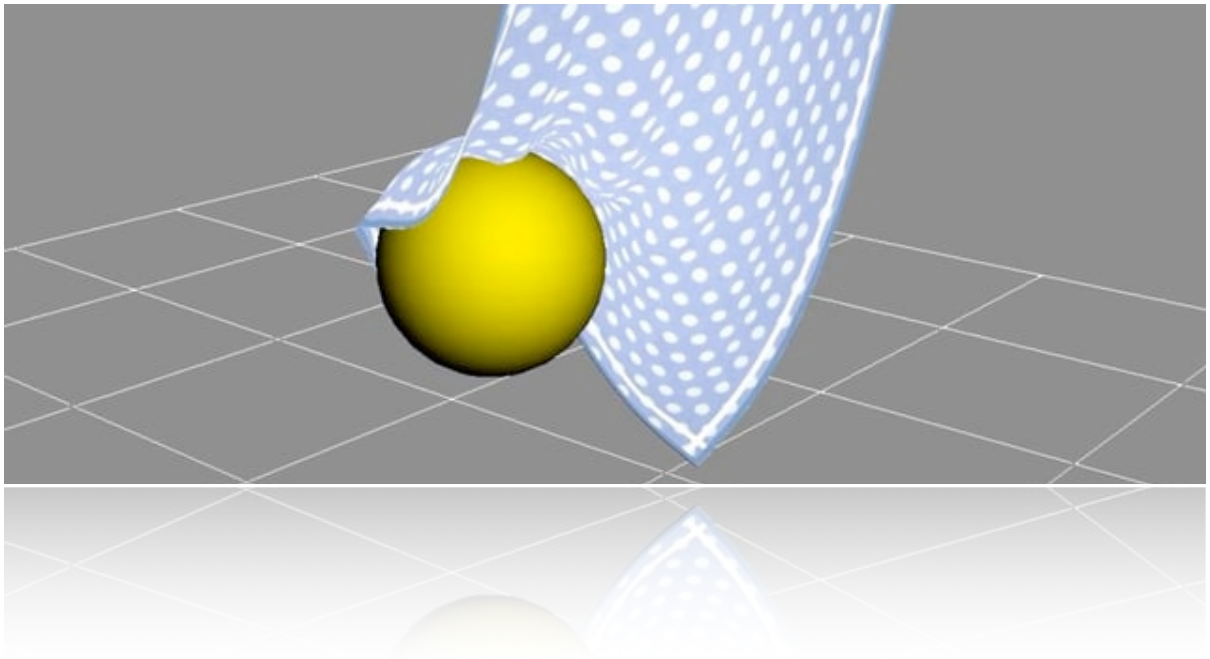


Mass-Spring Systems

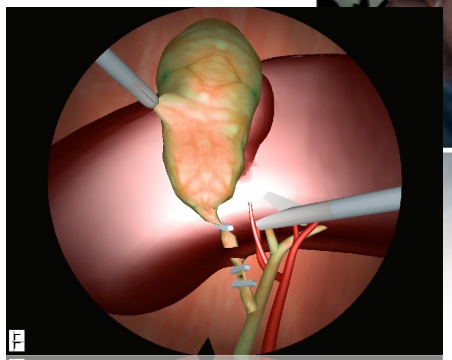
Thuerey / Game Physics

Mass-Spring Systems

- Not only:

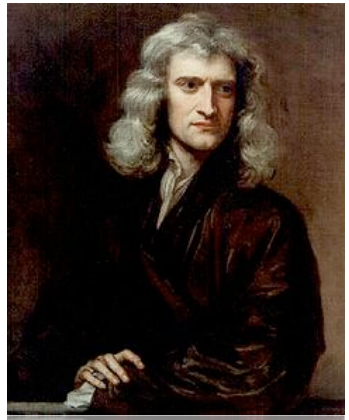
Applications

- Cloth
- Deformable / elastic objects
- Ropes etc.



Recap Projectile Motion

- Particle, with position $\boldsymbol{x}(t)$
- Velocity $\boldsymbol{v}(t)$, acceleration $\boldsymbol{a}(t)$
- Velocity from position: time derivative
- Position from velocity: integrate over time
- Newton's 2nd law: $\boldsymbol{F}=m\boldsymbol{a}$
- E.g., gravity: $\boldsymbol{F}=(0, -mg)^T$



Recap Projectile Motion

- We have: $\mathbf{v}(t) = \frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t}$ $\mathbf{a}(t) = \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t}$
- Analytic solution: $\mathbf{x}(t) = -\frac{1}{2}gt^2 + \mathbf{v}_0 t + \mathbf{x}_0$
- Note - mass doesn't matter...
- Only possible for very simple force functions!

Motivation

- We want: world state at time *t*
- What if acceleration *a=f(...)* and/or velocity *v=f(...)* are really *complicated* functions?
- Better: discretize & integrate numerically

Learning Outcomes

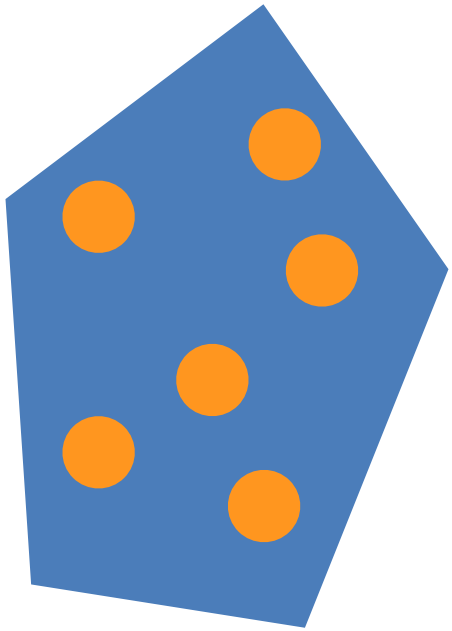
- Know how to simulate deformable objects with mass-spring systems
- Physical model for springs
- Numerical integration for ODEs
  - Properties of different methods
  - Be able to apply them
- How to implement

Mass-Spring Systems

Mass-Spring System

- Discretization of object into mass points (e.g. elastic object, fluid, inelastic object etc. ...)
- Interaction between points i and j based on internal force $\mathbf{F}_{ij}^{int}$
- All other forces: external $\mathbf{F}_{i}^{ext}$
- Overall force $\mathbf{F}_i = \mathbf{F}_{ij}^{int} + \mathbf{F}_{i}^{ext}$
- With $\mathbf{F}_{ij}^{int} = -\mathbf{F}_{ji}^{int}$, $\sum_i \sum_j \mathbf{F}_{ij}^{int} = 0$

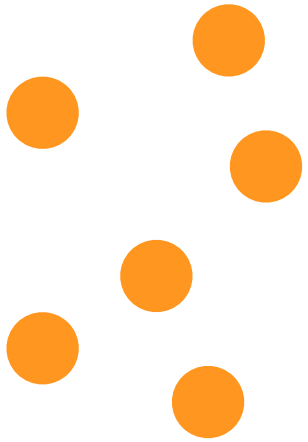
Mass Points



- Sample with mass points
- Mass of object: *M*
- Number of points: *n*
- Assuming uniform mass, for each point: $m=M/n$

➡ Simulate the motion of each mass point

Dynamics

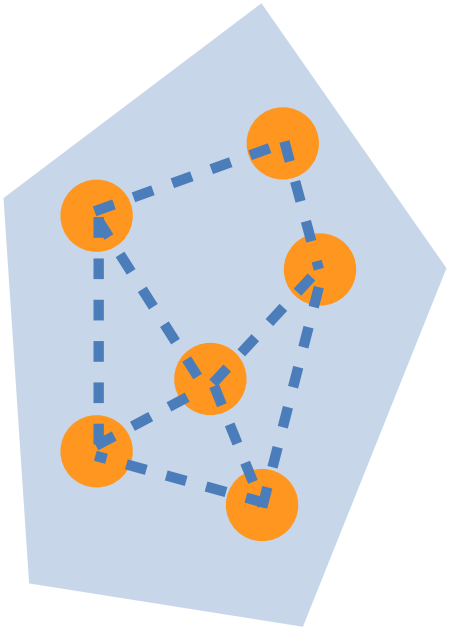


- Physically-based model
- Newtons 2nd law should hold:

$$\sum_j \mathbf{F}_{ij}^{int} + \mathbf{F}_i^{ext} = m\mathbf{a}_i$$

- How to compute internal forces?

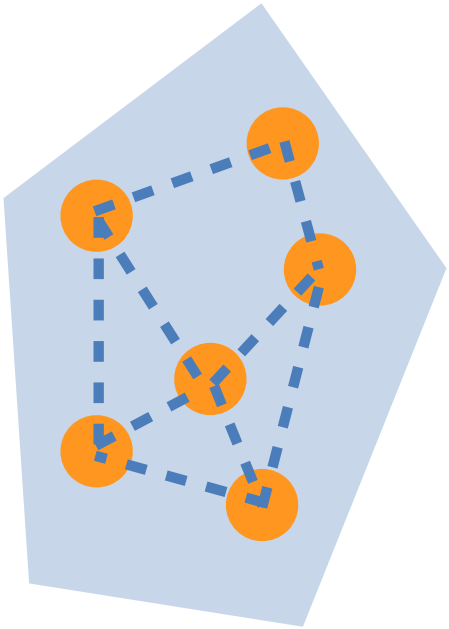
Springs



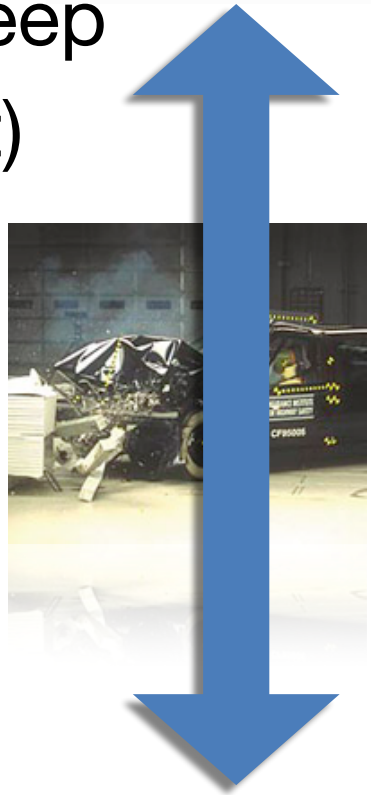
- Model elasticity with springs
- Hooke's law:
  - Force needed to extend (or compress) a spring by a distance is proportional to that distance

Elasticity

- Elasticity: return to rest state

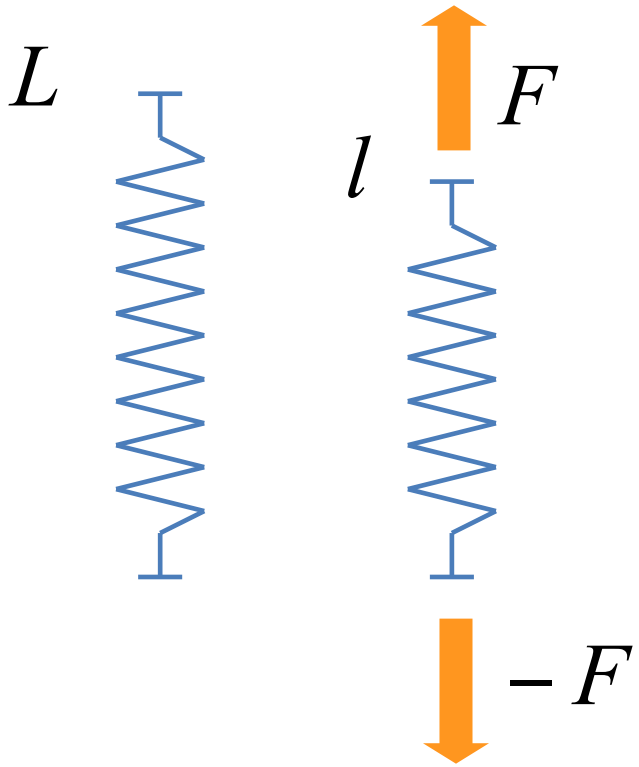


- In contrast to plasticity: keep current state (or parts of it)



- Extreme: fluid

Hookean Spring

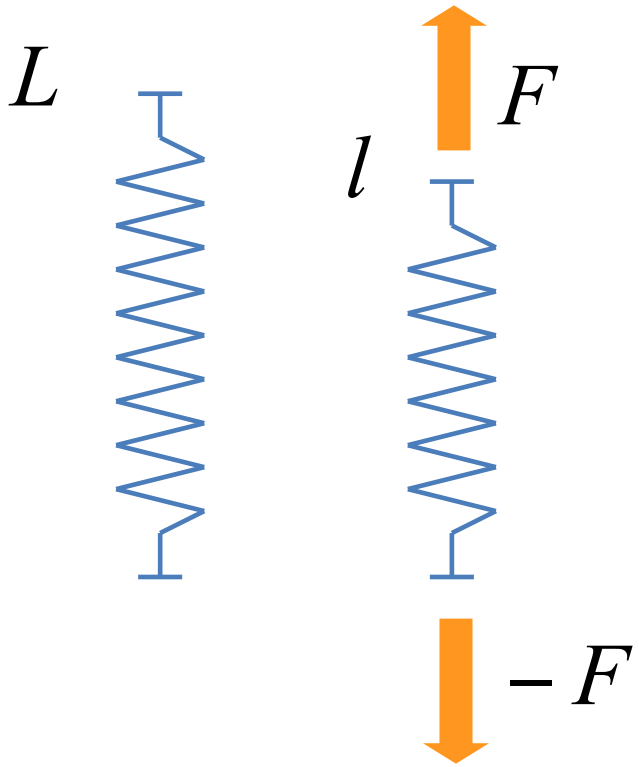


- Spring stiffness: *k*
- Initial length: *L*
- Current length: *l*

- Force linear in deformation:

$$F = -k(l - L)$$

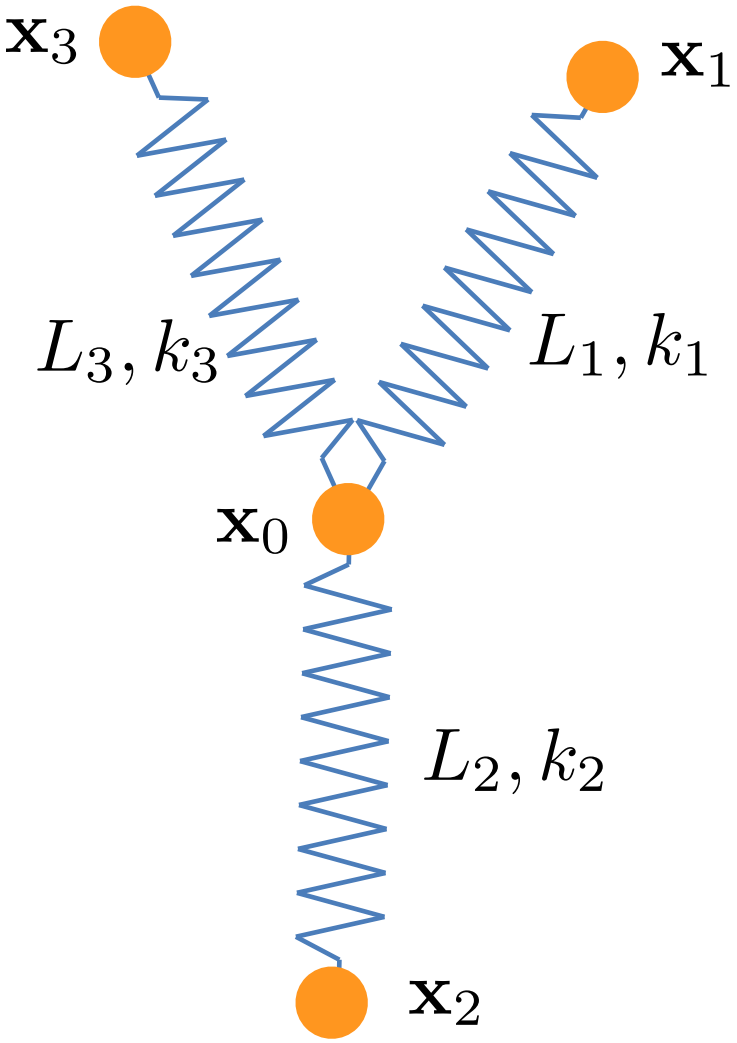
Directed Force



- Force has direction given by endpoints of spring
- In vector notation:

$$\mathbf{F}_{ij} = -k(l - L)\frac{\mathbf{x}_i - \mathbf{x}_j}{l}$$

Total Internal Force



- Summation of spring forces:

$$\mathbf{F}_0^{int} = -\sum_{i \in \{1,2,3\}} k_i (l_i - L_i) \frac{\mathbf{x}_0 - \mathbf{x}_i}{l_i}$$

$$\mathbf{F}_i = \mathbf{F}_i^{int} + \mathbf{F}_i^{ext}$$

- External forces: gravity, user interaction, etc. ...

Dissipative Forces

- Damping: $\mathbf{F}^{damp}(t) = -\gamma \mathbf{v}(t)$

- Models internal friction
- Removes energy from the system
- In practice: often necessary for stabilization
- Counteract numerical errors

System Equations

- Equations of motion for one mass point:

$$m_i \frac{\mathrm{d}^2 \mathbf{x}_i(t)}{\mathrm{d}t^2} = \mathbf{F}_i^{int}(\mathbf{x}_i(t)) + \mathbf{F}_i^{ext}(t) - \gamma \frac{\mathrm{d}\mathbf{x}_i(t)}{\mathrm{d}t}$$

Acceleration

Elastic forces
Note: depend on current position!

Damping
Note: depends on velocities

Implementation

- Straight forward approach
  - We know $\mathbf{v}(t) = \frac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t}$, $\mathbf{a}(t) = \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t}$
  - Euler step (cf. Game Engine Design lecture)
  - Integrate velocity and position in time

C++ Suggestions

- Spring class:

```
class Spring
{
  public:
    int    point1;
    int    point2;
    float  stiffness;
    float  initialLength;
   [float  currentLength;]   (not necessary, but can be useful...)
    ...

}
```

C++ Suggestions

- Mass point class:

```
class Point
{
  public:
    Vector3  position;
    Vector3  velocity;
    Vector3  force;
    float    mass;
    float    damping;

    ...

}
```

C++ Suggestions

- Simulation loop:

```
while(...)
{
    for all points i
        point[i].clearForce()
        point[i].addGravity()
    for all springs s
        spring[s].computeElasticForces()
        spring[s].addToEndPoints()
    // …evaluate other forces, eg, user interaction…
    integratePositions()
    integrateVelocity()
}
```

Alternative *(cf. I. Millington)*

- Generic force generators
  - Linked to one mass point
  - Evaluates force function, and accumulates
- Pros / Cons
  - Clean interface for force generators
  - Spring stores other point, but only adds to source
  - Two spring instances needed for one spring

Simple Collisions

- Ground plane at *z=0*
  - Compute new positions
  - Then enforce *z>=0* for all points
  - Spring forces will keep whole objects above ground

More on collisions later!

Implementation Notes

- Heavy objects
  - Give very high mass -> no movement
  - Better: don’t even update positions!
  - Spring forces will influence other points nonetheless
  - E.g., add flag to Point class: `bool isFixed;`

Analytic Integration Methods

System Equations

- Equations of motion: $m_i \frac{\mathrm{d}^2 \mathbf{x}_i(t)}{\mathrm{d}t^2} = \mathbf{F}_i^{int}(\mathbf{x}_i(t)) + \mathbf{F}_i^{ext}(t) - \gamma \frac{\mathrm{d}\mathbf{x}_i(t)}{\mathrm{d}t}$
- Problem: we want *x(t+h)* , but acceleration depends on complicated function in *x(t)*!
- Rewrite & classify:

$$m \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = g + f(x) - \gamma \frac{\mathrm{d}x}{\mathrm{d}t}$$

$$x'' = f(t, x, x')$$

ODE!

Note

- Change of Notation
  - For simulations usually: ***x*** denotes position (vector)
  - In the following: *x* is free variable
  - Function is *y(x)* (e.g., to compute state of object)
  - ...historical reasons
  - Thus: *x* corresponds to simulation time *t* (for now)

Classification

- One free variable *x*
- Function of highest derivative of solution *y(x):*

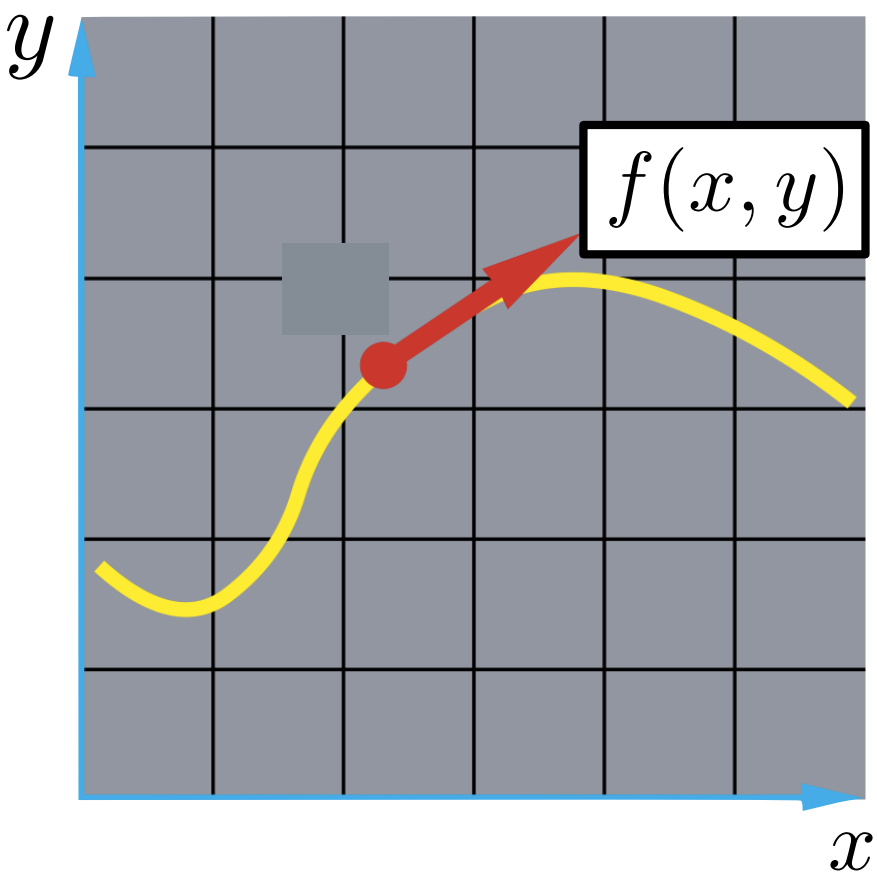
$$y^{(n)} = f(x, y, y', y'', .., y^{n-1})$$

–> Ordinary differential equation (ODE)

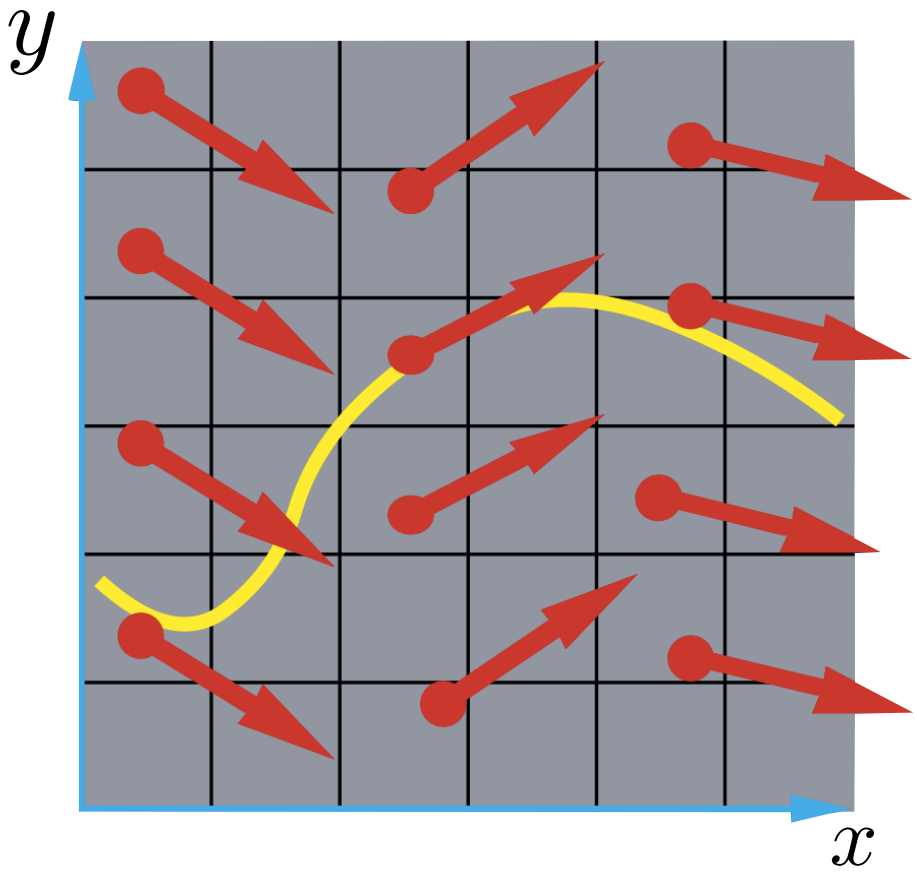
- Highest derivative defines order of ODE

Example: 1st Order ODE

Example Trajectory



Whole Vector Field



[Baraff&Witkin, Siggraph'97 course]

ODEs

- ODE of first order: $y' = f(x, y)$

Remember: *y(x)* written as *y*

- Example $\frac{\mathrm{d}y}{\mathrm{d}x} + \frac{y}{x} = -4\ ,\ x > 0$

- Solution $y = \frac{C}{x} - 2x$

- C is integration constant, to determine it...
- Initial condition needed, e.g. , *y(1)=1*
- Gives *C=3*

ODEs

- Second order ODE: $y'' = f(x, y, y')$
- Solution defined by two initial conditions
- I.e. $y(x_0) = y_0 \ , \ y'(x_0) = y'_0$

- Can be converted into a system of two first order ODEs:

$$y'_1(x) = y_2(x)$$
$$y'_2(x) = f(x, y_1, y_2)$$

ODEs

- Problems for which...
  - n initial conditions are given: initial value problems
  - conditions are distributed to different locations: boundary value problems
  - (The latter are usually trickier to solve...)

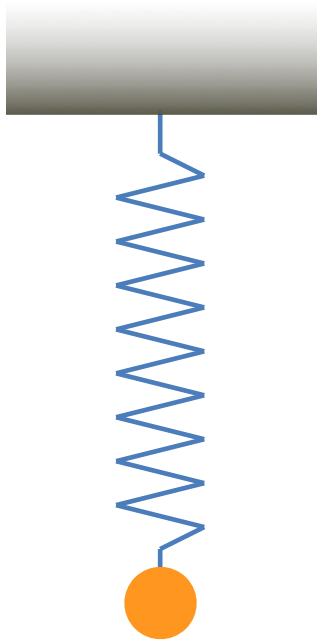
Outlook

- Partial differential equations (PDEs): derivatives with respect to different free variables
- Example: 1D diffusion

$$\frac{\partial y(z,t)}{\partial t} = k\frac{\partial^2 y(z,t)}{\partial^2 z}$$

- Note: boundary value problems with PDEs very important in practice

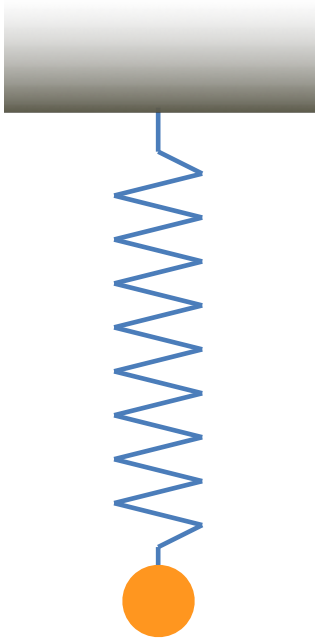
Analytics Solutions for Mass Spring Systems

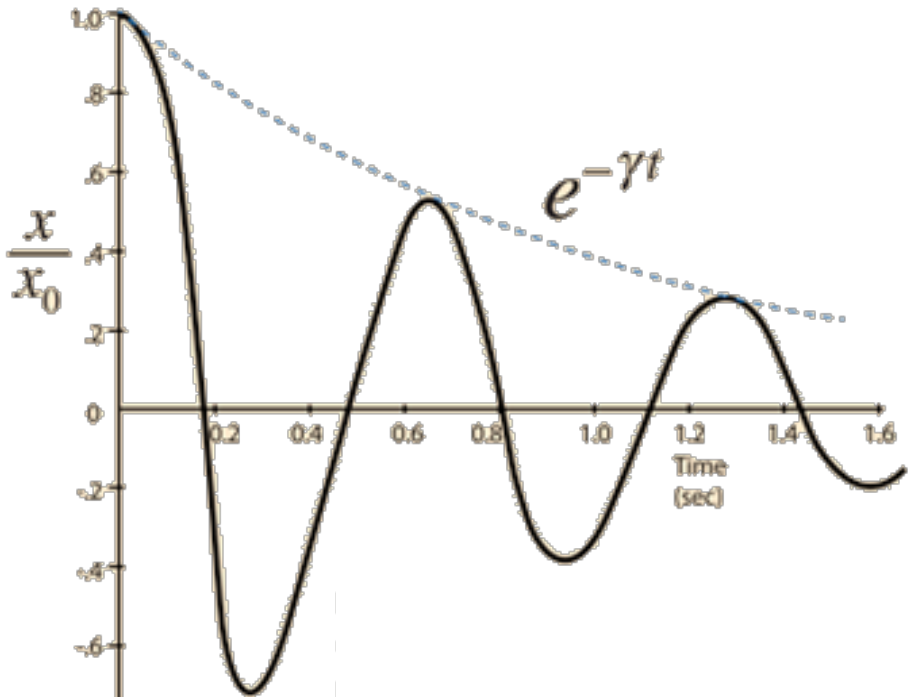


- Simplified setup: 1D Spring, fixed at top (3D spring possible, though)
- Displacement from rest x, initial d
- Thus $F = -kx$
- Resulting ODE: $x'' = -\frac{k}{m}x$
- Harmonic oscillator $x(t) = d\cos(\sqrt{\frac{k}{m}}t)$

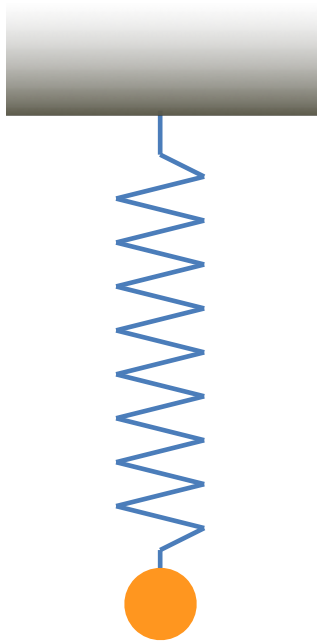
Analytics Solutions for Mass Spring Systems

- We can include damping!
- Gives: $x(t) = e^{-\gamma t} d \cos(\sqrt{\frac{k}{m}} t)$





Summary



- Analytic solution:
  - gives position for arbitrary time
- Will not blow up, no drift!
- Still - not useful in practice
  - Only for simplified setups
  - Not possible for more complex (i.e. realistic) spring networks

Numerical Integration

Notation for ODEs

y	target function to compute
x	free variable
y'	derivative w.r.t. *x*
h	discrete step size
y_n	n-th approximation of y

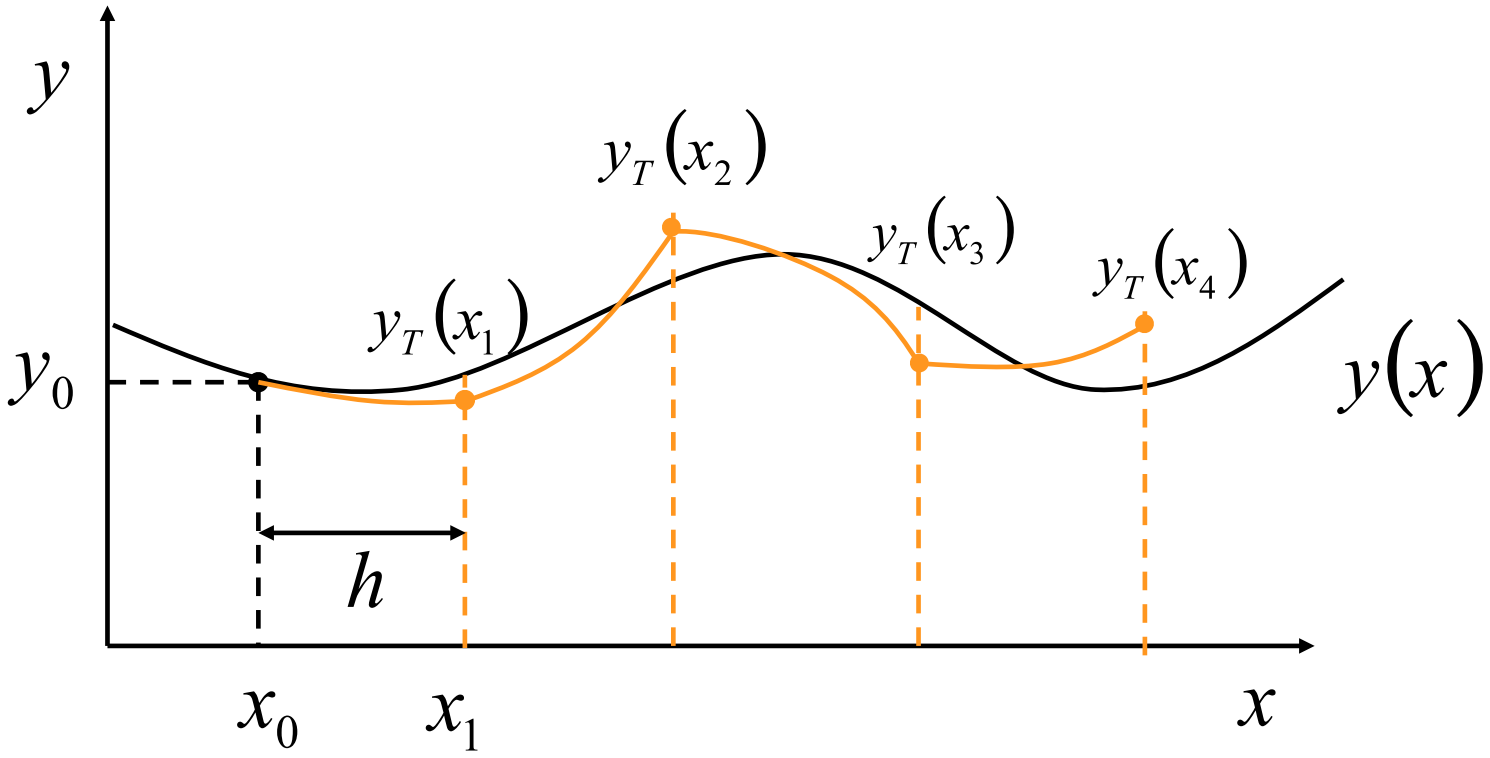
Taylor Expansion

- Start with local Taylor series of the solution
- Discrete step in free variable: $x_{n+1} = x_n + h$
- Expansion:

$$y(x_{n+1}) = y(x_n) + \frac{y'(x_n)}{1!}h + \frac{y''(x_n)}{2!}h^2 + \ldots$$

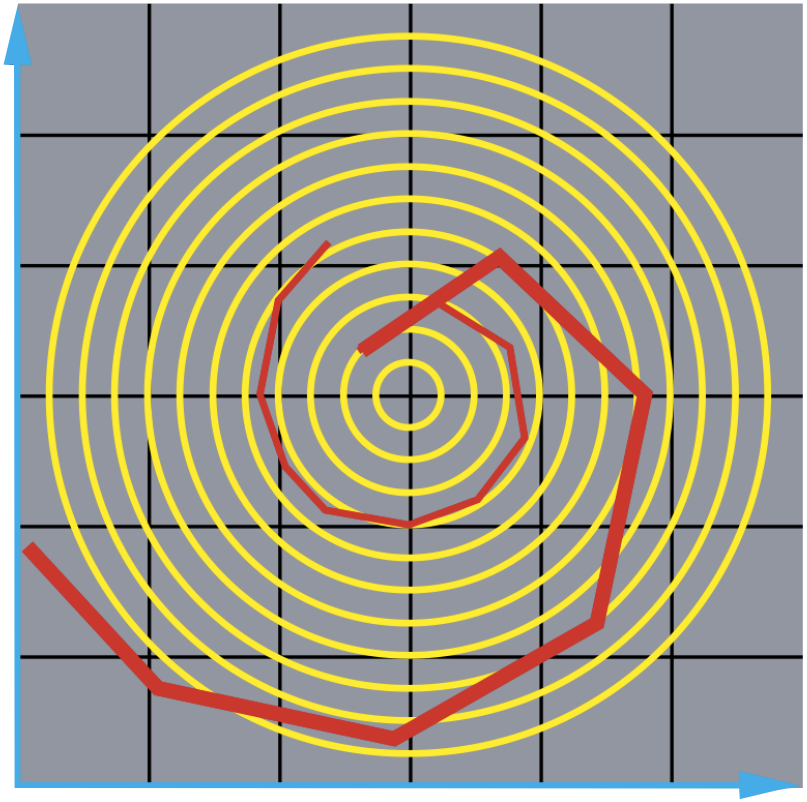
Generic Example with Second Order

$$y(x_{n+1}) = y(x_n) + \frac{y'(x_n)}{1!}h + \frac{y''(x_n)}{2!}h^2 + E$$

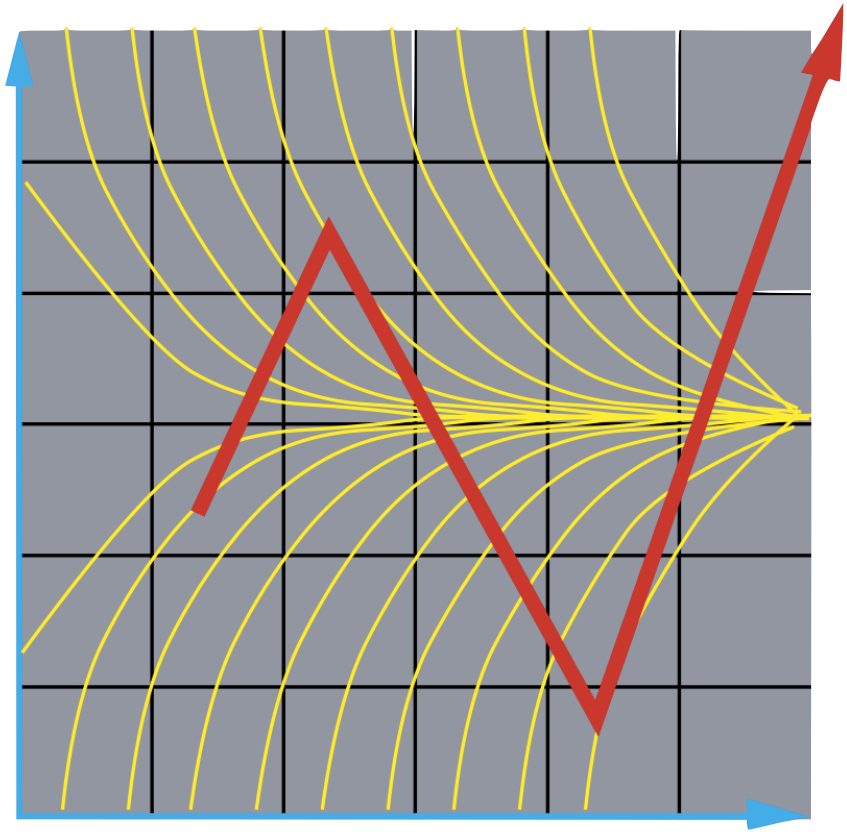


Integration Problems

Getting off-track...



Oscillation / explosion



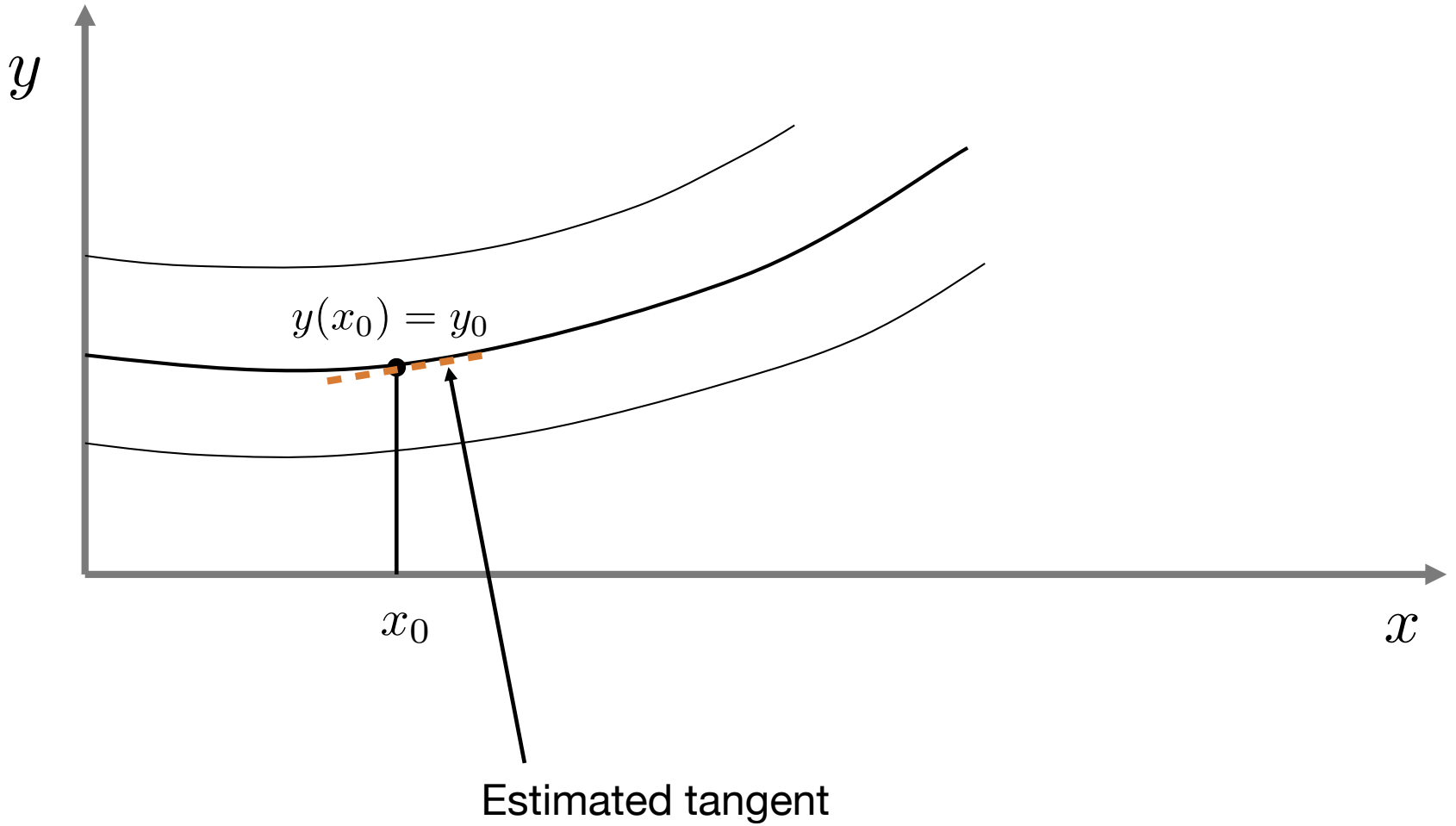
[Baraff&Witkin, Siggraph'97 course]

Euler's Method

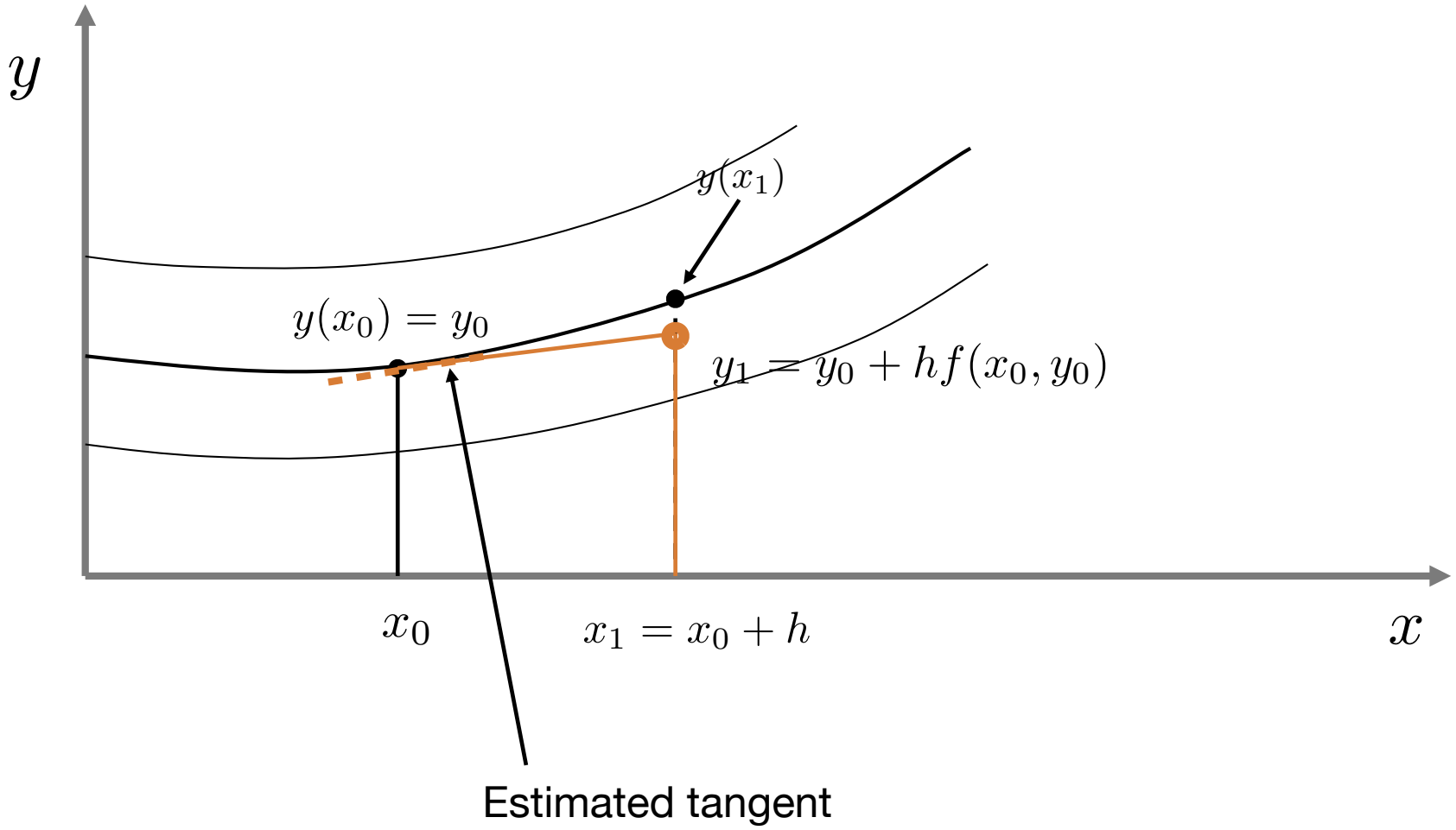
- Euler (*1768*!): start with initial value, then step along tangent
- Given: first order ODE $y' = f(x, y)$
- Initial conditions y_0 , x_0 and step size h
- Step with: $y_1 = y_0 + hf(x_0, y_0)$
- y_1 is an approximation for y(x_0 +h)
- Now we can compute f(x_1 ,y_1) , and so forth...
- Attention: $y_n \neq y(x_0 + nh)$

Notation for approximation	Actual function, we don't know this one!

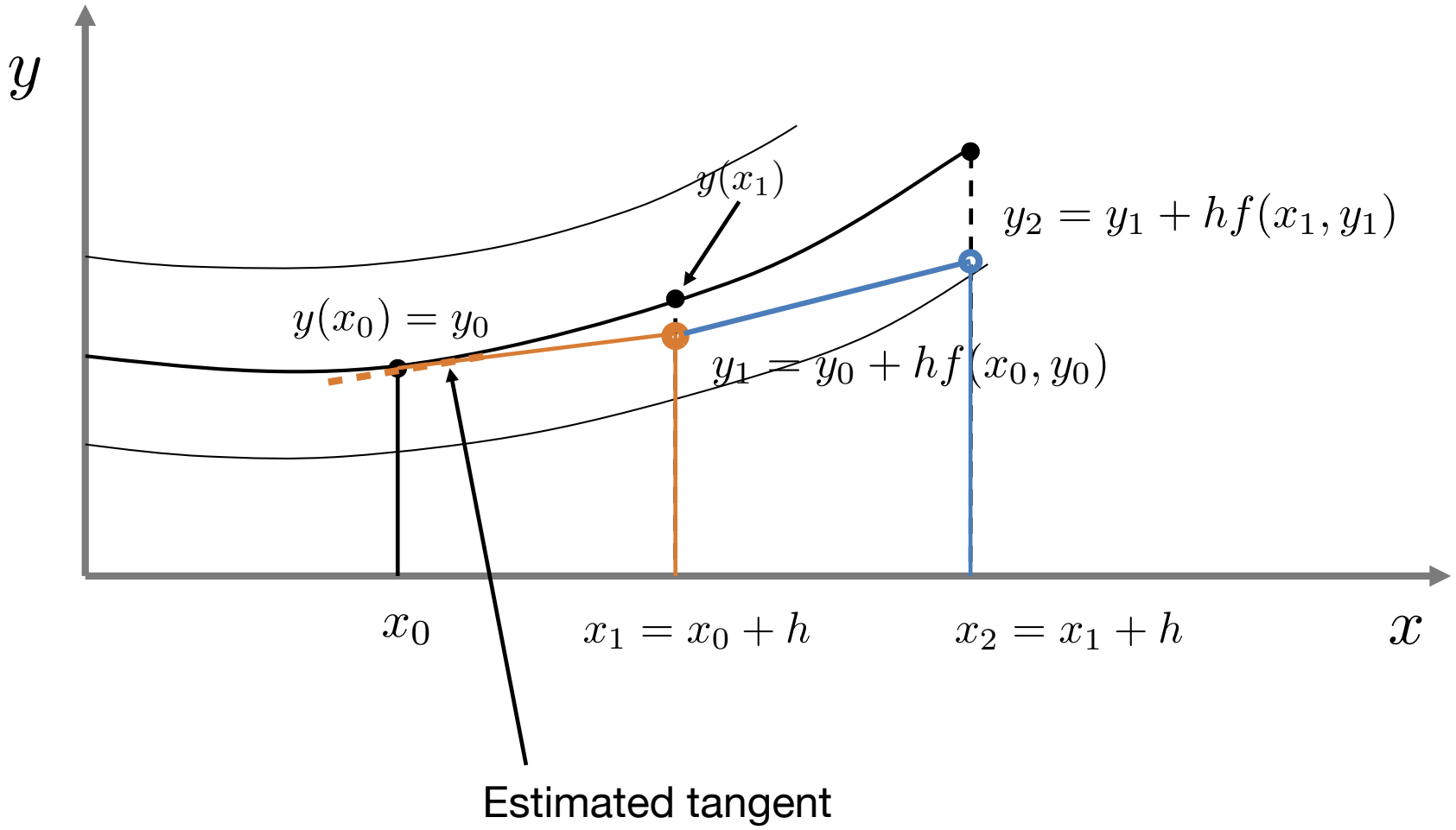
Euler's Method



Euler's Method



Euler's Method



Euler's Method

- Summary
  - Update with $y_{n+1} = y_n + hf(x_n, y_n)$
  - Approximate slope of the tangent based on the current solution
  - Step in direction of the estimated tangent
  - First order method
- How can we do better... ?

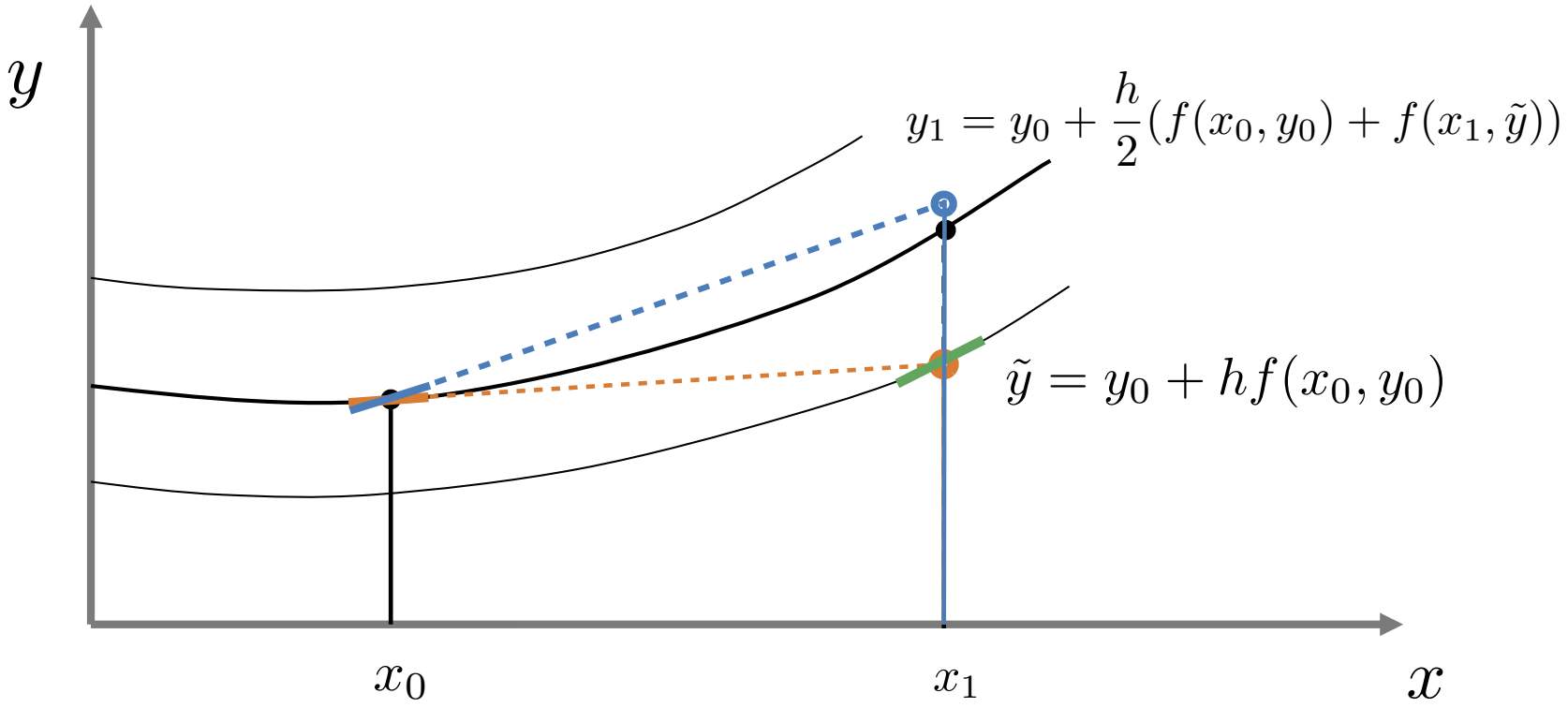
Heun's Method

- Idea: do a step with an averaged tangent
  - based on current state
  - and from an Euler-step
- Thus: $\tilde{y} = y_0 + hf(x_0, y_0)$

$$y_1 = y_0 + \frac{h}{2}(f(x_0, y_0) + f(x_1, \tilde{y}))$$

- Second order!

Heun's Method



Predictor-corrector method: predict slope, then correct and step forward

Midpoint Method

- Evaluate derivative at an intermediate half step
- Actually: Runge Kutta 2nd order
- Gives:

$$\tilde{y} = y_0 + \frac{h}{2} f(x_0, y_0)$$

$$y_1 = y_0 + h\ f(x_0 + h/2, \tilde{y})$$

- Like Heun: second order!

Runge-Kutta Methods

- 4th order RK: "Mother of all integrators"
- Compute a first estimate of the slope

$$k_1 = f(x_0, y_0)$$

- Predict the tangent at midpoint

$$k_2 = f(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_1)$$

- Correct the estimate

$$k_3 = f(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}k_2)$$

4th Order Runge-Kutta

- Predict slope with full step

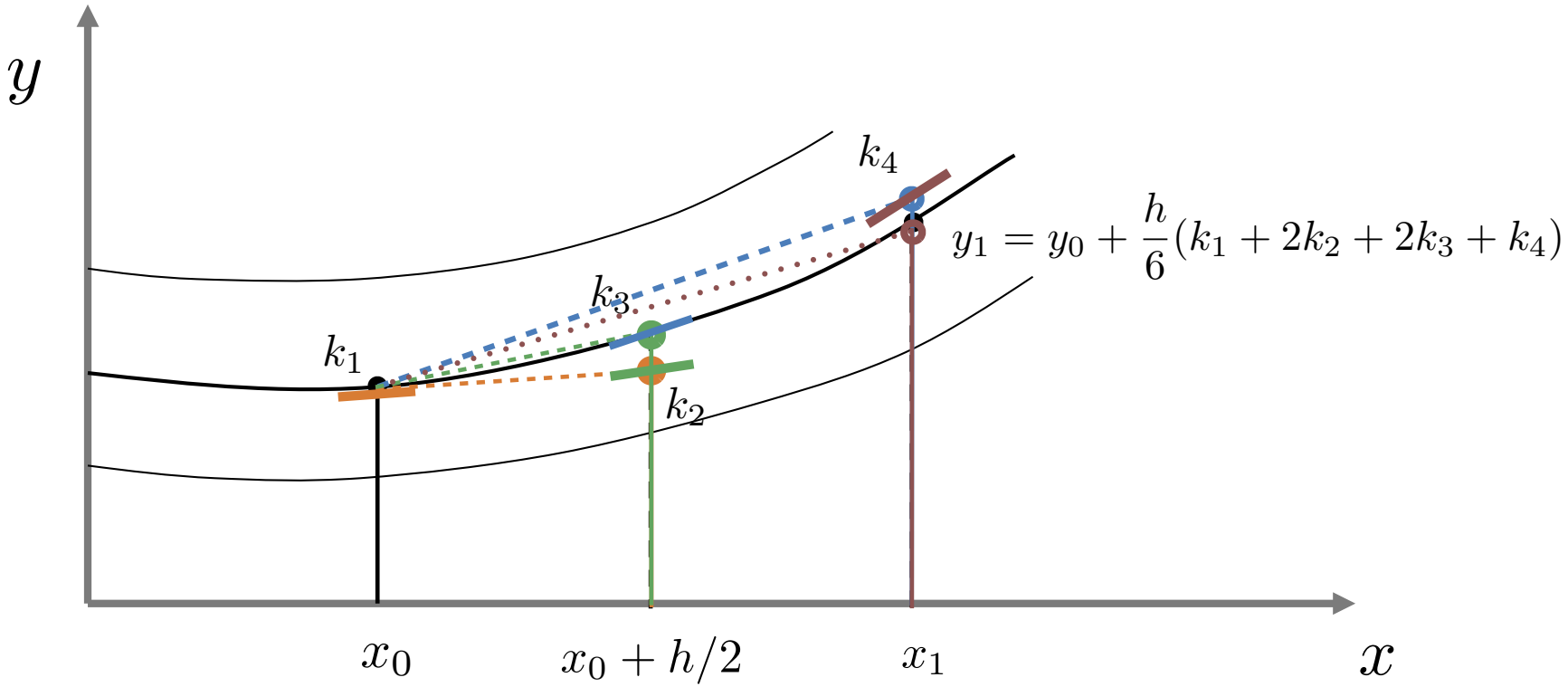
$$k_4 = f(x_0 + h, y_0 + hk_3)$$

- Finally, perform step with weighted slopes

$$y_1 = y_0 + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

- Fourth order method!

RK4 Method



Runge Kutta Methods

- Can be used to construct integrators of arbitrary order...
- In practice: 4th order RK good compromise
- Also popular: 2nd order RK (midpoint method)

Accuracy

- Euler - first order accuracy, 1 evaluation of *f*
- Heun - 2nd order accuracy, 2 evaluations of *f*
- RK4 - 4th order accuracy, 4 evaluations of *f*
- Instead of RK4, why not simply do 4 Euler steps?

Accuracy

- Evaluate with (very simple) sample ODE:

$$y' = y;\ x_0 = 0,\ y_0 = 1$$

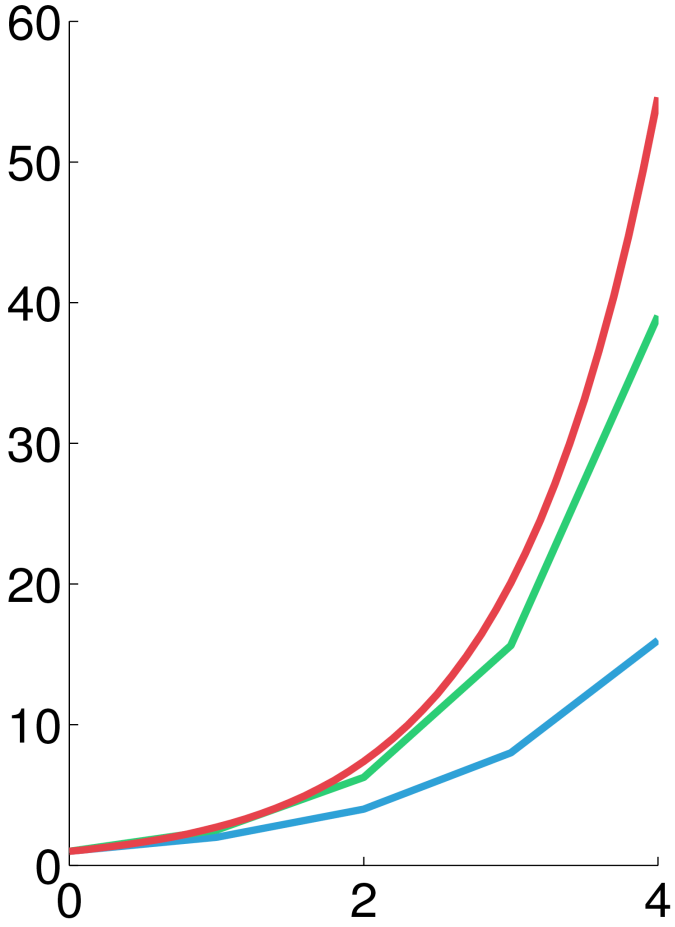
- Real solution: $y(x) = e^x$
- E.g.: calculate Euler step with $y(x) = y + h * y$
- Evaluate in interval *[0, x]* with *n* steps
- We can easily compute error of current solution:

$$|y_n - e^x|$$

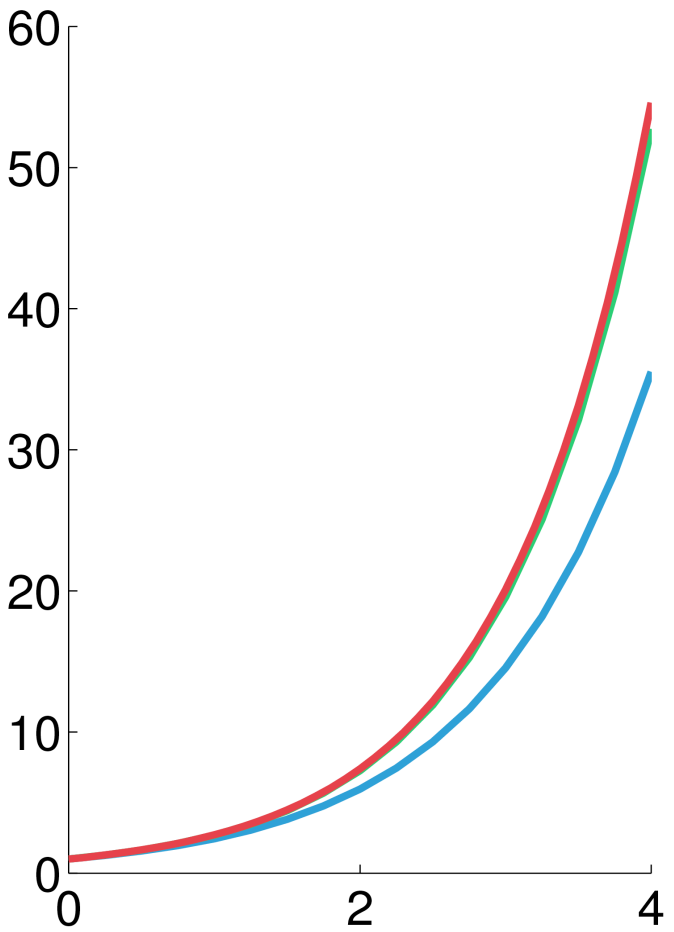
Accuracy

- Simple C++ test
- Aiming for error < *1e-06*
- Required number of steps to reach accuracy:
  - *Simply try out…*

Accuracy



h=1



h=1/4

Reference

Euler

Heun

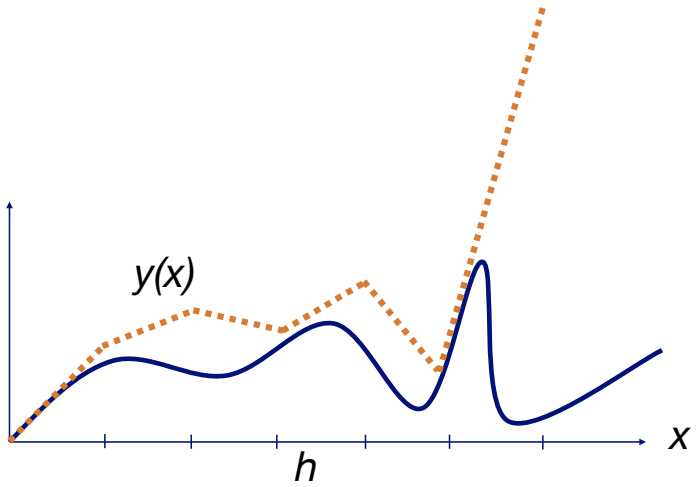
Accuracy - Euler

- Numerical integration is inaccurate

$$y(x+h) = y(x) + hy'(x) + O(h^2)$$

Euler step | Error

- Inaccuracy can cause instability



Bounds of error *e* for Euler step:

$$0 \leq e \leq \frac{h^2}{2} y''(x_e), x_e \in [x, x+h]$$

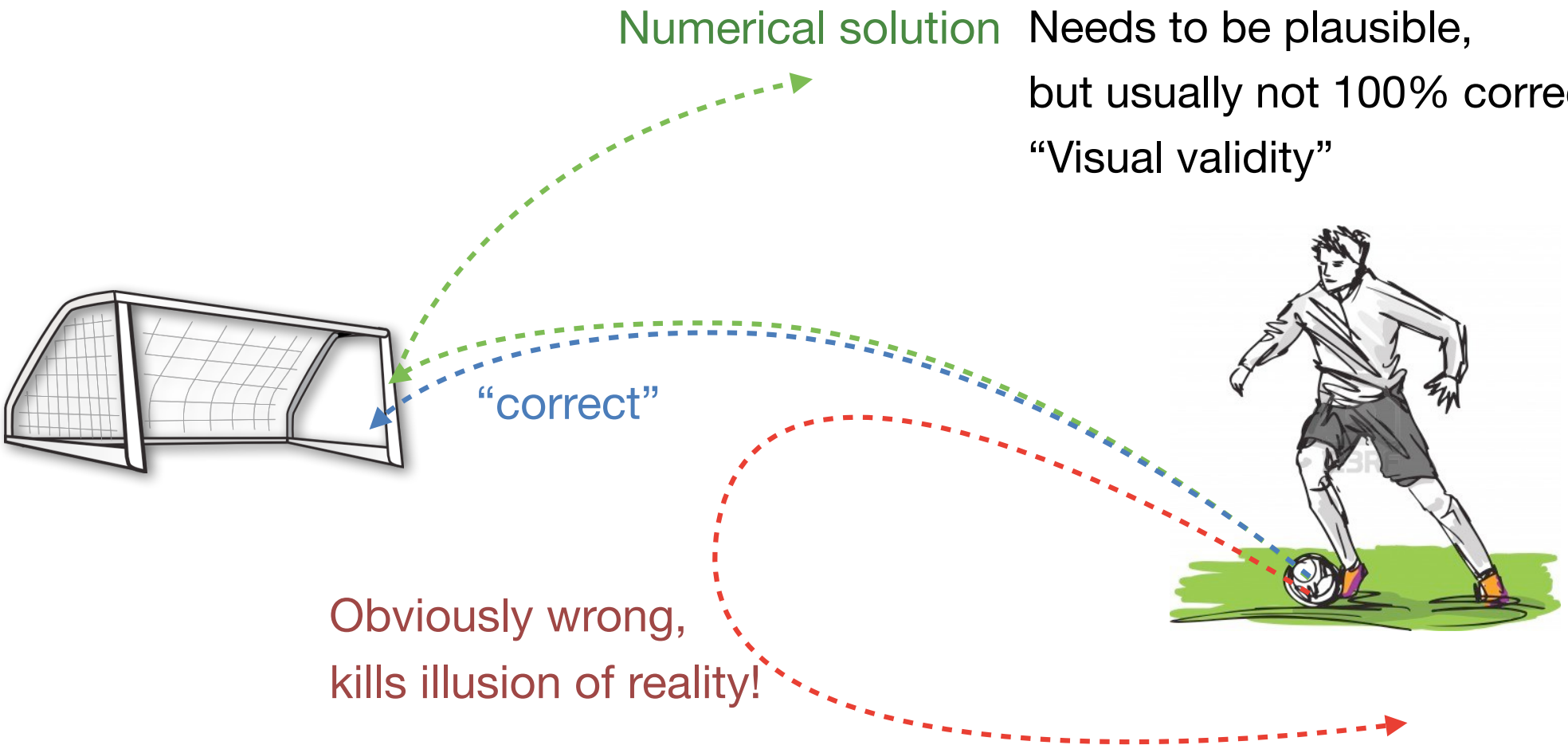
Accuracy - Discussion

- Error with *n* steps of a *1st* order method is usually much higher than *n-th* order method
- Theory:
  - Higher order method require more computations per time step (usually n times more)
  - Allow for larger time steps to be taken
  - Error is bounded by n-th derivative in interval

$$0 \leq e \leq \frac{h^n}{n} y^n$$

Accuracy - Discussion

- In practice



Alternative 1: State-based Methods

- Previously: achieve higher order accuracy by more accurate evaluation of derivatives
- Alternative: perform higher order interpolation of previous computational states to increase accuracy
- Consider 2nd order ODE: $y'' = f(x, y, y')$

Velocity Verlet

- Approach:

$$y_{n+1} = y_n + hy'_n + \frac{h^2}{2}y''_n + O(h^3)$$

From previous step…

Calculate

$$y'_{n+1} = y'_n + h\frac{y''_n + y''_{n+1}}{2} + O(h^3)$$

- Compute y''_{n+1} with y_{n+1}
- Use stored y''_n - data vs. calculation trade off (compare to e.g. midpoint method)

Alternative 2: Implicit Integration

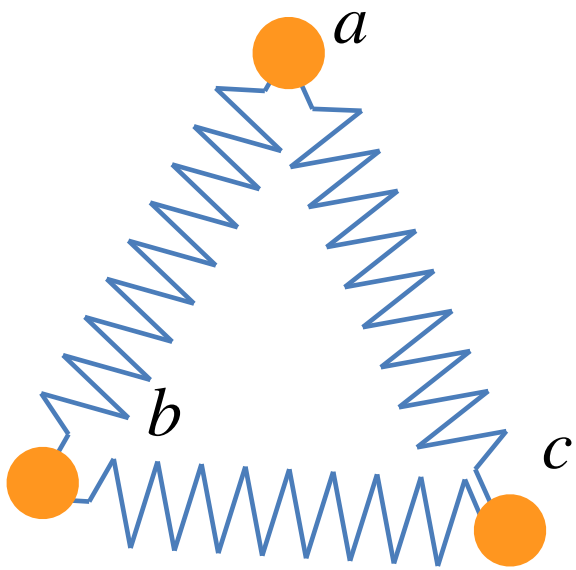
- Implicit Euler: approximate derivative at next time step (instead of current one): $y_{n+1} = y_n + h\, f(x_{n+1}, y_{n+1})$
- Re-arrange and collect coefficients for y_{n+1} in matrix A: $Ay_{n+1} = b$, where b now contains remaining y_n, x_{n+1}
- Now we “just” need bring over A: $y_{n+1} = A^{-1}b$

Alternative 2: Implicit Integration

- Matrix A encodes connectivity of system

- Sketch:

$$Ay \approx \left(\begin{array}{c|c|c} I & b \to a & c \to a \\ \hline a \to b & I & c \to b \\ \hline a \to c & b \to c & I \end{array}\right) \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$



Alternative 2: Implicit Integration

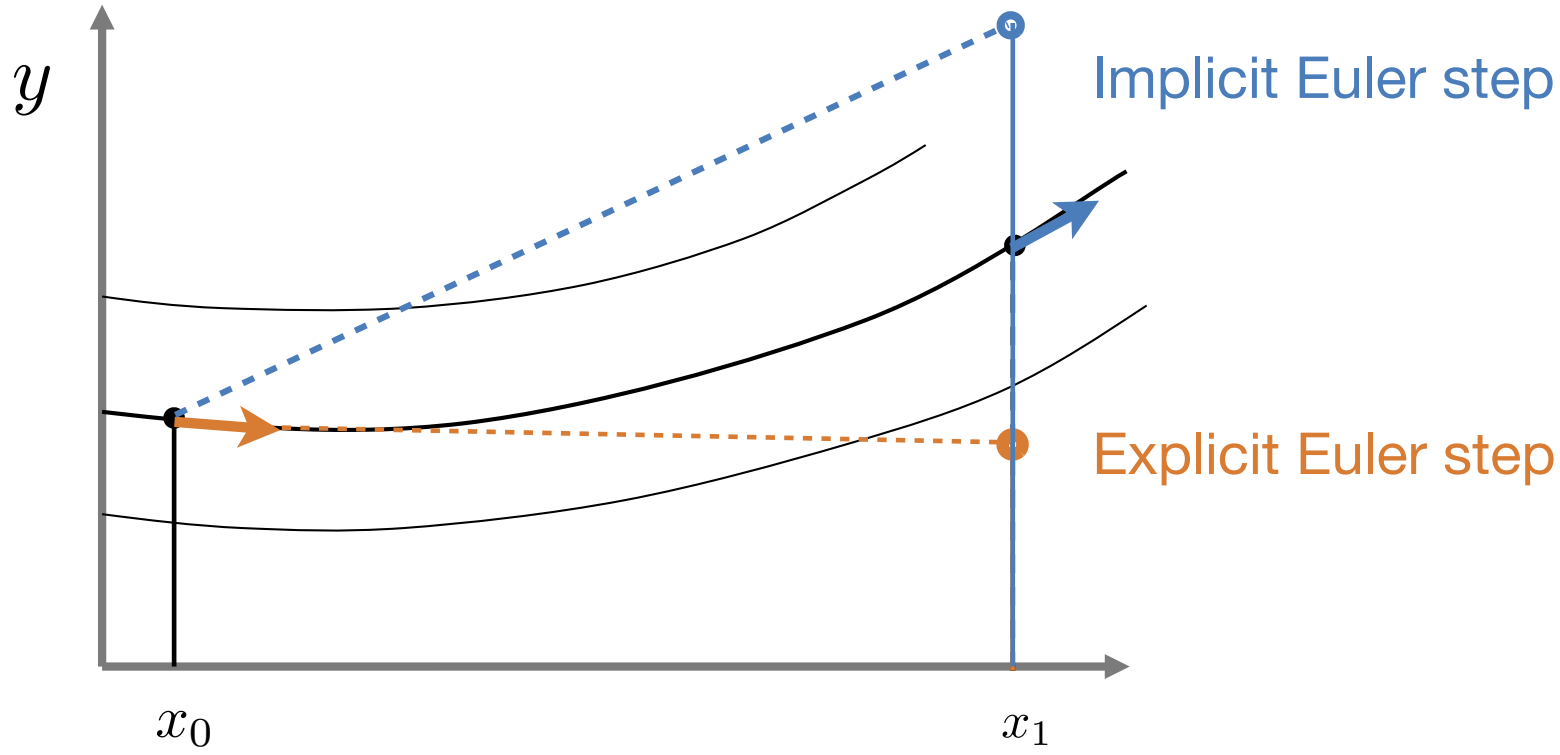
- Implicit Euler: approximate derivative at next time step (instead of current one): $y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$
- Leads to a system of equations that has to be solved

$$Ay_{n+1} = b$$

- Where matrix A encodes dependencies of state variables
  - Fine as long as the matrix is linear
  - Non-linear dependencies (resulting in a non-linear matrix) can be very difficult to solve...

Implicit Integration

- Implicit vs. Explicit Euler:



Implicit Integration

- Ideal for games: fully implicit schemes are unconditionally stable
- In practice:
  - Complex systems can give huge matrices (such as large mass spring networks)
  - Performance depends on solver for linear system (e.g. CG)
  - Non-linear systems are usually uncontrollable
  - Strong implicit damping can look boring
  - Partial implicit integration (e.g., only forces) can still blow up
- Not much used in games; more for, e.g., medical training etc.

Summary

- Instead of analytic evaluation at desired time...
- Numerically approximate (up to certain order) next solution based on current state
  - –> In line with typical computer applications
  - Start with a given initial state
  - Step forward in time & display result
  - Get player input and compute reaction

Summary

- Euler step the most basic (and least accurate)
- Higher-order methods can give significant gains in stability, at higher computational cost
  - Note: largest error dominates in more complex systems
- Implicit methods are often unconditionally stable, but are more difficult to apply
- Be aware of methodology - also when using external software, e.g., physics engines

Integration of Mass-Spring Systems

Notation Reminder for M.S.-systems

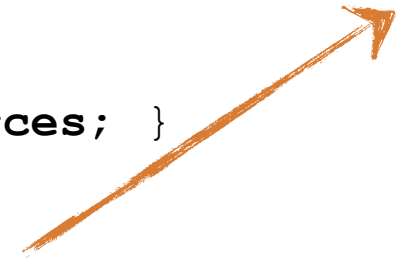
$\mathbf{x}$	position
$\mathbf{v}$	velocity
$\mathbf{a}$	acceleration
m	mass
t	time
h	time step size
k	spring stiffness
γ	damping coefficient
a_i	subscript i: sth. from mass point *i*

Mass Spring Systems

- So far: $m_i \frac{\mathrm{d}^2 \mathbf{x}_i(t)}{\mathrm{d}t^2} = \mathbf{F}_i^{int}(\mathbf{x}_i(t)) + \mathbf{F}_i^{ext}(t) - \gamma \frac{\mathrm{d}\mathbf{x}_i(t)}{\mathrm{d}t}$
- Integrate based on: $\mathbf{x}'(t) = \mathbf{v}(t) \;\; , \mathbf{v}'(t) = \frac{\mathbf{F}(\mathbf{x}(t)) - \gamma \mathbf{v}(t)}{m}$
- Euler step is straight forward:

```
for each timestep {
    for all springs s {
        spring[s].computeElasticForces; }
    for all points i {
        point[i].integratePosition;
        point[i].integrateVelocity; }
}
```

```
newpos = pos + dt * vel;
```



Important: don't re-use data,

Euler means evaluate at time previous time!

Midpoint Integration

- We need to integrate both position and velocity with midpoint method
  - We have $\mathbf{x}(t)$, $\mathbf{v}(t)$
  - We know we need for the position:

$$\tilde{\mathbf{x}} = \mathbf{x}(t) + h/2\mathbf{v}(t, \mathbf{x}(t))$$

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\tilde{\mathbf{v}}$$

  - And for the velocities:

$$\tilde{\mathbf{v}} = \mathbf{v}(t) + h/2\ \mathbf{a}(t, \mathbf{x}(t), \mathbf{v}(t))$$

$$\mathbf{v}(t+h) = \mathbf{v}(t) + h\ \mathbf{a}(t + h/2, \tilde{\mathbf{x}}, \tilde{\mathbf{v}})$$

  - Note: inter-dependencies!

Midpoint Integration

- Which gives the following order:
  - We have *v(t), x(t)*
  - Compute *xtmp* at *t+h/2* based on *v(t)*
  - Compute *a(t)* (i.e., evaluate elastic forces)
  - Compute *vtmp* at *t+h/2* based on *a(t)*
  - Compute *x* at *t+h*
  - Compute *a* at *t+h/2* based on *xtmp* and *vtmp* (elastic forces again!)
  - Compute *v* at *t+h*

$$\tilde{\mathbf{x}} = \mathbf{x}(t) + h/2\mathbf{v}(t, \mathbf{x}(t))$$

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\tilde{\mathbf{v}}$$

$$\tilde{\mathbf{v}} = \mathbf{v}(t) + h/2\,\mathbf{a}(t, \mathbf{x}(t), \mathbf{v}(t))$$

$$\mathbf{v}(t+h) = \mathbf{v}(t) + h\,\mathbf{a}(t + h/2, \tilde{\mathbf{x}}, \tilde{\mathbf{v}})$$

Rk4 Integration

- In line with midpoint method
- Compute interleaved intermediate *x* and *v*
- Then evaluate weighted final value
- Needs more storage: 8 vectors per point!

Leap-Frog Integrator

German: “Bocksprung”

- Special: just one force evaluation, but 2nd order accuracy!
- Interleaved update of velocity and position:

$$\mathbf{v}(t + h/2) = \mathbf{v}(t - h/2) + h\ \mathbf{a}(t)$$
$$\mathbf{x}(t + h) = \mathbf{x}(t) + h\ \mathbf{v}(t + h/2)$$

- Trivial implementation! But significantly more stable than explicit Euler

Implementation

Euler

```
...

computeElasticF;
intEuler_Positions;
intEuler_Velocity;
```

"Correct" Euler step:
both using previous data!

Leap-Frog

(More important for engineering applications - no initial drift!)

```
initVelocity(h/2);

...

computeElasticF;
intEuler_Velocity;
intEuler_Positions;
```

Using current positions!

In practice, doesn't matter that velocity is at (t+h/2)...

Why?

- Overview of derivation: $\mathbf{v}(t + h/2) = \mathbf{v}(t - h/2) + h\ \mathbf{a}(t)$

$$\mathbf{x}(t + h) = \mathbf{x}(t) + h\ \mathbf{v}(t + h/2)$$

- We can insert (1) in (2):

$$\mathbf{x}(t + h) = \mathbf{x}(t) + h\mathbf{v}(t - h/2) + h^2\ \mathbf{a}(t)$$

- Compare to real Taylor expansion:

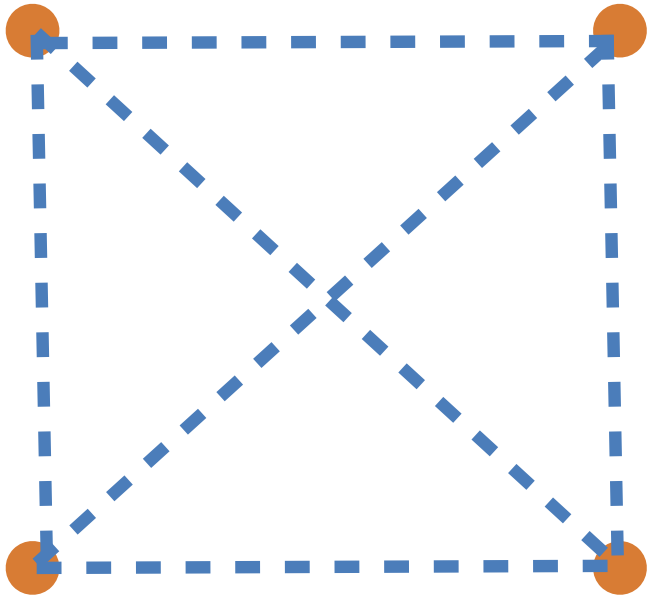
$$\mathbf{x}(t + h) = \mathbf{x}(t) + h\mathbf{v}(t) + h^2/2\ \mathbf{a}(t) + O(h^3)$$

Why?

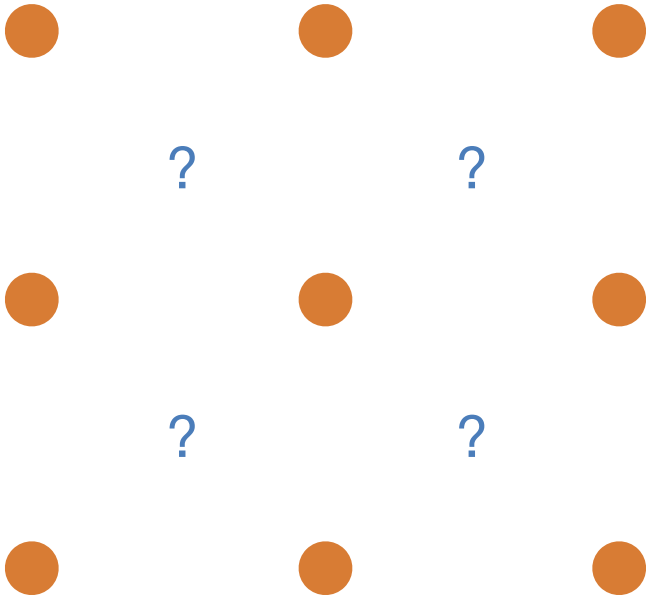
- Overview of derivation:
$$\mathbf{v}(t+h/2) = \mathbf{v}(t-h/2) + h\ \mathbf{a}(t)$$
$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\ \mathbf{v}(t+h/2)$$
- Do a two-way expansion for the velocity
$$\mathbf{v}(t+h) = \mathbf{v}(t) + h\mathbf{v}'(t) + \frac{1}{2}h^2\mathbf{v}''(t) + O(h^3)$$
$$\mathbf{v}(t-h) = \mathbf{v}(t) - h\mathbf{v}'(t) + \frac{1}{2}h^2\mathbf{v}''(t) - O(h^3)$$
- Subtract the two:
$$\mathbf{v}(t+h) - \mathbf{v}(t-h) = 2h\mathbf{v}'(t) + O(h^3)$$
- Second order central difference for acceleration!

Discretization

- Initial model



- Refined model, should give similar behavior



Difficult: How to connect springs? How to choose spring stiffness?

Mass-Spring Systems - Wrap-up

Conclusions - Numerical Integration

- Be aware of integration methods at hand
- For mass spring systems:
  - Not much reason to use Explicit Euler
  - At least do Leap-Frog...
- Try to prevent spring inversions when discretizing volumes
- Needed “everywhere”: from physics time steps to updating neural network weights…

What you should know by now

- How to construct & parametrize mass-spring models, and how to compute their dynamics
- Calculate the solutions for simple spring networks using pen and paper
- What are limitations & properties of the physical model and the chosen discretizations?
- The most common numerical integrators (and when to use which one)