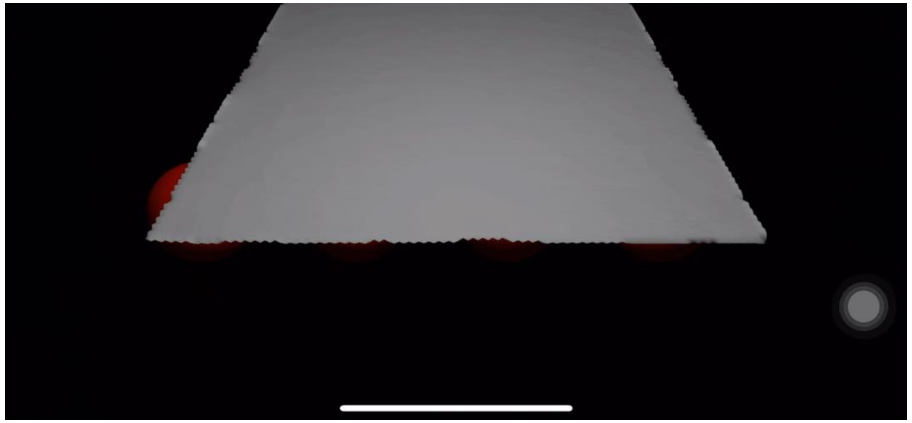
太极图形课

第09讲 Deformable Simulation 02: The Implicit Integration Methods

Where are we?

Procedural Animation

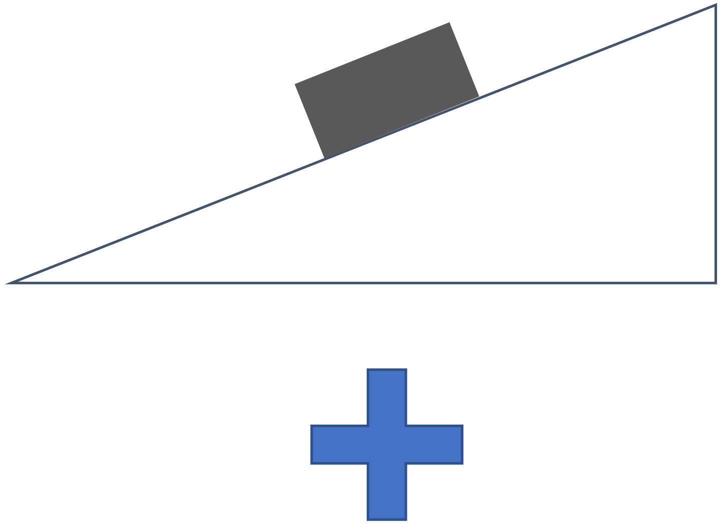
Rendering



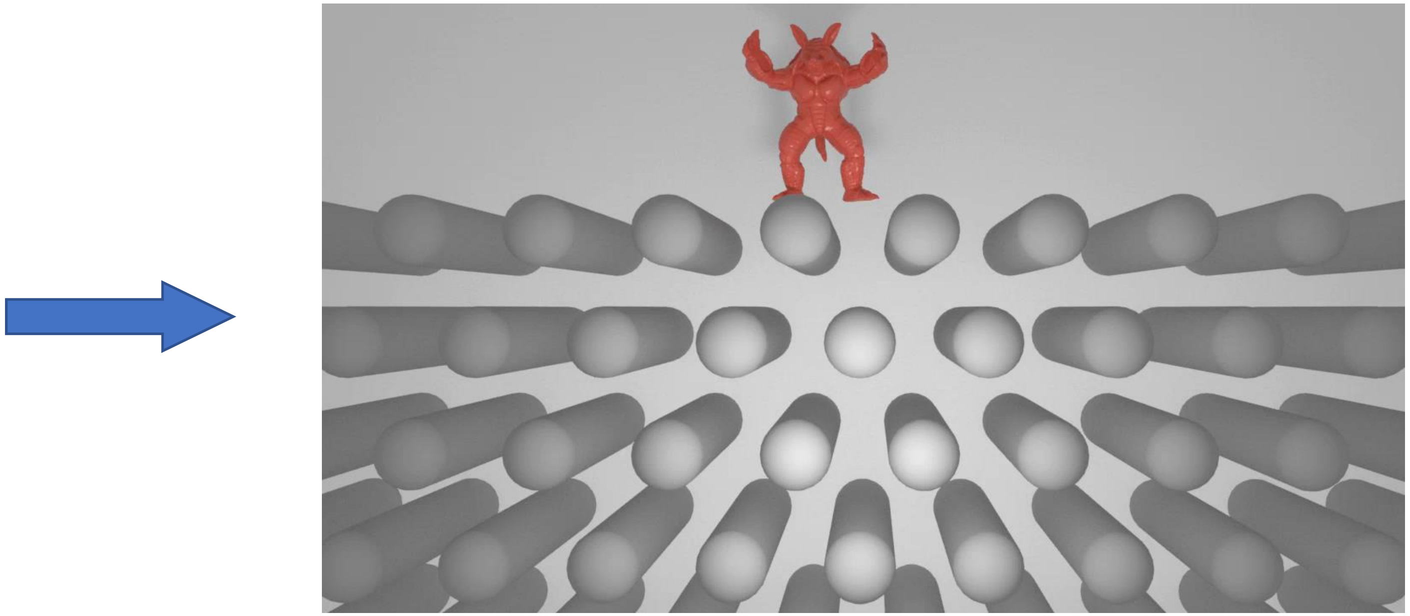
Deformable Simulation

Fluid Simulation

Previously in this Taichi graphics course:
A practitioner's guide to build your first deformable object simulator

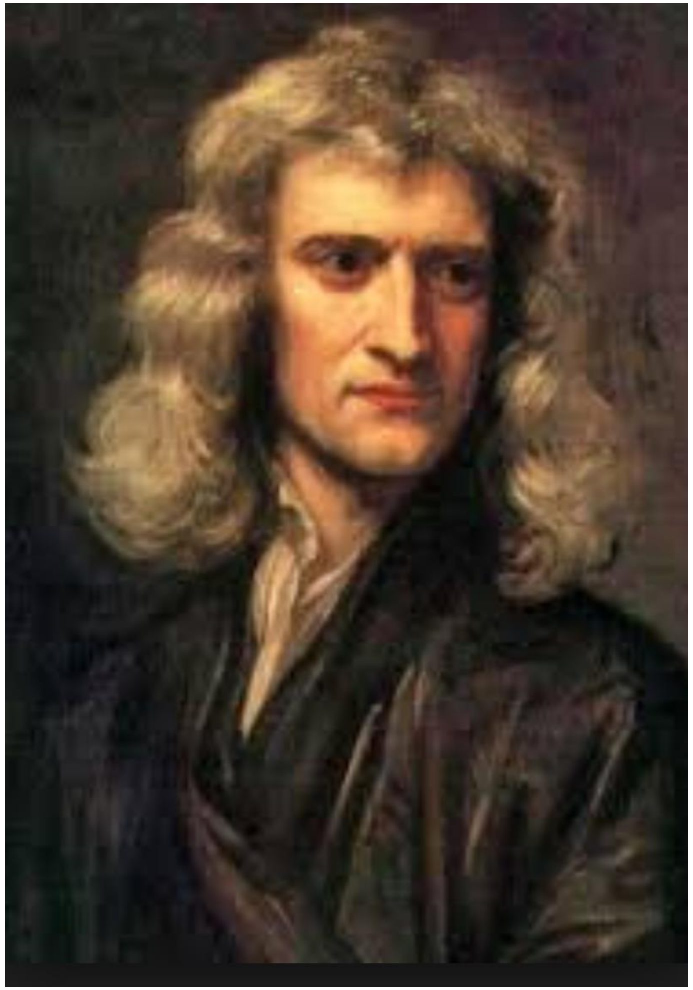


Some details



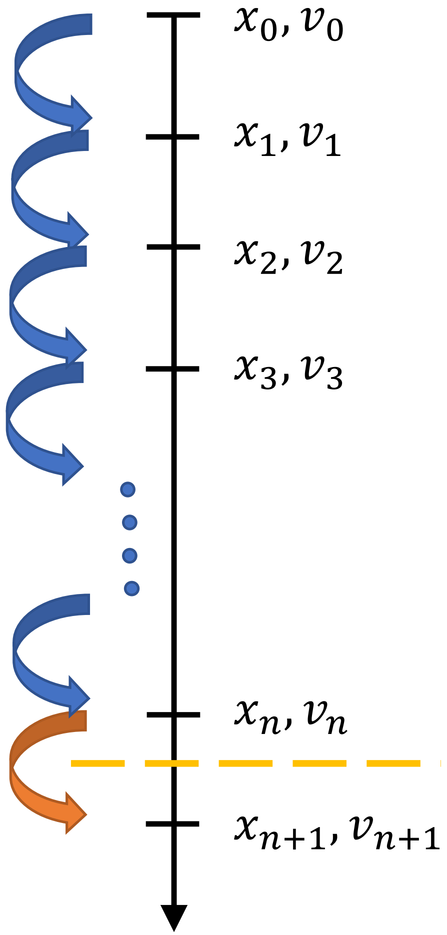
Recap

- Laws of physics
  - Equations of motion
- Integration in time
- Integration in space
  - A simple (but useful) model: mass-spring system
  - Constitutive models
  - The finite element method



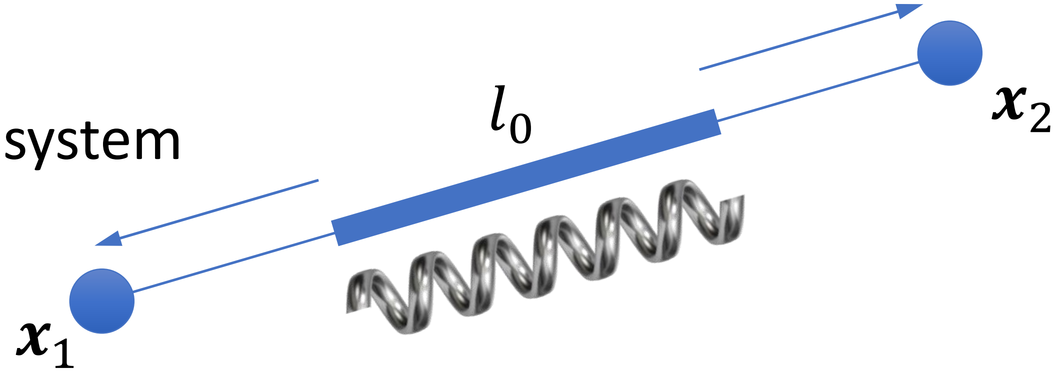
Recap

- Laws of physics
  - Equations of motion
- Integration in time
  - Numerical quadrature
- Integration in space
  - A simple (but useful) model: mass-spring system
  - Constitutive models
  - The finite element method



Recap

- Laws of physics
  - Equations of motion
- Integration in time
  - Numerical quadrature
- Integration in space
  - A simple (but useful) model: mass-spring system
  - Constitutive models
  - The finite element method



Recap

- Laws of physics
  - Equations of motion
- Integration in time
  - Numerical quadrature
- Integration in space
  - A simple (but useful) model: mass-spring system
  - Constitutive models
  - The finite element method

$$\phi$$

$$F$$

$$\epsilon$$

$$\Psi\big(\epsilon(F)\big)$$

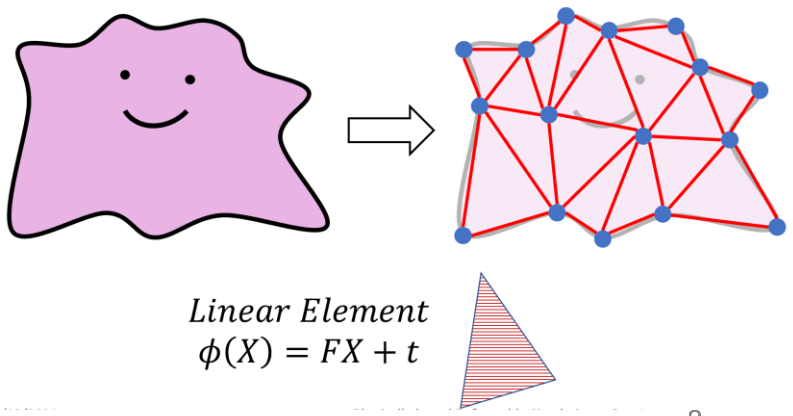
$$P$$

$$E = \int \Psi$$

$$f = -\frac{\partial E}{\partial x}$$

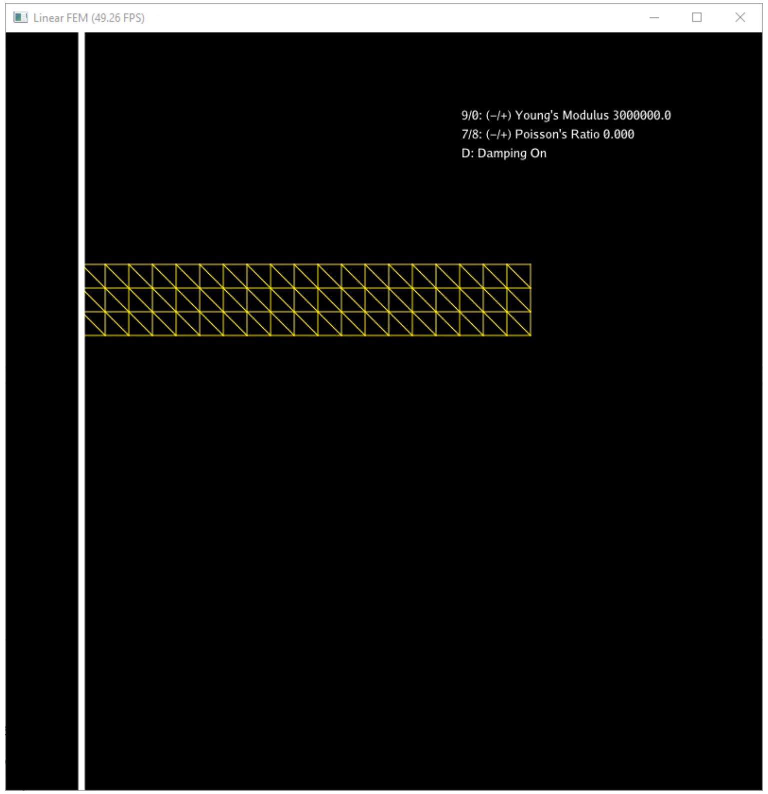
Recap

- Laws of physics
  - Equations of motion
- Integration in time
  - Numerical quadrature
- Integration in space
  - A simple (but useful) model: mass-spring system
  - Constitutive models
  - The finite element method



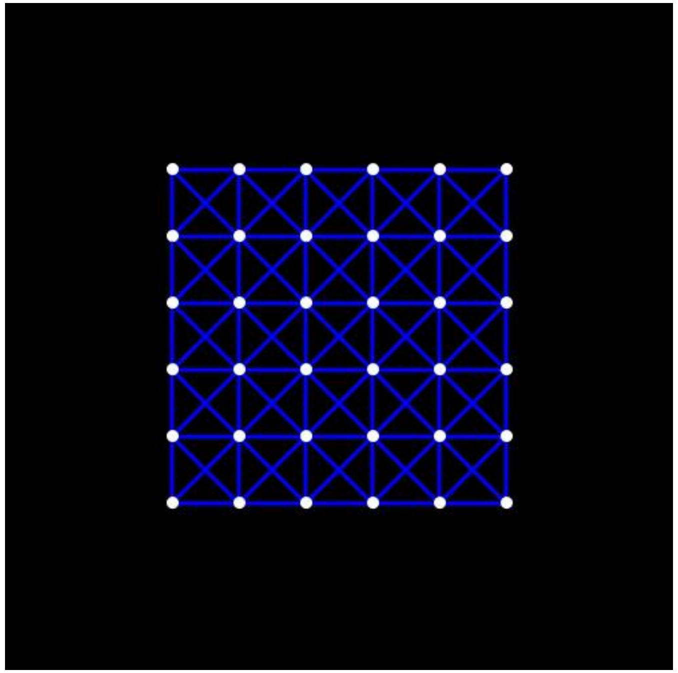
Recap

- Laws of physics
  - Equations of motion
- Integration in time
  - Numerical quadrature
- Integration in space
  - A simple (but useful) model: mass-spring system
  - Constitutive models
  - The finite element method



Code of the day

- https://github.com/taichiCourse01/--Deformables



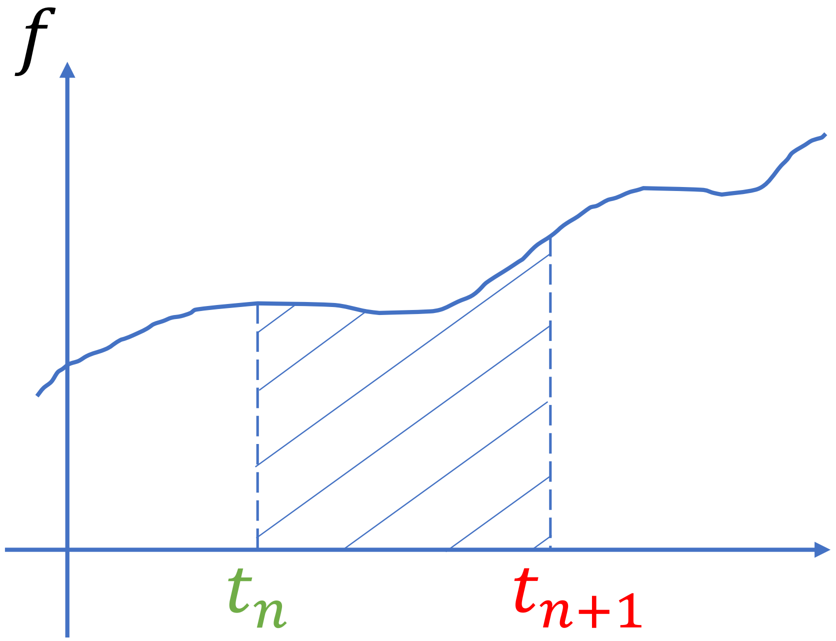
Outline today

- The implicit Euler integration
- Numerical recipes for implicit integrations
- Linear solvers

The implicit Euler integration

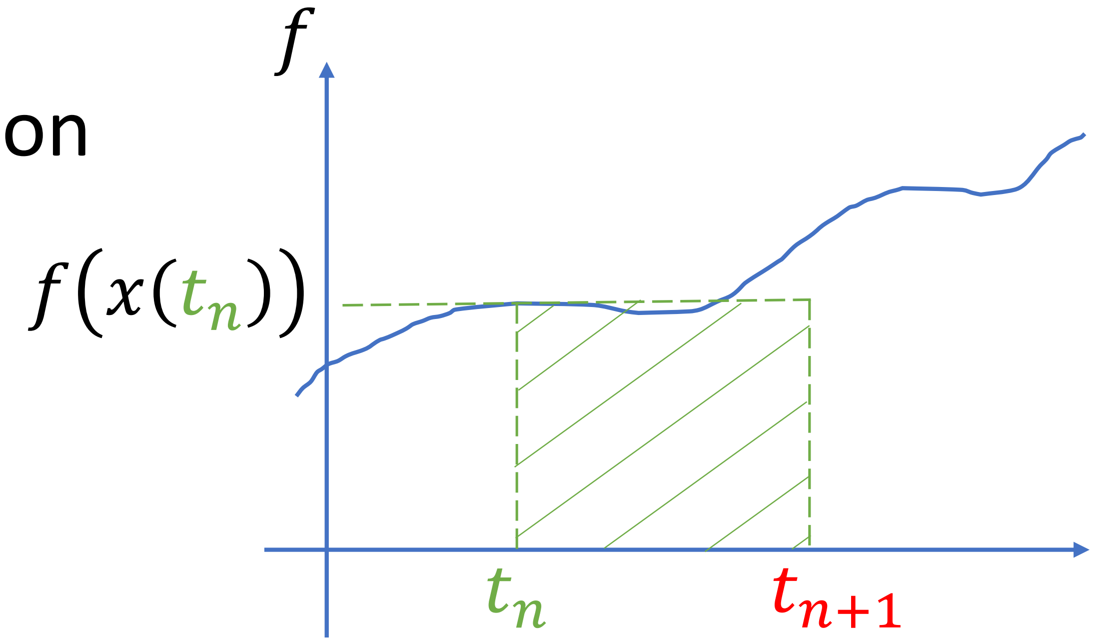
Time integration

- $x(t_n + h) = x(t_n) + \int_0^h v(t_n + t)dt$
- $v(t_n + h) = v(t_n) + \int_0^h M^{-1} f(t_n + t)dt$



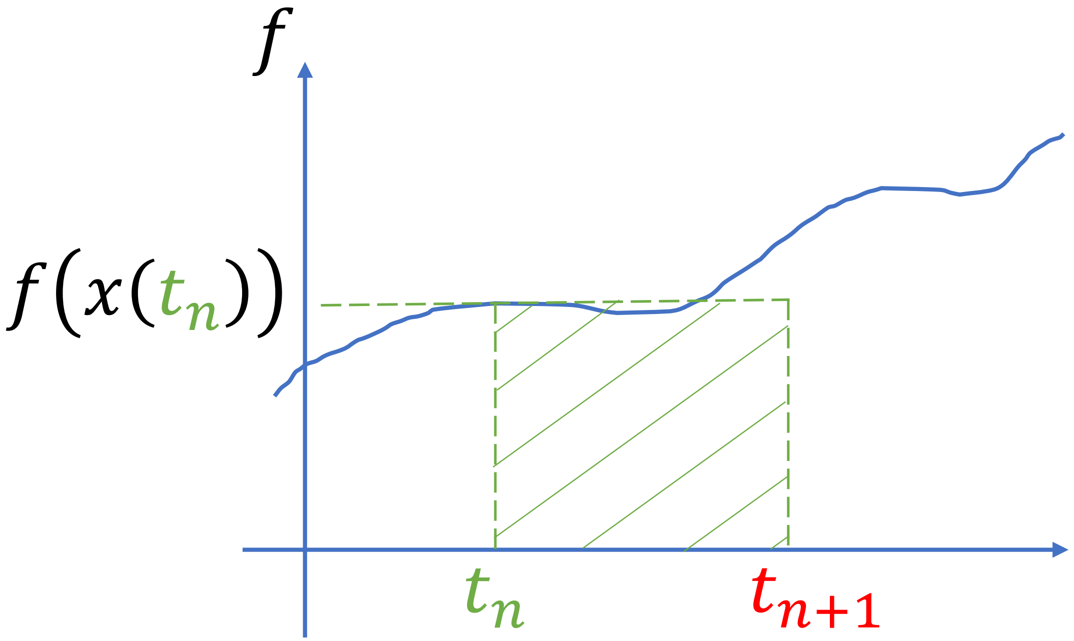
Time integration (explicit)

- Explicit(forward) Euler integration
  - $x_{n+1} = x_n + hv_n$
  - $v_{n+1} = v_n + hM^{-1}f(x_n)$



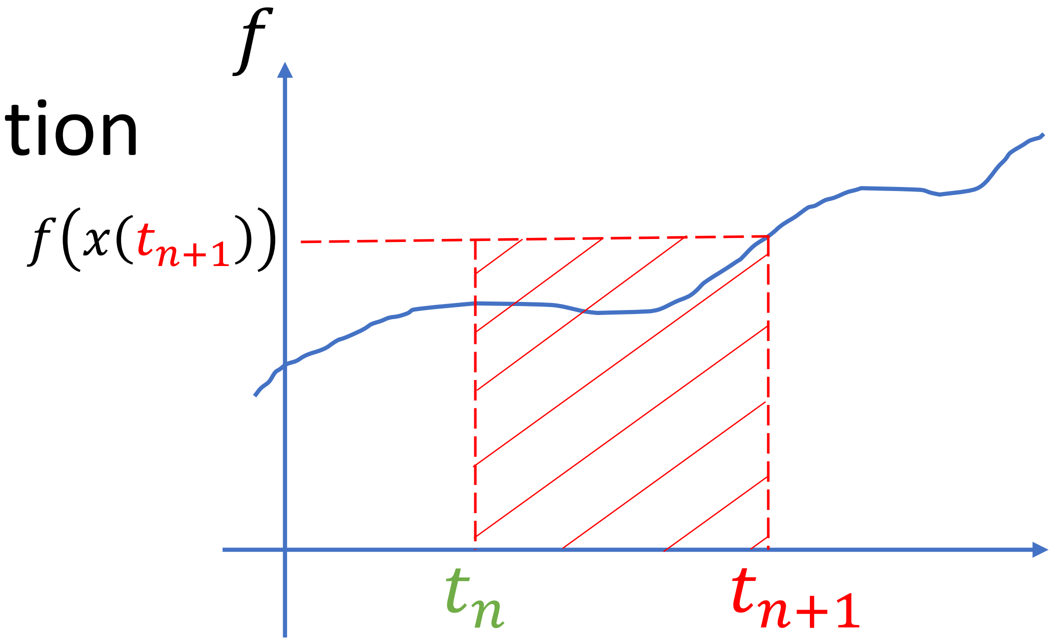
Time integration (explicit)

- Symplectic Euler integration
  - $v_{n+1} = v_n + hM^{-1}f(x_n)$
  - $x_{n+1} = x_n + hv_{n+1}$



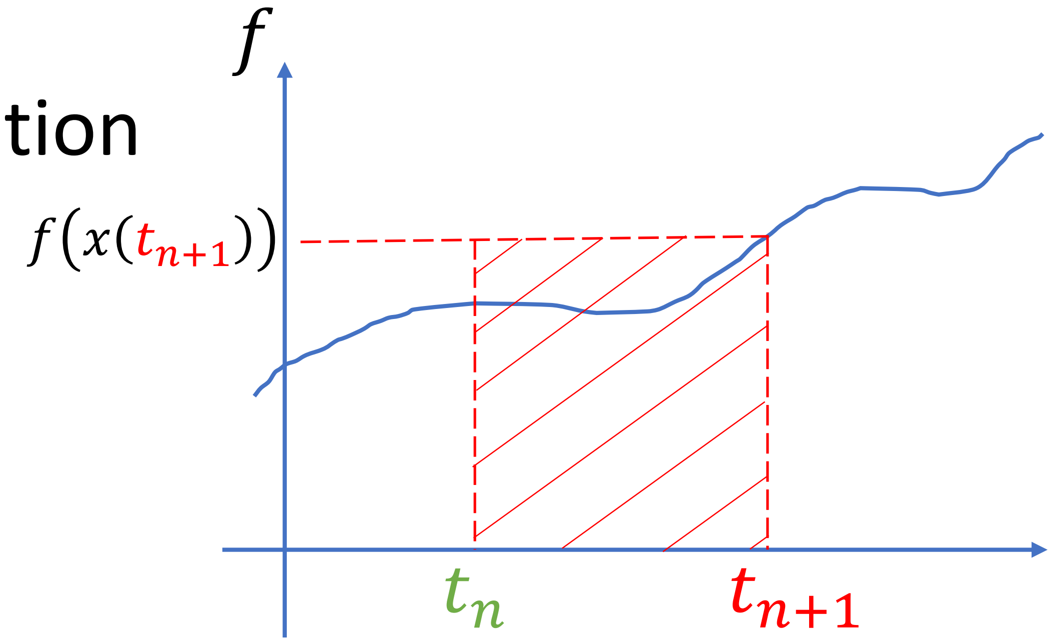
Time integration (implicit)

- Implicit (backward) Euler integration
  - $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
  - $x_{n+1} = x_n + hv_{n+1}$



Why implicit integration?

- Implicit (backward) Euler integration
  - $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
  - $x_{n+1} = x_n + hv_{n+1}$

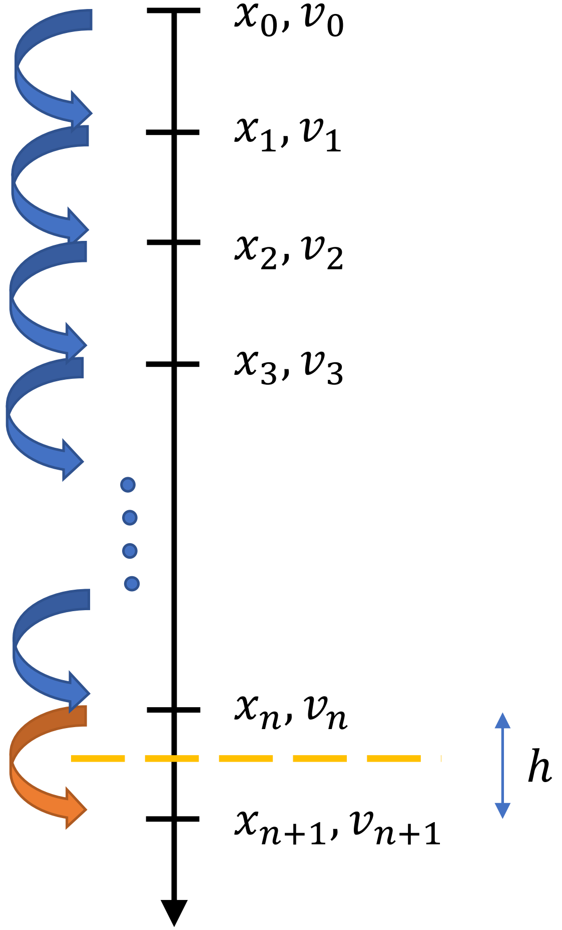


Note: Implicit Euler is often **expensive** due to the nonlinear optimization, it **damps the Hamiltonian** from the oscillating components, it is often **stable for large time-steps** and is widely used in **performance-centric applications**. (game / MR / design / animation)

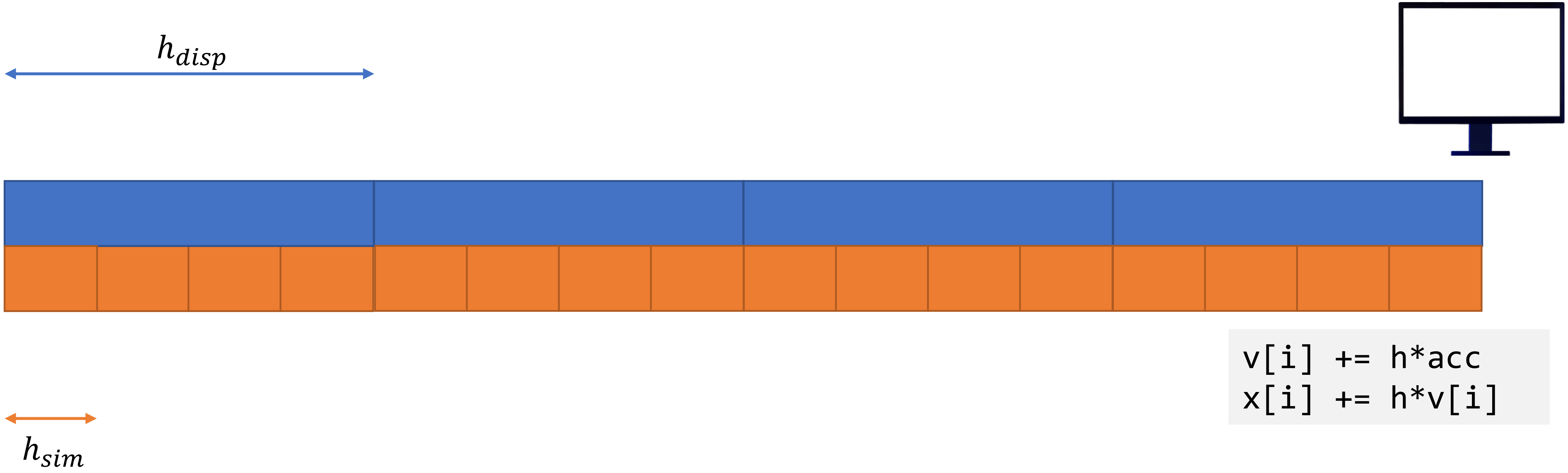
Time

- A time-step in simulation:
  - The time difference between the adjacent ticks on the temporal axis for your simulation h_{sim}
    - `v[i] += h*acc`
    - `x[i] += h*v[i]`

- A time-step in your display:
  - The time difference between two images displayed on your screen
    - For a 60-Hz application, the time between two images is $h_{disp} = \frac{1}{60}$ seconds



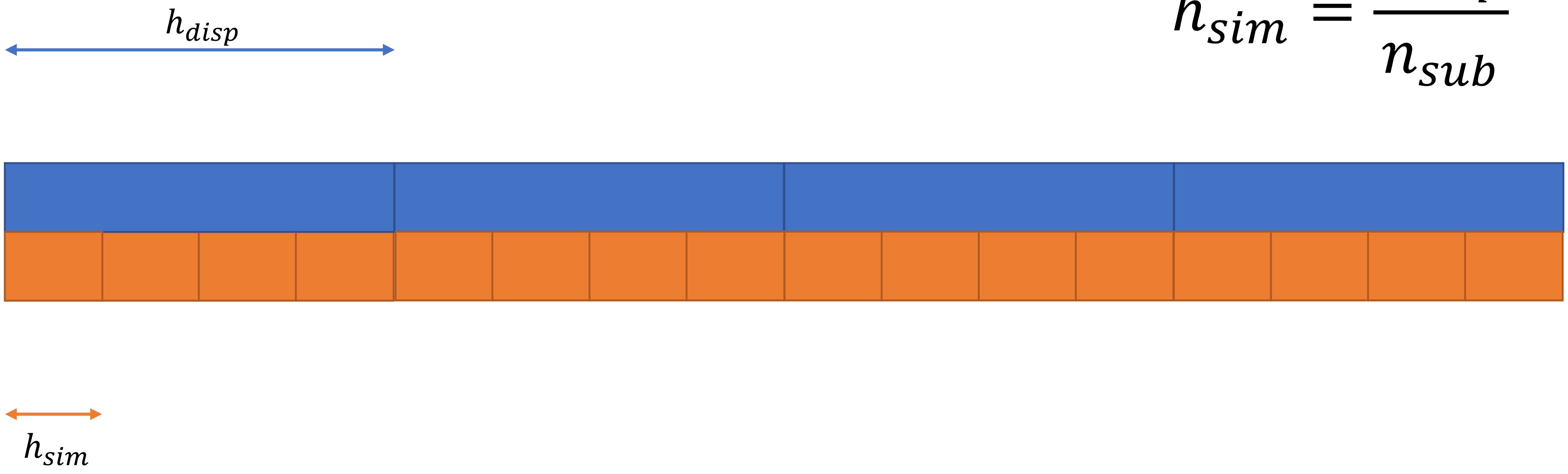
Time




```
v[i] += h*acc
x[i] += h*v[i]
```

Sub-(time)-stepping: n_{sub}

$$h_{sim} = \frac{h_{disp}}{n_{sub}}$$



The smaller n_{sub}, the larger h_{sim}

$$h_{sim} = \frac{h_{disp}}{n_{sub}}$$

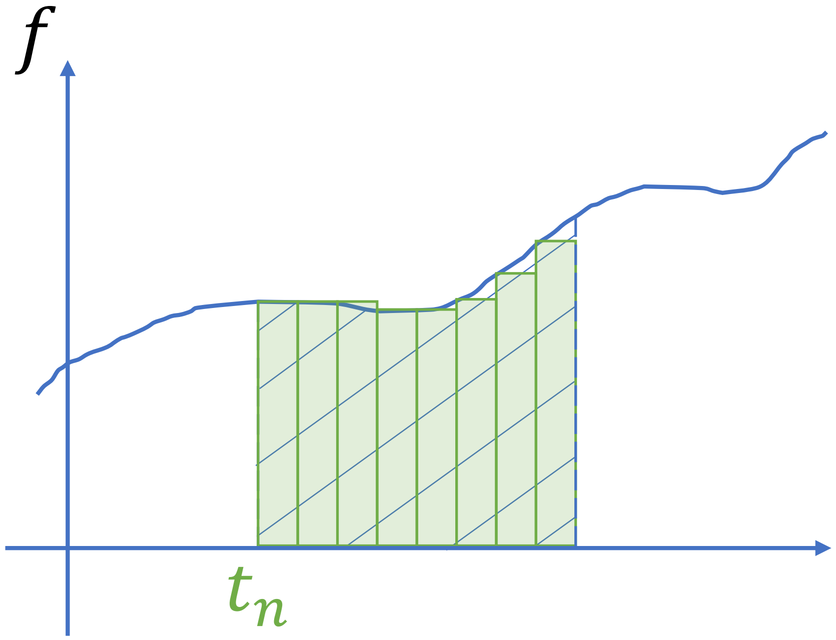
The smaller n_{sub}, the larger h_{sim}

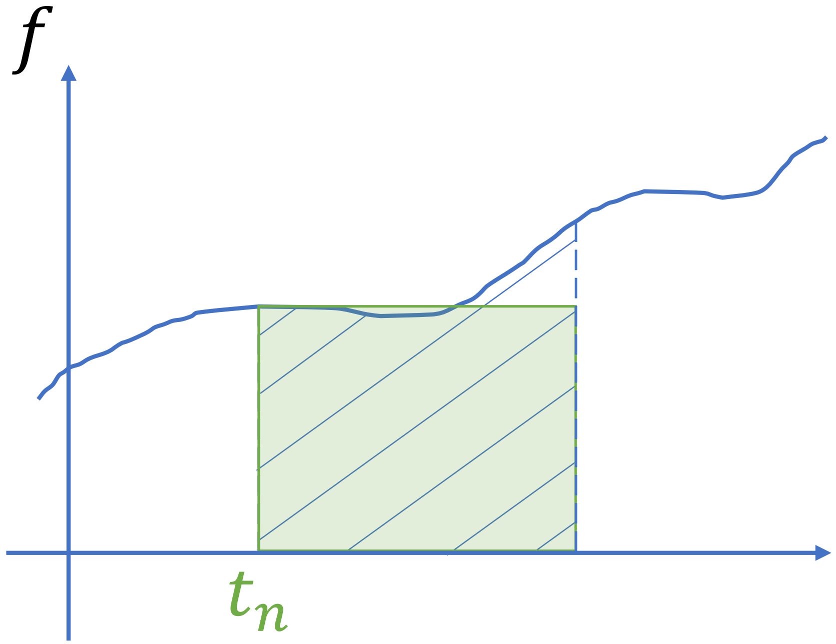
$$h_{sim} = \frac{h_{disp}}{n_{sub}}$$

The smaller n_{sub}, the larger h_{sim}

$$h_{sim} = \frac{h_{disp}}{n_{sub}}$$

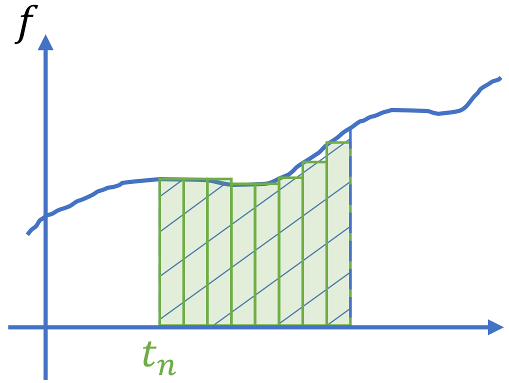
How to pick a proper h_{sim}?

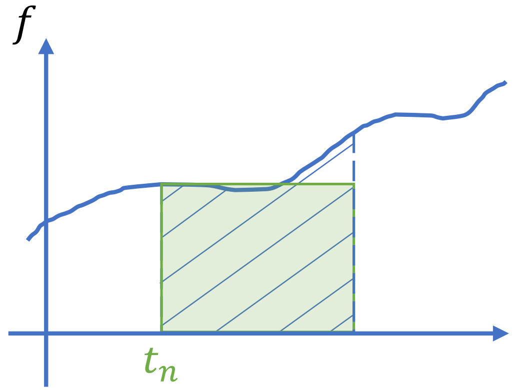




How to pick a proper h_{sim}?

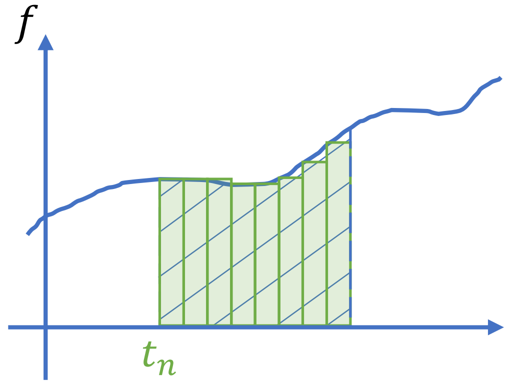
- Pick a:
  - Small h_{sim} for accuracy-centric apps
  - Larger h_{sim} for performance-centric apps

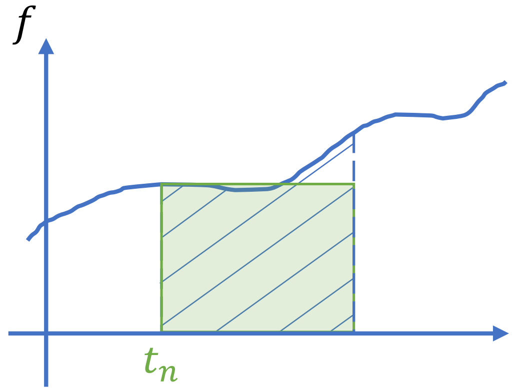




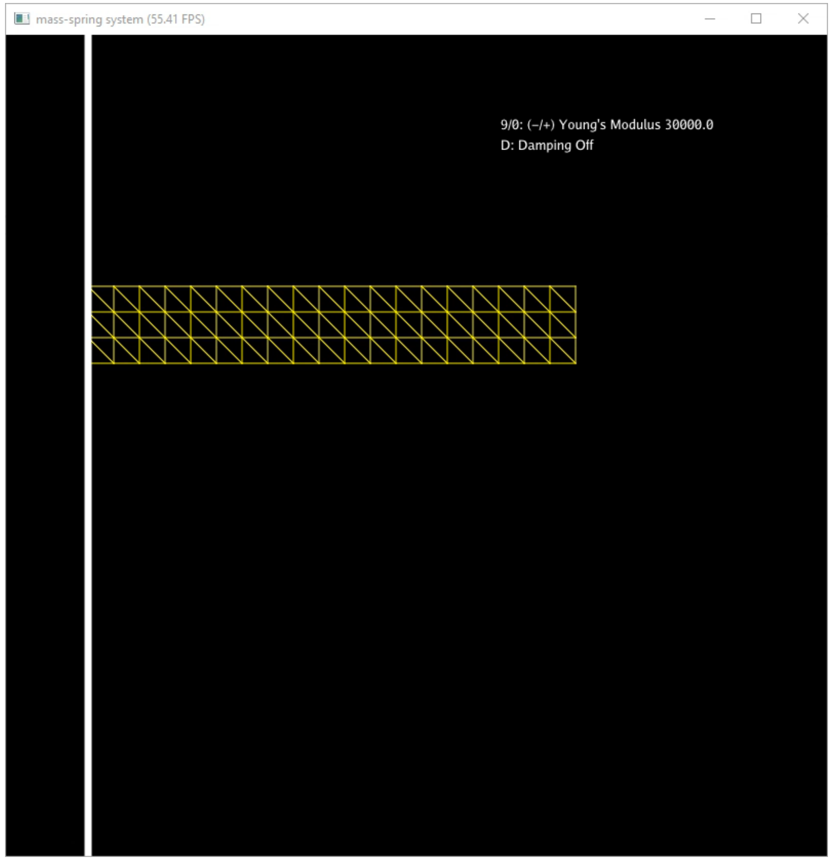
How to pick a proper h_{sim}?

- Pick a:
  - Small h_{sim} for accuracy-centric apps
  - Larger h_{sim} for performance-centric apps
    - Can we set h_{sim} to h_{disp} for real-time applications?

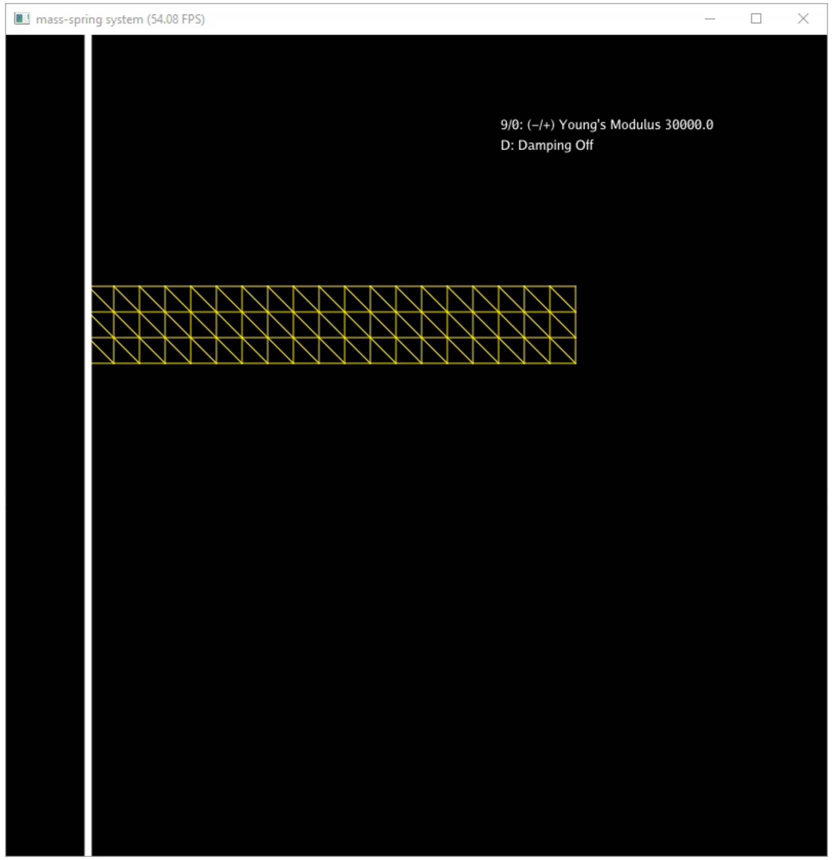




A failure case using explicit integration



$h_{disp} = 1/60\ s$
$n_{sub} = 100$



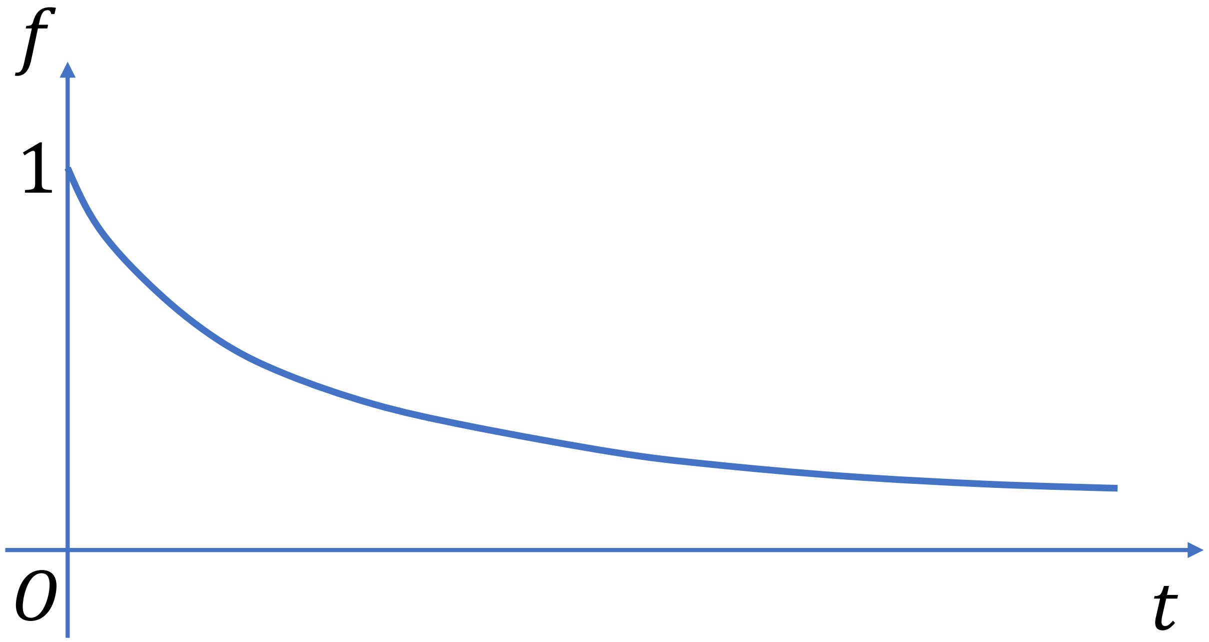
$h_{disp} = 1/60\ s$
$n_{sub} = 10$

Take-away

- We (usually) can not use large time-steps (~10ms) in explicit integration schemes
  - Unless you like EXPLOSION! :P

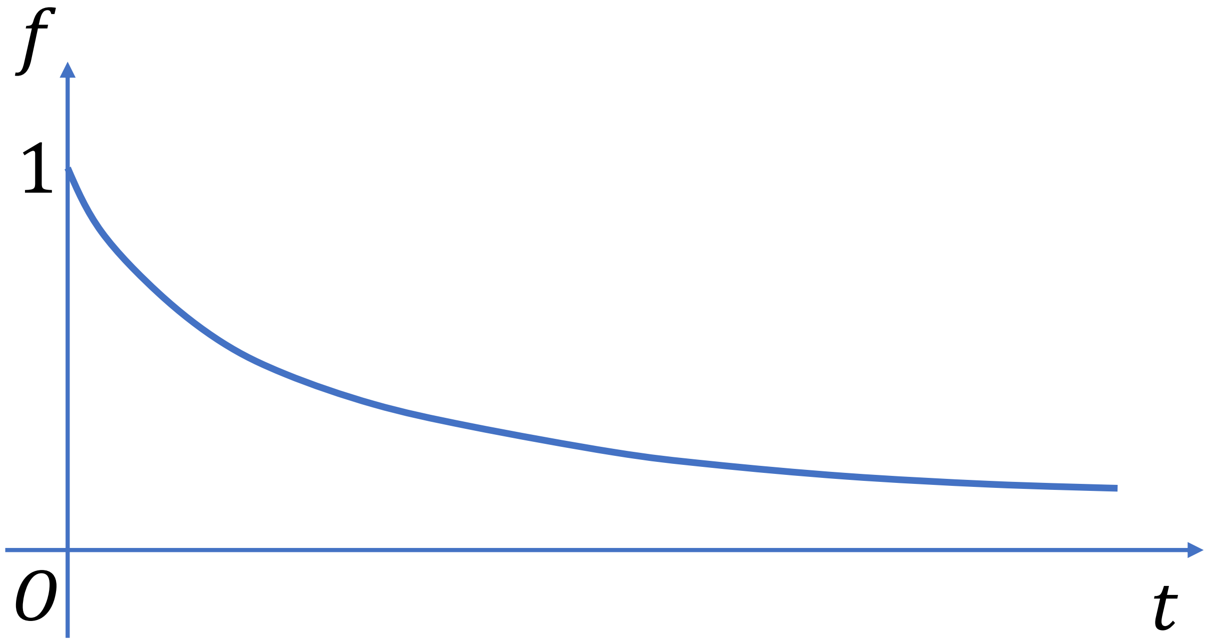
A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- Analytical solution:
  - $f(t) = e^{-\lambda t}$



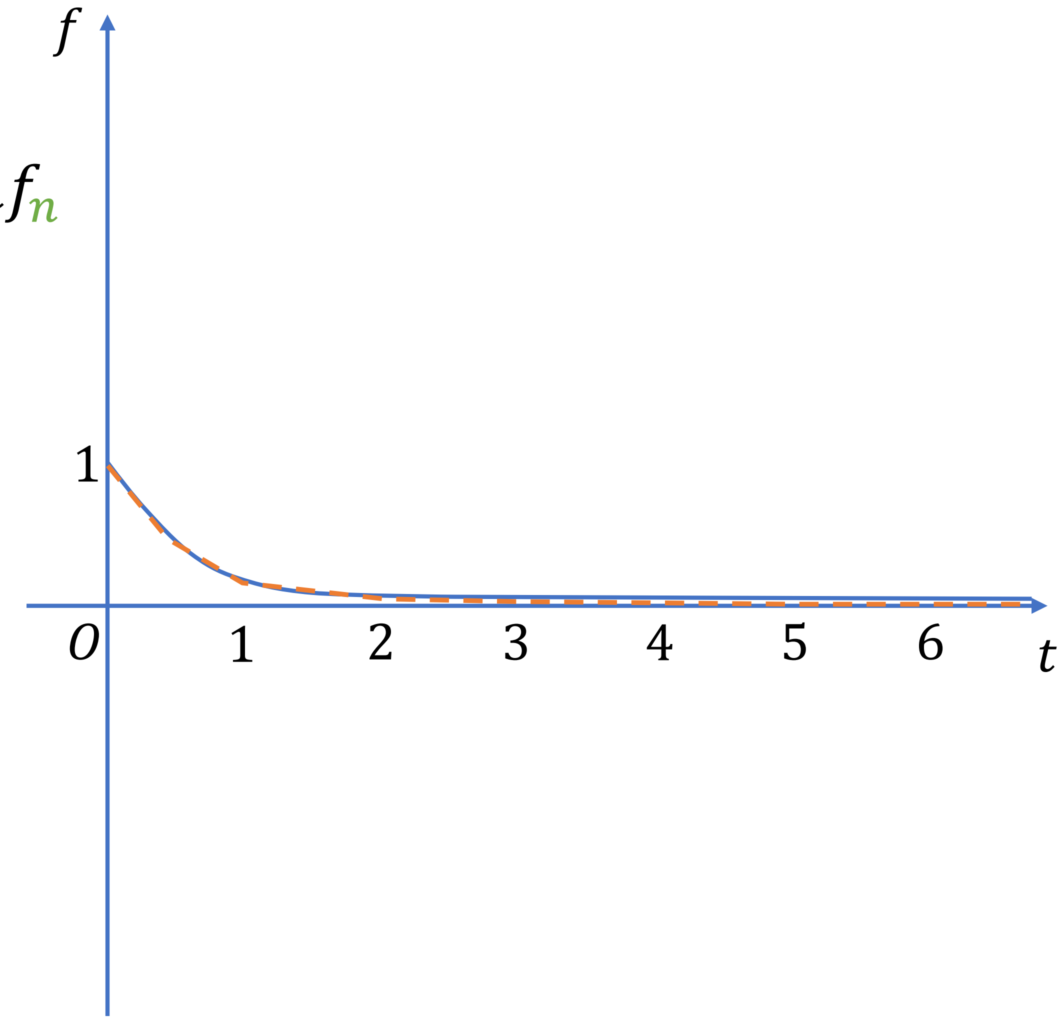
A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- Analytical solution:
  - $f(t) = e^{-\lambda t}$
- Explicit Euler:
  - $f_0 = 1$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_n) = f_n - h\lambda f_n$
  - ➔ $f_{n+1} = (1 - h\lambda)^{n+1}$
  - Converges iff $|1 - h\lambda| < 1$
    - ➔ $h < \frac{2}{\lambda}$



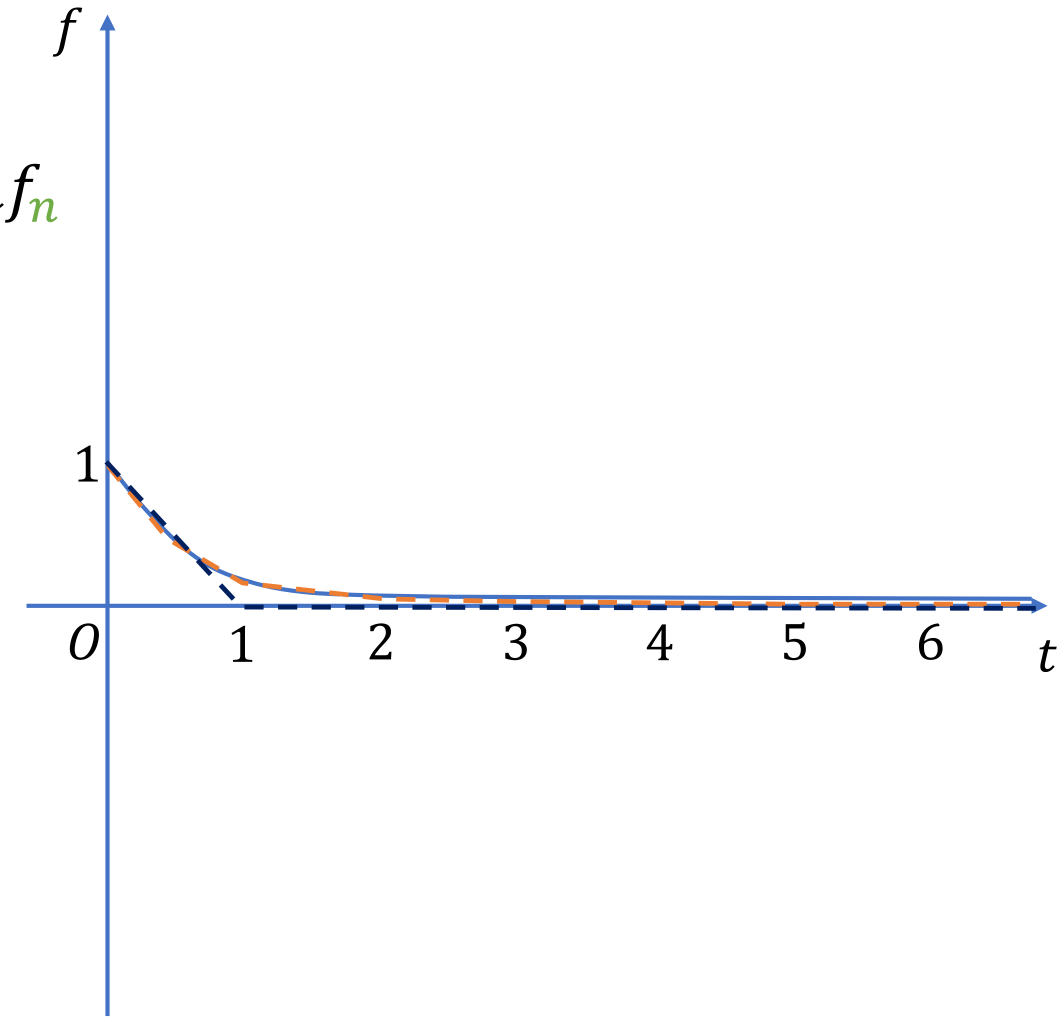
A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- When $\lambda = 1$ ➔ $\frac{2}{\lambda} = 2$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_n) = f_n - h\lambda f_n$
- Explicit Euler
  - $h = 0.5$



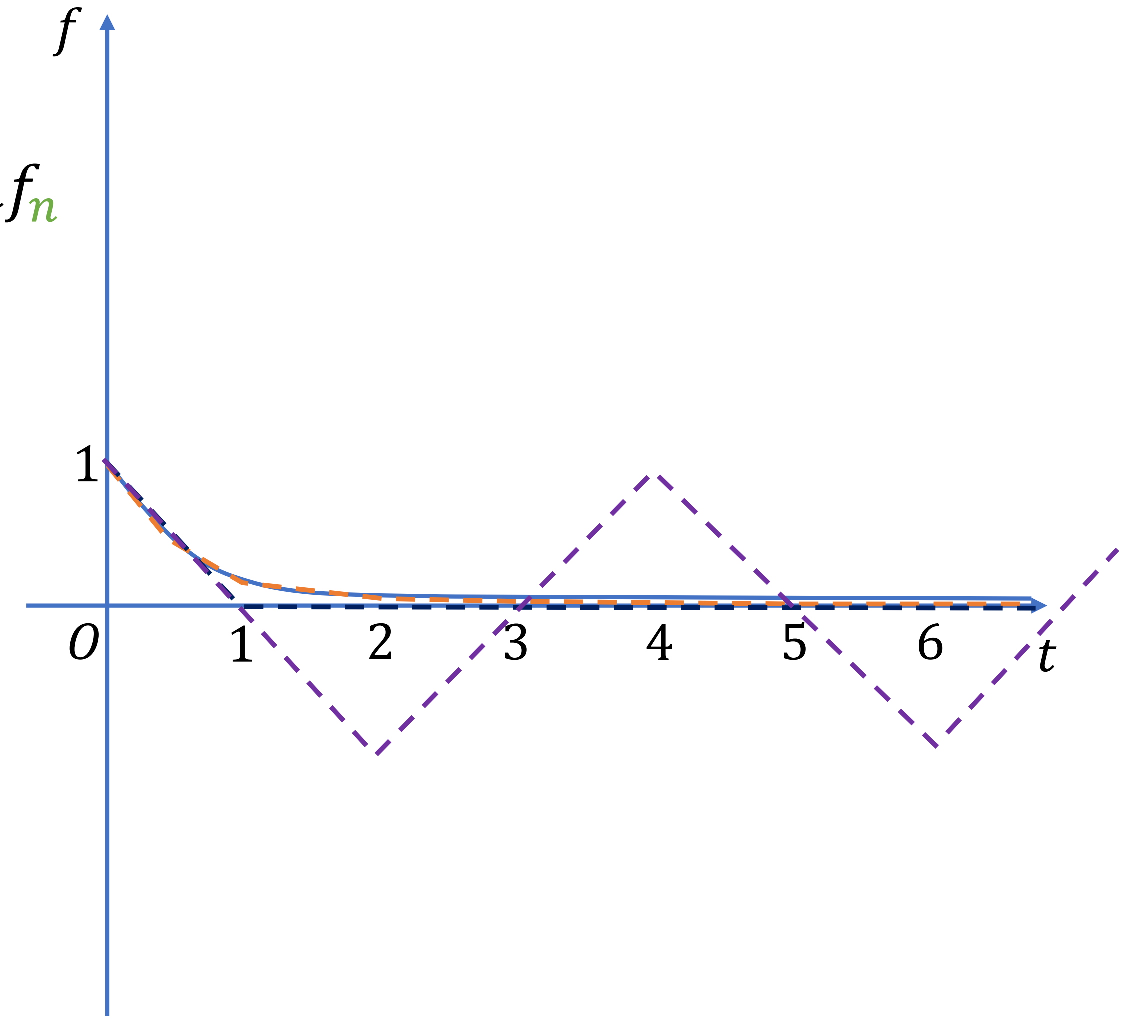
A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- When $\lambda = 1$ ➔ $\frac{2}{\lambda} = 2$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_n) = f_n - h\lambda f_n$
- Explicit Euler
  - $h = 0.5$
  - $h = 1$



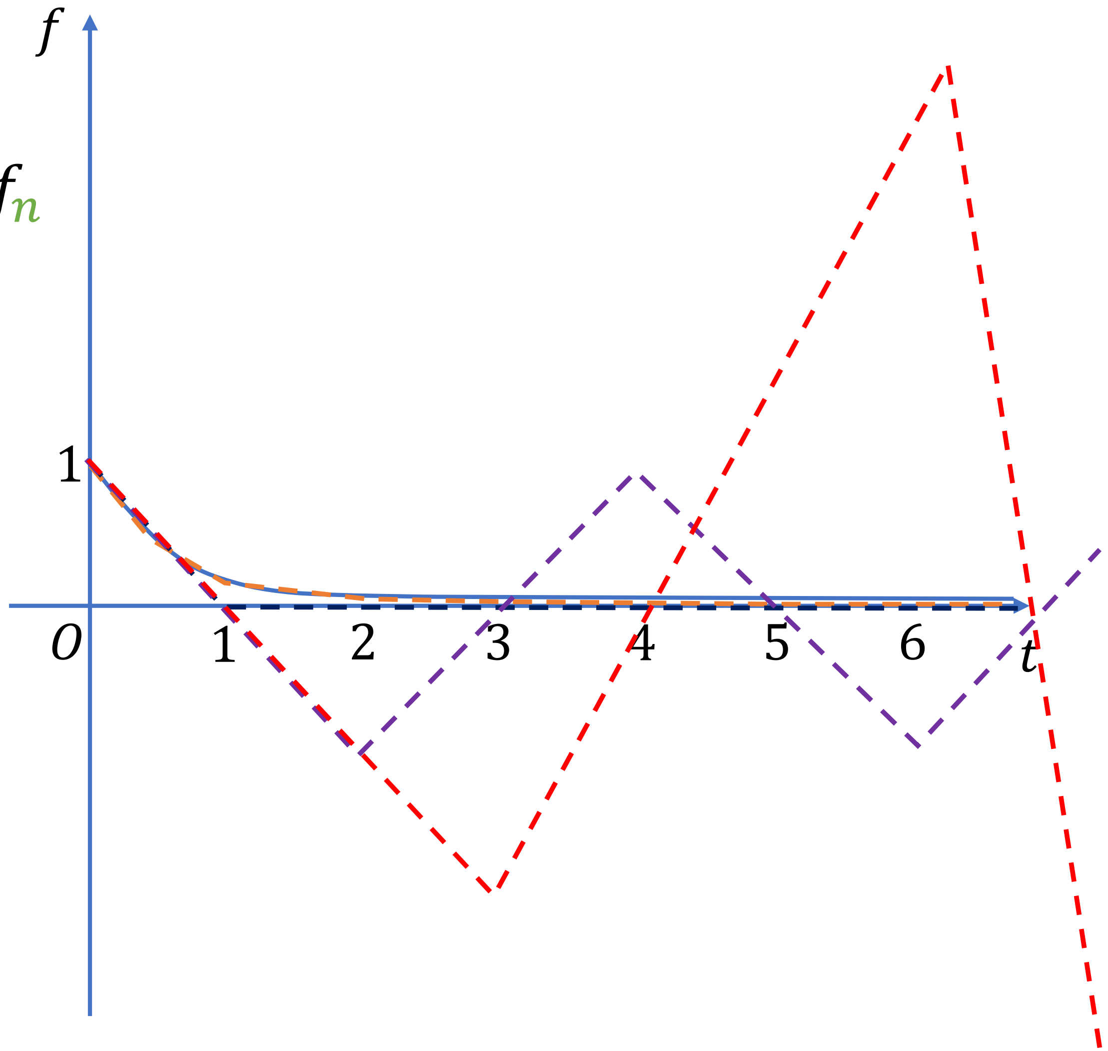
A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- When $\lambda = 1$ ➔ $\frac{2}{\lambda} = 2$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_n) = f_n - h\lambda f_n$
- Explicit Euler
  - $h = 0.5$
  - $h = 1$
  - $h = 2$



A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

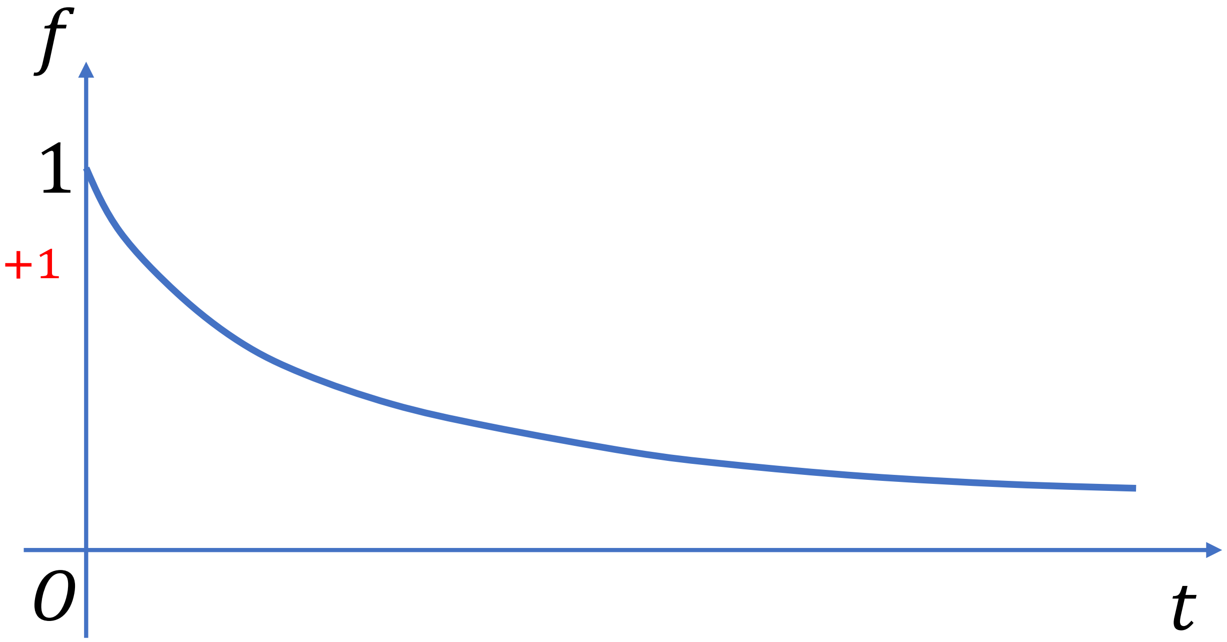
- When $\lambda = 1$ ➔ $\frac{2}{\lambda} = 2$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_n) = f_n - h\lambda f_n$
- Explicit Euler
  - $h = 0.5$
  - $h = 1$
  - $h = 2$
  - $h = 3$



A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

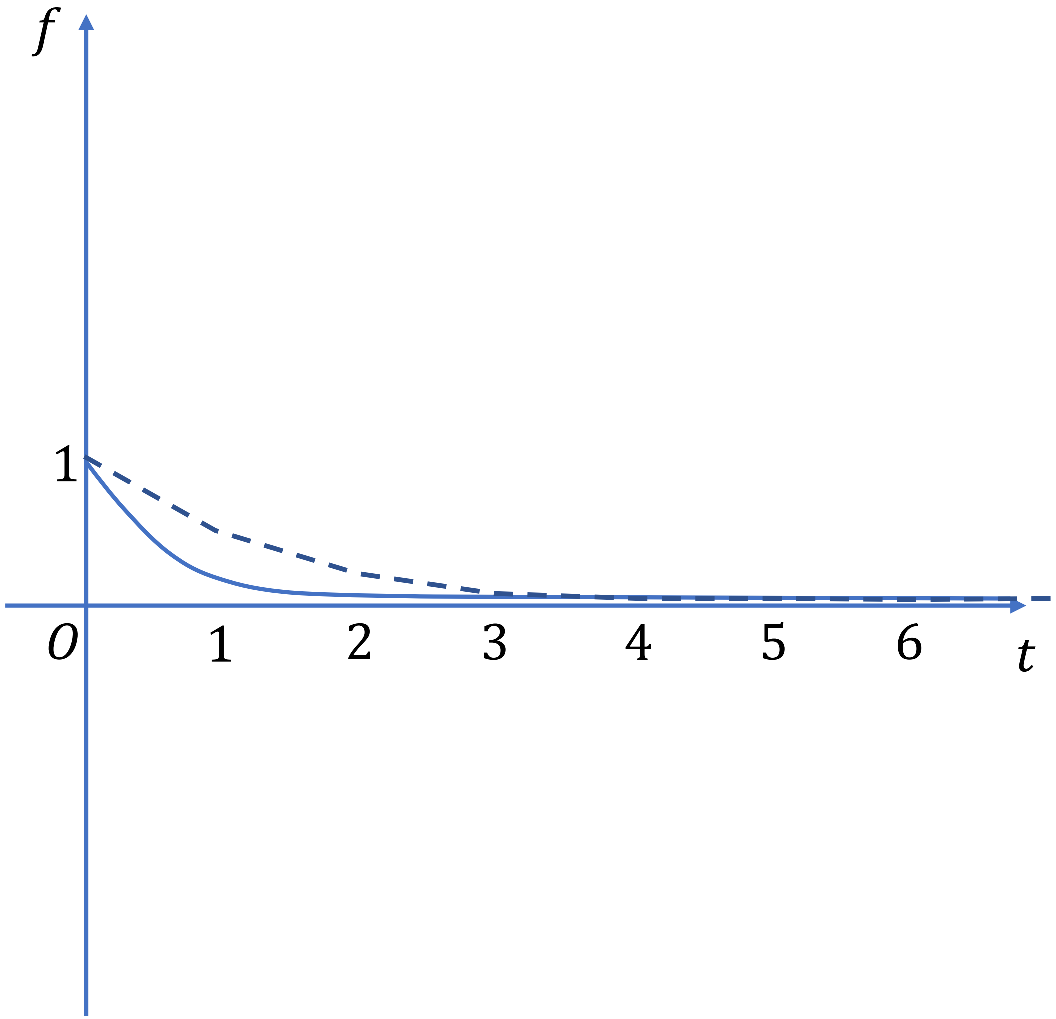
- Analytical solution:
  - $f(t) = e^{-\lambda t}$

- Implicit Euler:
  - $f_0 = 1$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_{n+1}) = f_n - h\lambda f_{n+1}$
  - ➔ $f_{n+1} = \left(\frac{1}{1+h\lambda}\right)^{n+1}$
  - Converges iff $\left|\frac{1}{1+h\lambda}\right| < 1$
    - ➔ converges for $\forall\ h > 0$



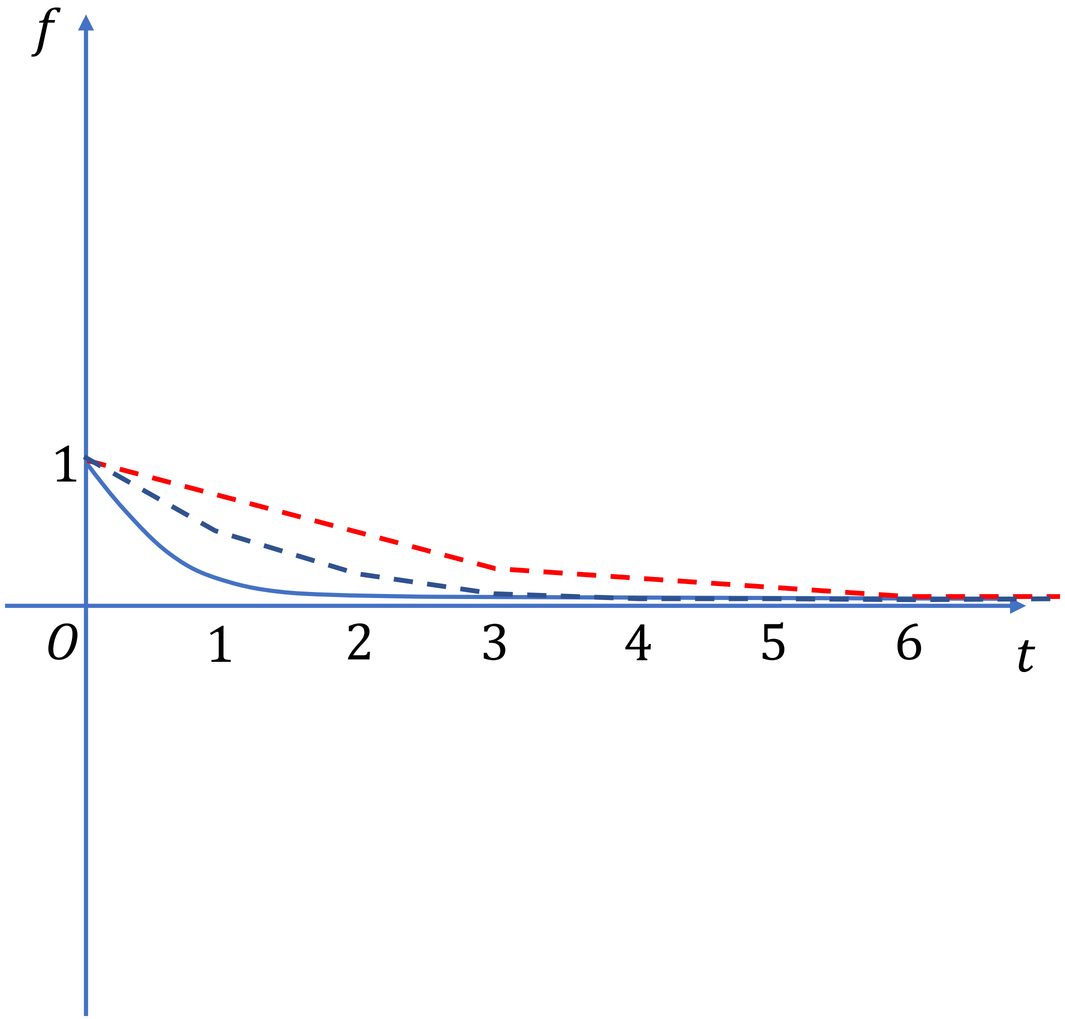
A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- When $\lambda = 1$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_{n+1}) = f_n - h\lambda f_{n+1}$
- Implicit Euler
  - $h = 1$



A toy example: $\frac{df}{dt}(t) = -\lambda f(t), f(0) = 1, \lambda > 0$

- When $\lambda = 1$
  - $f_{n+1} = f_n + h\frac{df}{dt}(t_{n+1}) = f_n - h\lambda f_{n+1}$
- Implicit Euler
  - $h = 1$
  - $h = 3$



Recap: the diffusion problem [Code]

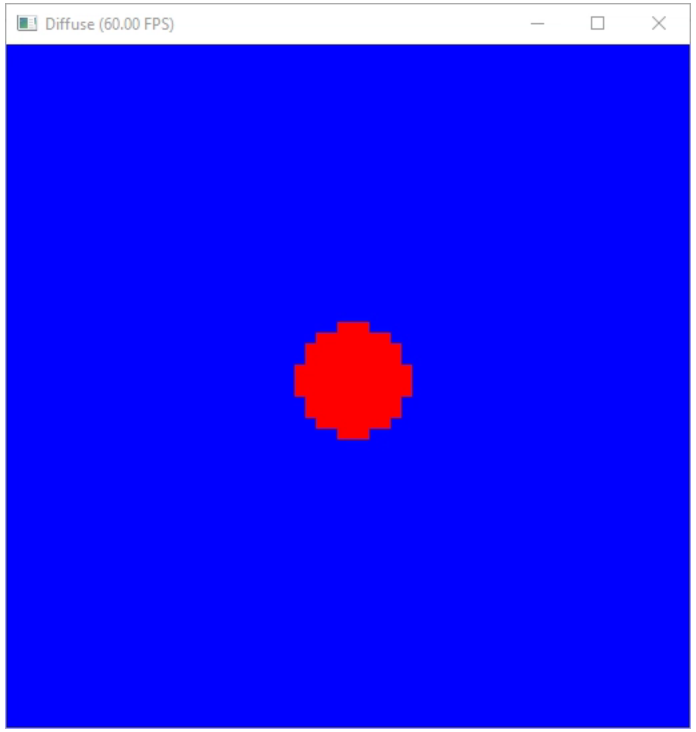
- Continuous form:
  - $\frac{\partial T}{\partial t} = \kappa \nabla^2 T$
- Explicit integration:
  - $\frac{T_{n+1} - T_n}{\Delta t} = \frac{\kappa}{\Delta x^2} \mathbf{D} T_{n} \rightarrow T_{n+1} = \left(\mathbf{I} + \frac{\Delta t * \kappa}{\Delta x^2} \mathbf{D}\right) T_n$
- Implicit integration:
  - $\frac{T_{n+1} - T_n}{\Delta t} = \frac{\kappa}{\Delta x^2} \mathbf{D} T_{n+1} \rightarrow T_{n+1} = \left(\mathbf{I} - \frac{\Delta t * \kappa}{\Delta x^2} \mathbf{D}\right)^{-1} T_n$

Recap: the diffusion problem

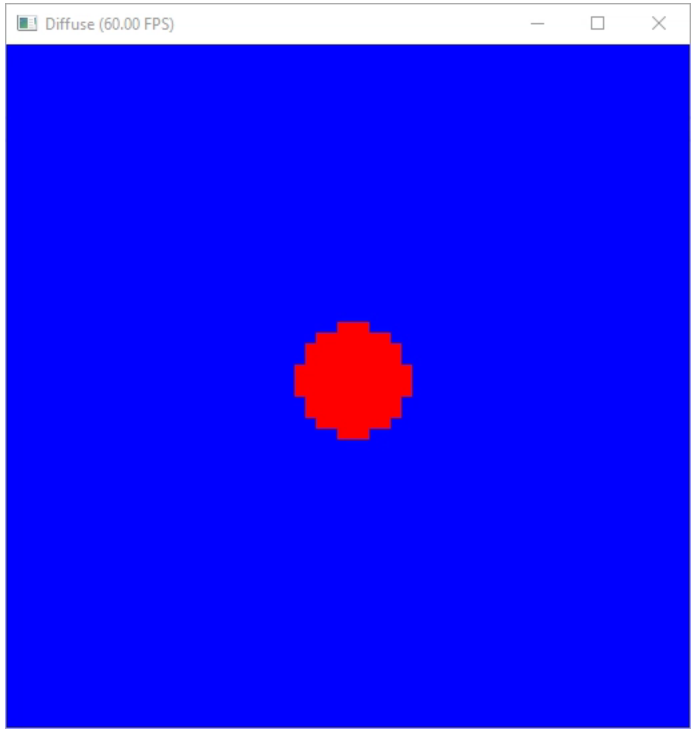
- Continuous form:
  - $\frac{\partial T}{\partial t} = \kappa \nabla^2 T$
- Explicit integration:
  - $\frac{T_{n+1} - T_n}{\Delta t} = \frac{\kappa}{\Delta x^2} \mathbf{D} T_{n} \Rightarrow T_{n+1} = \left(\mathbf{I} + \frac{\Delta t * \kappa}{\Delta x^2} \mathbf{D}\right) T_n$
- Implicit integration:
  - $\frac{T_{n+1} - T_n}{\Delta t} = \frac{\kappa}{\Delta x^2} \mathbf{D} T_{n+1} \Rightarrow T_{n+1} = \left(\mathbf{I} - \frac{\Delta t * \kappa}{\Delta x^2} \mathbf{D}\right)^{-1} T_n$

$\kappa = 2000$

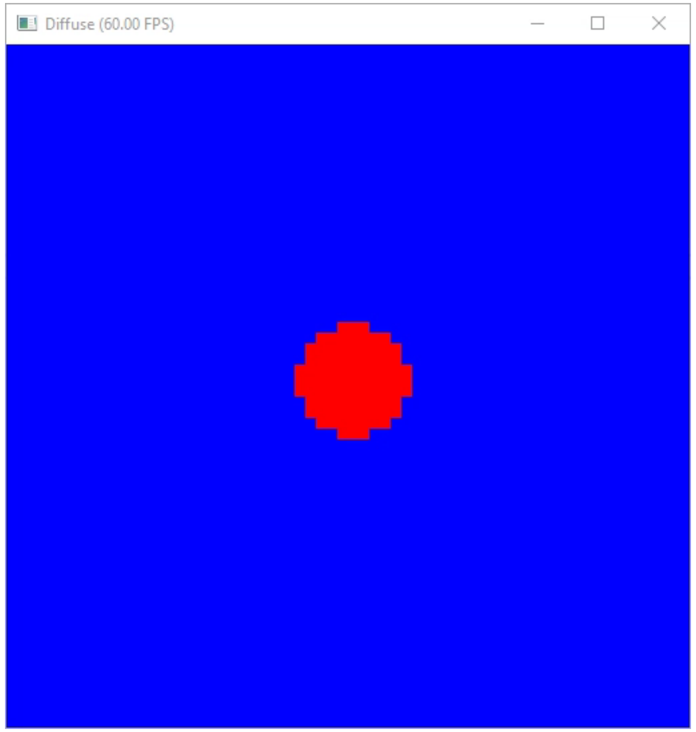
Recap: the diffusion problem ($\kappa = 2000$)



Explicit
$h_{disp} = 1ms$
$h_{sim} = 0.1ms$



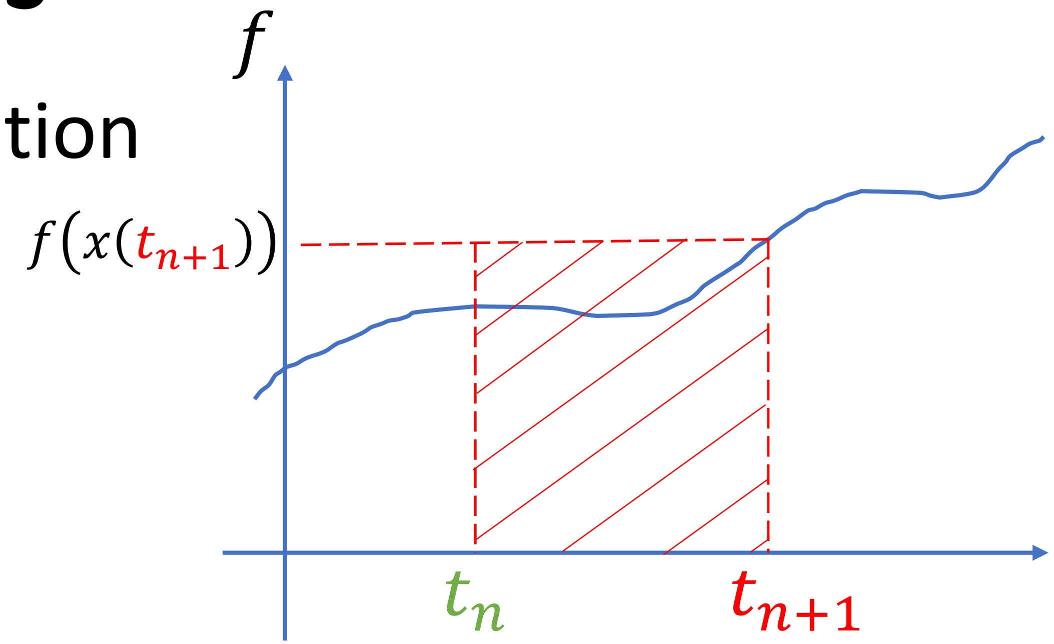
Explicit
$h_{disp} = 1ms$
$h_{sim} = 1ms$
(explodes)



Implicit
$h_{disp} = 1ms$
$h_{sim} = 1ms$

That's why we need implicit integrations – for **larger time-steps**

- Implicit (backward) Euler integration
  - $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
  - $x_{n+1} = x_n + hv_{n+1}$



Note: Implicit Euler is often **expensive** due to the nonlinear optimization, it **damps the Hamiltonian** from the oscillating components, it is often **stable for large time-steps** and is widely used in **performance-centric applications**. (game / MR / design / animation)

Numerical recipes for implicit integrations

The implicit Euler problem

- Implicit Euler:
  - $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
  - $x_{n+1} = x_n + hv_{n+1}$

Reformulating the implicit Euler problem

- Implicit Euler:
  - $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
  - $x_{n+1} = x_n + hv_{n+1}$

- Substituting the first equation into the second one gives us:
  - $x_{n+1} = x_n + hv_n + h^2M^{-1}f(x_{n+1})$

The [Baraff and Witkin, 1998] style solution [Link]

- Goal: solving $x_{n+1} = x_n + hv_n + h^2 M^{-1} f(x_{n+1})$

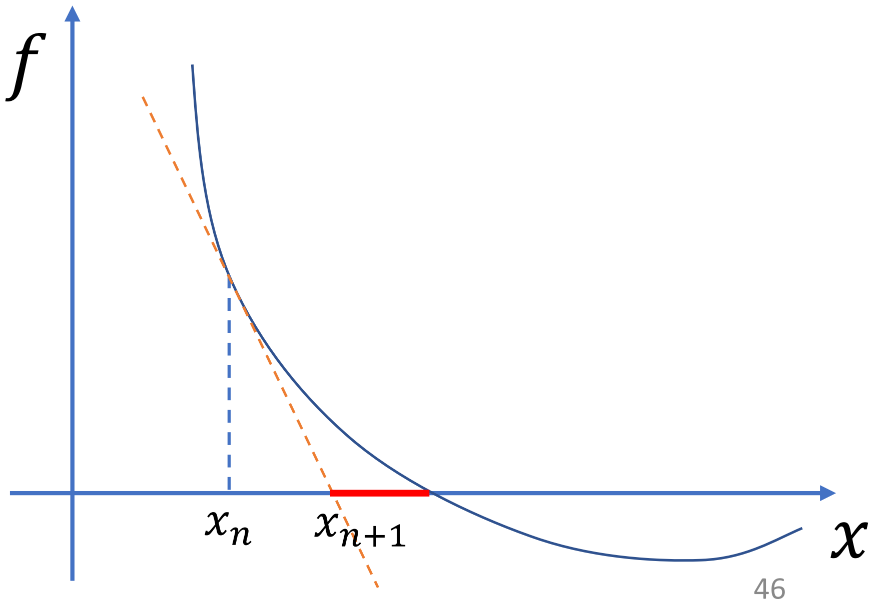
- Assumption: x_{n+1} is not too far away from x_n

- Algorithm:
  - Let $\delta x = x_{n+1} - x_n$, we have $f(x_{n+1}) \approx f(x_n) + \nabla_x f(x_n)\delta x$
  - Substitute this approximation into the goal:
    - $x_n + \delta x = x_n + hv_n + h^2 M^{-1}(f(x_n) + \nabla_x f(x_n)\delta x)$
    - ➔ $\left(M - h^2 \nabla_x f(x_n)\right)\delta x = hMv_n + h^2 f(x_n)$

The [Baraff and Witkin, 1998] style solution [Link]

- Algorithm:
  - Let $\delta x = x_{n+1} - x_n$, we have $f(x_{n+1}) \approx f(x_n) + \nabla_x f(x_n)\delta x$
  - Substitute this approximation into the goal:
    - $x_n + \delta x = x_n + hv_n + h^2 M^{-1}(f(x_n) + \nabla_x f(x_n)\delta x)$
    - ➔ $\left(M - h^2 \nabla_x f(x_n)\right)\delta x = hMv_n + h^2 f(x_n)$
    - $x_{n+1} = x_n + \delta x, v_{n+1} = \delta x / h$

- The Baraff-Witkin style solution is:
  - One iteration of Newton's method
  - Sometimes referred as semi-implicit Euler



Reformulating the implicit Euler problem

- Implicit Euler:
  - $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
  - $x_{n+1} = x_n + hv_{n+1}$

- Substituting the first equation into the second one gives us:
  - $x_{n+1} = x_n + hv_n + h^2M^{-1}f(x_{n+1})$

- Integrating the nonlinear root finding problem over x gives us:
  - $x_{n+1} = argmin_x\left(\frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2E(x)\right)$, given $f(x) = \nabla_x E(x)$

Note (Matrix Norm): $\|x\|_A = \sqrt{x^TAx}$; Note (Vector Derivative): $\nabla_x(x^TAx) = (A + A^T)x$

Minimization problem v.s. nonlinear root-finding problem

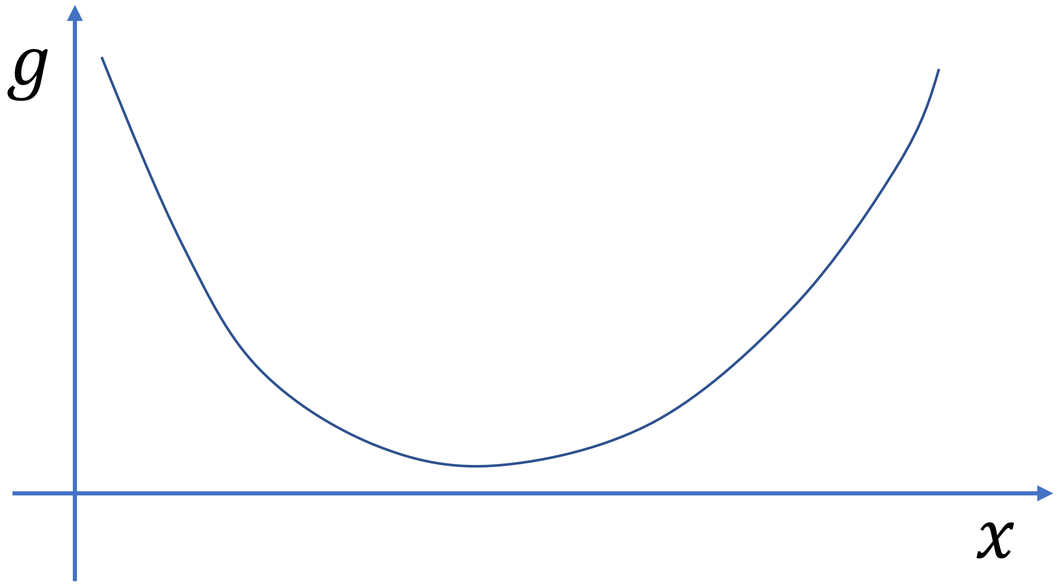
- Let: $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$
- We have: $\nabla_x g(x_{n+1}) = M\big(x_{n+1} - (x_n + hv_n)\big) - h^2 f(x_{n+1})$
- For nonsingular M: we have:
  - $\nabla_x g(x_{n+1}) = 0 \Leftrightarrow x_{n+1} = (x_n + hv_n) + h^2 M^{-1} f(x_{n+1})$

Note (Matrix Norm): $\|x\|_A = \sqrt{x^T A x}$; Note (Vector Derivative): $\nabla_x(x^T A x) = (A + A^T)x$

Convex minimization problem: $\min g(x)$

- The general descent method:

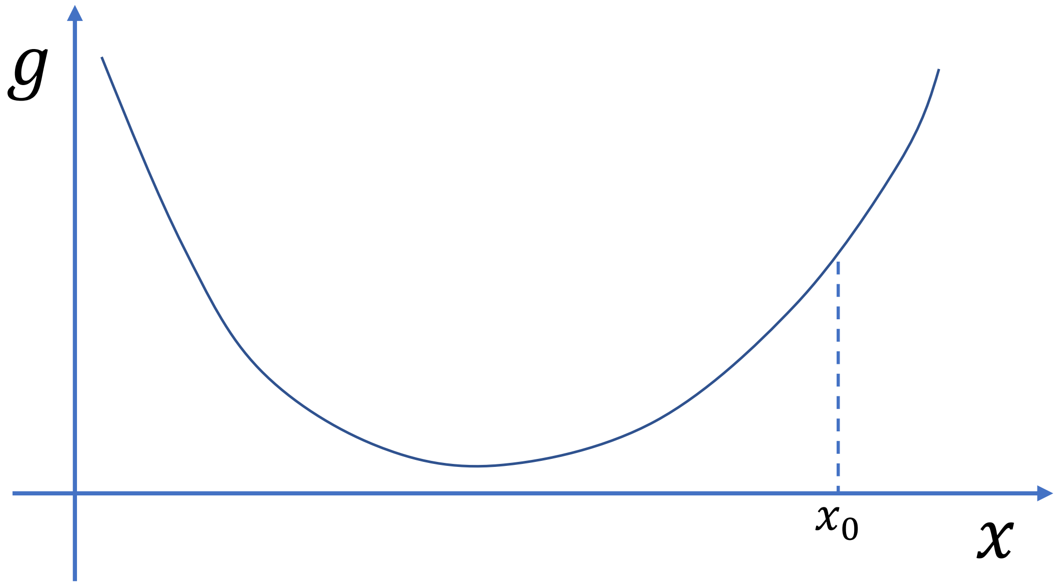
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\min g(x)$

- The general descent method:

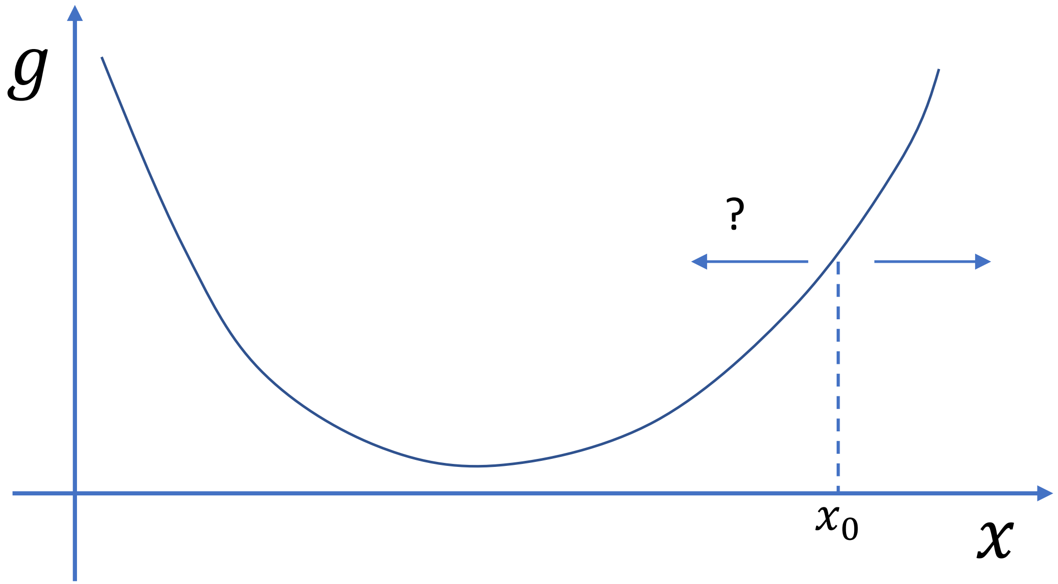
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\mathbf{min}\ g(x)$

- The general descent method:

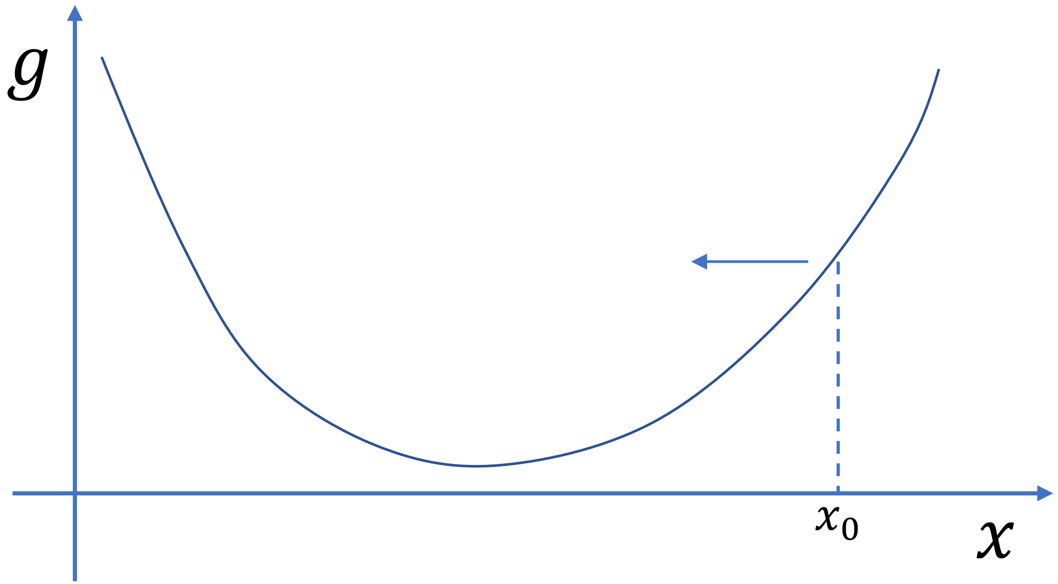
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\mathbf{min}\ g(x)$

- The general descent method:

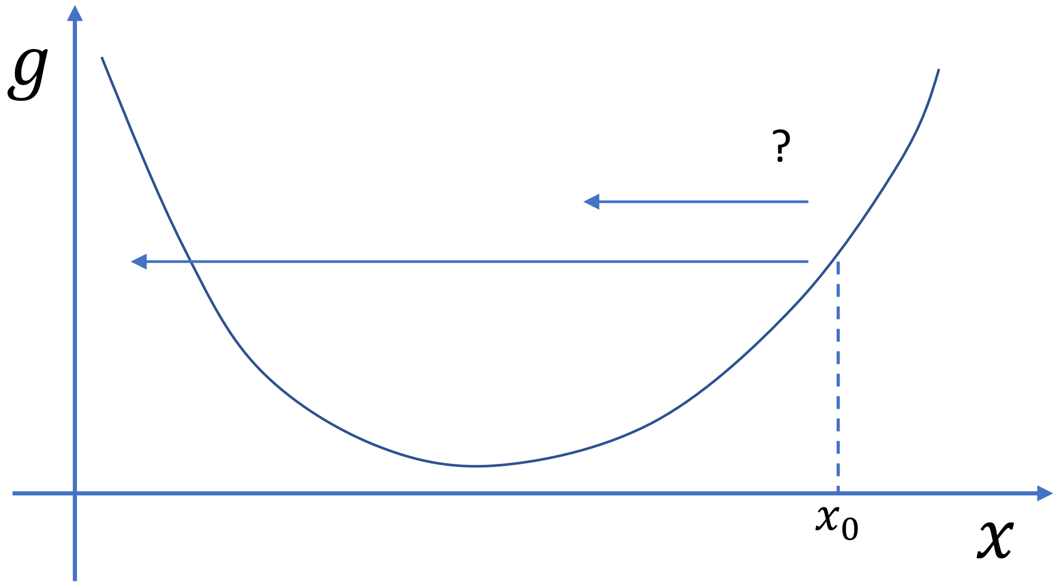
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\min g(x)$

- The general descent method:

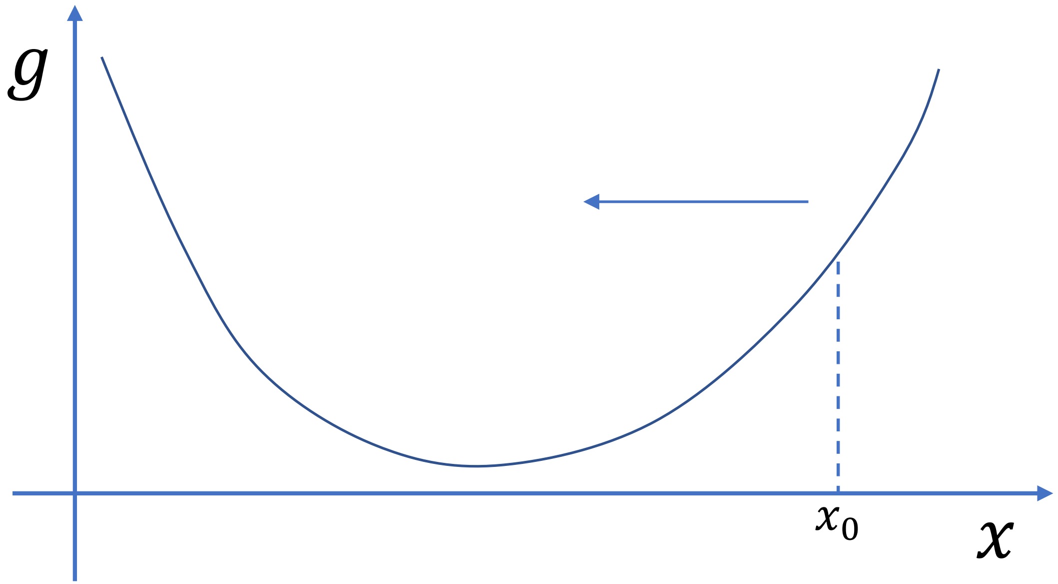
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\min g(x)$

- The general descent method:

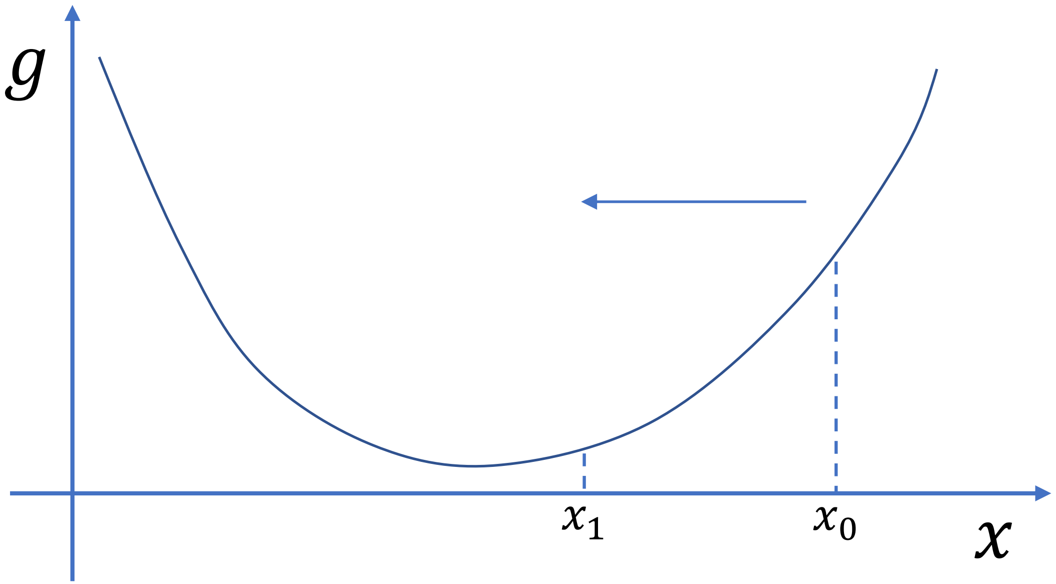
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\min g(x)$

- The general descent method:

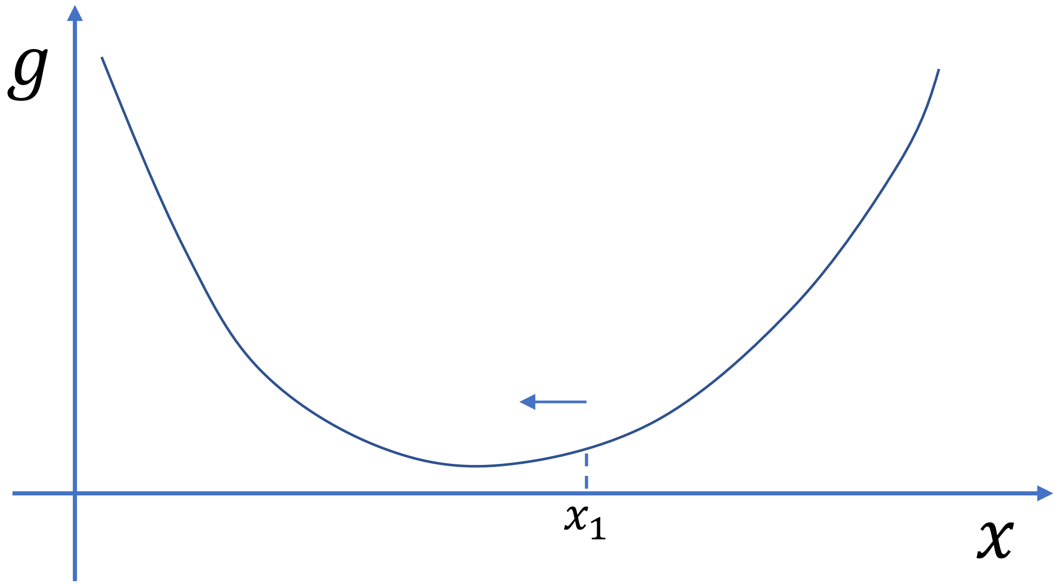
```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\min g(x)$

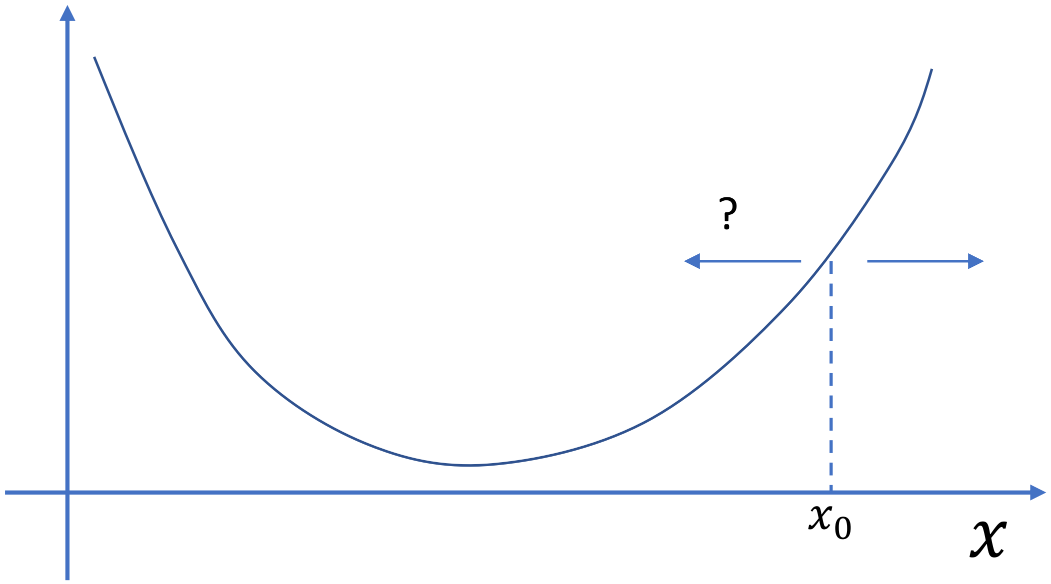
- The general descent method:

```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```



Convex minimization problem: $\min g(x)$

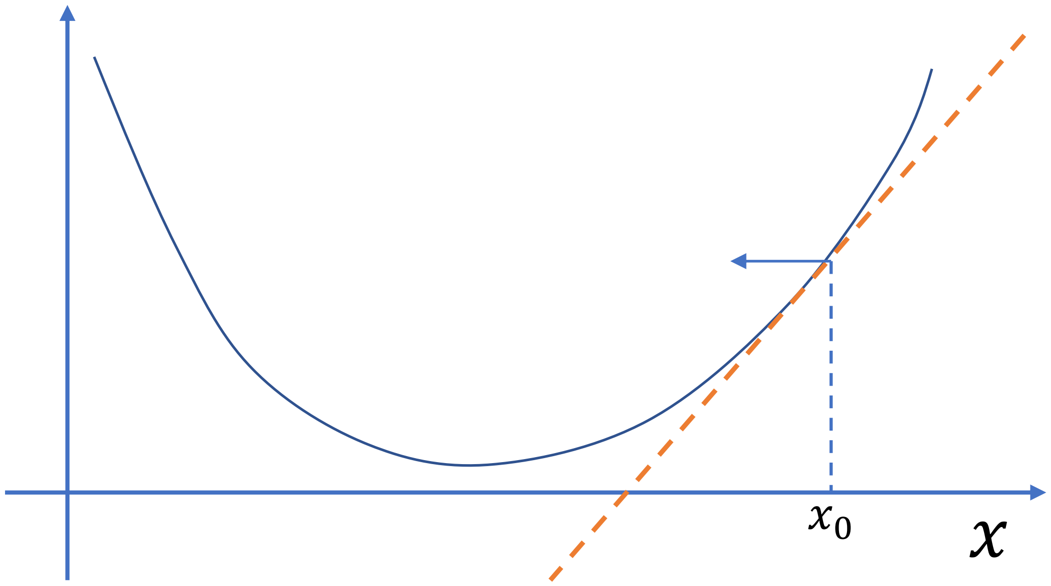
- Determine a descent direction: dx



Convex minimization problem: $\min g(x)$

- Determine a descent direction: dx

- Option I: Gradient Descent
  - $dx = -\nabla_x g(x)$

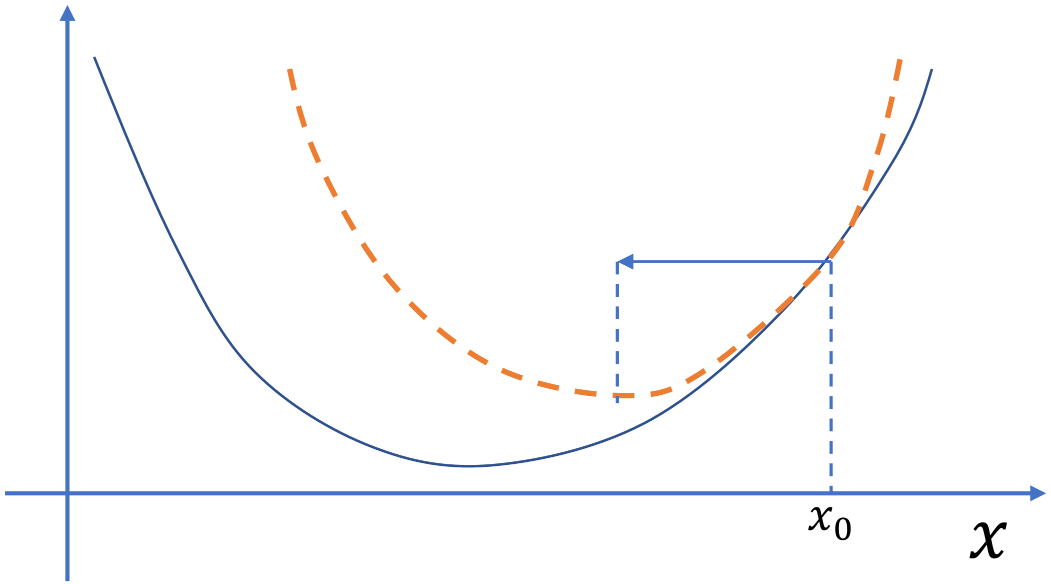


Convex minimization problem: $\min g(x)$

- Determine a descent direction: dx

- Option I: Gradient Descent
  - $dx = -\nabla_x g(x)$

- Option II: Newton's Method
  - $dx = -\left(\nabla_x^2 g(x)\right)^{-1} \nabla_x g(x)$



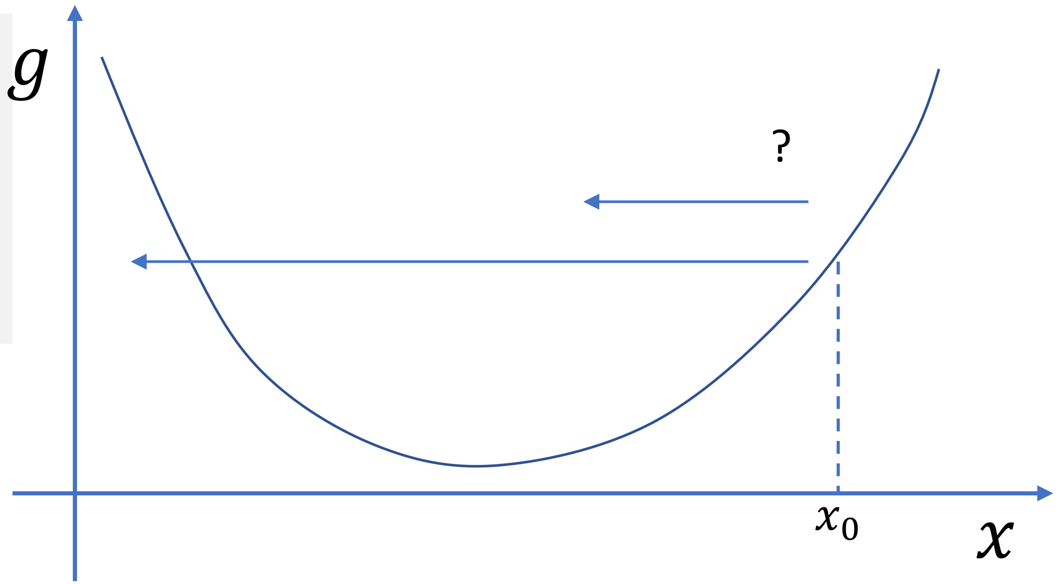
Convex minimization problem: $\min g(x)$

- Line search: choose a step size t > 0 given a dx

- Backtracking:

```
def line_search():
    t = t_0
    while g(x) + alpha*t*grad_g(x).dot(dx) < g(x+t*dx):
        t = t * beta

    return t
```



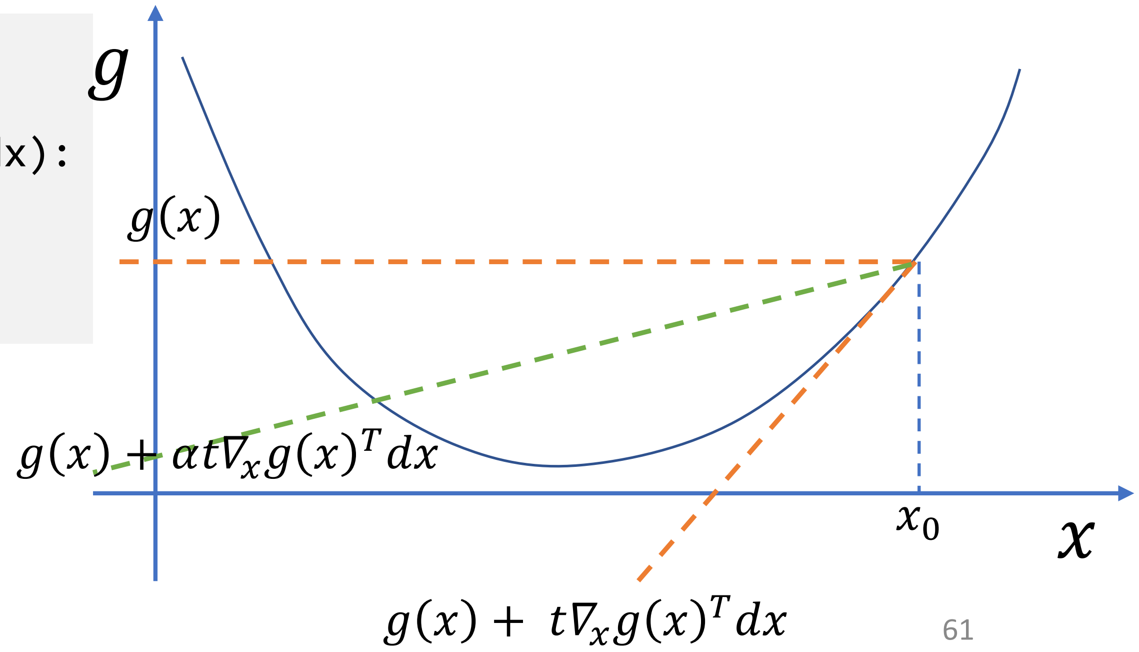
Convex minimization problem: $\min g(x)$

- Line search: choose a step size t > 0 given a dx

- Backtracking:

```
def line_search():
    t = t_0
    while g(x) + alpha*t*grad_g(x).dot(dx) < g(x+t*dx):
        t = t * beta

    return t
```



Convex minimization problem: $\min g(x)$

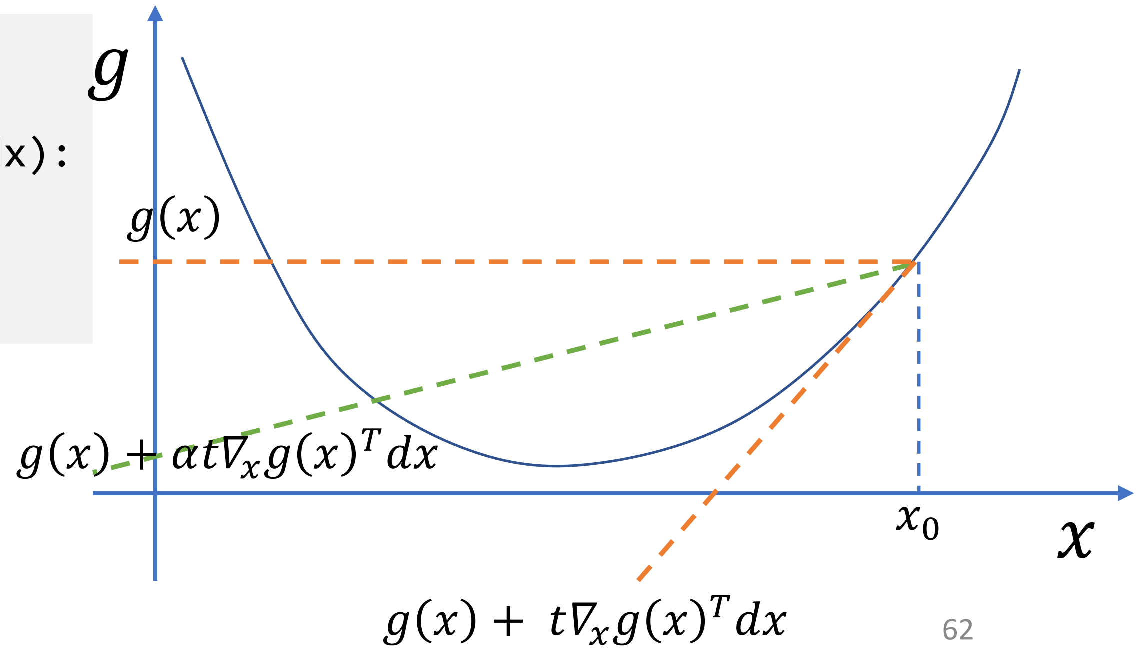
- Line search: choose a step size t > 0 given a dx

- Backtracking:

```python
def line_search():
    t = t_0
    while g(x) + alpha*t*grad_g(x).dot(dx) < g(x+t*dx):
        t = t * beta

    return t
```

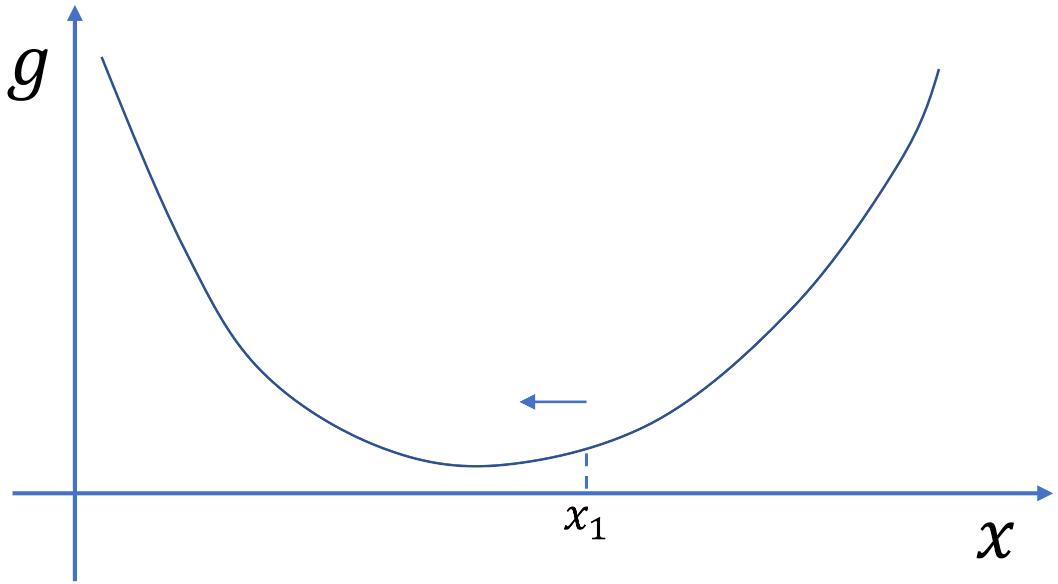
  - $\alpha \in (0,0.5), \beta \in (0,1)$
  - Common choice: $\alpha = 0.03, \beta = 0.5$



Convex minimization problem: $\min g(x)$

- The convex minimization problem:
  - Pick a descent direction:
    - Gradient descent
    - Newton's method
  - Pick a step size:
    - Back-tracking line-search

```
def minimize_g():
    x = x_0
    while grad_g(x).norm() > EPSILON:
        Determine a descent direction: dx
        Line search: choose a step size t > 0
        Update: x = x + t*dx
```

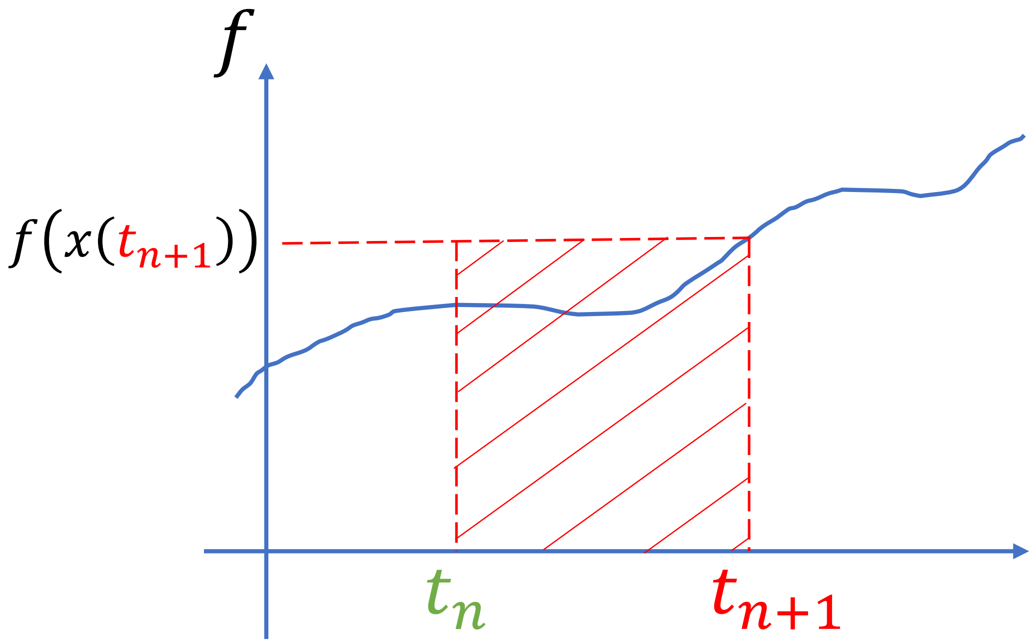


- Further Reading:
  - Convex Optimization [Link][Video]

Back to our problem

- $v_{n+1} = v_n + hM^{-1}f(x_{n+1})$
- $x_{n+1} = x_n + hv_{n+1}$

- $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2E(x)$
- $x_{n+1} = argmin_x g(x)$



Minimizing $g(x)$

- $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$

- Step 1: Initial guess: $x = x_n$ or $x = x_n + hv_n$

Minimizing $g(x)$

- $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$

- Step 1: Initial guess: $x = x_n$ or $x = x_n + hv_n$
- Step 2: While not converged:
  - Compute descent direction: $dx = -\nabla_x g(x)$ or $dx = -\left(\nabla_x^2 g(x)\right)^{-1} \nabla_x g(x)$
  - Line search to determine the step size: t
  - Update: $x = x + t * dx$

Minimizing $g(x)$

- $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$

- Step 1: Initial guess: $x = x_n$ or $x = x_n + hv_n$
- Step 2: While not converged:
    - Compute descent direction: $dx = -\nabla_x g(x)$ or $dx = -\left(\nabla_x^2 g(x)\right)^{-1} \nabla_x g(x)$
    - Line search to determine the step size: t
    - Update: $x = x + t * dx$

Minimizing $g(x)$

- $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$

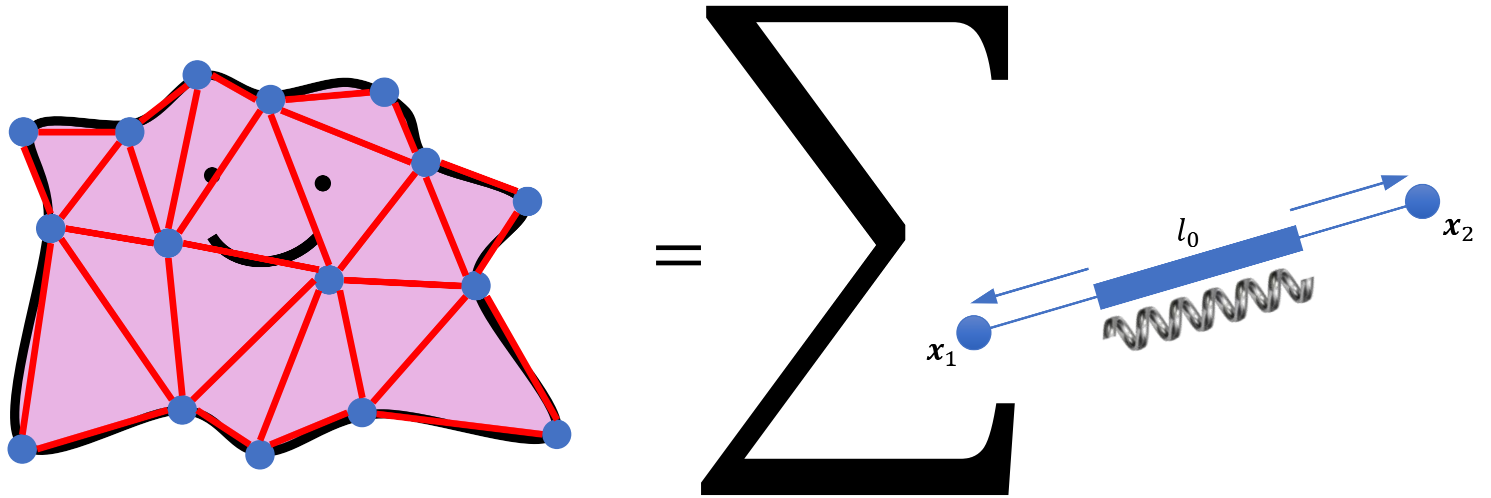
- The gradient:
  - $\nabla_x g(x) = M\big(x - (x_n + hv_n)\big) + h^2 \nabla_x E(x)$

- The Hessian:
  - $\nabla_x^2 g(x) = M + h^2 \nabla_x^2 E(x)$

Model-dependent

Example: (mass-spring system)

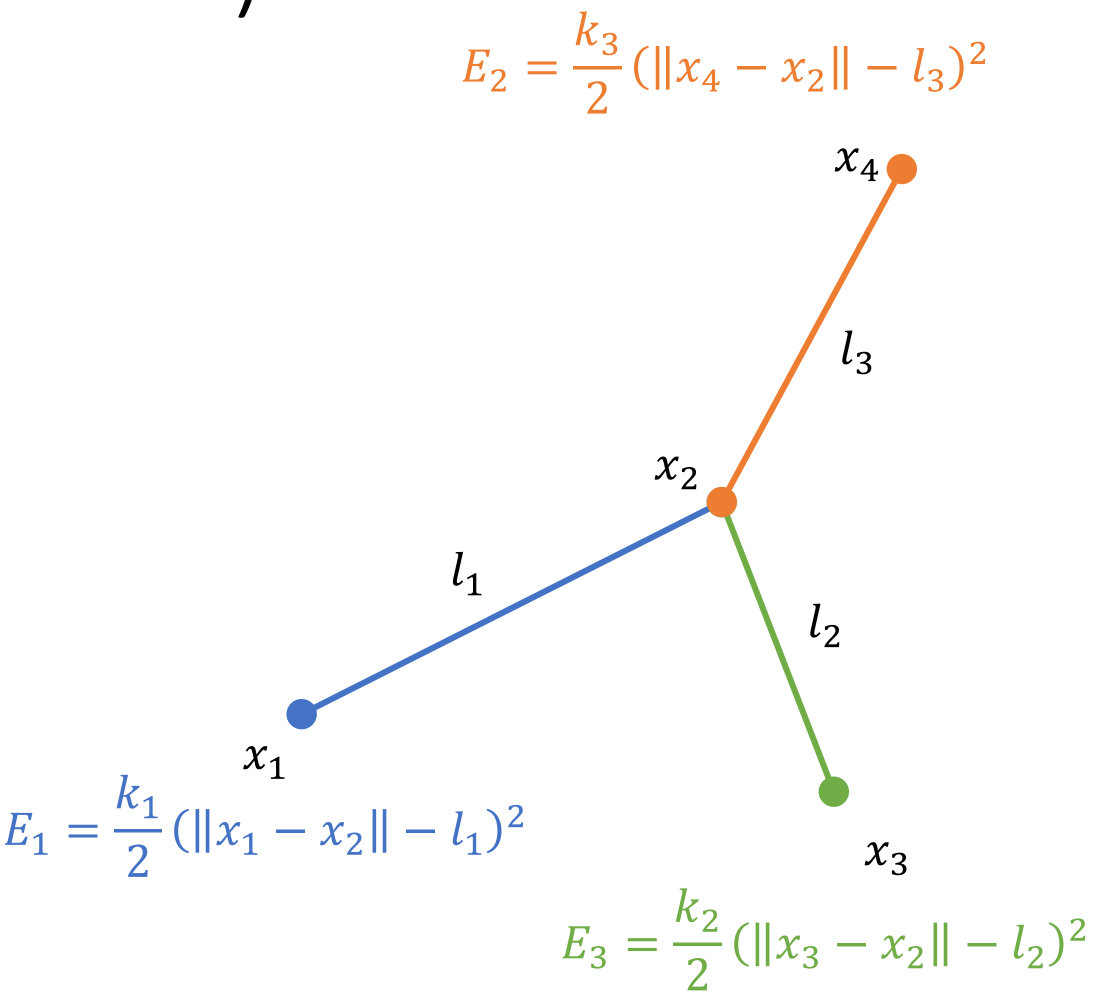
- Total energy:
  - $E(x) = \sum_{j=1}^{m} \frac{k_j}{2} \left(\|x_{j1} - x_{j2}\| - l_{j0} \right)^2$



Example: (mass-spring system)

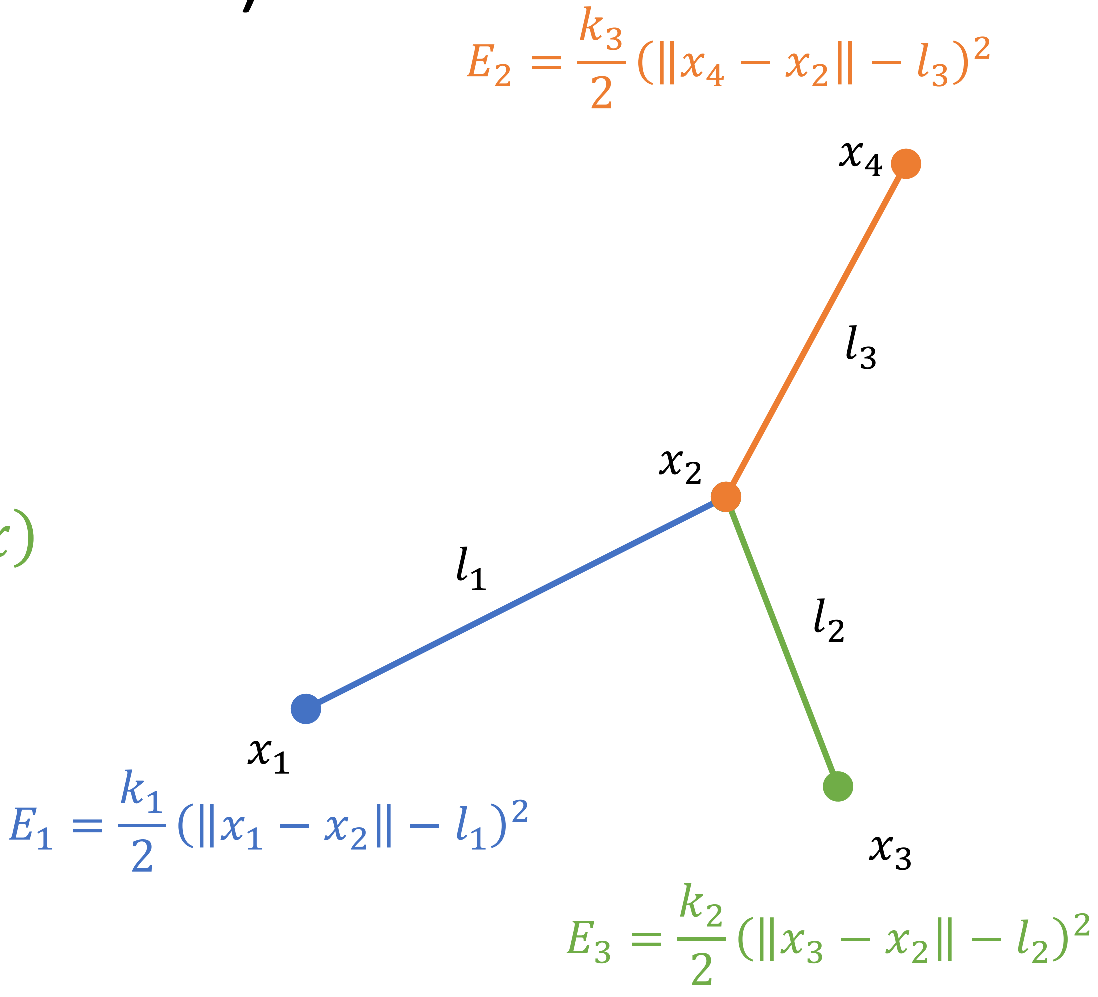
- Total energy:
  - $E(x) = \sum_{j=1}^{m} \frac{k_j}{2}\left(\|x_{j1} - x_{j2}\| - l_{j0}\right)^2$

- In the system shown right:
  - $E(x) = E_1(x) + E_2(x) + E_3(x)$



Example: (mass-spring system)

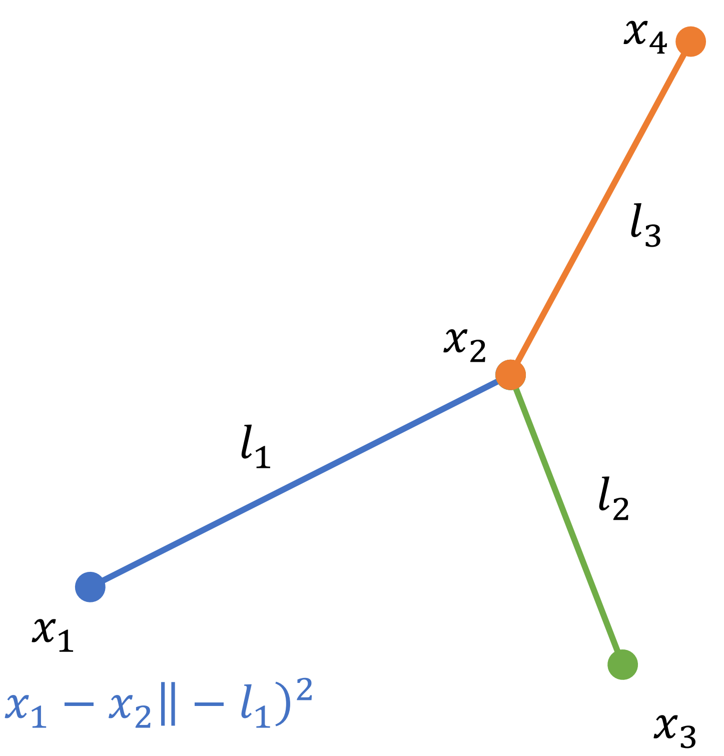
- Energy:
  - $E(x) = E_1(x) + E_2(x) + E_3(x)$
- Gradient:
  - $\nabla_x E(x) = \nabla_x E_1(x) + \nabla_x E_2(x) + \nabla_x E_3(x)$



Example: (mass-spring system)

- Energy:
  - $E(x) = E_1(x) + E_2(x) + E_3(x)$
- Gradient:
  - $\nabla_{x_1} E(x) = \nabla_{x_1} E_1(x) + \nabla_{x_1} E_2(x) + \nabla_{x_1} E_3(x)$
  - $\nabla_{x_1} E_1(x) = k_1(\|x_1 - x_2\| - l_1)\frac{x_1 - x_2}{\|x_1 - x_2\|}$
  - $\nabla_{x_2} E_1(x) = -\nabla_{x_1} E_1(x)$

$E_2 = \frac{k_3}{2}(\|x_4 - x_2\| - l_3)^2$



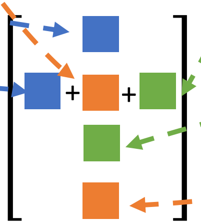
$E_1 = \frac{k_1}{2}(\|x_1 - x_2\| - l_1)^2$

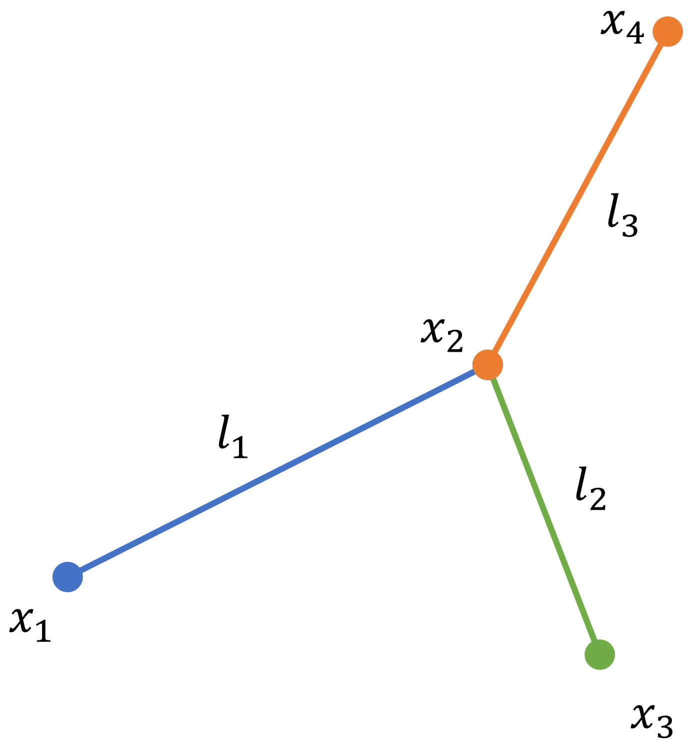
$E_3 = \frac{k_2}{2}(\|x_3 - x_2\| - l_2)^2$

Example: (mass-spring system)

- Element-wise gradient:
  - $\nabla_{x_1} E_1(x) = k_1(\|x_1 - x_2\| - l_1)\frac{x_1 - x_2}{\|x_1 - x_2\|}$
- Total gradient:
  - $\nabla_x E(x) = \nabla_x E_1(x) + \nabla_x E_2(x) + \nabla_x E_3(x)$

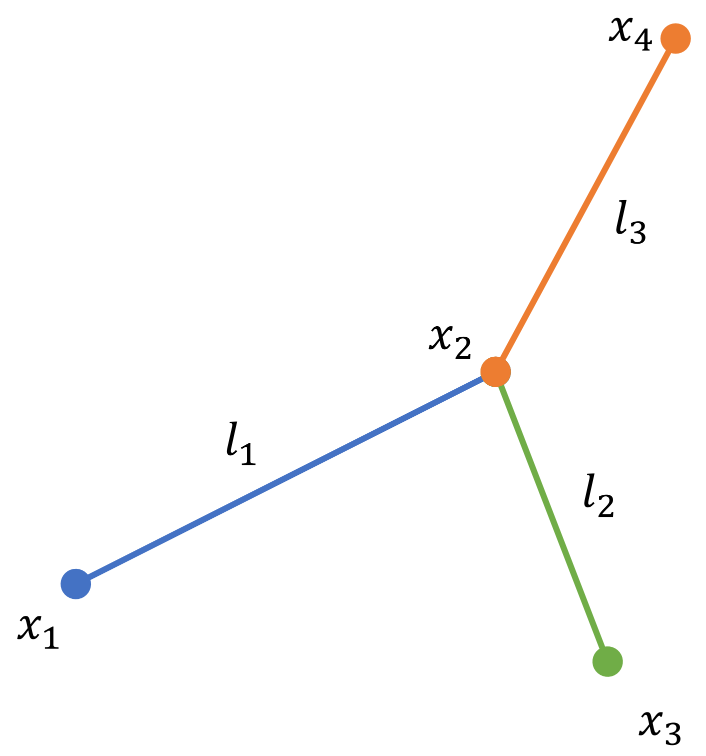
$$\begin{bmatrix} \nabla_{x_1} E_1(x) \\ -\nabla_{x_1} E_1(x) \end{bmatrix} \quad \begin{bmatrix} \nabla_{x_2} E_2(x) \\ -\nabla_{x_2} E_2(x) \end{bmatrix} \quad \begin{bmatrix} \nabla_{x_2} E_3(x) \\ -\nabla_{x_2} E_3(x) \end{bmatrix}$$

$\nabla_x E(x) =$ 



Example: (mass-spring system)

- Element-wise gradient:
  - $\nabla_{x_1} E_1(x) = k_1(\|x_1 - x_2\| - l_1)\frac{x_1-x_2}{\|x_1-x_2\|}$
- Element-wise Hessian:
  - $\nabla^2_{x_1x_1} E_1(x) = \overbrace{k_1\left(I - \frac{l_1}{\|x_1-x_2\|}\left(I - \frac{(x_1-x_2)(x_1-x_2)^T}{\|x_1-x_2\|^2}\right)\right)}^{K_1}$
  - $\nabla^2_{x_1x_2} E_1(x) = -K_1$
  - $\nabla^2_{x_2x_1} E_1(x) = -K_1$
  - $\nabla^2_{x_2x_2} E_1(x) = K_1$



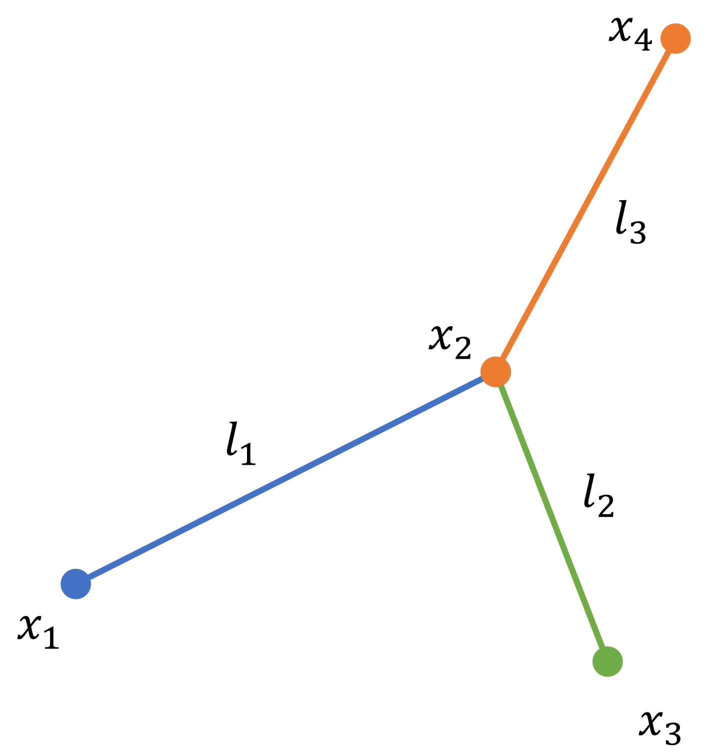
Example: (mass-spring system)

- Element-wise Hessian :
  - $\nabla^2_{x_1 x_1} E_1(x) = k_1 \left(I - \frac{l_1}{\|x_1 - x_2\|} \left(I - \frac{(x_1 - x_2)(x_1 - x_2)^T}{\|x_1 - x_2\|^2} \right) \right)$

- Total Hessian:
  - $\nabla^2_x E(x) = \nabla^2_x E_1(x) + \nabla_x E_2(x) + \nabla_x E_3(x)$

$$\begin{bmatrix} K_1 & -K_1 \\ -K_1 & K_1 \end{bmatrix} \quad \begin{bmatrix} K_2 & -K_2 \\ -K_2 & K_2 \end{bmatrix} \quad \begin{bmatrix} K_3 & -K_3 \\ -K_3 & K_3 \end{bmatrix}$$

$$\nabla^2_x E(x) = \begin{bmatrix} K_1 & -K_1 & & \\ -K_1 & K_1 + K_2 + K_3 & -K_3 & -K_2 \\ & -K_3 & K_3 & \\ & -K_2 & & K_2 \end{bmatrix}$$



Example: (linear FEM)

- Elastic energy:
  - $E_i(x) = w_i \Psi(F_i(x))$

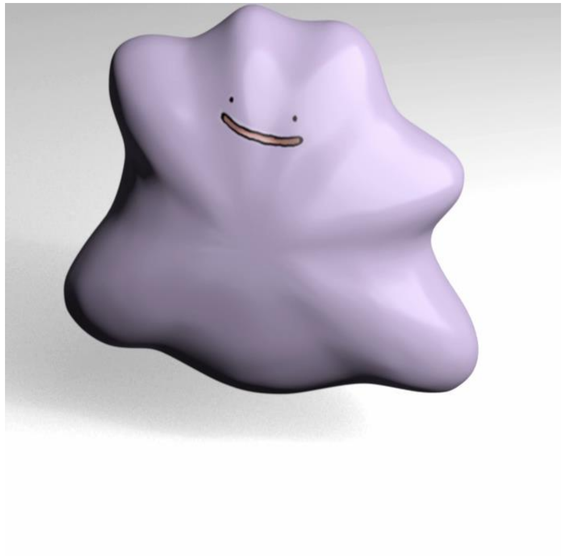
- Gradient:
  - $\frac{\partial E_i}{\partial x} = w_i \frac{\partial F_i}{\partial x} : P_i$

- Hessian:
  - $\frac{\partial^2 E_i}{\partial x \partial x} = w_i \frac{\partial F_i}{\partial x} : \frac{\partial P_i}{\partial F_i} : \frac{\partial F_i}{\partial x}^T$

A $(2n \times 1) \times (2 \times 2)$ tensor

A $(2 \times 2) \times (2 \times 2)$ tensor

A $(2 \times 2) \times (2n \times 1)$ tensor

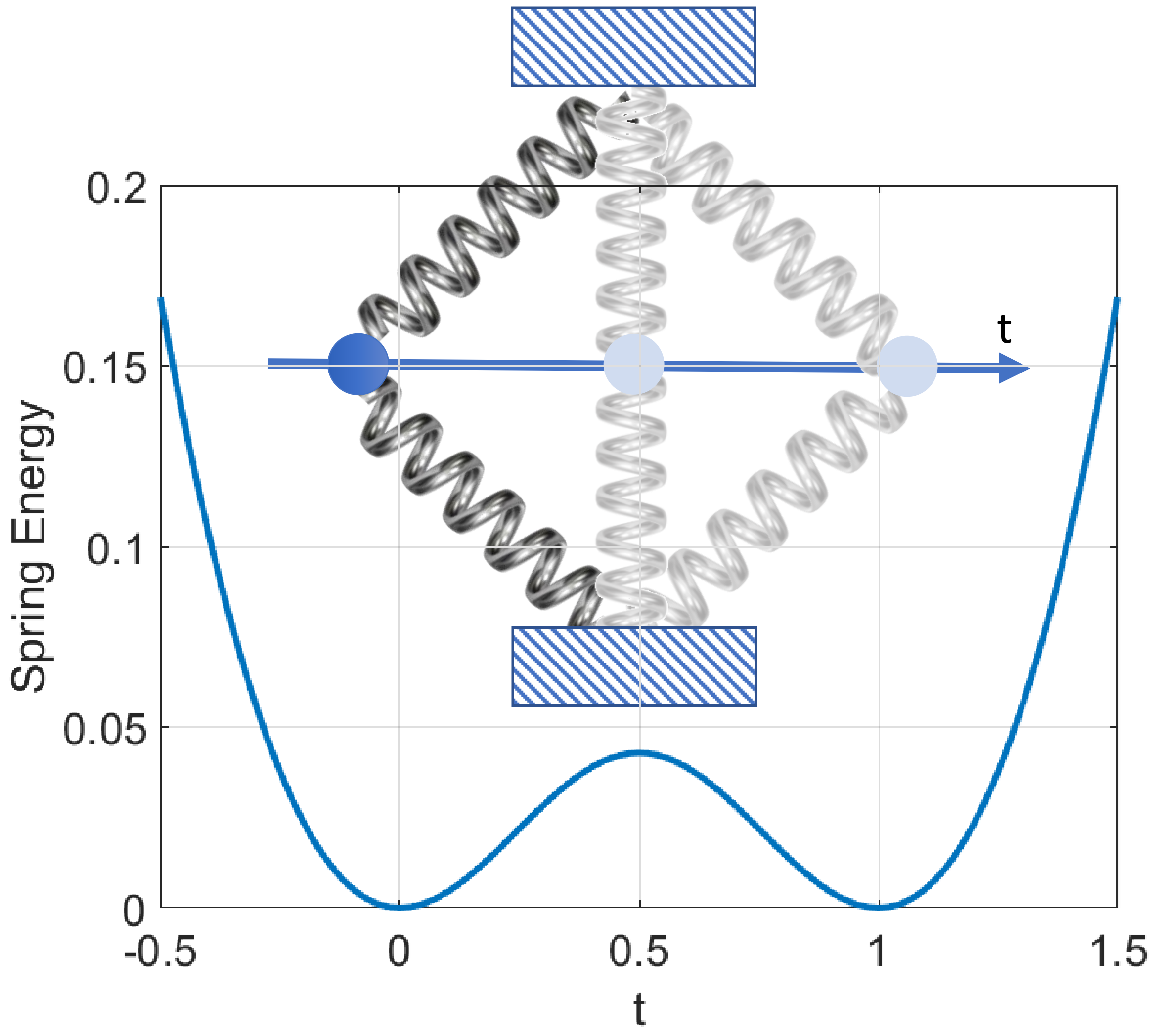


Almost a complete Newton's method

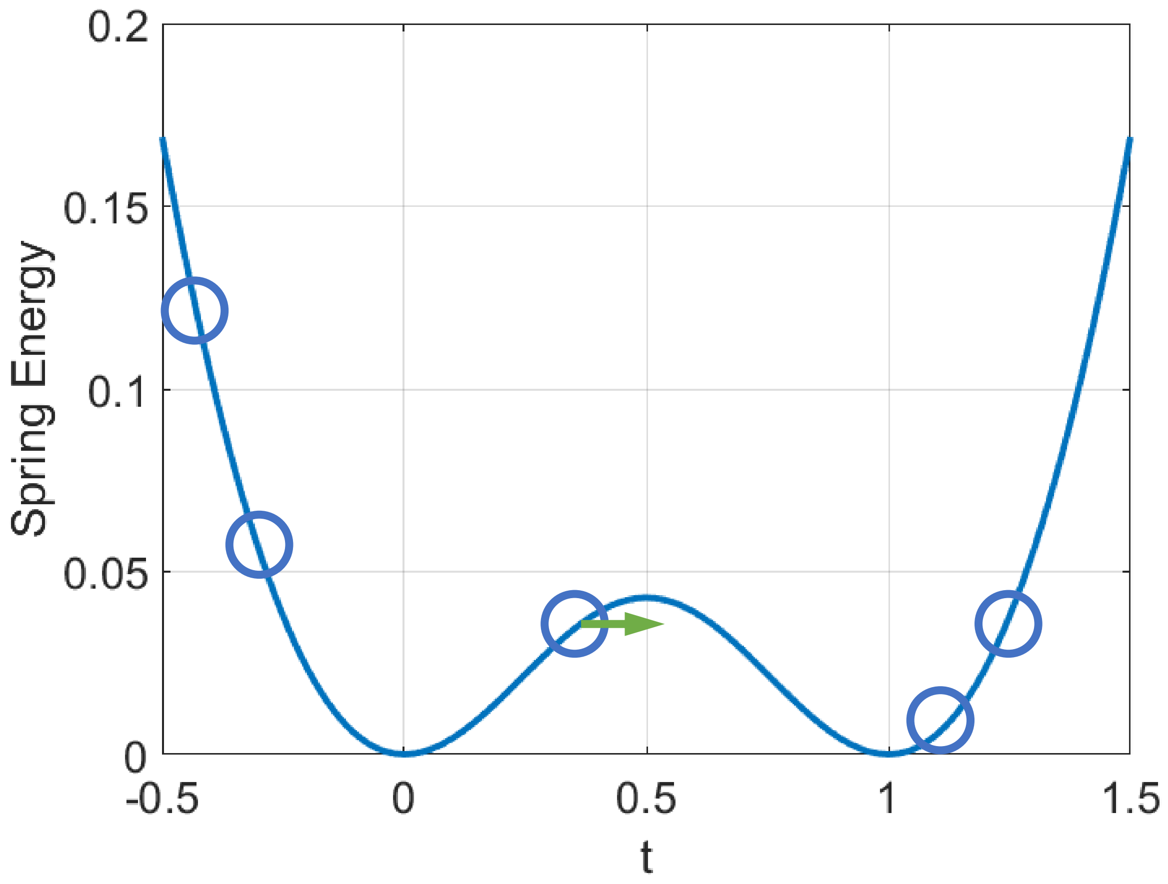
- $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$

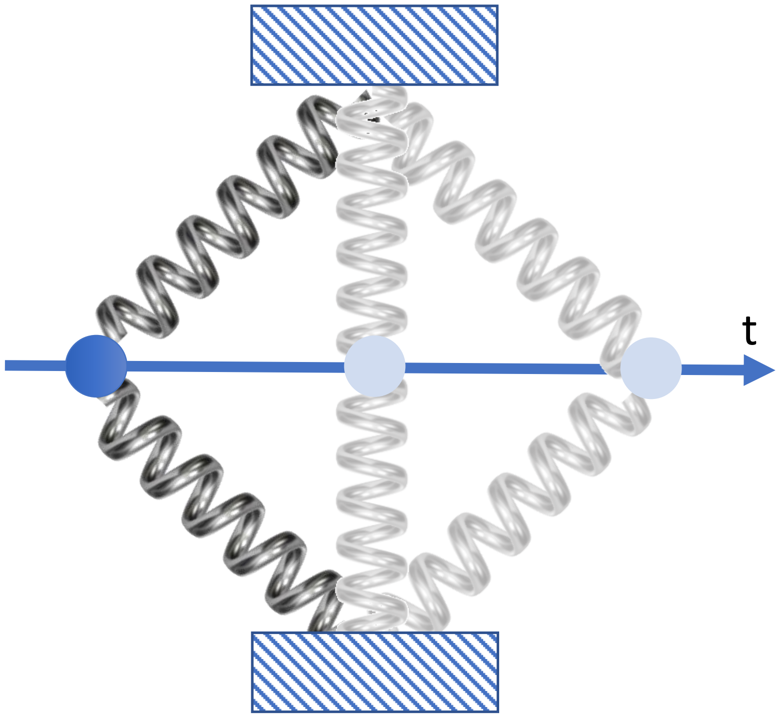
- Step 1: Initial guess: $x = x_n$ or $x = x_n + hv_n$
- Step 2: While not converged:
  - Compute descent direction: $dx = -\left(\nabla_x^2 g(x)\right)^{-1} \nabla_x g(x)$
  - Line search to determine the step size: t
  - Update: $x = x + t * dx$

Most deformable bodies have non-convex energies



One failure case for Newton's method





Vanilla Newton’s method (for convex)

- Step 1: Initial guess: $x = x_n$ or $x = x_n + hv_n$
- Step 2: While not converged:
  - Compute gradient direction: $\nabla_x g(x)$
  - Compute Hessian: $H = \nabla_x^2 g(x)$
  - Compute descent direction: $dx = -(H)^{-1} \nabla_x g(x)$
  - Line search to determine the step size: t
  - Update: $x = x + t * dx$

Definiteness-fixed Newton's method (for general cases)

- Step 1: Initial guess: $x = x_n$ or $x = x_n + hv_n$
- Step 2: While not converged:
  - Compute gradient direction: $\nabla_x g(x)$
  - Compute Hessian: $H = \nabla_x^2 g(x)$
  - Fix Hessian to positive definite: $\tilde{H} = fix(H)$
  - Compute descent direction: $dx = -\left(\tilde{H}\right)^{-1} \nabla_x g(x)$
  - Line search to determine the step size: t
  - Update: $x = x + t * dx$

Definiteness-fix: $\widetilde{H} = fix(H)$

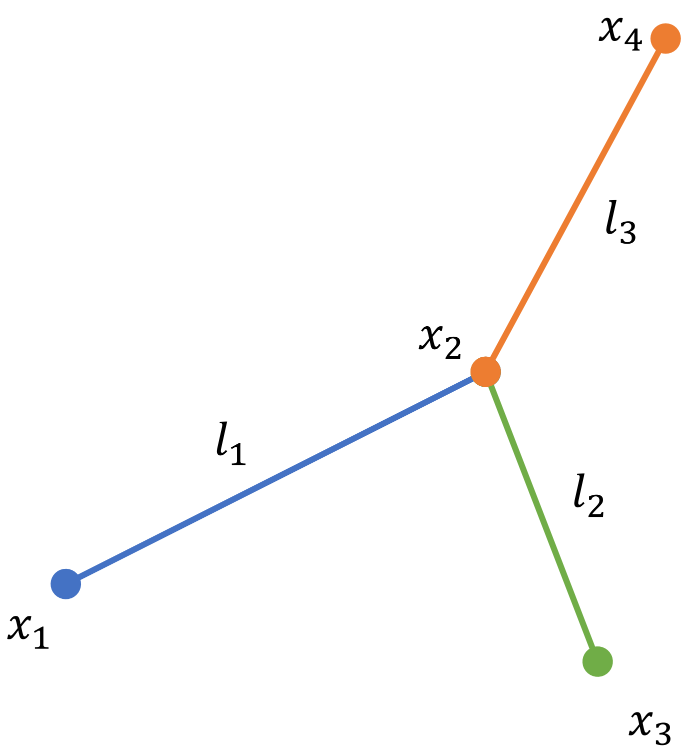
- Option I: global regularization
  - Init: $\widetilde{H} = H$, flag = False, reg = 1
  - while not flag:
    - flag, L = factorize($\widetilde{H}$) # try to factorize $\widetilde{H} = LL^T$
    - $\widetilde{H} = H + reg * I$, reg = reg * 10

Definiteness-fix: $\widetilde{H} = fix(H)$

- Option II: local regularization
  - $\nabla_x^2 g(x) = M + h^2 \nabla_x^2 E(x) = M + h^2 \sum_{j=1}^{m} \nabla_x^2 E_j(x)$
    - $K_1 \succcurlyeq 0 \Rightarrow \begin{bmatrix} K_1 & -K_1 \\ -K_1 & K_1 \end{bmatrix} = K_1 \otimes \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \succcurlyeq 0$
    - $\Rightarrow \nabla_x^2 E(x) = \nabla_x^2 E_1(x) + \nabla_x^2 E_2(x) + \nabla_x^2 E_3(x) \succcurlyeq 0$
    - $\Rightarrow \nabla_x^2 g(x) = M + h^2 \nabla_x^2 E(x) \succ 0$

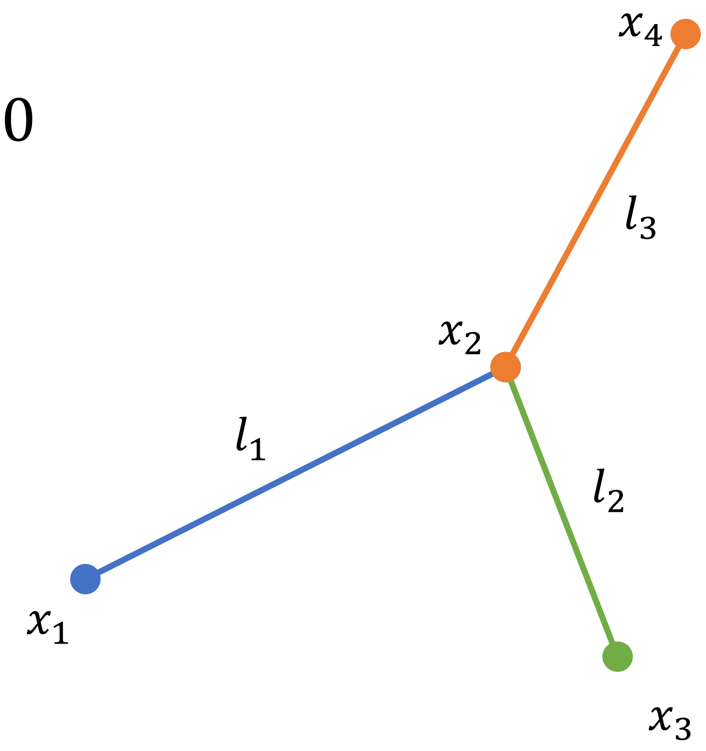
$$\begin{bmatrix} K_1 & -K_1 \\ -K_1 & K_1 \end{bmatrix} \begin{bmatrix} K_2 & -K_2 \\ -K_2 & K_2 \end{bmatrix} \begin{bmatrix} K_3 & -K_3 \\ -K_3 & K_3 \end{bmatrix}$$

$$\nabla_x^2 E(x) = \begin{bmatrix} K_1 & -K_1 & & \\ -K_1 & K_1 + K_2 + K_3 & -K_3 & -K_2 \\ & -K_3 & K_3 & \\ & -K_2 & & K_2 \end{bmatrix}$$



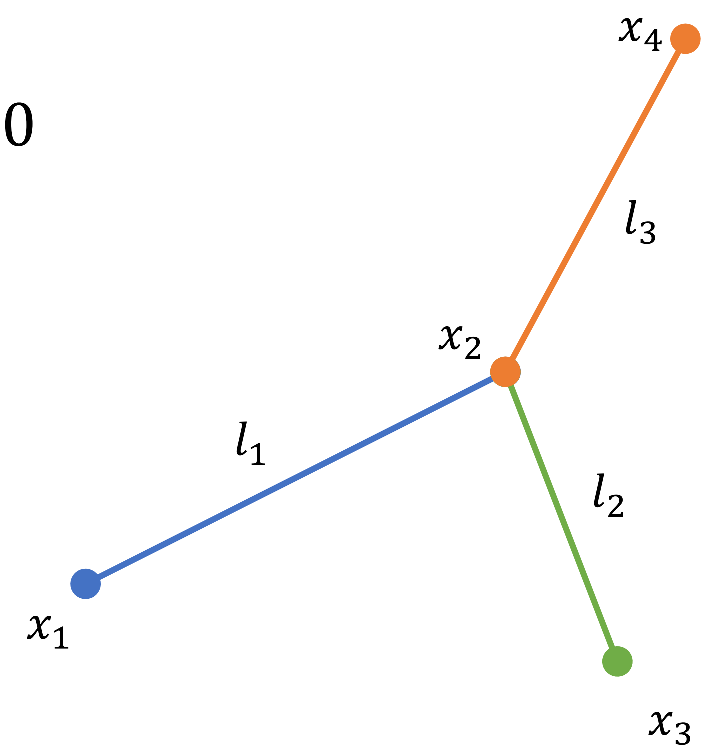
Definiteness-fix: $\widetilde{H} = fix(H)$

- Option II: local regularization
  - $\nabla_x^2 g(x) = M + h^2 \nabla_x^2 E(x) = M + h^2 \sum_{j=1}^{m} \nabla_x^2 E_j(x) \succ 0$
    - Has a sufficient condition: $K_1 \succcurlyeq 0, K_2 \succcurlyeq 0, K_3 \succcurlyeq 0$

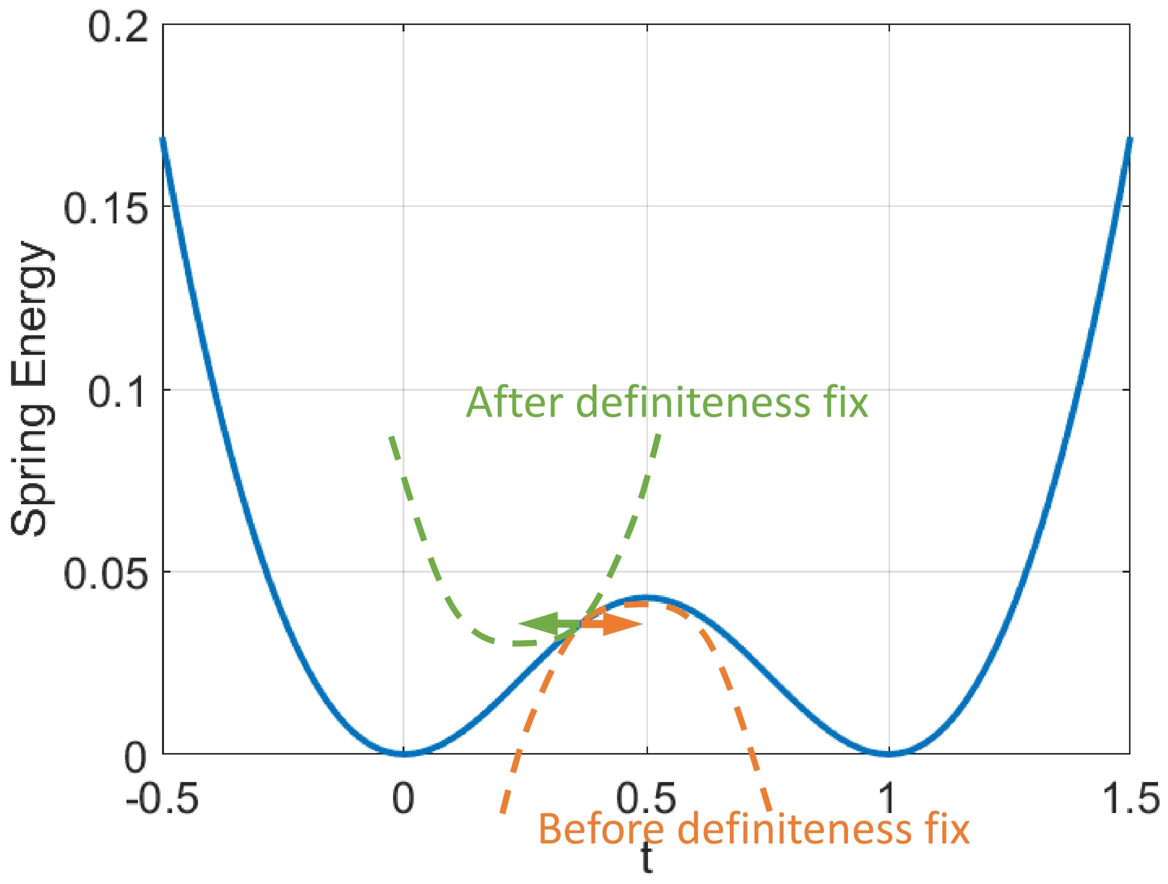


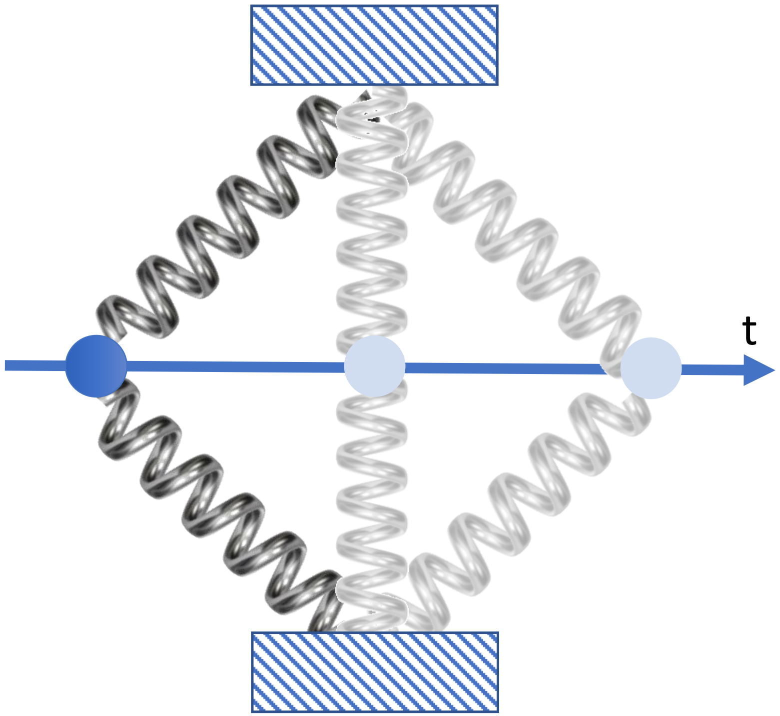
Definiteness-fix: $\widetilde{H} = fix(H)$

- Option II: local regularization
  - $\nabla_x^2 g(x) = M + h^2 \nabla_x^2 E(x) = M + h^2 \sum_{j=1}^{m} \nabla_x^2 E_j(x) \succ 0$
    - Has a sufficient condition: $K_1 \succcurlyeq 0, K_2 \succcurlyeq 0, K_3 \succcurlyeq 0$
  - $K_1 = k_1 \left(I - \frac{l_1}{\|x_1 - x_2\|} \left(I - \frac{(x_1 - x_2)(x_1 - x_2)^T}{\|x_1 - x_2\|^2} \right) \right) \in \mathbb{R}^{2\times 2}$
    - $K_1 = Q\Lambda Q^T$ ← Eigen value decomposition
    - $\widetilde{\Lambda} = \max(0, \Lambda)$ ← Clamp the negative eigen values
    - $\widetilde{K_1} = Q\widetilde{\Lambda}Q^T$ ← Construct the p.s.d. projection



A graphical interpretation of the definiteness-fix





Numerical recipes

- Goal: solve for $x_{n+1} = x_n + hv_n + h^2 M^{-1} f(x_{n+1})$

- The [Baraff and Witkin, 1998] style:
  - $\left(M - h^2 \nabla_x f(x_n)\right)\delta x = hMv_n + h^2 f(x_n) \Rightarrow x_{n+1} = x_n + \delta x, v_{n+1} = \frac{\delta x}{h}$

- Descent method:
  - Promote the root-finding problem to: $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$
  - Run Newton's method.
    - For each iteration, we want to evaluate $\nabla_x g, \nabla_x^2 g$, fix definiteness, perform a linear solve and run a line search.

One more thing…

Numerical recipes

- Goal: solve for $x_{n+1} = x_n + hv_n + h^2 M^{-1} f(x_{n+1})$

- The [Baraff and Witkin, 1998] style:
  - $\left(M - h^2 \nabla_x f(x_n)\right)\delta x = hMv_n + h^2 f(x_n) \Rightarrow x_{n+1} = x_n + \delta x, v_{n+1} = \frac{\delta x}{h}$

- Descent method:
  - Promote the root-finding problem to: $g(x) = \frac{1}{2}\|x - (x_n + hv_n)\|_M^2 + h^2 E(x)$
  - Run Newton's method.
    - For each iteration, we want to evaluate $\nabla_x g, \nabla_x^2 g$, fix definiteness, perform a linear solve and run a line search.
    - $dx = -\left(\widetilde{\nabla_x^2 g}\right)^{-1} \nabla_x g$

Linear solvers $Ax = b$

Linear solvers $Ax = b$

- Direct solvers:
  - Inversion: $x = A^{-1}b$

Linear solvers $Ax = b$

- Direct solvers:
  - Inversion: $x = A^{-1}b$
  - Factorization: $A = \begin{cases} LU & , if\ A\ is\ a\ square\ matrix \\ LDL^T & , if\ A = A^T \\ LL^T & , if\ A = A^T\ and\ A \succ 0 \end{cases}$

Linear solvers $Ax = b$

- Direct solvers:
    - Inversion: $x = A^{-1}b$
    - Factorization: $A = \begin{cases} LU & , if\ A\ is\ a\ square\ matrix \\ LDL^T & , if\ A = A^T \\ LL^T & , if\ A = A^T\ and\ A \succ 0 \end{cases}$
- Iterative solvers:
    - Stationary iterative linear solvers: Jacobi / Gauss-Seidel / SOR / Multigrid

Linear solvers $Ax = b$

- Direct solvers:
  - Inversion: $x = A^{-1}b$
  - Factorization: $A = \begin{cases} LU & , if\ A\ is\ a\ square\ matrix \\ LDL^T & , if\ A = A^T \\ LL^T & , if\ A = A^T\ and\ A \succ 0 \end{cases}$
- Iterative solvers:
  - Stationary iterative linear solvers: Jacobi / Gauss-Seidel / SOR / Multigrid
  - Krylov subspace methods: Conjugate Gradient (CG) / biCG / CR / MinRes / GMRes

Linear solvers $Ax = b$

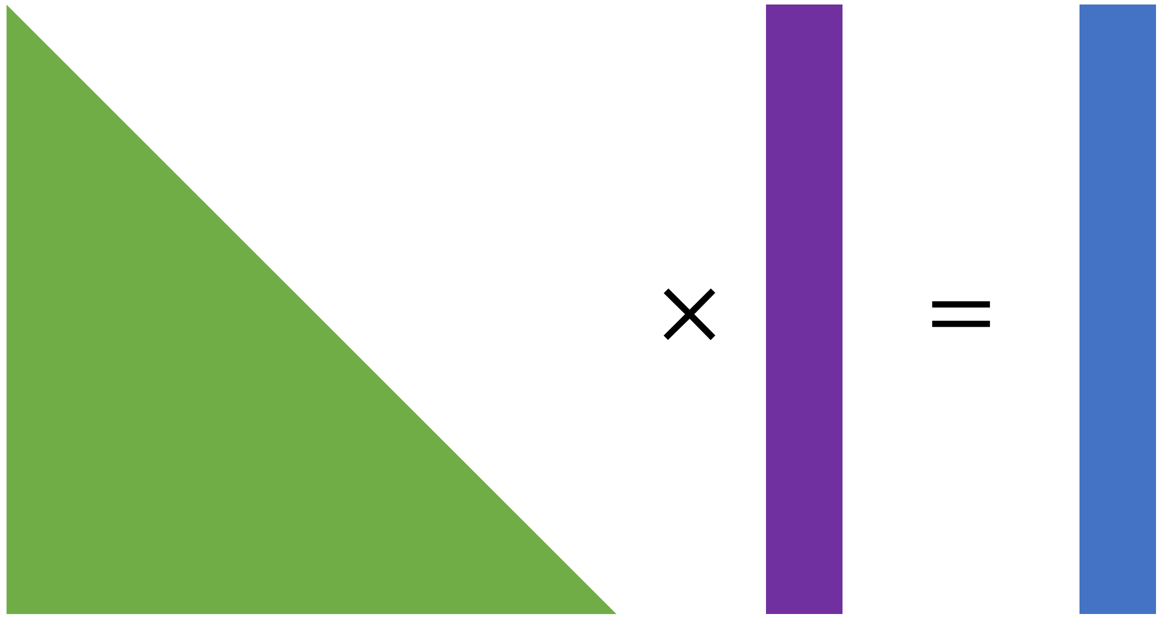
- Direct solvers:
  - Inversion: $x = A^{-1}b$
  - Factorization: $A = \begin{cases} LU & , if\ A\ is\ a\ square\ matrix \\ LDL^T & , if\ A = A^T \\ LL^T & , if\ A = A^T\ and\ A \succ 0 \end{cases}$
- Iterative solvers:
  - Stationary iterative linear solvers: Jacobi / Gauss-Seidel / SOR / Multigrid
  - Krylov subspace methods: Conjugate Gradient (CG) / biCG / CR / MinRes / GMRes

Factorization

- Factorize $A = LL^T$

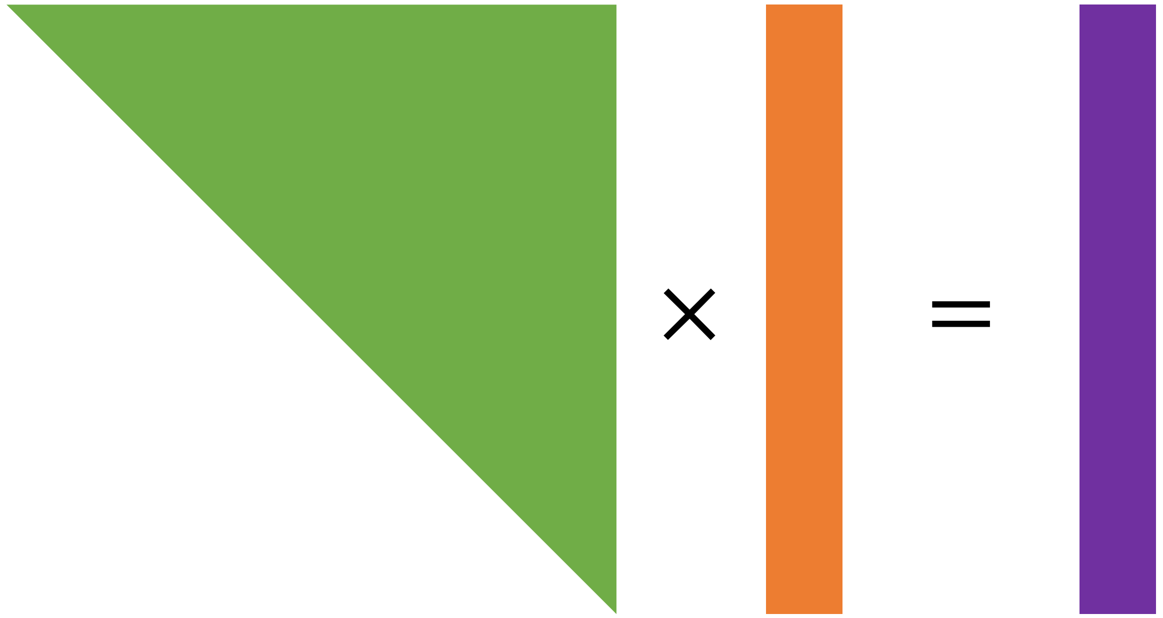
Factorization

- Solve $Ax = b$ is equivalent to $LL^T x = b$
- Let's first solve for $Ly = b$, which only requires a forward substitution



Factorization

- Once $Ly = b$ is solved
- We can solve for $L^T x = y$ by another backward substitution



Factorization using Taichi

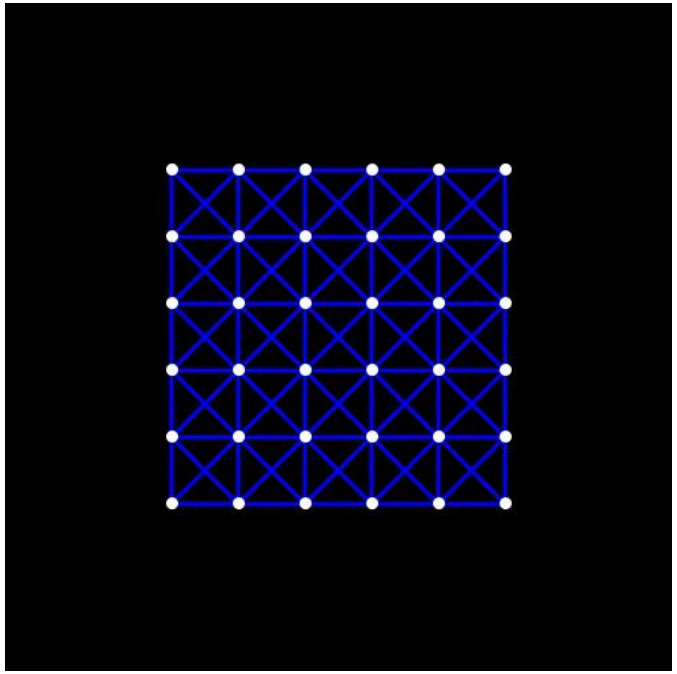
- Current APIs and docs: [Link]
  - [SparseMatrixBuilder] = ti.linalg.SparseMatrixBuilder()
  - [SparseMatrix] = [SparseMatrixBuilder].build()
  - [SparseMatrixSolver] = ti.linalg.SparseSolver(solver_type, ordering)
  - [NumpyArray] = [SparseMatrixSolver].solve([Field])

- Supports the CPU backend only

- Matrix-vector multiplication and sparse linear system solver returns
  - A numpy array

- Useful for the purpose of prototyping

Factorization using Taichi



Implicit Mass-spring
Simulation @禹鹏
[code]

The conjugate gradient (CG) method

- Works for any symmetric positive definite matrix $A = A^T, A \succ 0$

The conjugate gradient (CG) method

- Works for any symmetric positive definite matrix $A = A^T, A \succ 0$
- Guarantees to converge in n iterations for $A \in \mathbb{R}^{n \times n}$

The conjugate gradient (CG) method

- Works for any symmetric positive definite matrix $A = A^T, A \succ 0$
- Guarantees to converge in n iterations for $A \in \mathbb{R}^{n \times n}$
- Works amazingly good if the condition number $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$ of A is small

The conjugate gradient (CG) method

- Works for any symmetric positive definite matrix $A = A^T, A \succ 0$
- Guarantees to converge in n iterations for $A \in \mathbb{R}^{n \times n}$
- Works amazingly good if the condition number $\kappa = \frac{\lambda_{max}}{\lambda_{min}}$ of A is small
- You can write it from scratch using ~20 lines of code

The conjugate gradient (CG) method

```python
def conjugate_gradient(A, b, x):
    i = 0
    r = b – A @ x
    d = r
    delta_new = r.dot(r)
    delta_0 = delta_new
    while i < i_max and delta_new/delta_0 > epsilon**2:
        q = A @ d
        alpha = delta_new / d.dot(q)
        x = x + alpha*d
        r = b - A @ x # r = r - alpha * q
        delta_old = delta_new
        delta_new = r.dot(r)
        beta = delta_new / delta_old
        d = r + beta * d
        i = i + 1
    return x
```

The conjugate gradient (CG) method

```python
def conjugate_gradient(A, b, x):
    i = 0
    r = b - A @ x
    d = r
    delta_new = r.dot(r)
    delta_0 = delta_new
    while i < i_max and delta_new/delta_0 > epsilon**2:
        q = A @ d
        alpha = delta_new / d.dot(q)
        x = x + alpha*d
        r = b - A @ x # r = r - alpha * q
        delta_old = delta_new
        delta_new = r.dot(r)
        beta = delta_new / delta_old
        d = r + beta * d
        i = i + 1
    return x
```

Accelerate the conjugate gradient method

- Reduce the time of sparse-matrix-vector multiplication:
  - Making this q = A @ d a matrix-free black box
  - Recall our problem:
    - $A = M + h^2 \sum_{j=1}^{m} \nabla_x^2 E_j(x)$
    - $Ad = Md + h^2 \sum_{j=1}^{m} \nabla_x^2 E_j(x) d$
  - Do not forget to use @ti.kernel to parallelize your multiplication

$$\begin{bmatrix} K_1 & -K_1 & & \\ -K_1 & K_1 + K_2 + K_3 & -K_3 & -K_2 \\ & -K_3 & K_3 & \\ & -K_2 & & K_2 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix}$$

```python
def conjugate_gradient(A, b, x):
    i = 0
    r = b - A@x
    d = r
    delta_new = r.dot(r)
    delta_0 = delta_new
    while i < i_max and delta_new/delta_0 > epsilon**2:
        q = A @ d
        alpha = delta_new/d.dot(q)
        x = x + alpha*d
        r = b - A @ x # r = r - alpha * q
        delta_old = delta_new
        delta_new = r.dot(r)
        beta = delta_new/delta_old
        d = r + beta * d
        i = i + 1
    return x
```

Accelerate the conjugate gradient method

- Reduce the time of dot product:
  - Write a ti.@kernel for your dot product
  - Taichi enables thread local storage automatically for this reduction problem

- Further readings:
  - Taichi TLS [Link]
  - CUDA Reduction Guide [Link]

```python
def conjugate_gradient(A, b, x):
    i = 0
    r = b - A@x
    d = r
    delta_new = r.dot(r)
    delta_0 = delta_new
    while i < i_max and delta_new/delta_0 > epsilon**2:
        q = A @ d
        alpha = delta_new/d.dot(q)
        x = x + alpha*d
        r = b - A @ x # r = r - alpha * q
        delta_old = delta_new
        delta_new = r.dot(r)
        beta = delta_new/delta_old
        d = r + beta * d
        i = i + 1
    return x
```

Accelerate the conjugate gradient method

- Reduce the condition number of A:
  - $\|e_i\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \|e_0\|_A$

- Instead of solving $Ax = b$, let us solve for $M^{-1}Ax = M^{-1}b$
  - The choice of the preconditioner: $M \approx A$:
    - Jacobi: $M = diag(A)$
    - Incomplete Cholesky: $M = \tilde{L}\tilde{L}^{\mathrm{T}}$
    - Multigrid

The conjugate gradient (CG) method

- Further reading:
  - An Introduction to the Conjugate Gradient Method Without the Agonizing Pain [Link]

Remark

Remark

- The implicit Euler integration
  - Why? Because we want large time-steps
  - How? Essentially a nonlinear root-finding problem
- Numerical recipes for implicit integrations
  - The [Baraff and Witkin 1998] style
  - General descent method (usually Newton's method)
  - Hessian evaluation, assembly, and definiteness-fix for the mass-spring system
- Linear solvers
  - Direct solvers (based on matrix factorization)
  - Iterative solvers (the conjugate gradient method)

Remark

- The implicit Euler integration
  - Why? Because we want large time-steps
  - How? Essentially a nonlinear root-finding problem
- Numerical recipes for implicit integrations
  - The [Baraff and Witkin 1998] style
  - General descent method (usually Newton's method)
  - Hessian evaluation, assembly, and definiteness-fix for the mass-spring system
- Linear solvers
  - Direct solvers (based on matrix factorization)
  - Iterative solvers (the conjugate gradient method)

Remark

- The implicit Euler integration
  - Why? Because we want large time-steps
  - How? Essentially a nonlinear root-finding problem
- Numerical recipes for implicit integrations
  - The [Baraff and Witkin 1998] style
  - General descent method (usually Newton's method)
  - Hessian evaluation, assembly, and definiteness-fix for the mass-spring system
- Linear solvers
  - Direct solvers (based on matrix factorization)
  - Iterative solvers (the conjugate gradient method)

Remark

- The implicit Euler integration
  - Why? Because we want large time-steps
  - How? Essentially a nonlinear root-finding problem
- Numerical recipes for implicit integrations
  - The [Baraff and Witkin 1998] style
  - General descent method (usually Newton’s method)
  - Hessian evaluation, assembly, and definiteness-fix for the mass-spring system
- Linear solvers
  - Direct solvers (based on matrix factorization)
  - Iterative solvers (the conjugate gradient method)

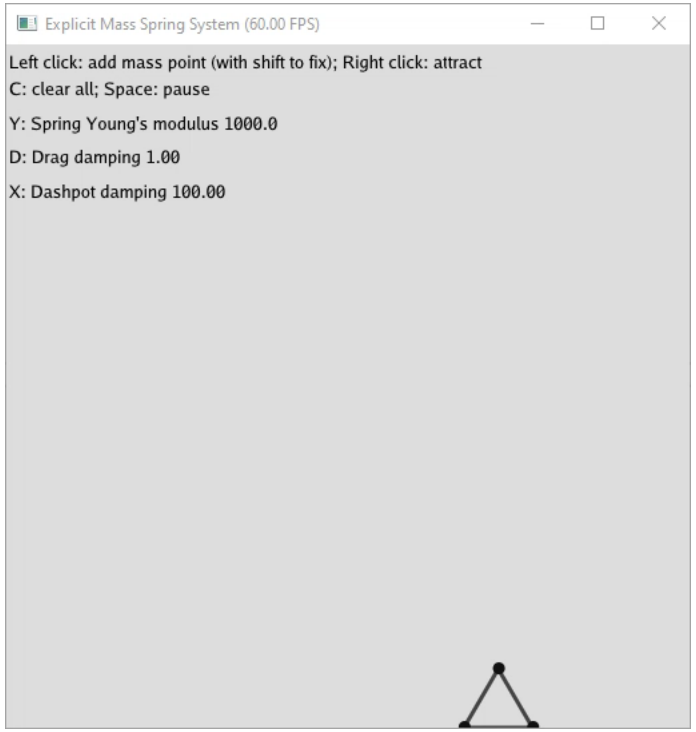
Further readings

- *Real Time Physics, Chapter 3,4* [SIGGRAPH 2008 Course] [Link]
- *Finite Element Method, Part I* [SIGGRAPH 2012 Course] [Link]
- *Dynamic Deformables: Implementation and Production Practicalities* [SIGGRAPH 2020 Course] [Link]

Homework

Homework Today

- Download (or pull from) the repo (--Deformables):
  - https://github.com/taichiCourse01/--Deformables
- Try:
  - turning the mass-spring game [Code] into an implicitly integrated one.
  - turning up the stiffness to see what happens using the implicit and explicit integrations.

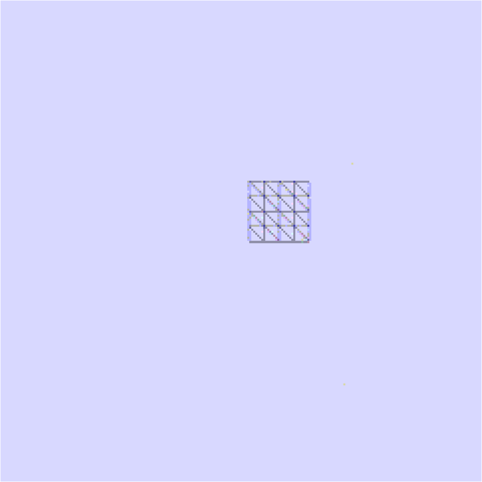


Start your final project if you are into deformable body simulation

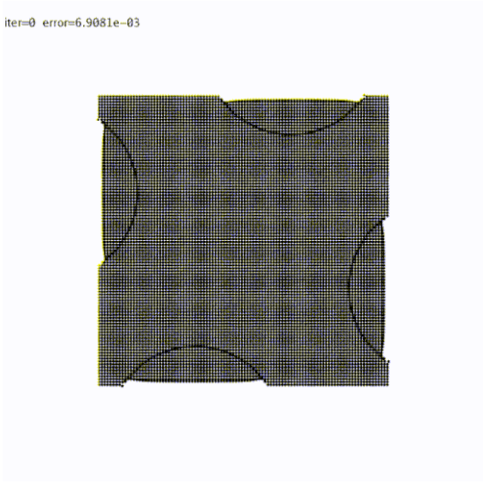
- Candidate topics:
  - Supporting collisions and contact (with friction) [Course]
  - Playing with material models [Bending][Stable NeoHookean][Hair]
  - Simulating linear FEM using implicit time integration [Course]
  - Render your deformables using your own renderer (You may want an obj reader/writer as well)

- Both 2D and 3D projects are great!
  - As long as your pictures look great ☺

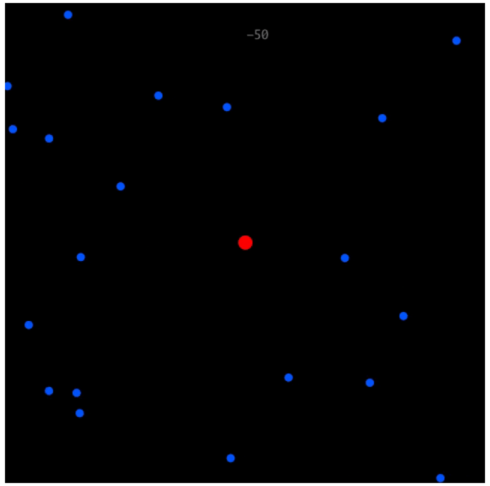
Excellent homework assignments



[@chunleili]

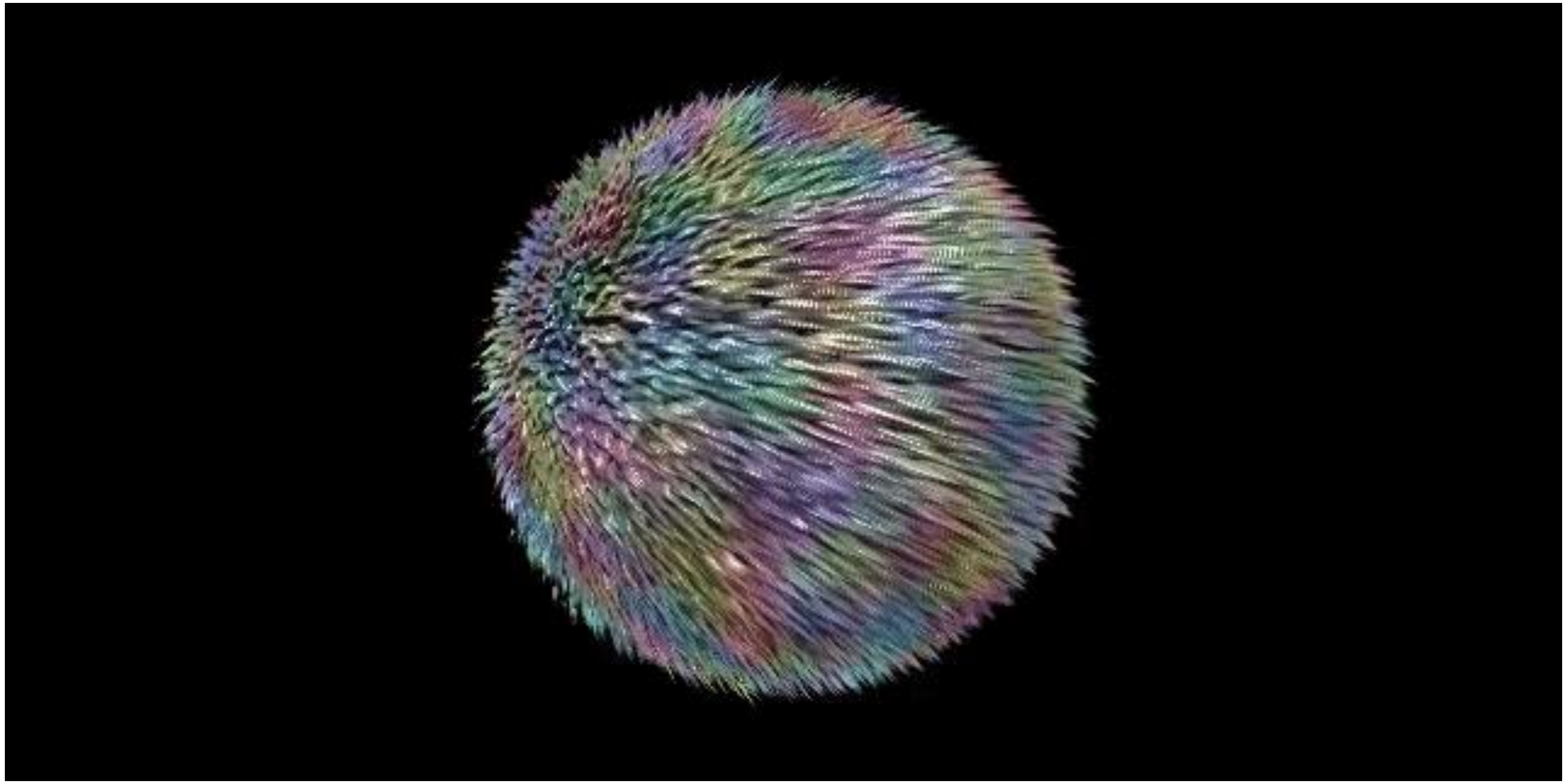


[@chunleili]



[@runck]

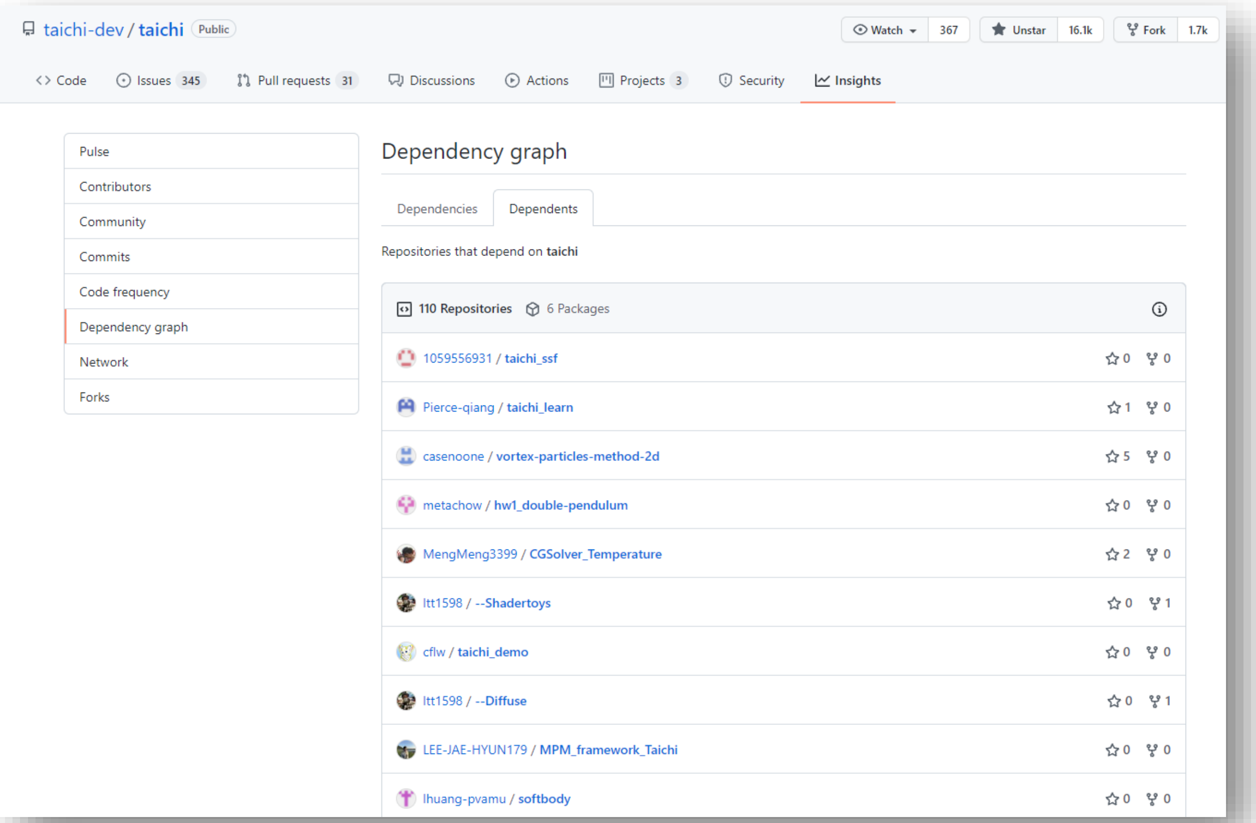
Excellent home work assignments

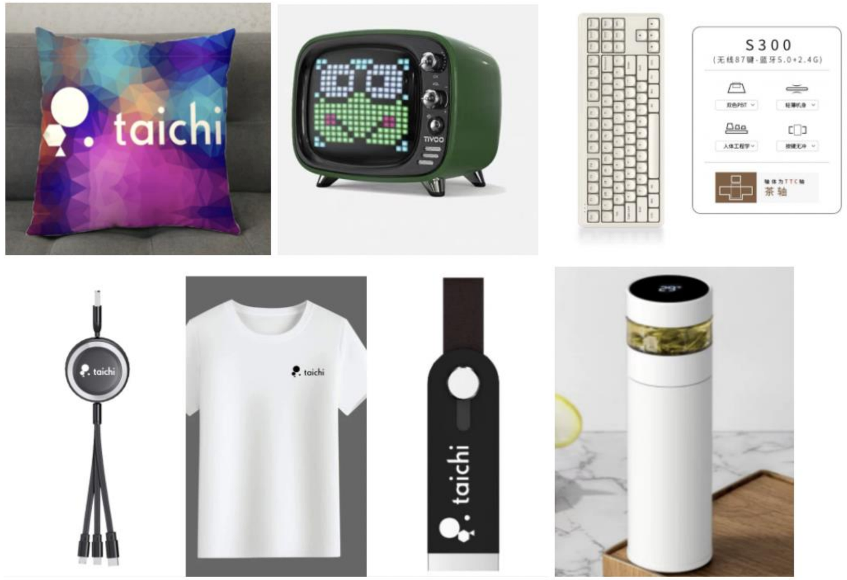


[@ruiwng]

Gifts for the gifted

- Use Template for your homework
- Next check Dec. 14, 2021





Questions?

本次答疑：11/25 🡐作业分享也在这里

下次直播：11/30

直播回放：Bilibili 搜索「太极图形」

主页&课件：https://github.com/taichiCourse01

主页&课件(backup)：https://docs.taichi.graphics/tgc01