CS 174C Week 7

Aidan Jan

February 24, 2024

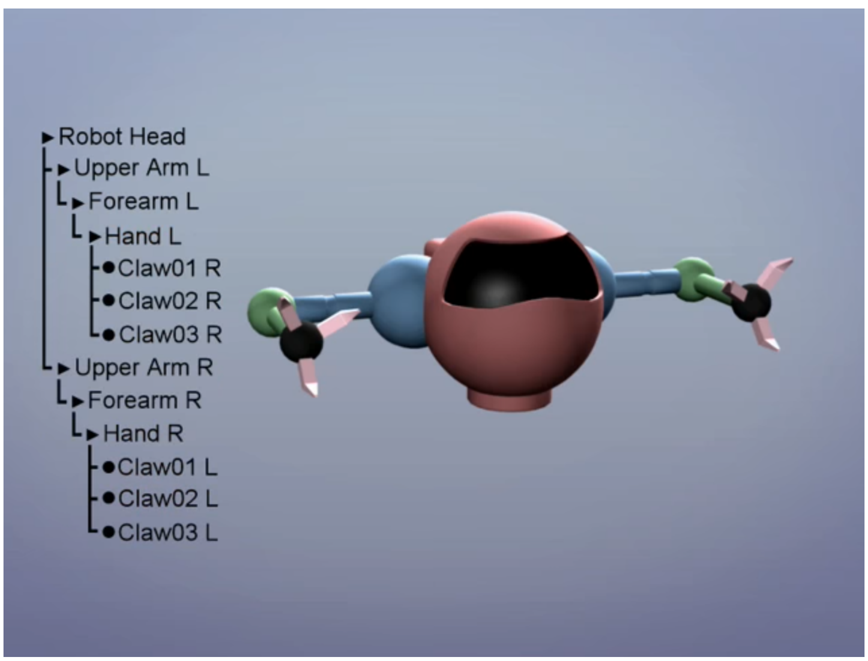
Articulated-Body Kinematics

- When animating characters, their movements should look realistic. Joints must bend in the correct directions, and all the parts should be animated together, not separately.

Hierarchies

- Hierarchies are used in order to determine how objects are linked together.
- Each individual object is referred to as a node, and one object is designated as a root node.
- From here, how objects are linked can be drawn as a graph, similar to a file directory.

Example: Building a Robot

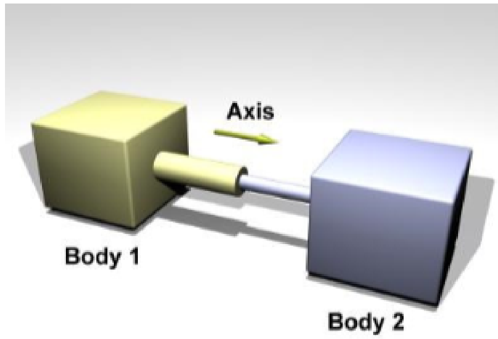


Joints

- A 2-rigid-body system has $2 \cdot 6 = 12$ degrees of freedom (DOF)
- Joints are essentially constraints that remove degrees of freedom
  - Implicitly (through forces)
  - Explicitly (through parameterization)

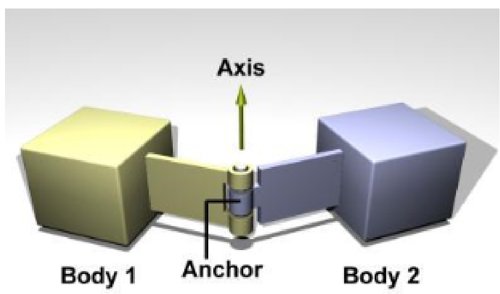
Slider Joints

- 1 Degree of freedom
  - One translational, defined by the axis



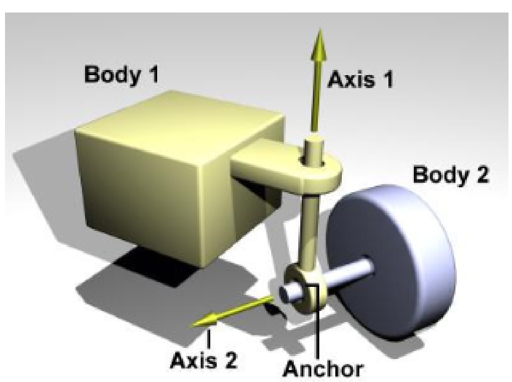
Hinge Joints

- 1 Degree of freedom
  - One rotational, defined by axis and anchor point



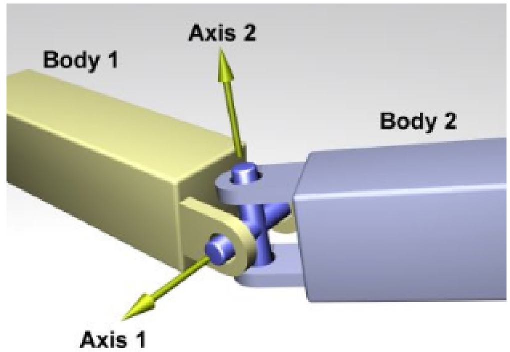
Hinge2 Joints

- 2 Degrees of freedom
  - Two rotational, defined by axis 1, axis 2, and anchor



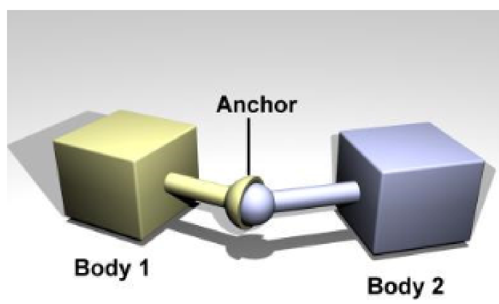
Universal Joints

- 2 Degrees of freedom
  - Two rotational, defined by axis 1, axis 2, and anchor



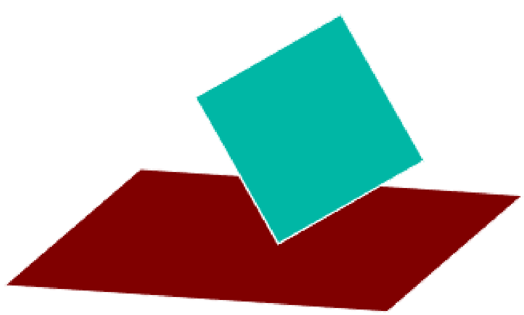
Ball and Socket Joints

- 3 Degrees of freedom
  - Three rotational, defined by anchor point
  - Usually represented as a quaternion or exponential map



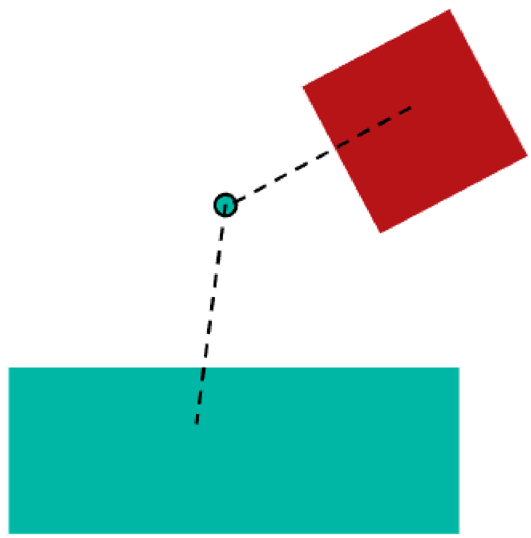
Planar Joints

- Point confined to move on a plane
  - Can be used to model non-penetration constraints

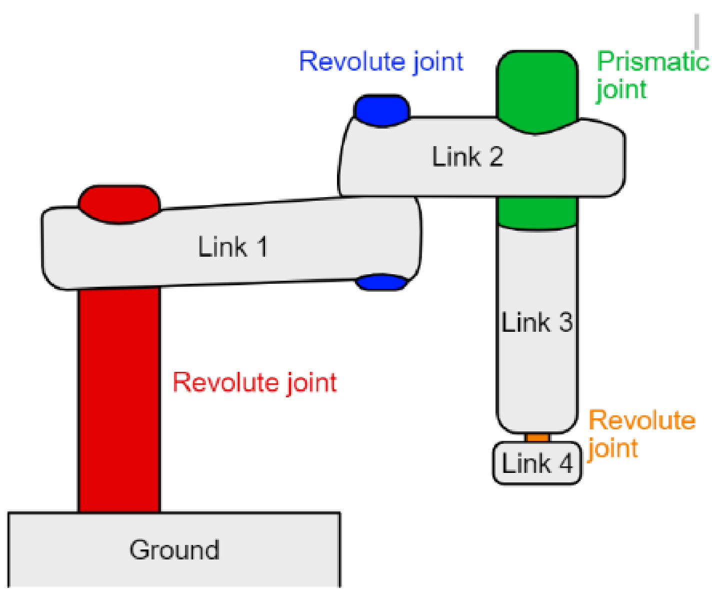


Free Joints

- 6 Degrees of freedom
  - Three translational
  - Three rotational
- Example: Free joint between root object and ground



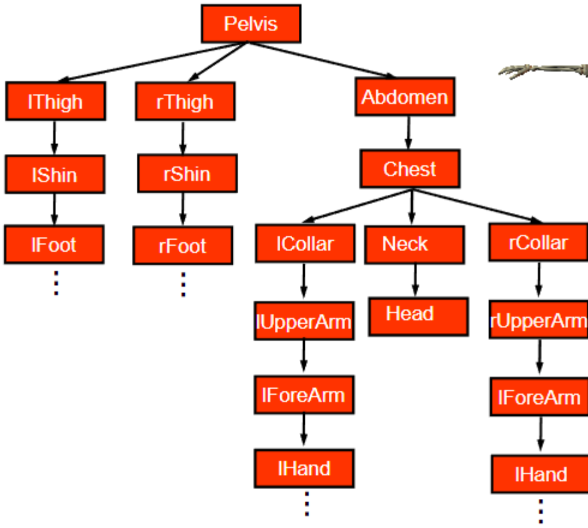
Example - SCARA Robotic Arm

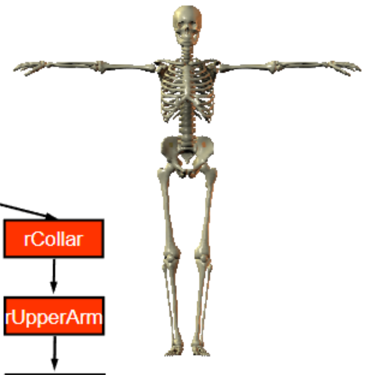


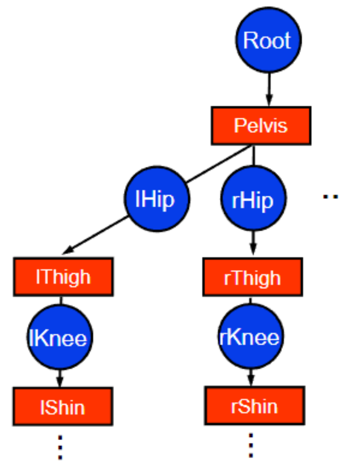
Parametric Representation of a Human Character

- Local Coordinate systems
  - Frames
- Child links can move with respect to their parent links

– Links are jointed by transformation matrices



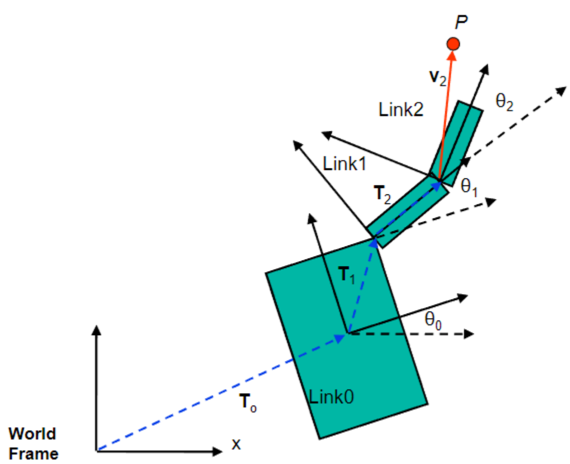




Forward Kinematics (FK) - Determine coordinates of a point P

In 2D: From Link 2 Frame to the "World Frame"

$$
\begin{aligned}
v_w &= T_0 R(\theta_0) v_0 \\
v_w &= T_0 R(\theta_0) T_1 R(\theta_1) v_1 \\
v_w &= T_0 R(\theta_0) T_1 R(\theta_1) T_2 R(\theta_2) v_2
\end{aligned}
$$



In 3D: 4x4 Rotation matrices:

- $R_i = R(\theta_x, \theta_y, \theta_z)$ or $R_i = F(quaternion)$

$$
\begin{aligned}
v_w &= T_0 R_0 v_0 \\
v_w &= T_0 R_0 T_1 R_1 v_1 \\
v_w &= T_0 R_0 T_1 R_1 T_2 R_2 v_2
\end{aligned}
$$

Let ${}_{i-1}M_i = T_i R_i$. Then, $v_w = {}_w M_0 \, {}_0 M_1 \, {}_1 M_2 v_2$.

In general, for a chain of n links:

$$P_{i-1} =_{i-1} M_i P_i \qquad\qquad P_i = {}_{i-1}M_i^{-1} P_{i-1}$$

$$P_w = \left(\prod_{i=0}^{n} {}_{i-1}M_i \right) P_n \qquad\qquad P_n = \left(\prod_{i=0}^{n} {}_{i-1}M_i \right)^{-1} P_w$$

where $i - 1 = w$ for $i = 0$.
If there is no scaling and shearing:

$$M = \begin{bmatrix} R_{3\times 3} & T_{3\times 1} \\ 0_{1\times 3} & 1 \end{bmatrix}$$

Articulated Model Data Structures

- Node:
  - dataPtr: Data (possibly shared by other nodes) that represent the geometry of this part of the figure
  - Tmatrix: Matrix to transform the node data into position to be articulated (e.g., put the point of rotation at the origin)
  - arcPtr: Pointer to a single child Arc
- Arc
  - nodePtr: Pointer to a node holding data to be articulated by the arcPtr
  - Lmatrix: Matrix that locates the following (child) node relative to the previous (parent) node
  - Amatrix: Matrix that articulates the node data; this is the matrix that is changed in order to (animate) articulate the linkage
  - arcPtr: Pointer to a sibling arc (another child of this arc's parent node; this is NULL if there are no more siblings)

Evaluation of an Articulated Model

- By a depth-first traversal from root to leaf nodes of the model hierarchy tree
  - Traverse from root node to leaf nodes
  - Backtrack up the tree until unexplored arc
  - Traversing arc down: Concatenate transform to that of parent node
  - Traversing arc up: Restore transform
- Implemented as a stack of transformations with push (down) and pop (up) operations

Example code:

```
traverse(arcPtr, matrix) {
  ; Get transformations of arc and concatenate arc matrices
  matrix = matrix*arcPtr->Lmatrix     ; concatenate location
  matrix = matrix*arcPtr->Amatrix     ; concatenate articulation
  ; Process data at node
  nodePtr = arcPtr->nodePtr           ; get the node of the arc
  push(matrix)                        ; save the matrix
  matrix = matrix * nodePtr->matrix   ; ready for articulation
  articulatedData = transformData(matrix, dataPtr) ; articulate the data
  draw(articulatedData)               ; and draw it
  matrix = pop                        ; restore matrix for children
  ; Process node's children
  if (nodePtr->arcPtr != NULL) {      ; if not a terminal node
    nextArcPtr = nodePtr->arcPtr      ; get first arc emanating from node
    while (nextArcPtr != NULL) {      ; while there's an arc to process
      Push(matrix)                    ; save matrix at node
      traverse(nextArcPtr, matrix)    ; traverse arc
      matrix = pop()                  ; restore matrix at node
      nextArcPtr = nextArcPtr->arcPtr ; set next child of node
    }
  }
}
```

Inverse Kinematics (IK)

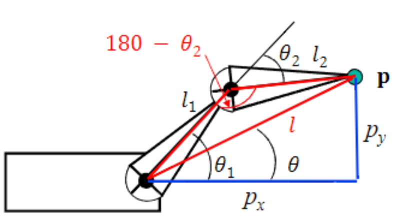
Given the DOFs, e.g., $q = [T \ \ R \ \ \theta]$, compute the position of any point of interest (e.g., the end effector)

- $x = f(q)$
- `traverse(rootArcPtr, I);`, where `I` is the identity matrix

A Simple (Analytic) Example

Direction IK solution, given $p = (p_x, p_y)$

- Solve for $\theta = (\theta_1, \theta_2)$



$$l = \sqrt{p_x^2 + p_y^2} \qquad \theta = \text{acos}\left(\frac{p_x}{l}\right)$$

$$\cos(\theta_1 - \theta) = \frac{l_1^2 + l^2 - l_2^2}{2 l_1 l} \quad \text{cosine rule}$$

$$\theta_1 = \text{acos}\left(\frac{l_1^2 + l^2 - l_2^2}{2 l_1 l}\right) - \theta$$

$$\cos(\pi - \theta_2) = -\cos\theta_2 = \frac{l_1^2 + l_2^2 - l^2}{2 l_1 l_2}$$

$$\theta_2 = \text{acos}\left(\frac{l_1^2 + l_2^2 - l^2}{2 l_1 l_2}\right)$$

Problems:

- Multiple Solutions
- Unreachable goals
- Most structures are too complex to solve analytically

Aside: The Jacobian

- Derivative of a one-variable scalar function

$$y = f(x) \to \frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{x \to 0} \frac{\Delta y}{\Delta x} = \frac{\delta y}{\delta x} \to \delta y = \frac{\partial f}{\partial x} \delta x$$

- Extension to multivariable vector functions

$$y = F(x) = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} f_1(x_1, x_2, x_3, x_4) \\ f_2(x_1, x_2, x_3, x_4) \\ f_3(x_1, x_2, x_3, x_4) \end{pmatrix}$$

$$\delta y_i = \frac{\partial f_i}{\partial x_1} \delta x_1 + \frac{\partial f_i}{\partial x_2} \delta x_2 + \frac{\partial f_i}{\partial x_3} \delta x_3 + \frac{\partial f_i}{\partial x_4} \delta x_4, \quad i = 1, 2, 3$$

- Jacobian Matrix

$$J = \frac{\partial F}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \frac{\partial f_1}{\partial x_4} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \frac{\partial f_2}{\partial x_4} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \frac{\partial f_3}{\partial x_4} \end{bmatrix}$$

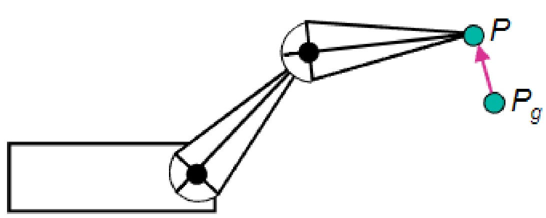
- This gives a linear mapping between instantaneous velocities
- At any instant in time, the Jacobian provides a linear mapping between the velocities in the neighborhood of x: $y' = J(x) \cdot x'$
- The Jacobian is a function of x.
- It enables us to linearize $y(x)$ with respect to x around the neighborhood of x.

$$\delta y = J(x) \delta x \Rightarrow \Delta y \approx J \Delta x$$

$$y(x) \approx y(x_0) + J(x_0)(x - x_0)$$

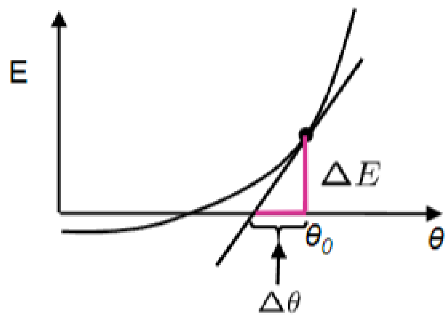
Back to Inverse Kinematics

- Move end effector to a given goal position P_g
- Error vector: $E = P - P_g$



- We want the error $E(\theta) = P - P_g$ to be 0.

Newton's Method



$$\frac{\partial E}{\partial \theta} = \frac{\Delta E}{\Delta \theta}$$

$$\Delta\theta = \left(\frac{\partial E}{\partial \theta}\right)^{-1} \Delta E$$

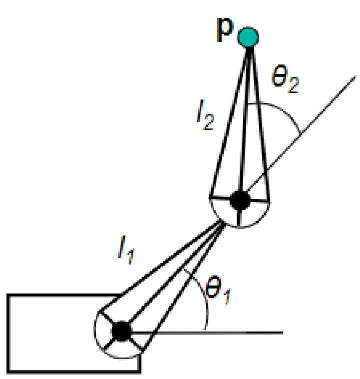
$$\theta' = \theta - \Delta\theta$$

$$\frac{\partial E}{\partial \theta} = \frac{\partial P}{\partial \theta}$$

For the Simple (Analytic) Example

Two-link planar arm:

$$p_x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$
$$p_y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$



$$\frac{\partial p_x}{\partial \theta_1} = -l_1 \sin(\theta_1) - l_2 \sin(\theta_1 + \theta_2)$$
$$\frac{\partial p_y}{\partial \theta_1} = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$
$$\frac{\partial p_x}{\partial \theta_2} = -l_2 \sin(\theta_1 + \theta_2)$$
$$\frac{\partial p_y}{\partial \theta_2} = l_2 \cos(\theta_1 + \theta_2)$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_1} & \frac{\partial p_x}{\partial \theta_2} \\ \frac{\partial p_y}{\partial \theta_1} & \frac{\partial p_y}{\partial \theta_2} \end{bmatrix}$$

The More General Formulation

- End effector position and orientation
  - $x = f(\theta)$ where:
    - $x = [p_x, p_y, p_z, \theta_x, \theta_y, \theta_z]^T$
    - $\theta = [\theta_1, \ldots, \theta_n]^T$

- End effector velocity
  - $x' = [v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]$
- Joint velocities
  - $\theta' = [\theta'_1, \ldots, \theta'_n]^T$
- Velocity relationship
  - $x' = J(\theta)\theta'$, where $J = \frac{\partial f}{\partial x}$ is a $6 \times n$ matrix.

Inverting the Jacobian

For inverse kinematics, we ideally need $\theta' = J^{-1}(\theta)x'$. However, for m functions and n DOFs, $J_{m \times n}$ is not square

- J^{-1} is not defined
- We use the pseudoinverse J^+

For full rank matrices:

- $m > n$: $J^{+}=(J^T J)^{-1} J^T$ Overconstrained, minimizes $\|J\theta' - x'\|$
- $m < n$: $J^+ = J^T(JJ^T)^{-1}$ Underconstrained, minimuzes $\|\theta'\|$
  - For rank deficient matrices, use the SVD or other methods

Secondary Tasks

- Obstacle Avoidance
- Joint limit constraints
- Singularity Avoidance
  - Pseudoinverse is unstable around singularities: $\det(J) = 0$

Adding Control

How can we bias the angular velocities without affecting the end effector velocity?

- Consider the control expression
$$\theta' = (J^+J - I)z$$
- Since $x' = J\theta'$
$$x' = J(J^+J - I)z = (JJ^+J - J)z = (J - J)z = 0$$
- Therefore, z does not change the end effector velocity

What is z?

- Let θ_i^c be preferred angles at each joint
$$H = \frac{1}{2}\sum_{i=1}^{n} \alpha_i(\theta_i - \theta_i^c)^2$$
$$z = \nabla_\theta H = [\alpha_1(\theta_1 - \theta_1^c), \ldots, \alpha_n(\theta_n - \theta_n^c)]^T$$
- Where α_i are the gains

How is z used?

- System

$$\theta' = J^+x' + (J^+J - I)\nabla_\theta H \cdots$$
$$\theta' = J^T[(JJ^T)^{-1}(x' + J\nabla_\theta H)] - \nabla_\theta H$$

- Set

$$\beta = (JJ^T)^{-1}(x' + J\nabla_\theta H)$$

- Solve for β
- Then, controlled angle velocities are:

$$\theta' = J^T\beta - \nabla_\theta H$$
$$x' = (JJ^T)\beta - J\nabla_\theta H$$

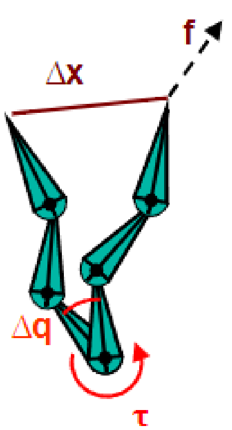
Jacobian Transpose Method

- Use the transpose of the Jacobian matrix rather than the pseudoinverse
  - Rather than: $\Delta\theta = J^+\Delta x$
  - Find $\Delta\theta$ by: $\Delta\theta = J^T\Delta x$
- Avoids expensive inversion
- Avoids singularity problems

But why does it work?

Principal of Virtual Work

- Virtual because amount is infinitesimal
- Work = force $\times$ distance
- Work = torque $\times$ angle



$\mathbf{f} \cdot \Delta\mathbf{x} = \boldsymbol{\tau} \cdot \Delta\boldsymbol{\theta}$ *(energy equal in either coordinates)*

$\mathbf{f}^T\Delta\mathbf{x} = \boldsymbol{\tau}^T\Delta\boldsymbol{\theta}$

$\Delta\mathbf{x} = \mathbf{J}\Delta\boldsymbol{\theta}$ **(forward kinematics)**

$\mathbf{f}^T\mathbf{J}\Delta\boldsymbol{\theta} = \boldsymbol{\tau}^T\Delta\boldsymbol{\theta}$ **(substitution)**

$\mathbf{f}^T\mathbf{J} = \boldsymbol{\tau}^T$

$\mathbf{J}^T\mathbf{f} = \boldsymbol{\tau}$ **(transpose both sides)**

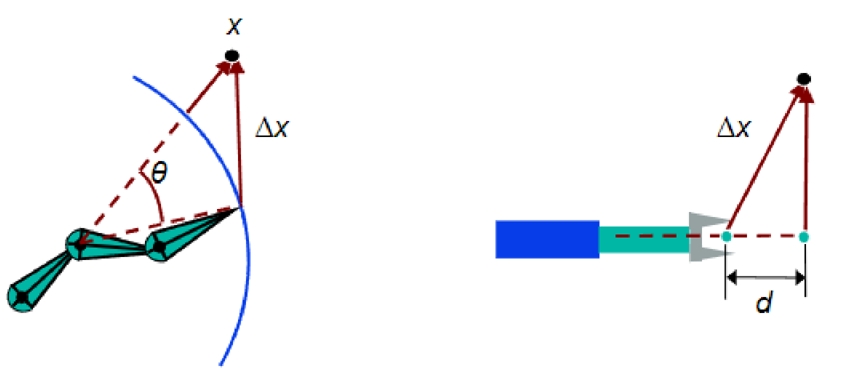
Jacobian Transpose Method

- The good and bad of J^T
- Pros:
  - Cheaper evaluation step than computing pseudoinverse
  - No singularities
- Cons:
  - Slower to converge than J^+
  - Scaling problems
    * J' has nice property that the solution has minimal norm at every step
    * J^T doesn't have this property. Joints far from the end effector experience larger torques, hence take larger steps
    * Can introduce a constant diagonal scaling matrix to counteract some scaling problems: $\mathrm{d}\theta/\mathrm{d}t = KJ^TF(\theta)$

Cyclic Coordinate Descent

A simple idea:

- Solve 1-DOF IK problems repeatedly up the chain
- 1-DOF problems are simple and have analytical solutions



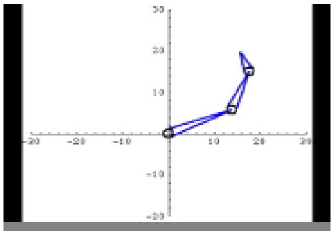
Rotational joint: Find θ that minimizes Δx for joint i

Translational joint: Find d that minimizes Δx for joint i

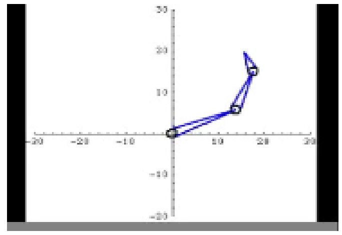
The good and bad of CCD:

- Pros:
  - Simple to implement
  - Often effective
  - Stable around singular configuration
  - Computationally cheap
  - Can combine withother more accurate optimization methods (such as Broyden-Fletcher-Shanno (BFS) when close enough)
- Cons:
  - Can lead to odd solutions if per-step deltas are not limited, making the method slow to converge
  - Doesn't necessarily lead to smooth motion

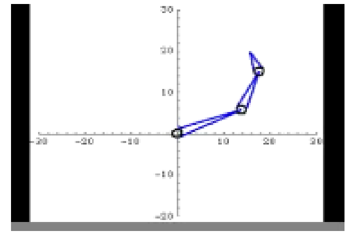
Comparison of the IK Methods



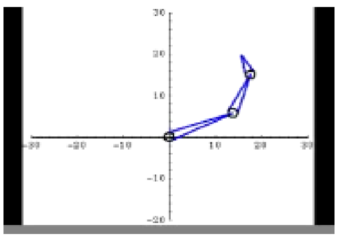
Jacobian



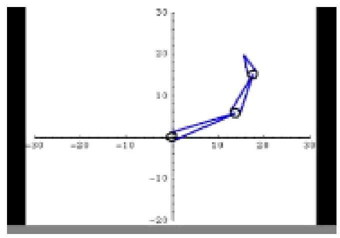
Damped Jacobian



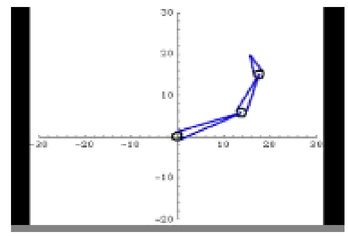
Controlled Jacobian



Alternative Jacobian



Jacobian Transpose



Cyclic Coordinate Descent