# Graphics
# HW 4

04/23/2018

Jackie Li

Wennie Tabib

**Carnegie Mellon University**

# Cubic Hermite Spline

- The Hermite form is given by:

$$p(t) = h_{00}(t)p_0 + h_{10}(t)m_0 + h_{01}(t)p_1 + h_{11}(t)m_1$$

$$h_{00}(t) = 2t^3 - 3t^2 + 1$$

$$h_{10}(t) = t^3 - 2t^2 + t$$

$$h_{01}(t) = -2t^3 + 3t^2$$

$$h_{11}(t) = t^3 - t^2$$

- Where m0,m1 are the endpoint tangents, p0,p1 are the endpoint positions, and hij are the Hermite bases.

# Cubic Hermite Spline

- Now, we take the first derivative of the function:

$$p'(t) = h'_{00}(t)p_0 + h'_{10}(t)m_0 + h'_{01}(t)p_1 + h'_{11}(t)m_1$$

$$h'_{00}(t) = 6t^2 - 6t$$

$$h'_{10}(t) = 3t^2 - 4t + 1$$

$$h'_{01}(t) = -6t^2 + 6t$$

$$h'_{11}(t) = 3t^2 - 2t$$

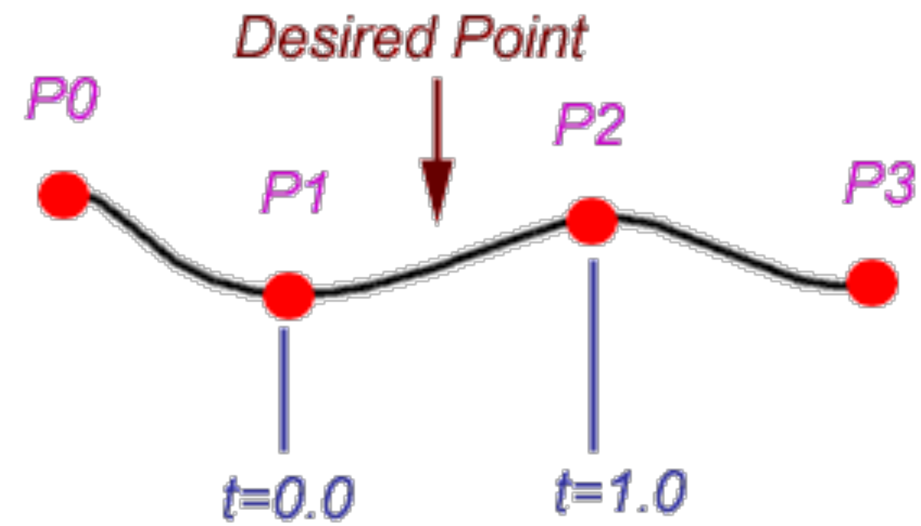
# Cubic Hermite Spline

- And the second:

$$
\begin{aligned}
p''(t) &= h''_{00}(t)p_0 + h''_{10}(t)m_0 + h''_{01}(t)p_1 + h''_{11}(t)m_1 \\
h''_{00}(t) &= 12t - 6 \\
h''_{10}(t) &= 6t - 4 \\
h''_{01}(t) &= -12t + 6 \\
h''_{11}(t) &= 6t - 2
\end{aligned}
$$

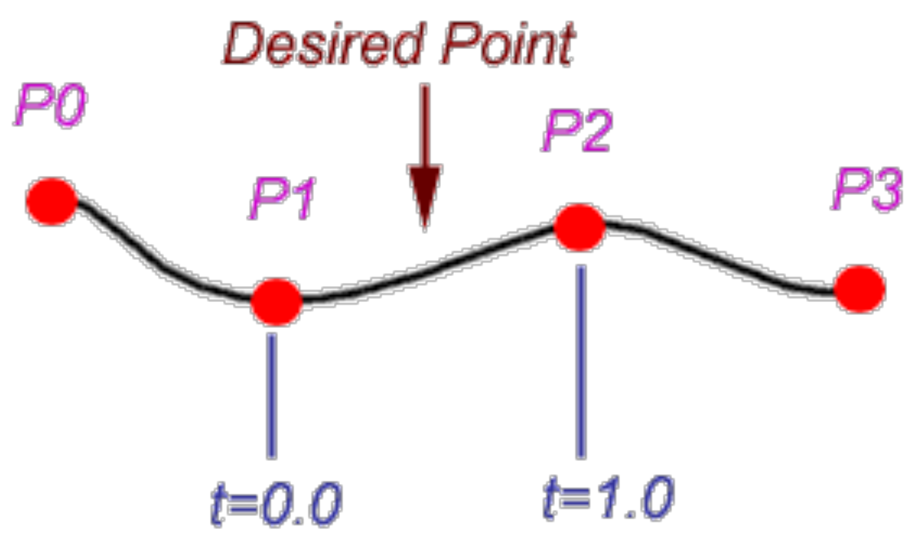
# Catmull-Rom spline

- We specify a series of points (knots) at intervals along a curve and define a function that allows additional points within an interval to be calculated.



# Catmull-Rom spline

- We specify a series of points (knots) at intervals along a curve and define a function that allows additional points within an interval to be calculated.
- For this task, you must find the 4 closest knots.
- Note that **KnotIter** is a **map<double, T>::iterator**
- Some useful functions:
  - **upper_bound**
  - **Next**
  - **Prev**
  - **First**
  - **Second**

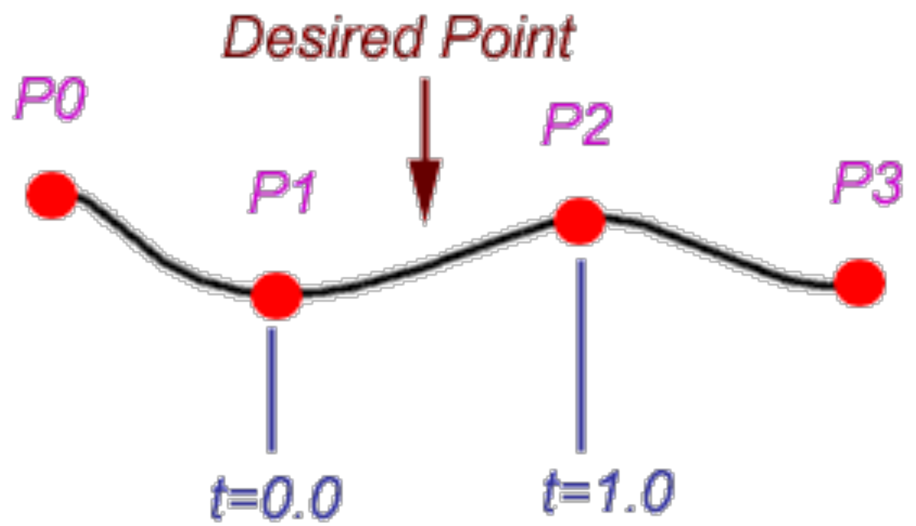


# Catmull-Rom spline

- Once you have the four closest points, call the **cubicSplineUnitInterval** with the the correct endpoints (p1, p2) and tangents:

$$m_1 = (p_2 - p_0)/(t_2 - t_0)$$

$$m_2 = (p_3 - p_1)/(t_3 - t_1)$$

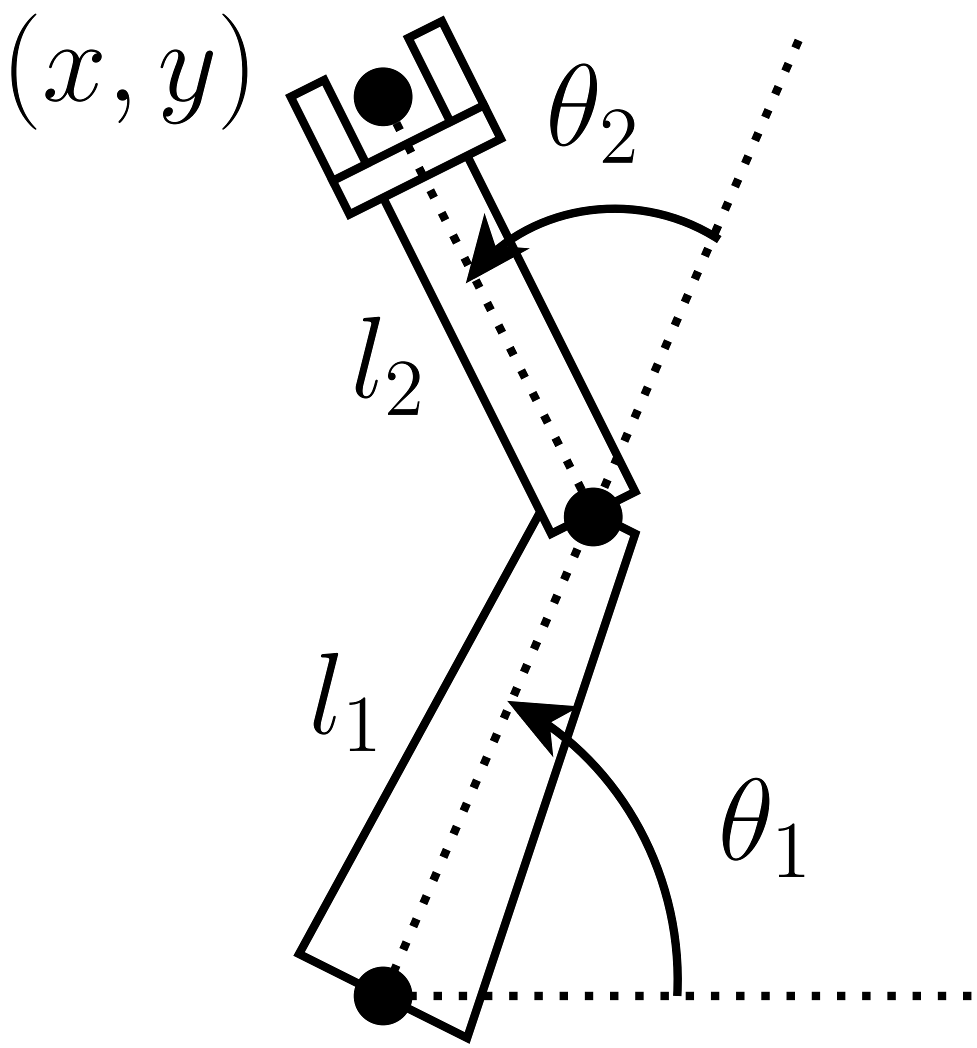



- Don’t forget to compute the appropriate time! It will no longer be **time**

# Kinematics

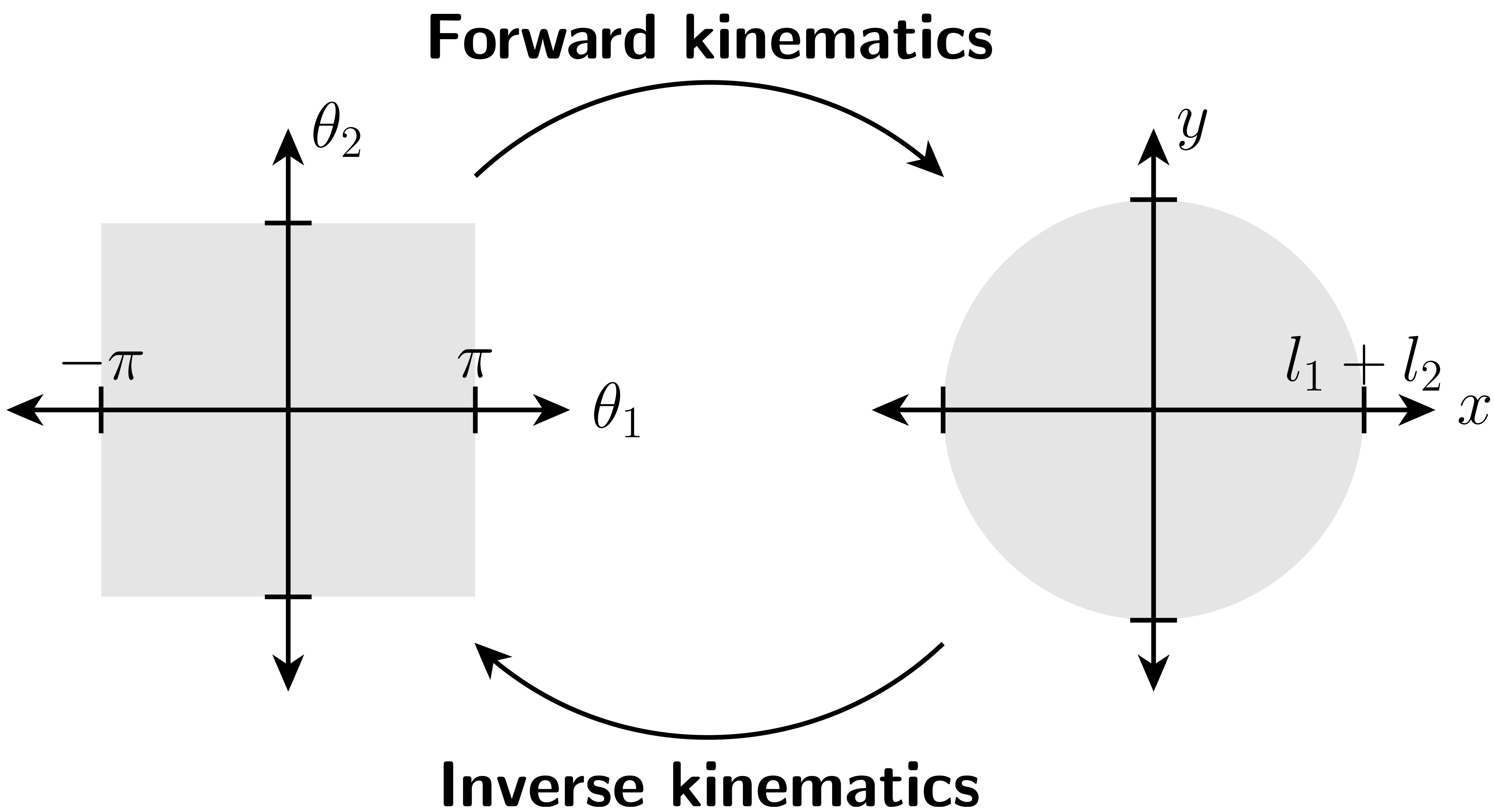
- Kinematics refers generally to the study of robot geometry
- Given a configuration of a robot (e.g., settings to joint angles), how does this affect the position of its parts?
- For a desired position of the robot end-effector, are there joint angles that achieve this position?

# Forward kinematics of two-link robot

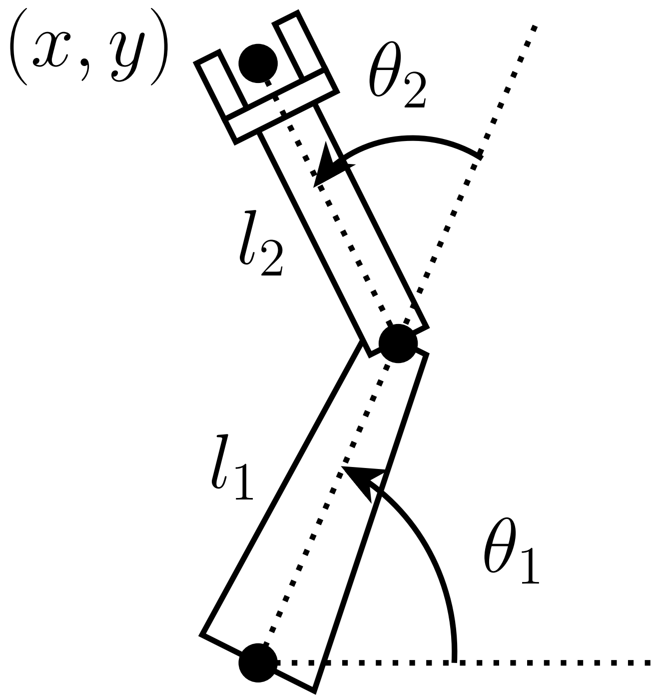



- $\theta_1, \theta_2$: joint angles of robot (configuration space, joint space)
- $l_1, l_2$: length of each link (robot parameters)
- $x, y$: position of end effector (task space)
- Kinematics is how we move back and forth between these representations

# Kinematics of two-link robot



# Forward kinematics of two-link robot




- Position of "elbow" $x_0, y_0$

$$x_0 = \ell_1 \cos(\theta_1)$$
$$y_0 = \ell_1 \sin(\theta_1)$$

- So, position of end effector $x, y$

$$x = \ell_1 \cos(\theta_1) + \ell_2 \cos(\theta_1 + \theta_2)$$
$$y = \ell_1 \sin(\theta_1) + \ell_2 \sin(\theta_1 + \theta_2)$$

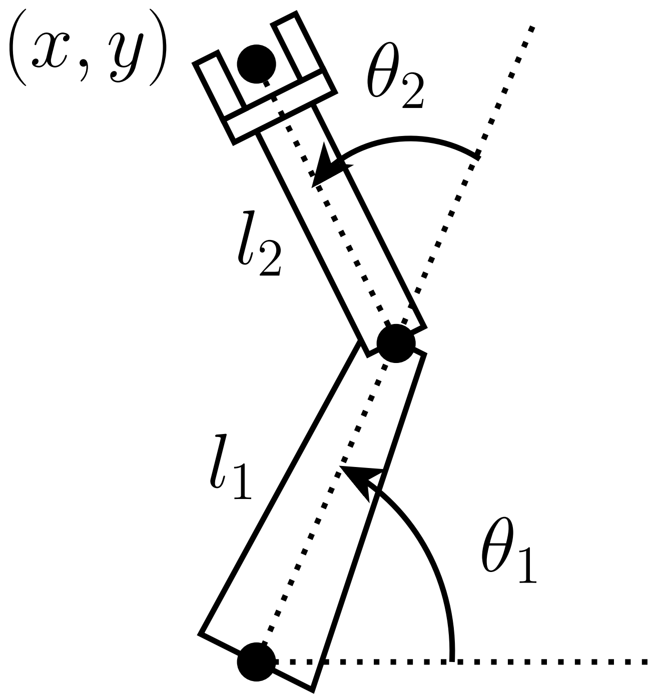
- For simplicity, we'll write this as

$$x = \ell_1 c_1 + \ell_2 c_{12}$$
$$y = \ell_1 s_1 + \ell_2 s_{12}$$

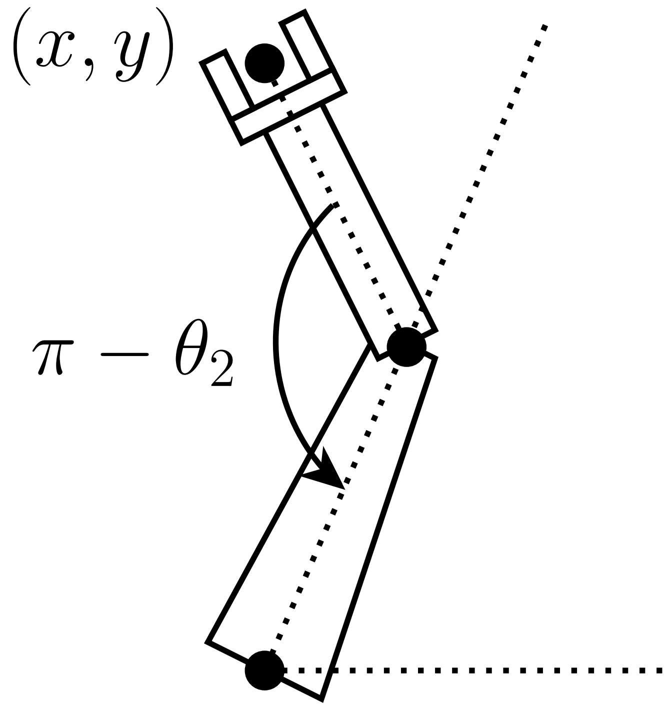
# Inverse kinematics of two-link robot

- Given $x, y$, can we find $\theta_1, \theta_2$ that achieve this position?
- This seems harder, there could be
  - Infinite solutions ($x = 0, y = 0$)
  - Two solutions ($\sqrt{x^2 + y^2} < \ell_1 + \ell_2$)
  - One solution ($\sqrt{x^2 + y^2} = \ell_1 + \ell_2$)
  - No solutions ($\sqrt{x^2 + y^2} > \ell_1 + \ell_2$)
- (Sometimes) can solve via inverse trigonometry functions

# Inverse kinematics of two-link robot



# Inverse kinematics of two-link robot

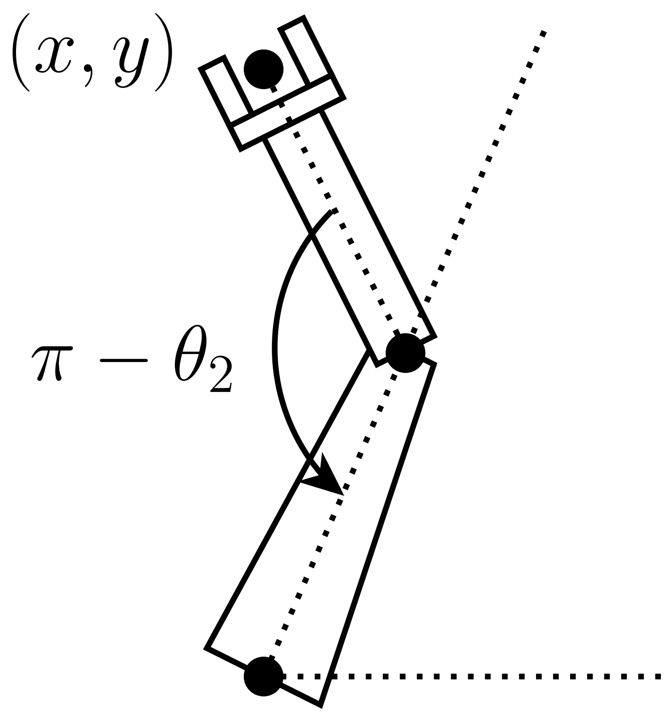



- From cosine rule

$$x^2 + y^2 = \ell_1^2 + \ell_2^2 - 2l_1 l_2 \cos(\pi - \theta_2)$$

$$\implies \quad \theta_2 = \pm \cos^{-1}\left(\frac{x^2 + y^2 - \ell_1^2 - \ell_2^2}{2l_1 l_2}\right)$$

# Inverse kinematics of two-link robot




- From cosine rule

$$x^2 + y^2 = \ell_1^2 + \ell_2^2 - 2 l_1 l_2 \cos(\pi - \theta_2)$$

$$\implies \quad \theta_2 = \pm \cos^{-1}\left(\frac{x^2 + y^2 - \ell_1^2 - \ell_2^2}{2 l_1 l_2}\right)$$

- Now solve for $\theta_1$

$$\tan\psi = y/x$$

$$\sin\phi = \frac{\ell_2 \sin(\theta_2)}{x^2 + y^2}$$

$$\implies \quad \theta_1 = \psi - \phi$$

$$= \tan^{-1}\left(\frac{y}{x}\right) - \sin^{-1}\left(\frac{\ell_2 \sin(\theta_2)}{x^2 + y^2}\right)$$

# Inverse kinematics of two-link robot

$$\theta_2 = \pm \cos^{-1}\left(\frac{x^2 + y^2 - \ell_1^2 - \ell_2^2}{2 l_1 l_2}\right)$$

$$\theta_1 = \tan^{-1}\left(\frac{y}{x}\right) - \sin^{-1}\left(\frac{\ell_2 \sin(\theta_2)}{x^2 + y^2}\right)$$

- What happens when $\sqrt{x^2 + y^2} > \ell_1 + \ell_2$?
- For general manipulators (more on this shortly), we may not be able to find a closed form solution.

# Inverse kinematics as optimization

**Challenges:**

- There may not always be a solution.
- If there is a solution, it may not always be the best.
- There may no closed form equation for the solution.

**Solution:**

- Iterative methods to approximate a good solution.
- For this, we need the Jacobian matrix!

# Jacobian

- *Jacobian* matrix contains derivatives of robot end effector with respect to joint angles

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} l_1 c_1 + l_2 c_{12} \\ l_1 s_1 + l_2 s_{12} \end{bmatrix}$$

so

$$\begin{aligned} J &= \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} \end{bmatrix} \\ &= \begin{bmatrix} -\ell_1 s_1 - \ell_s s_{12} & -\ell_2 s_{12} \\ \ell_1 c_2 + \ell_2 c_{12} & \ell_2 c_{12} \end{bmatrix} \end{aligned}$$

# Jacobian

- Jacobian also provides (instantaneous) relationship between joint velocities and velocities of end effector

- Let $\theta_1(t), \theta_2(t)$ be time-varying angles

- Then by chain rule

$$\frac{\partial x(t)}{\partial t} = \frac{\partial x(t)}{\partial \theta_1(t)} \frac{\partial \theta_1(t)}{\partial t} + \frac{\partial x(t)}{\partial \theta_2(t)} \frac{\partial \theta_2(t)}{\partial t}$$

i.e.

$$\left[ \begin{array}{c} \frac{\partial x(t)}{\partial t} \\ \frac{\partial y(t)}{\partial t} \end{array} \right] = J \left[ \begin{array}{c} \frac{\partial \theta_1(t)}{\partial t} \\ \frac{\partial \theta_2(t)}{\partial t} \end{array} \right]$$

# Jacobian Transpose

- Assume we have the following

$$\vec{\mathbf{s}} = (\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_k)^T\text{: positions of end effectors}$$

$$\vec{\mathbf{t}} = (\mathbf{t}_1, \mathbf{t}_2, ..., \mathbf{t}_k)^T\text{: target positions of end effectors}$$

$$\boldsymbol{\theta} = (\theta_1, \theta_2, ...\theta_n)^T\text{: joint angles}$$

$$\mathbf{e}_i = \mathbf{t}_i - \mathbf{s}_i, \text{ the desired change in position of the ith end effector}$$

Certain points on the links are identified as *end effectors*

# Jacobian Transpose

- The basic equation for forward dynamics that describes the velocities of the end effectors can be written as follows (using dot notation for the first derivatives

$$\dot{\vec{\mathbf{s}}} = J(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}.$$

- We seek an update value $\Delta\theta$ for the purpose of incrementing the joint angles $\theta$ by $\Delta\theta$

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \Delta\boldsymbol{\theta}.$$

- The change in joint angles can be estimated as

$$\Delta\vec{\mathbf{s}} \approx J\,\Delta\boldsymbol{\theta}.$$

# Jacobian Transpose

- The Jacobian transpose is a method that uses the transpose of *J* instead of the inverse of *J* for the inverse kinematics.
- In this formulation,

$$\Delta\boldsymbol{\theta} = \alpha J^T \vec{\mathbf{e}},$$

- For some appropriate scale factor $\alpha$.

$$\alpha = \frac{\langle \vec{\mathbf{e}}, JJ^T\vec{\mathbf{e}} \rangle}{\langle JJ^T\vec{\mathbf{e}}, JJ^T\vec{\mathbf{e}} \rangle}.$$

**Theorem 1** *For all $J$ and $\vec{\mathbf{e}}$,* $\langle JJ^T\vec{\mathbf{e}}, \vec{\mathbf{e}} \rangle \geq 0.$

**Proof** The proof is trivial: $\langle JJ^T\vec{\mathbf{e}}, \vec{\mathbf{e}} \rangle = \langle J^T\vec{\mathbf{e}}, J^T\vec{\mathbf{e}} \rangle = ||J^T\vec{\mathbf{e}}||^2 \geq 0.$ □

# Summary: Kinematics

- Two-link planar robot is not that useful in practice
- To manipulate objects in 3D space, we typically want full control over 3D position *and* 3D orientation of end effector $\Longrightarrow$ at least 6 joint angles
- Forward kinematics still easy to solve (just be careful with representing 3D rotations)
- Inverse kinematics often solvable too, but much more complicated

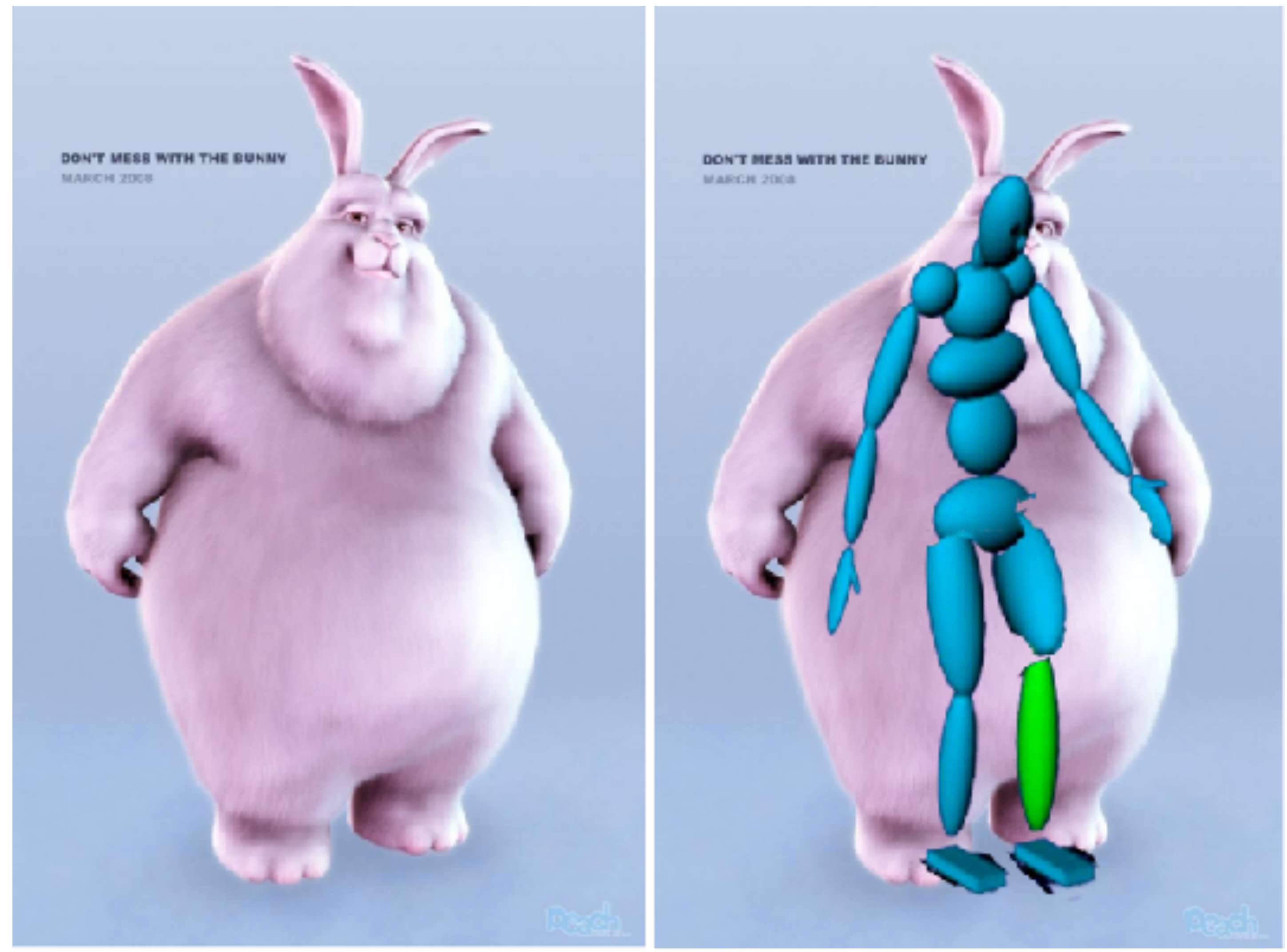
# More Information: Kinematics

- See the following resources:
- http://www.cs.cmu.edu/~zkolter/course/15-780-s14/robotics.pdf
- http://graphics.cs.cmu.edu/nsp/course/15464-s17/lectures/iksurvey.pdf

# Skinning Characters

## *Overview*

- What we have
  - skeleton
  - mesh
- Goal
  - embed the skeleton into the mesh



# Skinning Characters

## *Blending*

- Associate each vertex with joints
  - Only animate joints. Skin (mesh) vertices will move as joints move

# Skinning Characters

## *Skinning*

- The process of associating skin vertices (mesh) with joints (skeleton)
  - Only animate joints Skin (mesh) vertices will move as joints move
    - We know the position of each joint at every time step
- Need to infer how skin deforms from joint transformations
- Most popular technique: Skeletal Subspace Deformation (SSD)
  - simply Skinning
  - aliases:
    - vertex blending
    - linear blend skinning

# Skinning Characters

***Skinning***

- What if we attach each vertex of the skin to a single joint, say the nearest joint?
  - Skin will be rigid, except at joints where it will stretch badly

# Skinning Characters

## *Skinning*

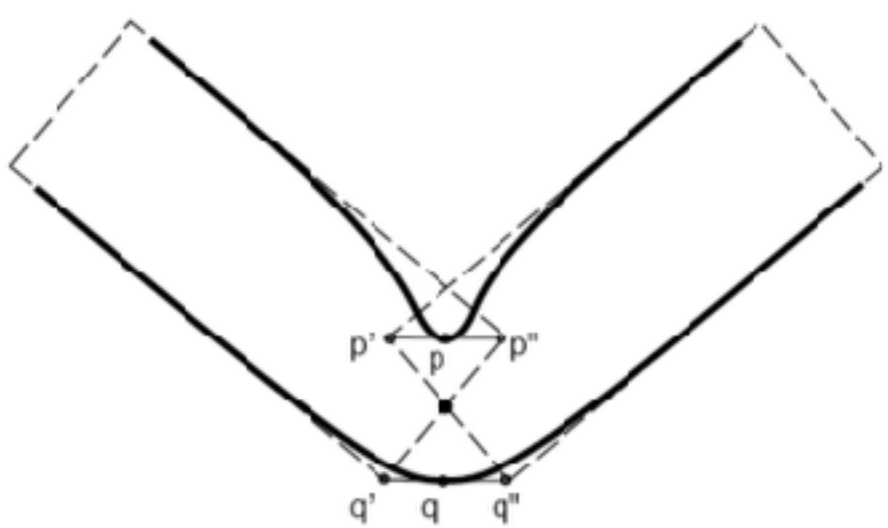
- What if we associate each vertex of the skin to a single joint, say the nearest joint?
  - Skin will be rigid, except at joints where it will stretch badly
- Solution:
  - associate a vertex to many joints!
  - skin is deformed according to a weighted combination of the joints

# Linear Blending

***Skinning (Skeletal Subspace Deformation  = SSD)***

- (At a fixed time step), a vertex on the deforming surface (skin / mesh) lies in the subspace defined by the rigid transformations of that point
  - i : index of the joint
  - p: position of the vertex at bind pos
    - for implementation, p is in the local coordinate frame of each joint
  - C_i : transformation of the i_th joint
  - w_i : weight scheme
    - How much vertex p should move with joint i

$$\bar{\mathbf{p}} = \sum w_i C_i(\mathbf{p})\mathbf{p}$$





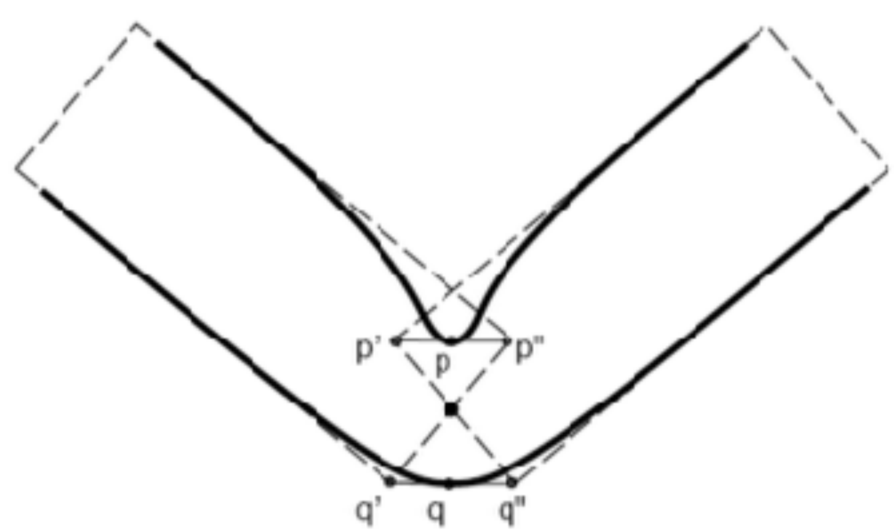

Carnegie Mellon
THE ROBOTICS INSTITUTE

# Linear Blending

## *Weight Scheme*

- Properties
    - w_i sum up to 1
    - should be non-negative
    - w_i = 1 means p is rigidly attached to joint i
    - w_i = 0 means p is not attached to joint i at all

$$\bar{\mathbf{p}} = \sum w_i C_i(\mathbf{p})\mathbf{p}$$

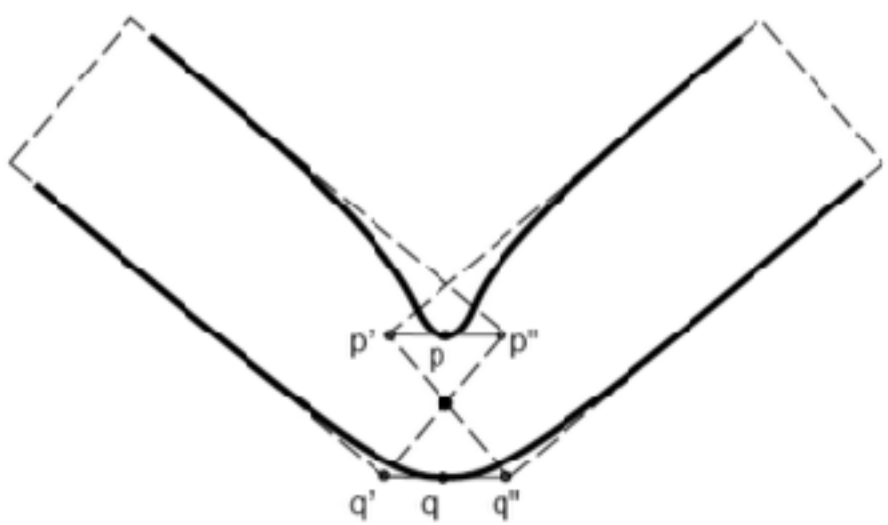


# Linear Blending

## *Weight Scheme*

- Properties
  - w_i sum up to 1
  - should be non-negative
  - w_i = 1 means p is rigidly attached to joint i
  - w_i = 0 means p is not attached to joint i at all

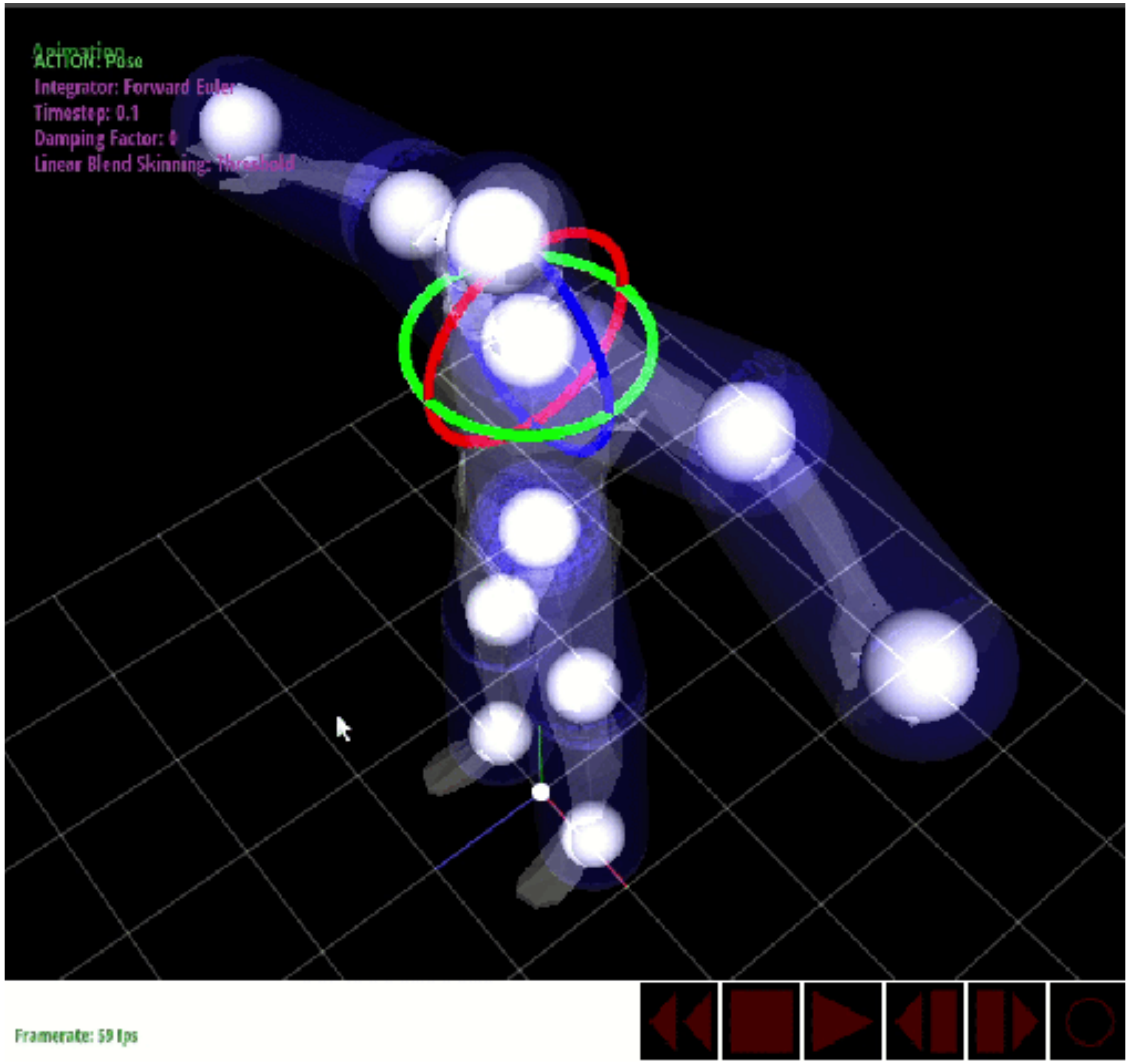
$$\bar{\mathbf{p}} = \sum w_i C_i(\mathbf{p})\mathbf{p}$$



# Linear Blending

## *Weight Scheme*

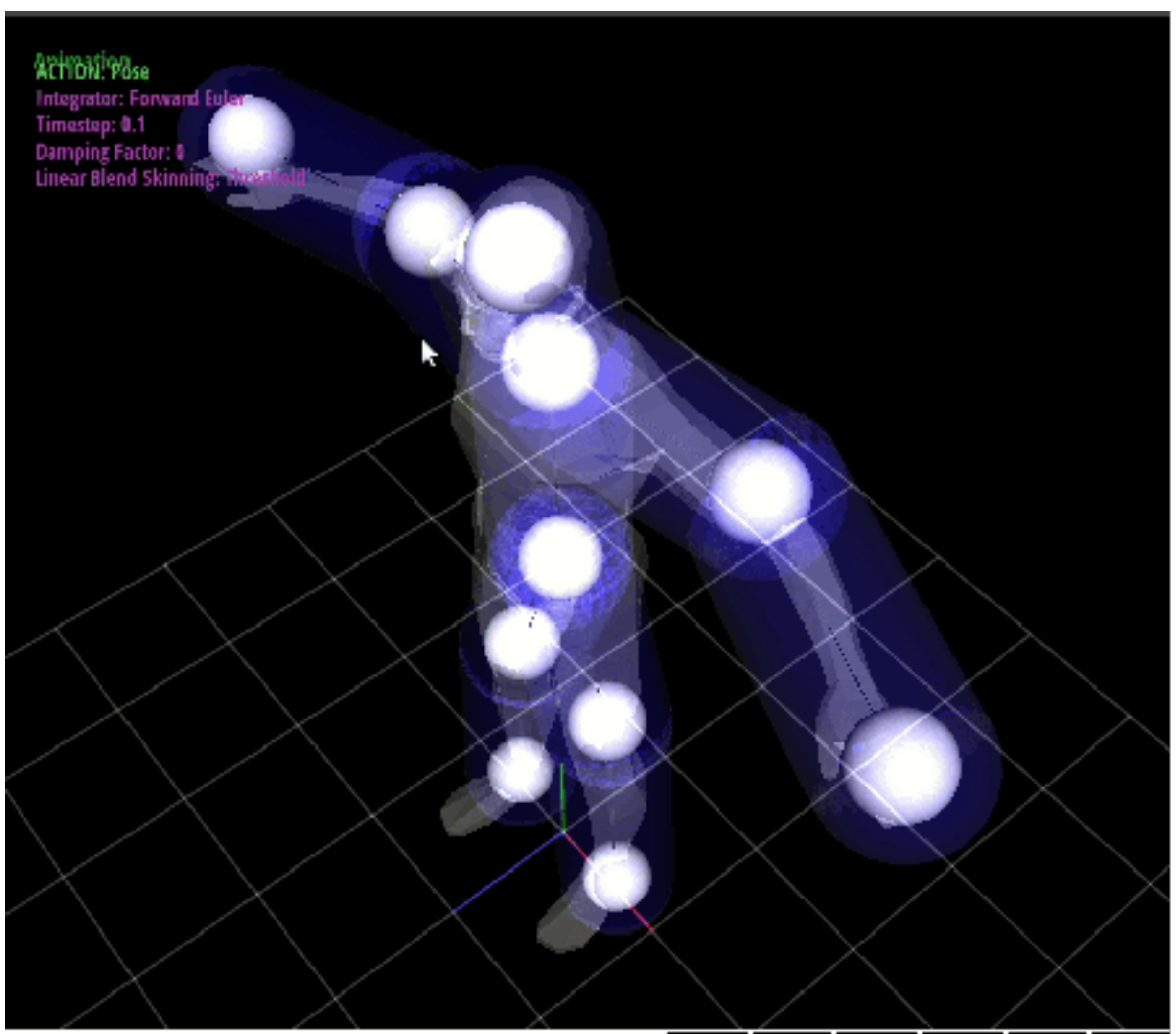
- Homework
  - Weight scheme: inverse distance
    - joints far away from the vertices should only slightly affect the vertex

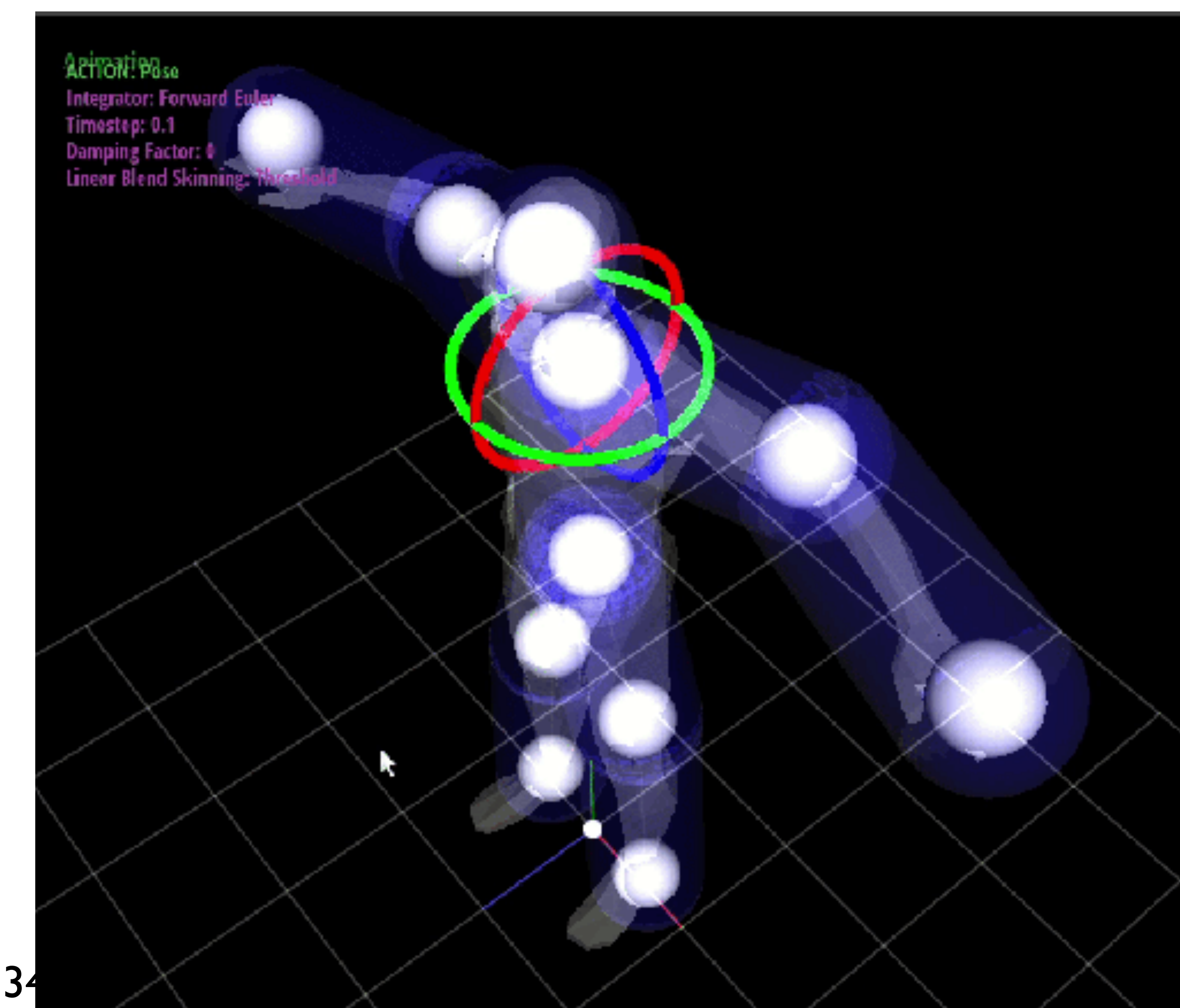


# Linear Blending

## *Weight Scheme*

- Homework
  - Weight scheme: inverse distance
    - apply threshold:
      - Given a fixed distance r, w_i = 0 if distance between joint and vertex is greater than r



# Physical Simulation

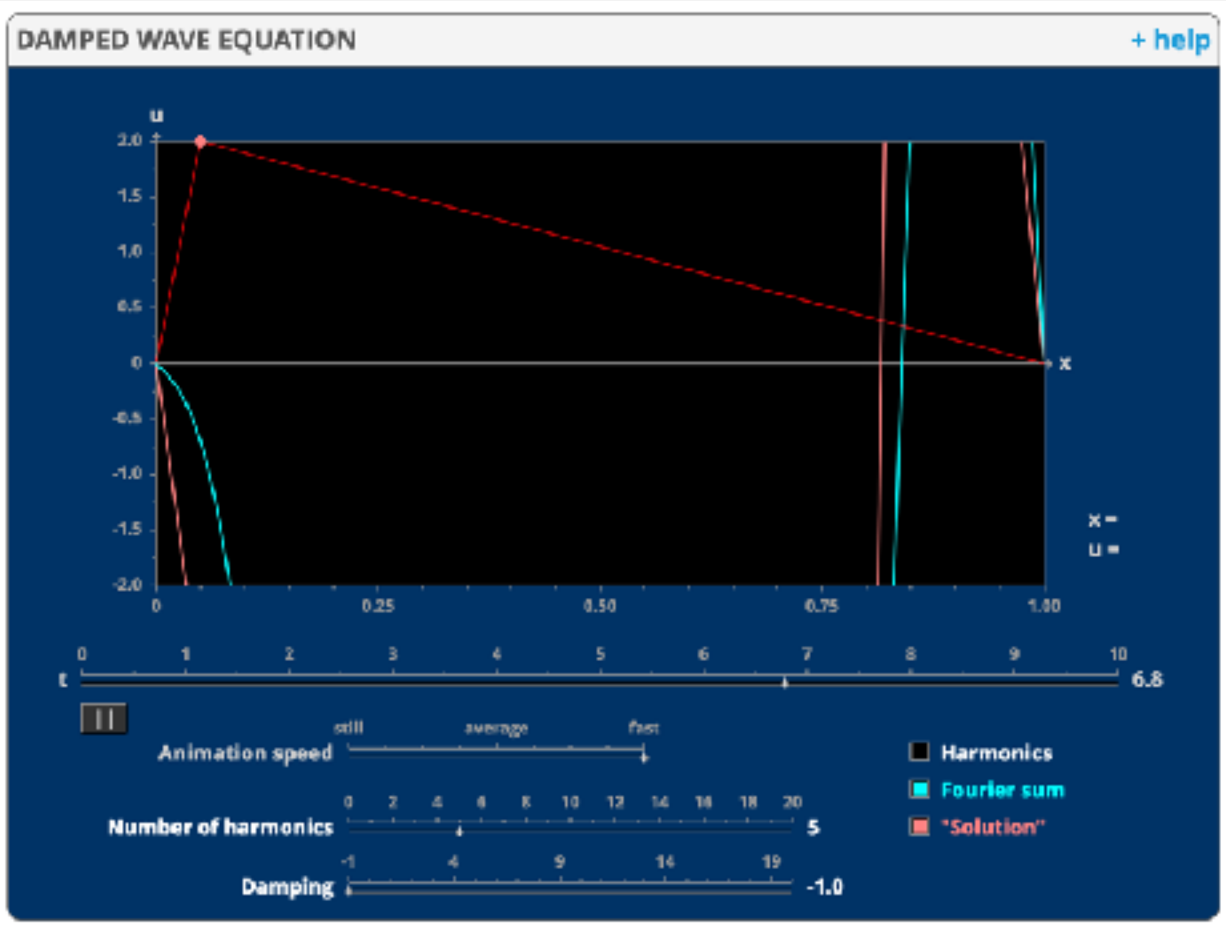
***Animation:***

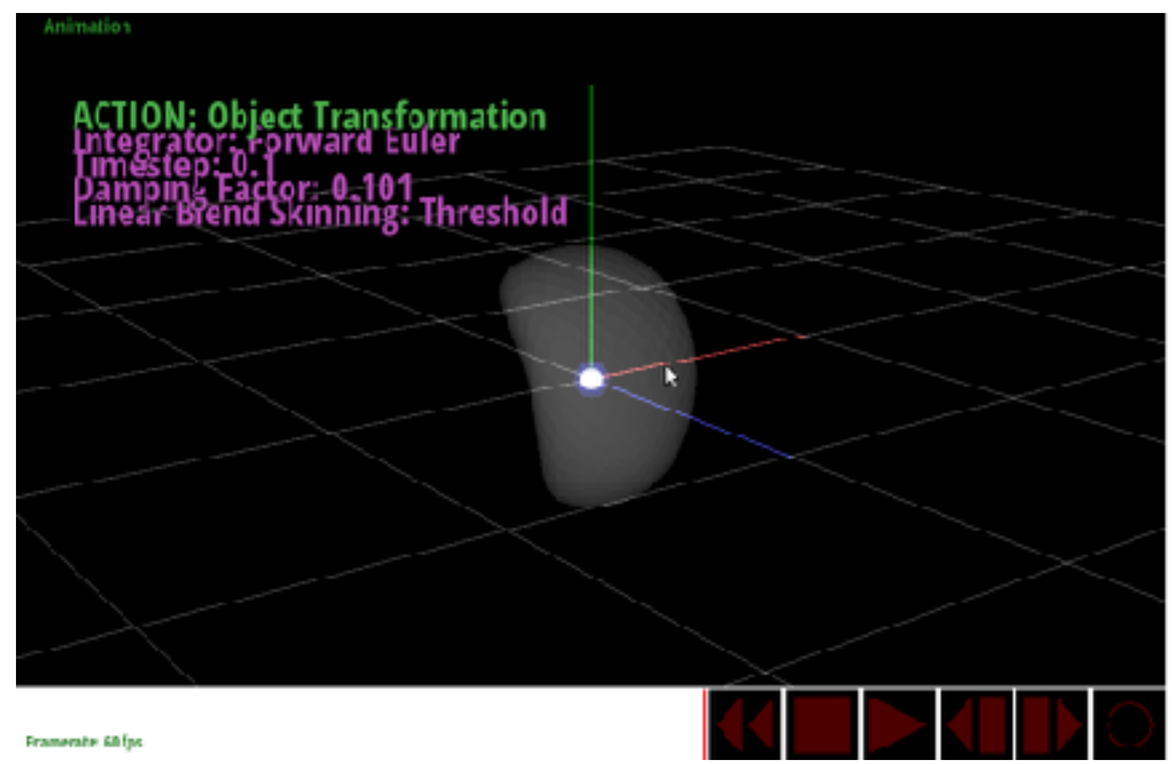
http://mathlets.org/mathlets/damped-wave-equation/

**nd order in time**

**2nd order in space**

$$u'' = \Delta u - \lambda u'$$



# Physical Simulation

- PDEs are difficult/impossible to solve analytically—especially if we want to incorporate data (e.g., user interaction)
- Must instead use numerical integration
- Basic strategy: as with ODEs, run a time-stepping algorithm
- Historically, very expensive—only for “hero shots” in movies
- Computers are ever faster...
- More & more use of PDEs in games, interactive tools, ...

# Physical Simulation

***Animation:***

http://mathlets.org/mathlets/damped-wave-equation/

$$u'' = \Delta u - \lambda u'$$

Laplace operator

$$\Delta u = \overset{\text{div}}{\nabla} \cdot \overset{\text{grad}}{\nabla} u$$

$$\Delta u = \frac{\partial u^2}{\partial x_1^2} + \cdots + \frac{\partial u^2}{\partial x_n^2}$$

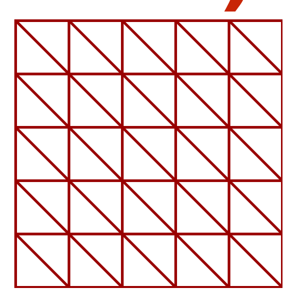
**GRID**

$h$

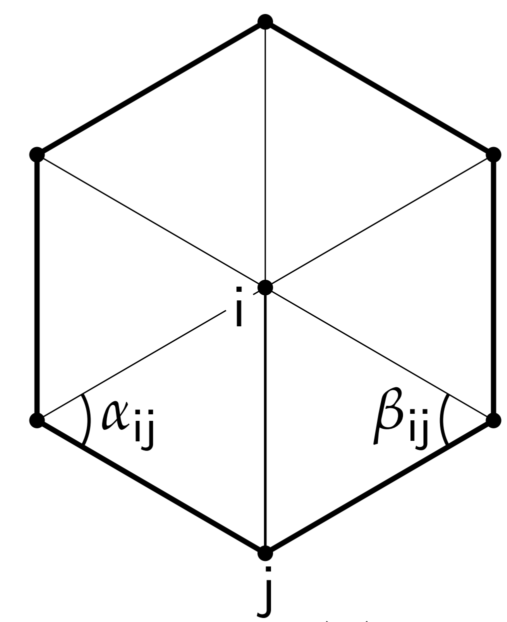
| | | |
|---|---|---|
| | 1 | |
| 1 | -4 | 1 |
| | 1 | |

$$\frac{4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}}{h^2}$$

**(actually, this becomes that)**



**TRIANGLE**




$$\frac{1}{2} \sum_j (\cot \alpha_{ij} + \cot \beta_{ij})(u_j - u_i)$$

# Finite differences

High-school reminder: definition of a derivative using forward difference

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Alternative: use central difference

$$f'(x) = \lim_{h \to 0} \frac{f(x+0.5h) - f(x-0.5h)}{h}$$

For discrete signals: Remove limit and set h = 2

$$f'(x) = \frac{f(x+1) - f(x-1)}{2}$$

| | | | |
|---|---|---|---|
| -1 | 0 | 1 | ? |

| | | | |
|---|---|---|---|
| 1 | 0 | -1 | ? |

1D derivative filter

| | | |
|---|---|---|
| 1 | 0 | -1 |

# Laplace filter

Basically a second derivative filter.

- We can use finite differences to derive it, as with first derivative filter.

first-order finite difference

$$f'(x) = \lim_{h \to 0} \frac{f(x+0.5h) - f(x-0.5h)}{h} \longrightarrow$$

1D derivative filter

| | | |
|---|---|---|
| 1 | 0 | -1 |

second-order finite difference

$$f''(x) = \lim_{h \to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \longrightarrow$$

Laplace filter

?

# Physical Simulation

Basically a second derivative filter.

- We can use finite differences to derive it, as with first derivative filter.

first-order finite difference

$$f'(x) = \lim_{h \to 0} \frac{f(x + 0.5h) - f(x - 0.5h)}{h} \longrightarrow$$

1D derivative filter

| | | |
|---|---|---|
| 1 | 0 | -1 |

second-order finite difference

$$f''(x) = \lim_{h \to 0} \frac{f(x + h) - 2f(x) + f(x - h)}{h^2} \longrightarrow$$

Laplace filter

| | | |
|---|---|---|
| 1 | -2 | 1 |

# Physical Simulation

second-order finite difference

$$f''(x) = \lim_{h \to 0} \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \longrightarrow$$

Laplace filter

| | | |
|---|---|---|
| 1 | -2 | 1 |

**GRID**

$h$

| | | |
|---|---|---|
| | 1 | |
| 1 | -4 | 1 |
| | 1 | |

$$\frac{4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}}{h^2}$$

$$\Delta u = \frac{\partial u^2}{\partial x_1^2} + \cdots + \frac{\partial u^2}{\partial x_n^2}$$

# Solving the Wave Equati

- **Finally, wave equation:**

$$\ddot{u} = \Delta u$$

  - **Not much different; now have 2nd derivative in time**
  - **By now we've learned two different techniques:**
    - **Convert to two 1st order (in time) equations:**

$$\dot{u} = v, \qquad \dot{v} = \Delta u$$

$$u'' = \Delta u - \lambda u'$$

# Forward Euler

- **Simplest scheme: evaluate velocity at current configuration**
- **New configuration can then be written *explicitly* in terms of known data:**

new configuration, current configuration, velocity at current time

$$q_{k+1} = q_k + \tau f(q_k)$$

# Forward Euler

- **Simplest scheme: evaluate velocity at current configuration**
- **New configuration can then be written *explicitly* in terms of known data:**

$$\underbrace{q_{k+1}}_{\text{new configuration}} = \underbrace{q_k}_{\text{current configuration}} + \tau \underbrace{f(q_k)}_{\text{velocity at current time}}$$

$$u'' = \Delta u - \lambda u'$$