Junjie Cao @ DLUT

Spring 2018

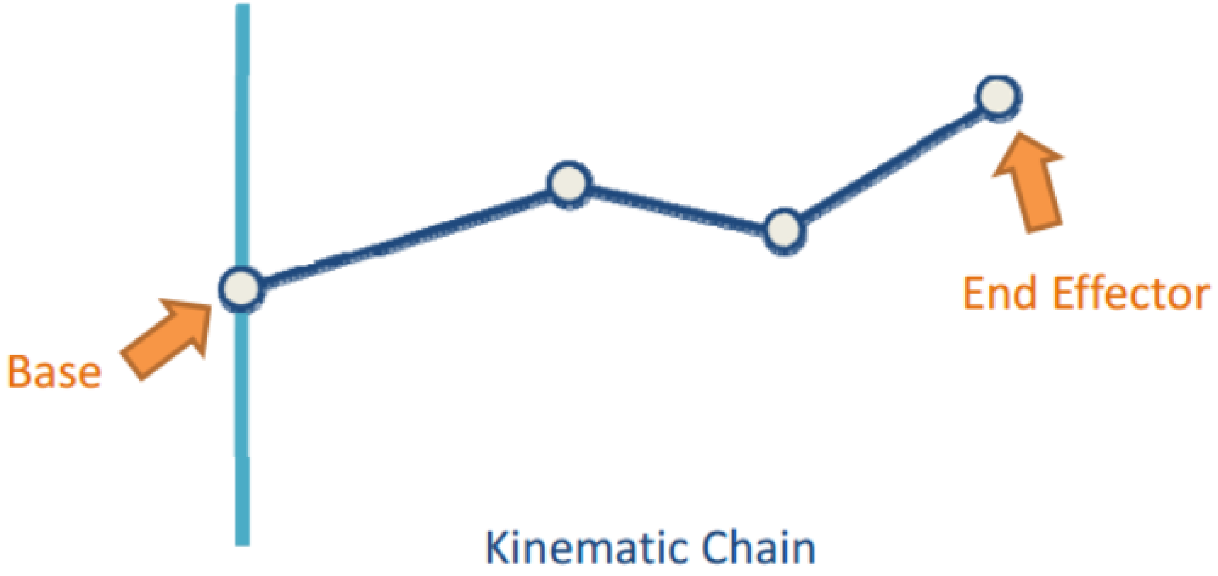
http://jjcao.github.io/ComputerGraphics/

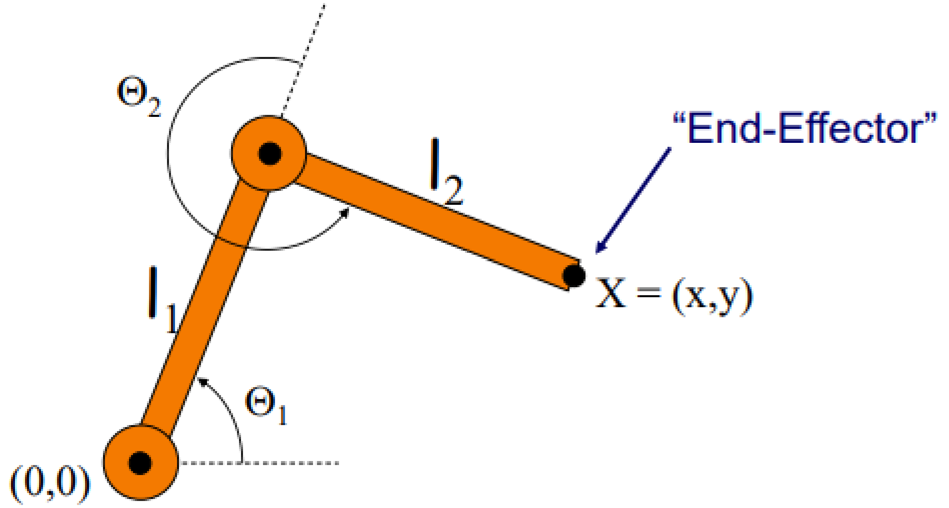
Overview

- Kinematics
- Forward Kinematics and Inverse Kinematics
- Jacobian
- Pseudoinverse of the Jacobian

Vocabulary of Kinematics

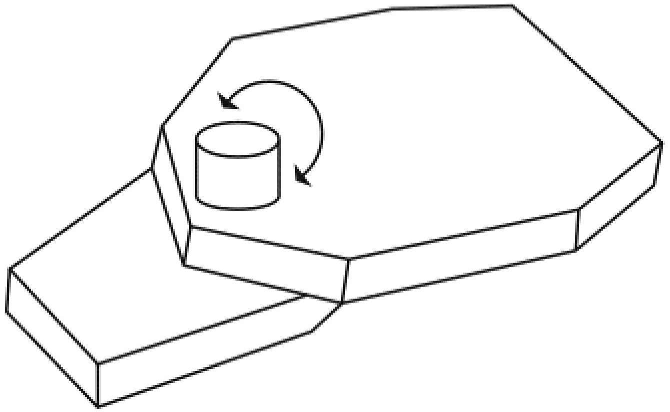
- Kinematics is the study of how things move, it describes the motion of a hierarchical skeleton structure.
- Base and End Effector.
- For today, we will limit our study to **linear kinematic chains**, rather than the more general hierarchies (i.e., stick with individual arms & legs rather than an entire body with multiple branching chains)
- 2-Link Structure: Two links connected by rotational joints



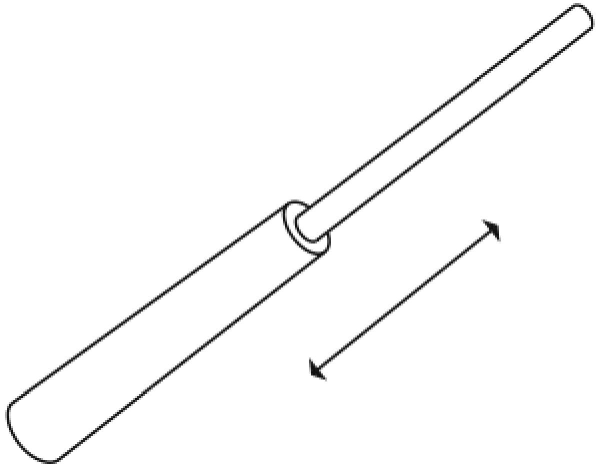


DOF (Degrees of freedom)

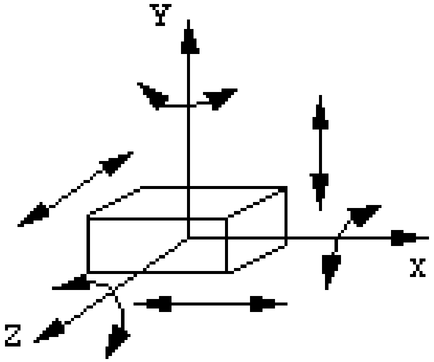
- Every joint allowing motion in one dimension is said to have one degree of freedom (DOF)
- An unrestrained rigid body in space has six degrees of freedom: three translating motions along the *x*, *y* and *z* axes and three rotary motions around the *x*, *y* and *z* axes respectively.
- Joints with 1 DOF

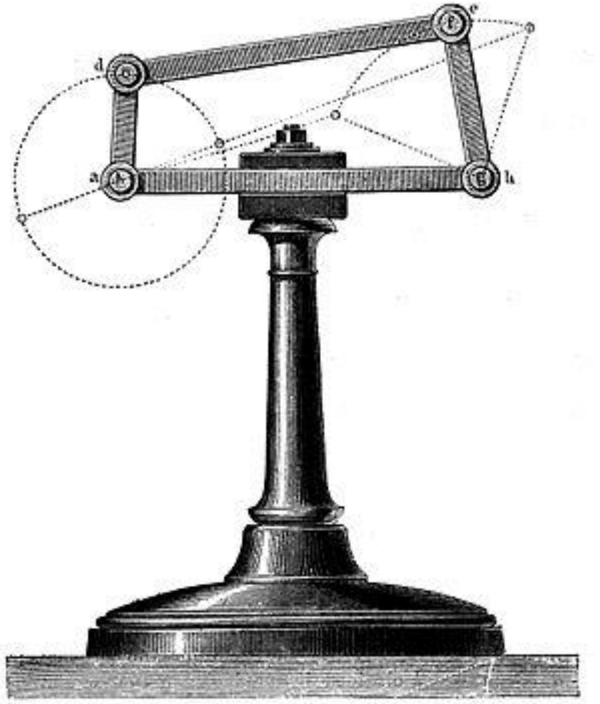


Revolute joint

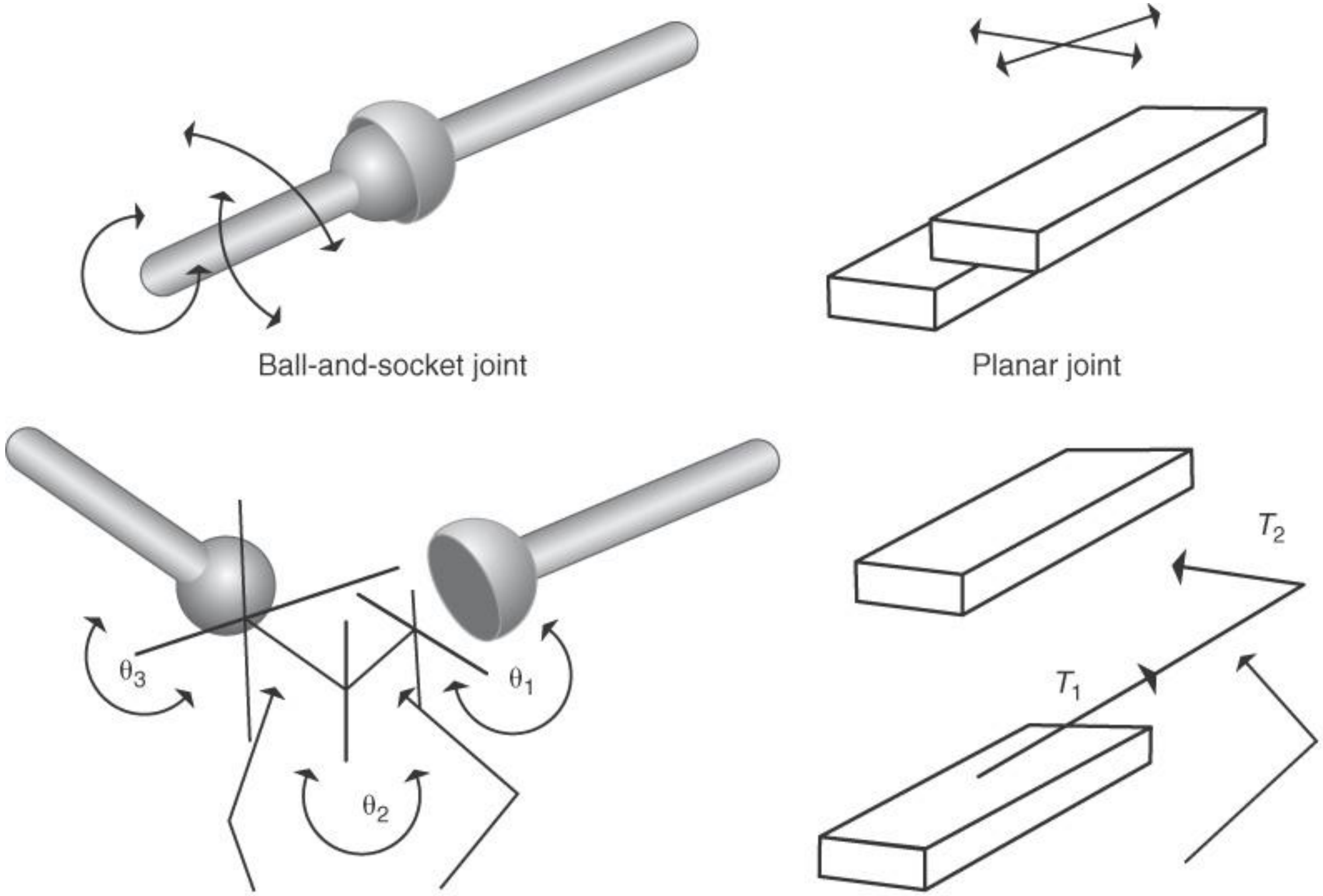


Prismatic joint





DOF



zero-length linkages

zero-length linkage

Ball-and-socket joint modeled as 3 one-degree joints with zero-length links

Planar joint modeled as 2 one-degree prismatic joints with zero-length links

DOF of human joints

- Root: 3 translational DOF + 3 rotational DOF
- Most of the joints are Rotational joints
- Each joints has at most 3 DOF
  - Shoulder: 3 DOF
  - Wrist: 2 DOF
  - Knee: 1 DOF

Overview

- Kinematics
- Forward Kinematics and Inverse Kinematics
- Jacobian
- Pseudoinverse of the Jacobian

Forward Kinematics

- We have joint DOF (Degrees of freedom) values:

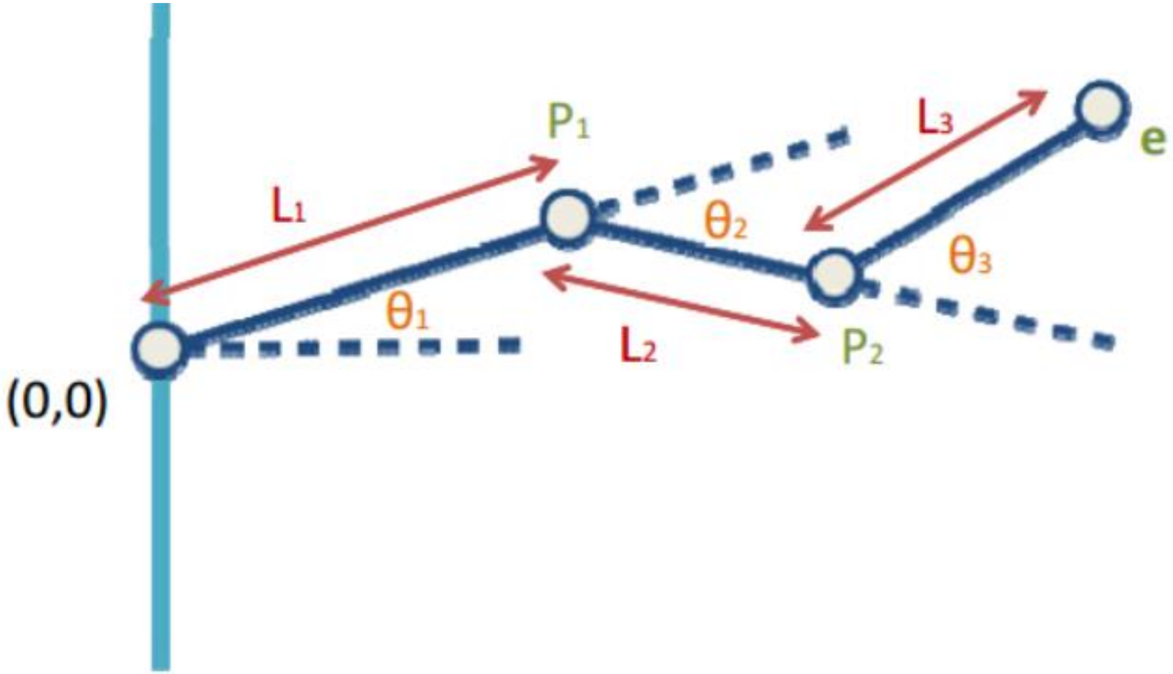
$$\boldsymbol{\theta} = [\theta_1 \ \theta_2 \ \cdots \ \theta_M]$$

- We want the end effector description in world space (N=3 in our case):

$$\mathbf{e} = [e_1 \ e_2 \ \cdots \ e_N]$$

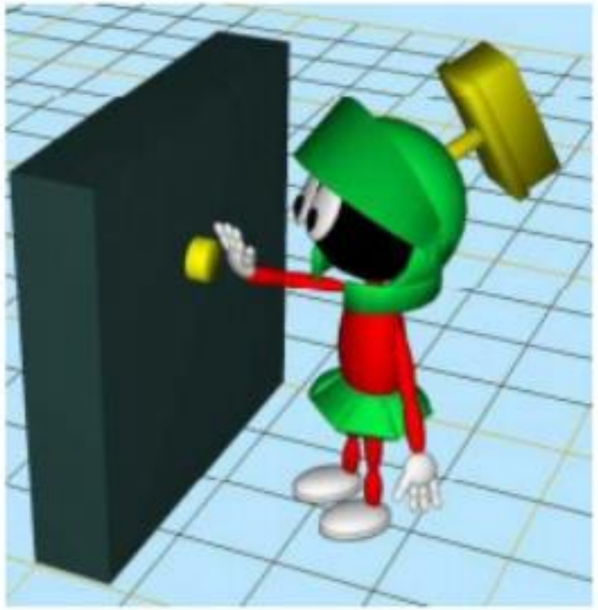
- FK gives us:

$$\mathbf{e} = \mathbf{f}(\boldsymbol{\theta})$$

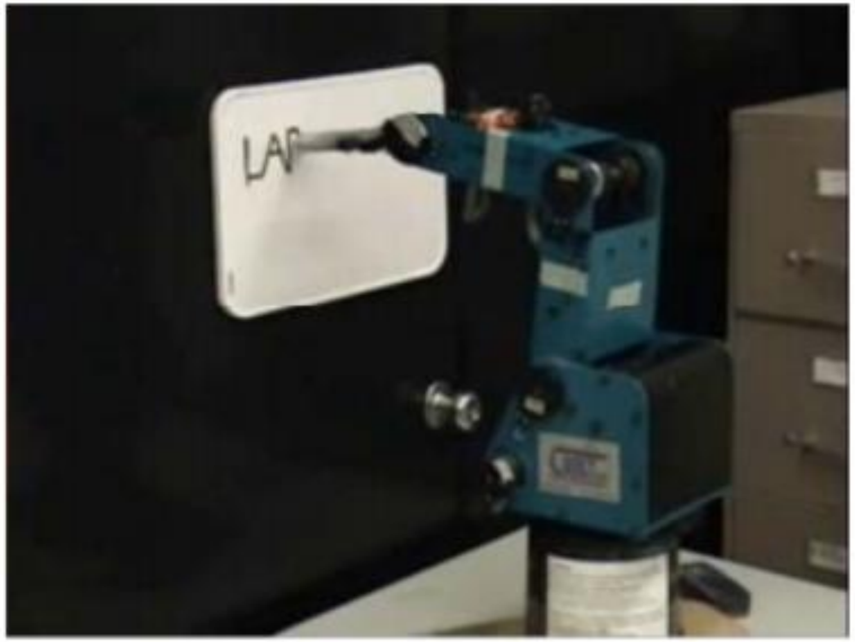


But Sometimes We Want the Opposite

- We want to know how the upper joints of the hierarchy would rotate if we want the end effector to reach some goal.



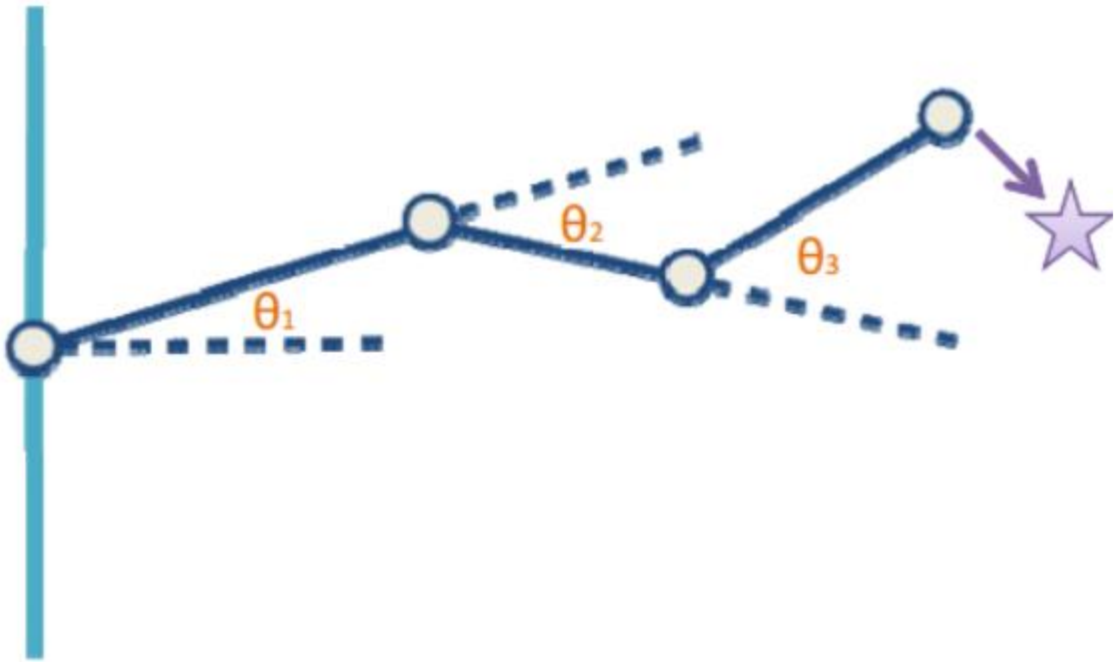
Animations



Robotics

Inverse Kinematics

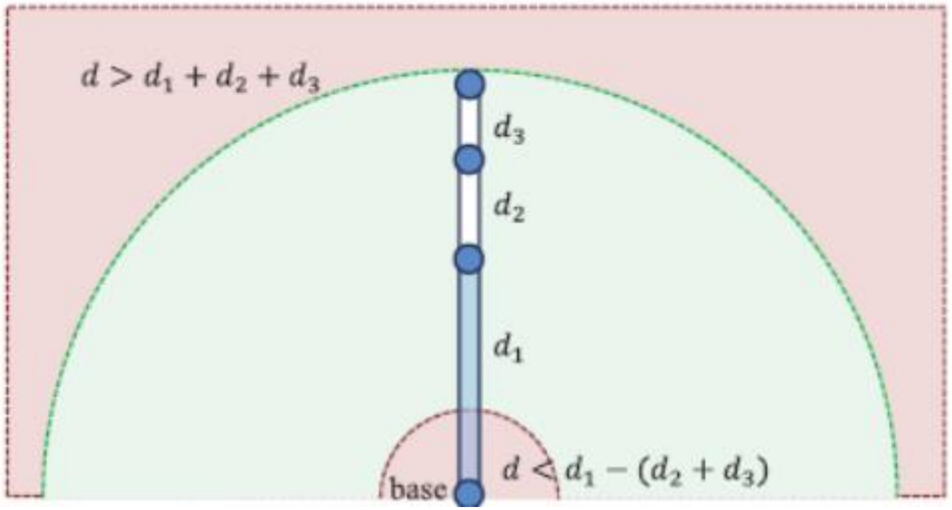
- The goal of inverse kinematics is to compute the vector of joint DOFs that will cause the end effector to reach some desired goal state
- We have: $$\mathbf{e} = [e_1 \ e_2 \ \cdots \ e_N]$$
- And we want: $$\boldsymbol{\theta} = [\theta_1 \ \theta_2 \ \cdots \ \theta_M]$$
- We need: $$\boldsymbol{\theta} = \mathbf{f}^{-1}(\mathbf{e})$$



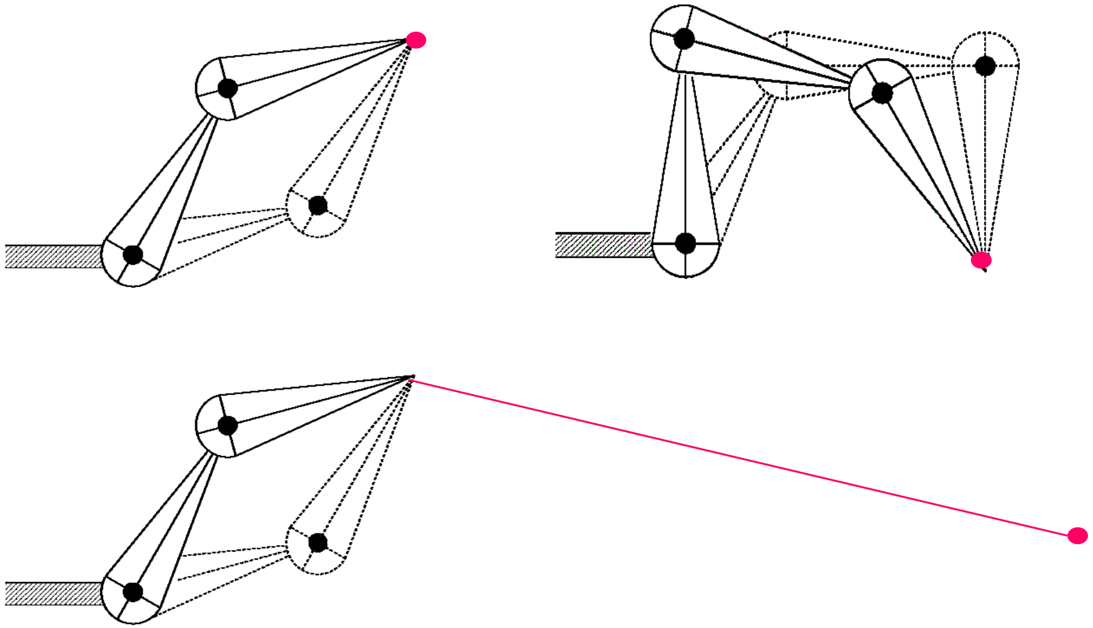
f is a Multivariate nonlinear function

Inverse Kinematics Issues

- While FK is relatively easy to evaluate.
- IK is more challenging: several possible solutions, or sometimes maybe no solutions.



Reachable and unreachable cases



- Require Complex and Expensive computations to find a solution.
- As a result, there are **many different approaches** to solving IK problems

Analytical vs. Numerical Solutions

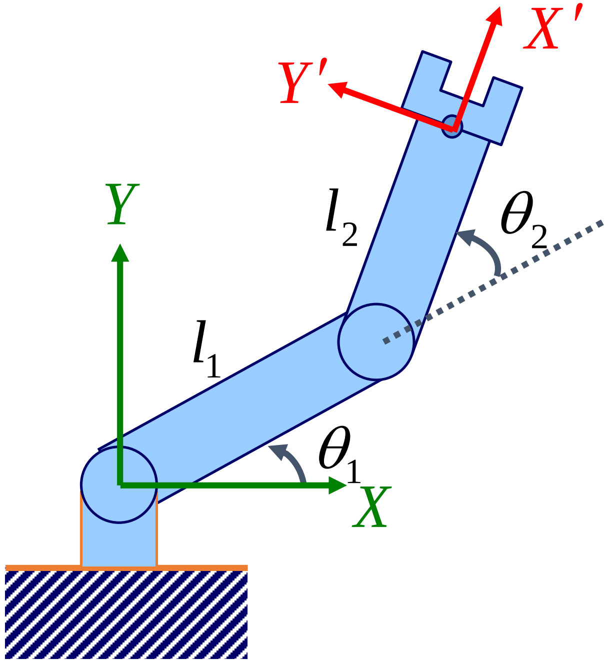
- One major way to classify IK solutions is into **analytical** and **numerical** methods

- Analytical methods attempt to mathematically solve an exact solution by directly inverting the forward kinematics equations. This is only **possible on relatively simple** chains.

- Numerical methods use **approximation and iteration** to converge on a solution. They tend to be more **expensive**, but far more **general** purpose.

Forward Kinematics: A Simple Example

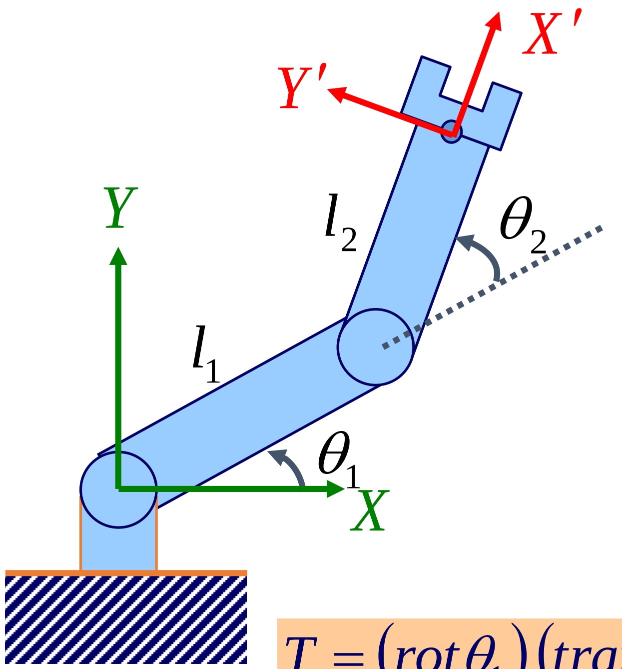
- Forward kinematics map as a coordinate transformation
  - The body local coordinate system of the end-effector was initially coincide with the global coordinate system
  - Forward kinematics map transforms the position and orientation of the end-effector according to joint angles



$$\begin{pmatrix} x_e \\ y_e \\ 1 \end{pmatrix} = \begin{pmatrix} & & \\ & T & \\ & & \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Example from Jehee Lee

A Chain of Transformations



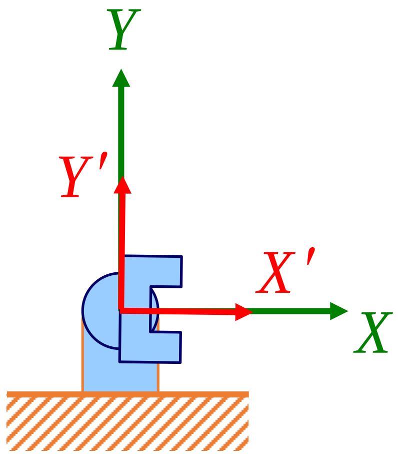
$$\begin{pmatrix} x_e \\ y_e \\ 1 \end{pmatrix} = \begin{pmatrix} & & \\ & T & \\ & & \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$T = (rot\,\theta_1)(transl_1)(rot\,\theta_2)(transl_2)$$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of global coordinate system

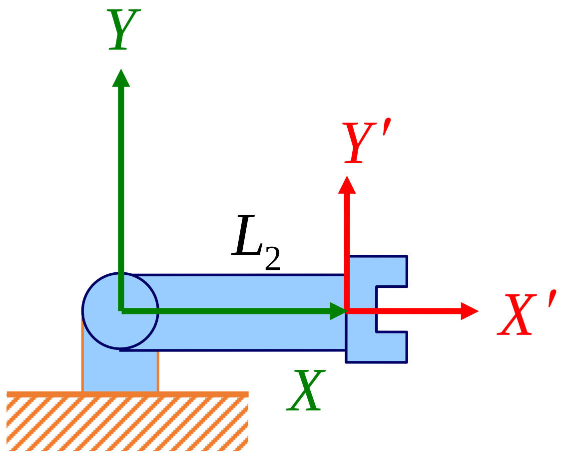


$$T = (rot\,\theta_1)(transl_1)(rot\,\theta_2)(transl_2)$$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of global coordinate system

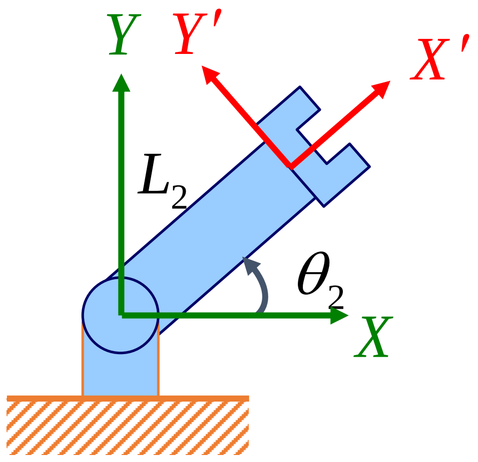


$$T = (rot\,\theta_1)(transl_1)(rot\,\theta_2)\boxed{(transl_2)}$$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \boxed{\begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}$$

Thinking of Transformations

- In a view of global coordinate system

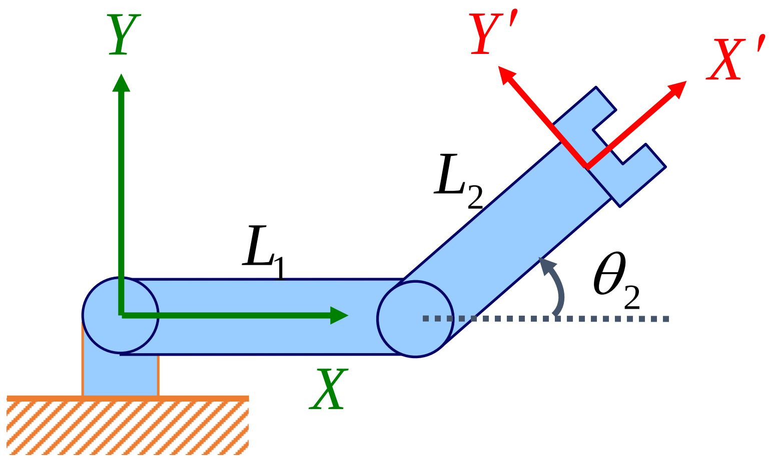


$$T = (rot\,\theta_1)(transl_1)\boxed{(rot\,\theta_2)(transl_2)}$$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \boxed{\begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}$$

Thinking of Transformations

- In a view of global coordinate system

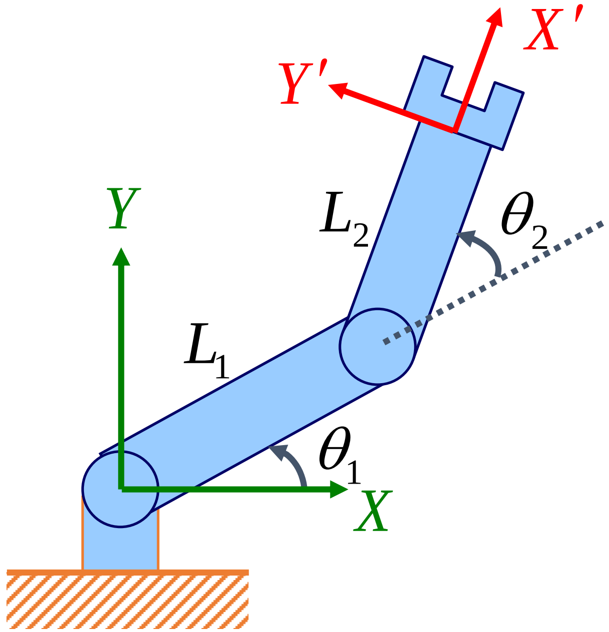


$$T = (rot\,\theta_1)(transl_1)(rot\,\theta_2)(transl_2)$$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of global coordinate system

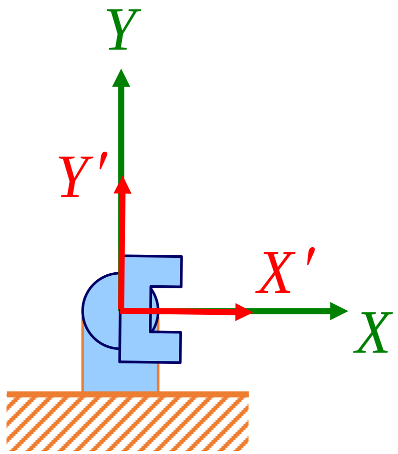


$$T = \boxed{(rot\,\theta_1)(trans\,l_1)(rot\,\theta_2)(trans\,l_2)}$$

$$= \boxed{\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}$$

Thinking of Transformations

- In a view of body-attached coordinate system

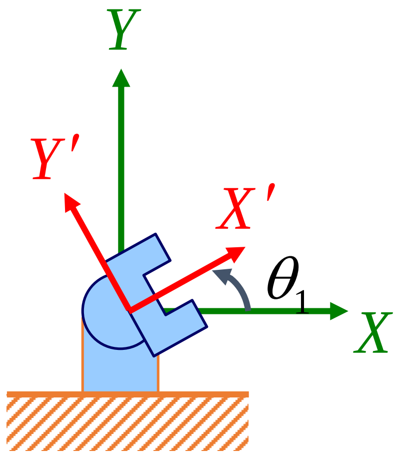


$$T = (rot\,\theta_1)(transl_1)(rot\,\theta_2)(transl_2)$$

$$= \begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of body-attached coordinate system

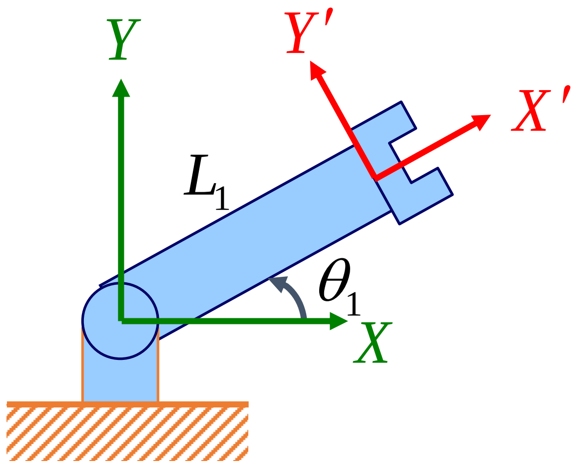


$$T = \boxed{(rot\,\theta_1)}(transl_1)(rot\,\theta_2)(transl_2)$$

$$= \boxed{\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}} \begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of body-attached coordinate system

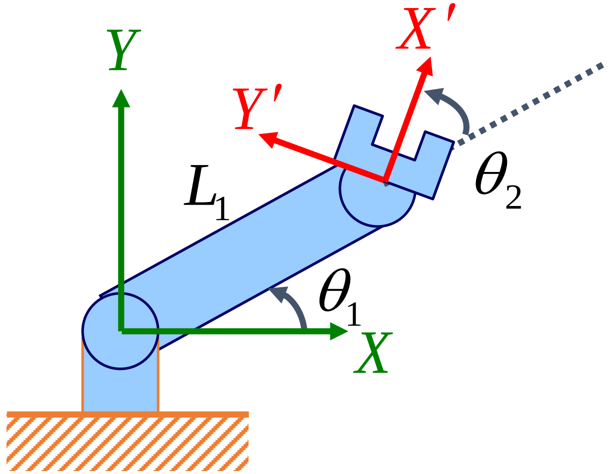


$$T = \boxed{(rot\,\theta_1)(transl_1)}(rot\,\theta_2)(transl_2)$$

$$= \boxed{\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}\begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of body-attached coordinate system

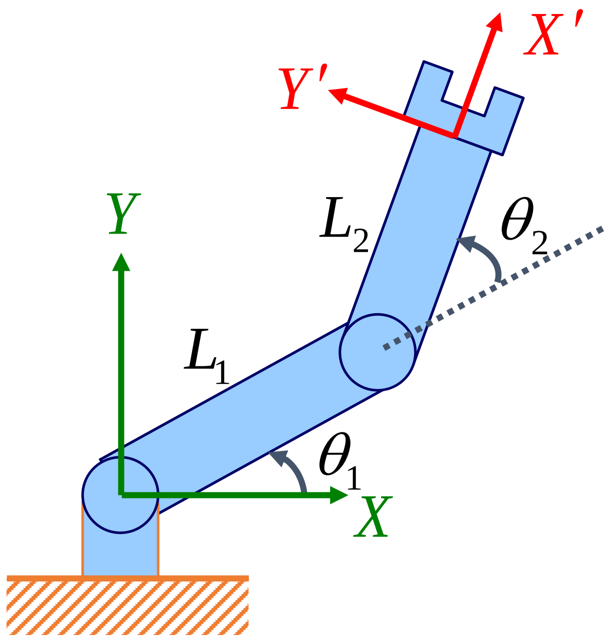


$$T = \boxed{(rot\,\theta_1)(transl_1)(rot\,\theta_2)}(transl_2)$$

$$= \boxed{\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}}\begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thinking of Transformations

- In a view of body-attached coordinate system

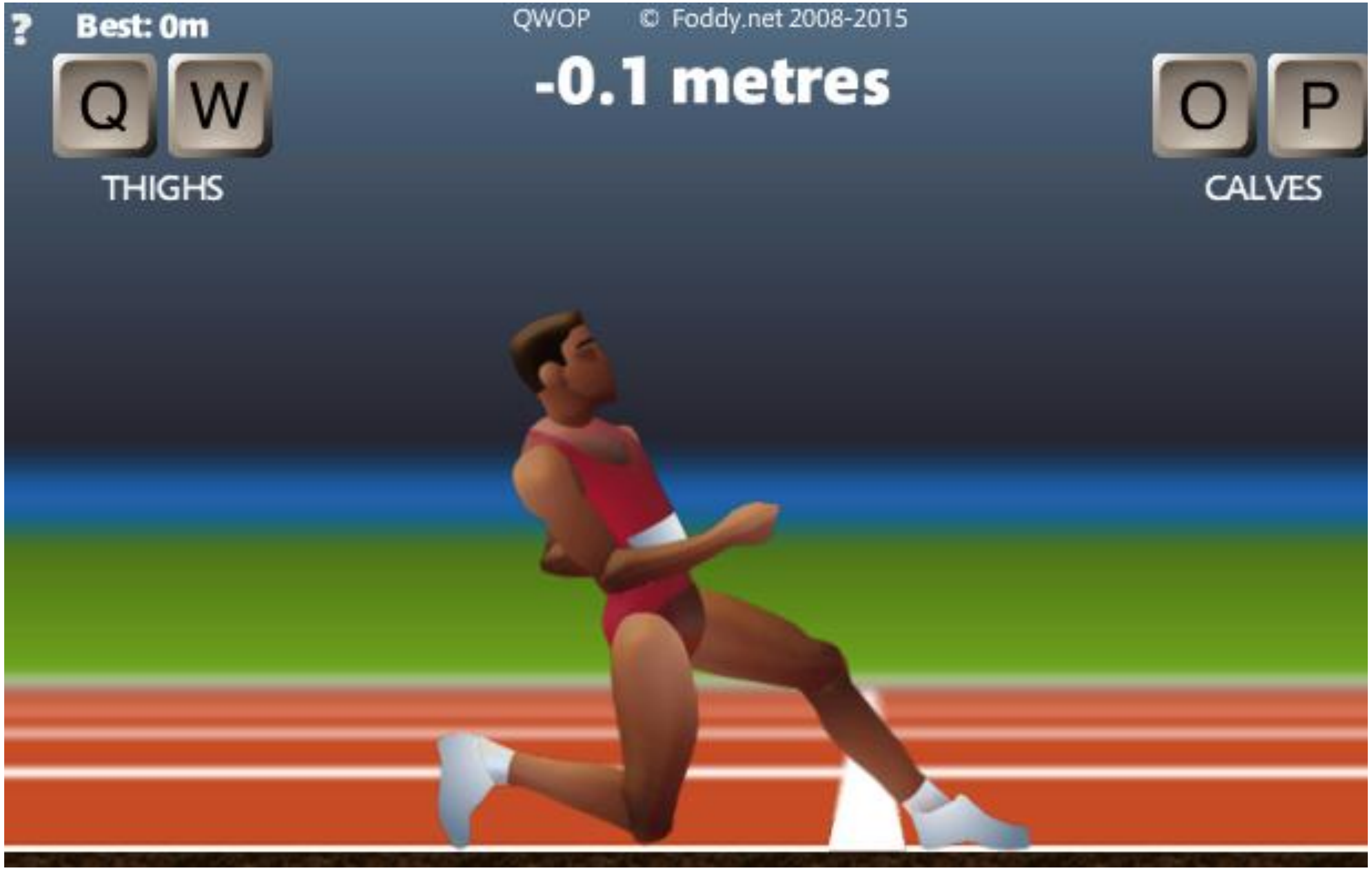


$$T = \boxed{(rot\,\theta_1)(transl_1)(rot\,\theta_2)(transl_2)}$$

$$= \boxed{\begin{pmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} \cos\theta_2 & -\sin\theta_2 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & l_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}$$

Real-time forward kinematics

- QWOP http://www.foddy.net/Athletics.html



Solutions of IK

- Analytic solutions
- Numerical Solutions
  - Jacobian inverse methods
    - **damped least squares (DLS)**
    - ***Selectively damped least squares (SDLS)***
  - Newton methods
  - Heuristic IK algorithms
    - **Cyclic Coordinate Descent (CCD)**
    - ***Forward and backward reaching inverse kinematics (FABRIK)***
- Data-driven methods
- Hybrid solvers
  - Statistical methods
  - Sequential IK

Multivariate nonlinear root finding

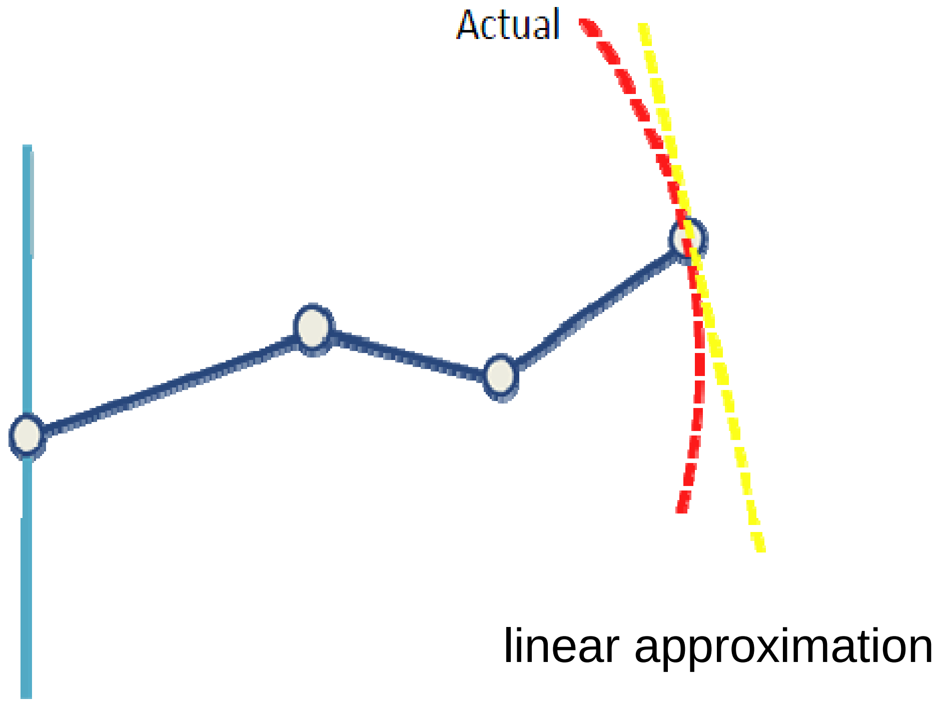
- Want to solve $f(\theta)-X=0$ numerically

- Given: current θ, $f(\theta)$ and target X
- How to find Δ such that $f(\theta+\Delta)=X$
  - Find Δ that gets closer
  - Then $\theta \leftarrow \theta+\Delta$ and repeat

Local Linearization

- Taylor series expansion:
  $f(\theta+\Delta)=f(\theta)+f'(\theta)\,\Delta+f''(\theta)\,\Delta^2/2+\ldots$
- Use first term of Taylor series:
  $\mathbf{f(\theta+\Delta)-}\ \boldsymbol{f(\theta)}\ +\!=\mathbf{J}\boldsymbol{(\theta)\ \Delta}$
- Jacobian matrix:

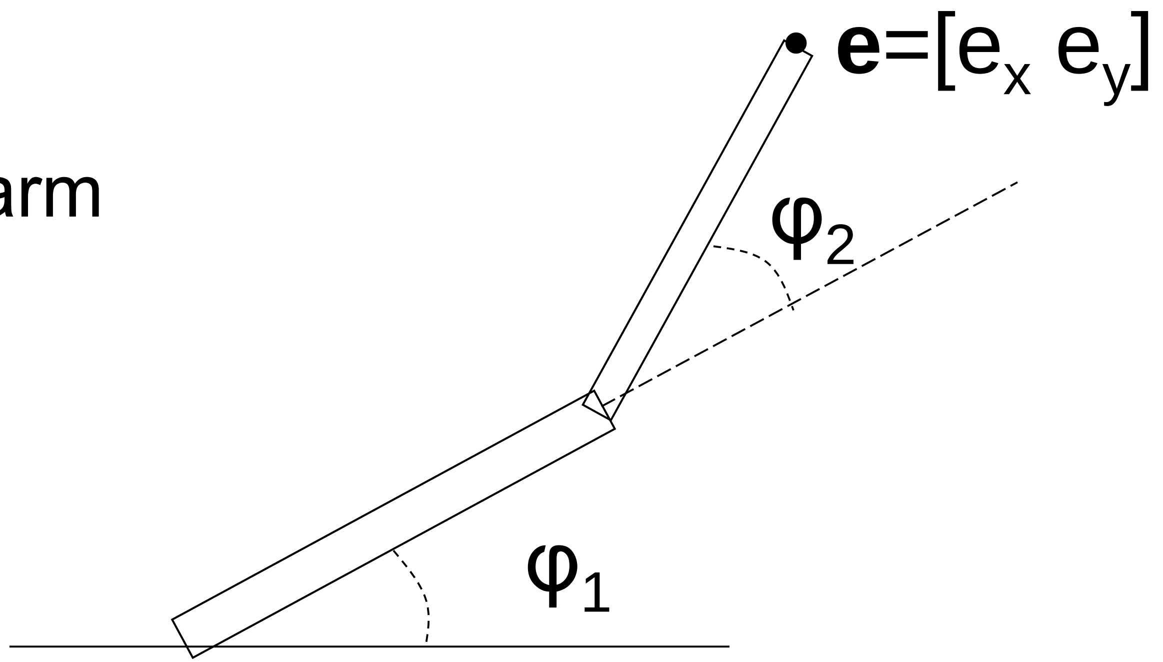
$$J(\mathbf{f},\theta)=\begin{bmatrix} \frac{\partial f_1}{\partial \theta_1} & \frac{\partial f_1}{\partial \theta_2} & \cdots & \frac{\partial f_1}{\partial \theta_N} \\ \frac{\partial f_2}{\partial \theta_1} & \frac{\partial f_2}{\partial \theta_2} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_M}{\partial \theta_1} & \cdots & \cdots & \frac{\partial f_M}{\partial \theta_N} \end{bmatrix}$$



- Matrix of partial derivatives of entire system.

Jacobians

- Let's say we have a simple 2D robot arm with two 1-DOF rotational joints:



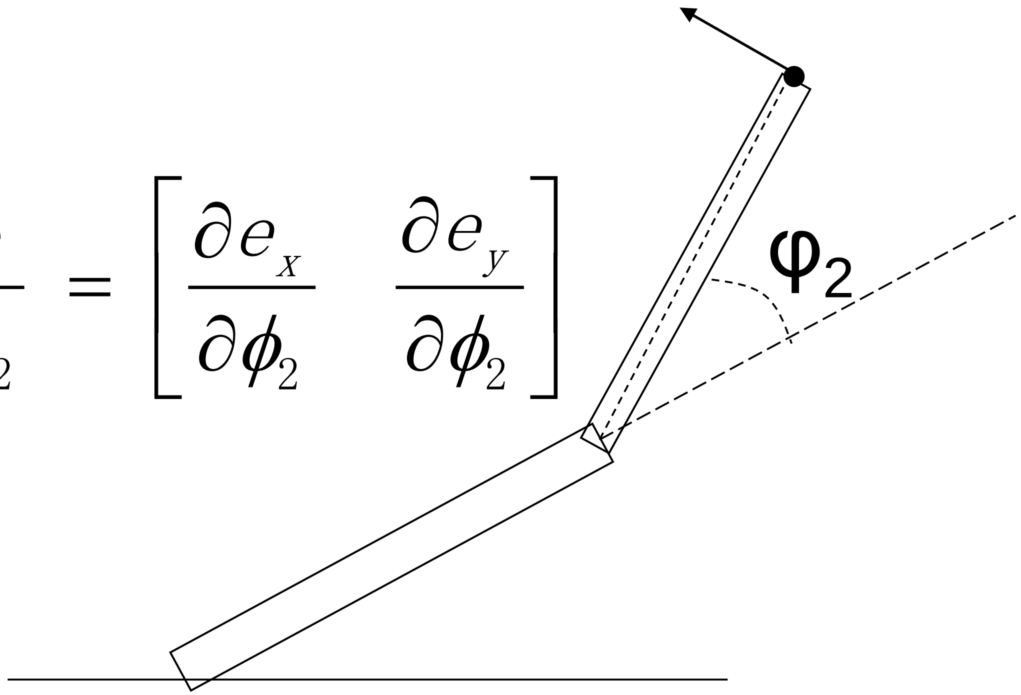
- The Jacobian matrix J(**e**,**Φ**) shows how each component of **e** varies with respect to each joint angle

$$J(\mathrm{e}, \Phi) = \begin{bmatrix} \frac{\partial e_x}{\partial \phi_1} & \frac{\partial e_x}{\partial \phi_2} \\ \frac{\partial e_y}{\partial \phi_1} & \frac{\partial e_y}{\partial \phi_2} \end{bmatrix}$$

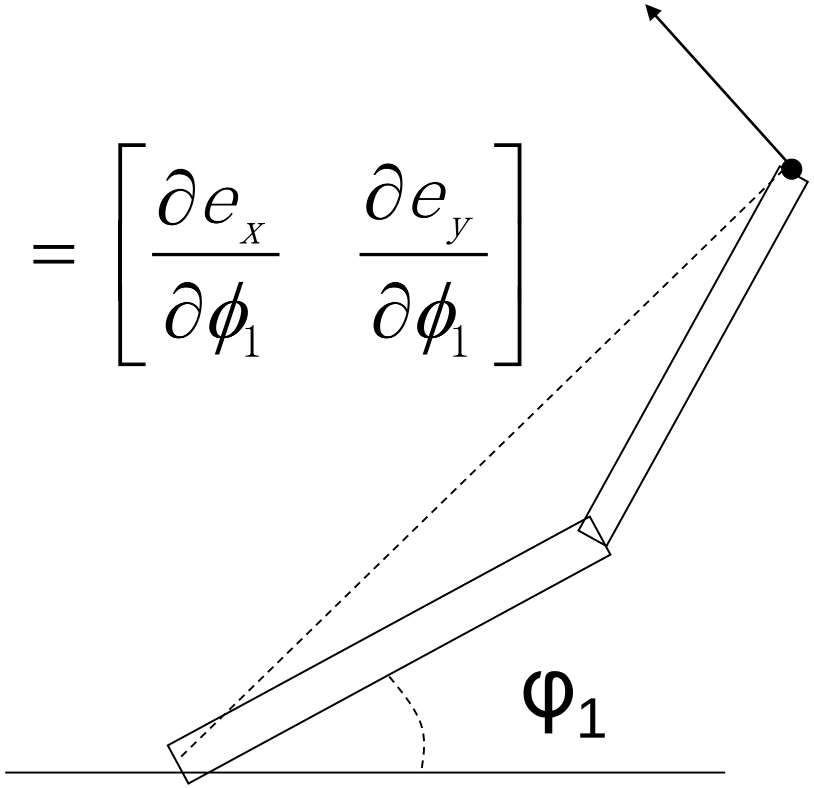
Jacobians

- Consider what would happen if we increased φ_2 by a small amount? What would happen to **e** ?
- What if we increased φ_1 by a small amount?

$$\frac{\partial \mathbf{e}}{\partial \phi_2} = \begin{bmatrix} \dfrac{\partial e_x}{\partial \phi_2} & \dfrac{\partial e_y}{\partial \phi_2} \end{bmatrix}$$



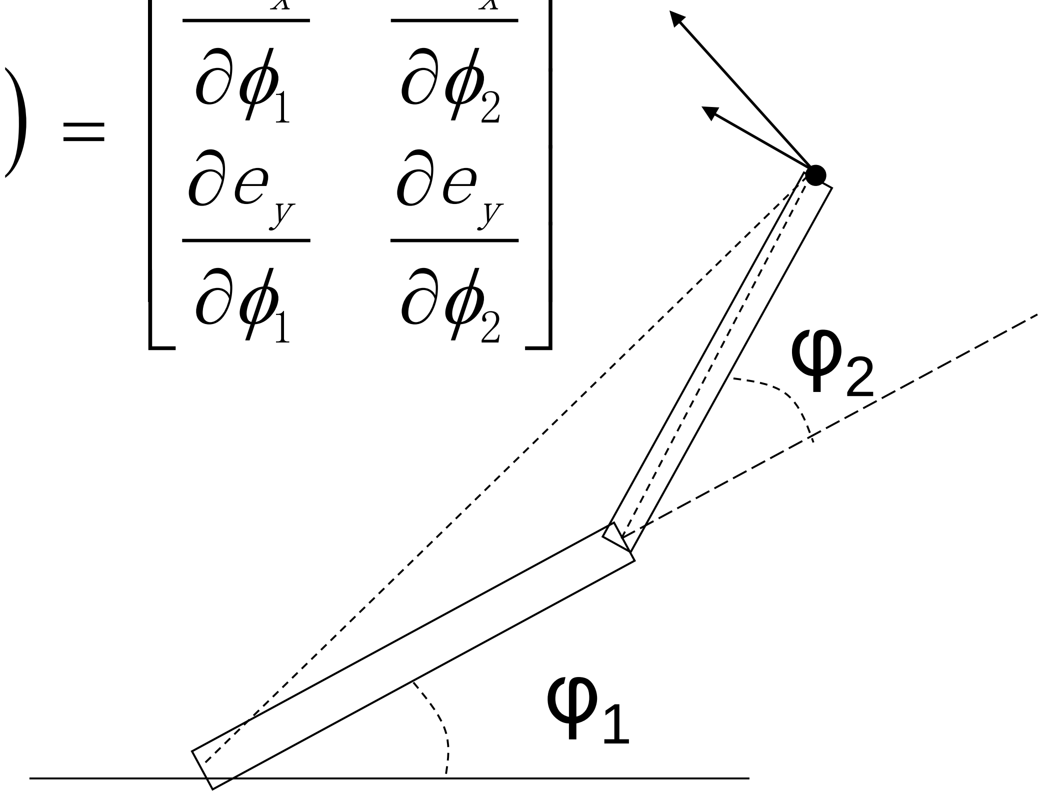
$$\frac{\partial \mathbf{e}}{\partial \phi_1} = \begin{bmatrix} \dfrac{\partial e_x}{\partial \phi_1} & \dfrac{\partial e_y}{\partial \phi_1} \end{bmatrix}$$



Jacobian for a 2D Robot Arm

- Defines how the end effector **e** changes relative to instantaneous changes of each joint angle

$$J(\mathrm{e}, \Phi) = \begin{bmatrix} \dfrac{\partial e_x}{\partial \phi_1} & \dfrac{\partial e_x}{\partial \phi_2} \\ \dfrac{\partial e_y}{\partial \phi_1} & \dfrac{\partial e_y}{\partial \phi_2} \end{bmatrix}$$



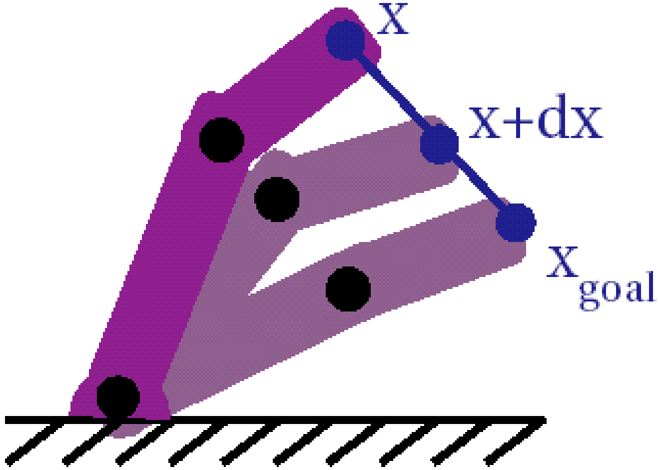
Solving IK—Incremental Changes

- $\boldsymbol{\theta}$**:** current set of joint DOFs;
- **e:** current end effector DOFs;
- **g:** goal DOFs that we want the end effector to reach
- Let $\Delta_{\mathrm{e}} = \boldsymbol{g} - \boldsymbol{e}(\boldsymbol{\theta}) \approx \frac{\boldsymbol{de}}{\boldsymbol{d\theta}} \Delta_{\boldsymbol{\theta}} = \boldsymbol{J}(\boldsymbol{e}, \boldsymbol{\theta}) \Delta_{\boldsymbol{\theta}}$ be the difference between e and g, then

$$\boldsymbol{J} \Delta_{\boldsymbol{\theta}} = \Delta_{\mathrm{e}}$$

solve for $\Delta_{\boldsymbol{\theta}}$

- $\Delta_{\boldsymbol{\theta}}$ moves end towards $\boldsymbol{g}$
  - Only valid for small $\Delta_{\boldsymbol{\theta}}$
  - Take series of small steps
  - Recompute **J(*θ*)** and Δ_{e} at each step



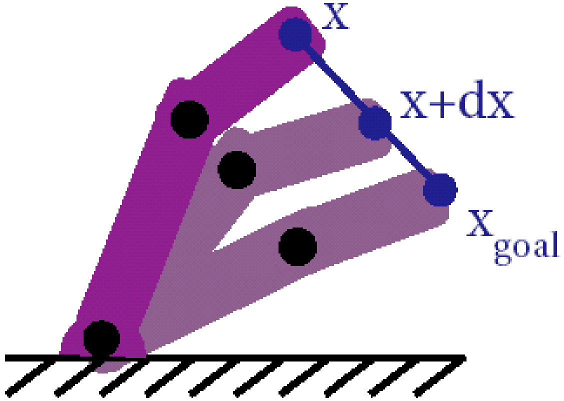
- How to determine length of step?
  - Could try to find optimum size
  - know we're doing rotations:
    - keep less than ~2 degrees

Algorithm

```
solve()
 {
   start with previous θ;
   E = target - computeEndPoint();
   for(k=0; k<max && |E| > eps; k++){
     J = computeJacobian();
     solve J Δ = E; // invert the Jacobian matrix
     if (max(Δ)>2) Δ = 2Δ/max(Δ);
     θ = θ + Δ;
     E = target - computeEndPoint();
   }
 }
```

$E := \Delta_e$

$\mathbf{e} = \mathbf{f}(\boldsymbol{\theta})$



- **Inverse and Forward kinematics demo application**
- http://www.fit.vutbr.cz/~dobsik/projects/kinem_INV/kinem_INV.html

Problems

- How to invert J?
  - Cheat by using transpose (Too easy, we don't do that)
  - Pseudoinverse of Jacobian (Required)
- How to compute J?
  - Numerically (Required)
  - Analytically (Extra Credit)

Inverting the Jacobian

- No guarantee it is invertible
  - Typically not a square matrix, in our case, 2 x N

$$\begin{bmatrix} \frac{\partial f_x}{\partial \theta_0} & \frac{\partial f_x}{\partial \theta_1} & \cdots & \frac{\partial f_x}{\partial \theta_N} \\ \frac{\partial f_y}{\partial \theta_0} & \frac{\partial f_y}{\partial \theta_1} & \cdots & \frac{\partial f_y}{\partial \theta_N} \end{bmatrix} \begin{bmatrix} \boldsymbol{D}_0 \\ \boldsymbol{D}_1 \\ \vdots \\ \boldsymbol{D}_N \end{bmatrix} = \begin{bmatrix} \boldsymbol{E}_x \\ \boldsymbol{E}_y \end{bmatrix}$$

  - Singularities.

Solving $\boldsymbol{J}\,\Delta = E$: pseudo inverse

- Trick: $\boldsymbol{J}^T\boldsymbol{J}$ is square. So:

  $$\boldsymbol{J}\,\Delta = E$$
  $$\boldsymbol{J}^T\boldsymbol{J}\,\Delta = \boldsymbol{J}^T E$$
  $$\Delta = (\boldsymbol{J}^T\boldsymbol{J})^{-1}\boldsymbol{J}^T E$$
  $$\Delta = \boldsymbol{J}^{+}E$$

- $\mathbf{J}^{+}=(\boldsymbol{J}^T\boldsymbol{J})^{-1}\boldsymbol{J}^T$ is the *pseudoinverse* of $\boldsymbol{J}$
  - Properties: $\boldsymbol{J}\boldsymbol{J}^{+}\boldsymbol{J}=\boldsymbol{J}$, $\boldsymbol{J}^{+}\boldsymbol{J}\boldsymbol{J}^{+}=\boldsymbol{J}^{+}$
  - same as $\boldsymbol{J}^{-1}$ when $\boldsymbol{J}$ is square and invertible
  - $\boldsymbol{J}$ is mxn => $\boldsymbol{J}^{+}$ is nxm

- How to compute pseudoinverse?
  - What if $(\boldsymbol{J}^T\boldsymbol{J})^{-1}$ is singular?

Solutions of IK

- Analytic solutions
- Numerical Solutions
  - Jacobian inverse methods
    - *Jacobian transpose*
    - ***Jacobian pseudo-inverse***
    - ***Singular value decomposition***
    - damped least squares (DLS)
    - *Selectively damped least squares (SDLS)*
  - Newton methods
  - Heuristic IK algorithms
    - Cyclic Coordinate Descent (CCD)
    - *Forward and backward reaching inverse kinematics (FABRIK)*
- Data-driven methods
- Hybrid solvers
  - Statistical methods
  - Sequential IK

Singular Value Decomposition

- Any *mxn* matrix **A** can be expressed by SVD
  - **A** = **U S V**^{T}
    - **U** is *m*xmin*(m,n)*, columns are orthogonal
    - **V** is *n*xmin*(m,n)*, columns are orthogonal
    - S is min*(m,n)*xmin*(m,n),* diagonal: *singular values*

$$A = \left(\vec{h}_1 \,\middle|\, \vec{h}_2 \,\middle|\, \cdots \,\middle|\, \vec{h}_N \right) \begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & s_N \end{pmatrix} \begin{pmatrix} \vec{a}_1 \\ \hline \vec{a}_2 \\ \hline \vdots \\ \hline \vec{a}_N \end{pmatrix}$$

  - unique up to sign and order of s_i values
    - canonical: positive, sorted largest to smallest
    - other properties: rank is # of non-zero values; determinant is product of all values, …

Pseudoinverse using SVD

- Given SVD, $\mathbf{A} = \mathbf{U}\,\mathbf{S}\,\mathbf{V}^T$
- pseudoinverse is easy: $\mathbf{A}^+ = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}^T$

$$\begin{pmatrix} s_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & s_N \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{s_1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \frac{1}{s_N} \end{pmatrix}$$

- *singular*: some $s_i = 0$,
- *ill-conditioned*: some $s_i \ll s_0$
  - use 0 instead of $1/s_i$ for those ("truncated")
  - choose small threshold ε, test $s_i < \varepsilon\, s_0$

Solving **A** $\boldsymbol{X}$ = $\boldsymbol{B}$ using SVD

- Using truncated $\mathbf{A}^{+}\boldsymbol{B}$ gives *least-squares solution*:
  - If no solution, gives $\boldsymbol{X}$ that minimizes $||\mathbf{A}\boldsymbol{X}-\boldsymbol{B}||^2$
  - If many solutions, minimizes $||\boldsymbol{X}||^2$ such that $\mathbf{A}\boldsymbol{X}=\boldsymbol{B}$
  - Numerically stable for ill-conditioned matrices
- SVD has many other properties.
  - rank of **A** is # non-zero singular values, determinant is product of all singular values, …
  - known algorithm to compute it
- SVD is a powerful hammer!
  - slow $O(n^3)$; there are faster algorithms.
  - but SVD always works, is fast enough for us
  - hard to implement. some libraries help

Damped least squares

Jacobian pseudo-inverse

$\Delta = (\boldsymbol{J}^T\boldsymbol{J})^{-1}\boldsymbol{J}^T E = \boldsymbol{J}^T (\boldsymbol{J}^T\boldsymbol{J})^{-1} E$

Damped least squares

$\Delta = \boldsymbol{J}^T (\boldsymbol{J}^T\boldsymbol{J}+\lambda^2\boldsymbol{I})^{-1} E$, λ:*non-zero damping constant*

Large damping constant makes the solutions for Δ well behaved near singularities, but also lowers the convergence rate, reduces the accuracy and generates oscillation and shaking.

Very popular

Selectively damped least squares

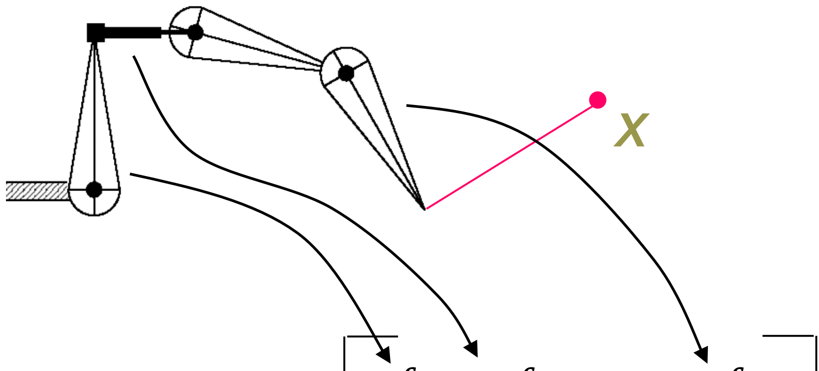
Select λ for each singular vector of the Jacobian SVD according to … presented by Buss and Kim in [BK05]

Back to IK

- Reminder: Let ***E(θ)* = *g*** – *e(**θ**),* error in the current pose

$$\mathbf{J}(\boldsymbol{\theta})\,\boldsymbol{\Delta} = \boldsymbol{E}$$

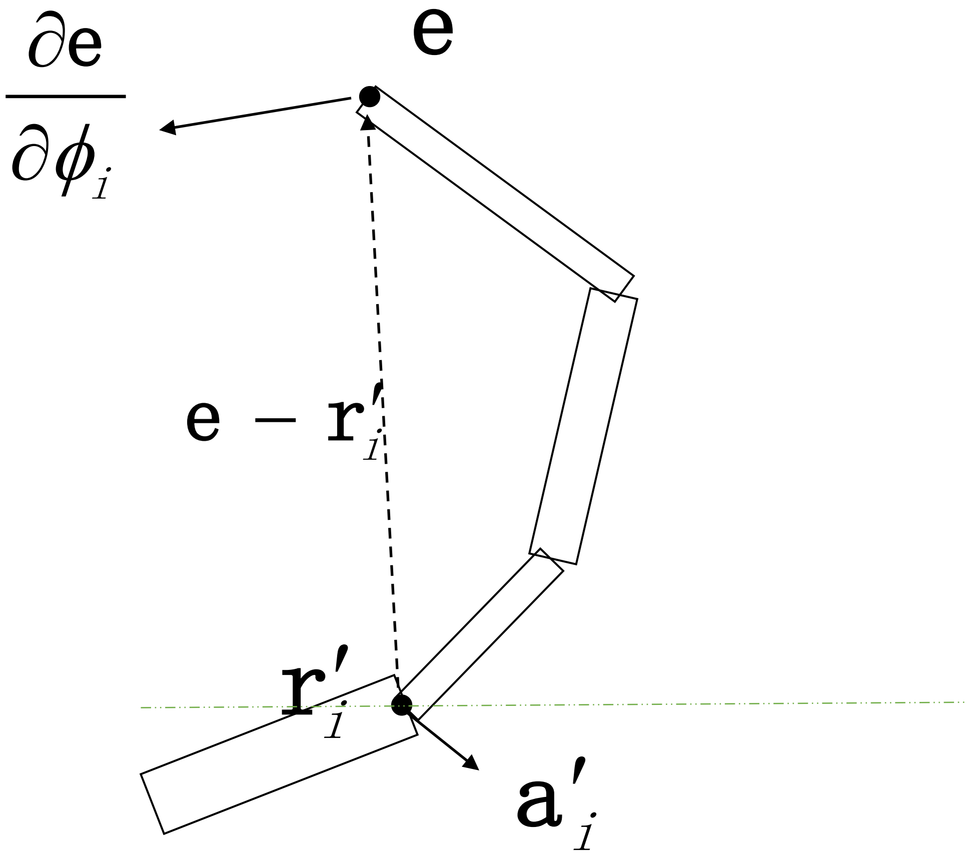
- solve for ***Δ***
    - *i*th column of **J** comes from link *i*



$$\begin{bmatrix} \frac{\partial f_x}{\partial \theta_0} & \frac{\partial f_x}{\partial \theta_1} & \cdots & \frac{\partial f_x}{\partial \theta_N} \\ \frac{\partial f_y}{\partial \theta_0} & \frac{\partial f_y}{\partial \theta_1} & \cdots & \frac{\partial f_y}{\partial \theta_N} \end{bmatrix}$$

Rotational DOFs

$$\frac{\partial \mathbf{e}}{\partial \phi_i} = \mathbf{a}'_i \times \left(\mathbf{e} - \mathbf{r}'_i\right)$$



a'$_i$: unit length rotation axis in world space

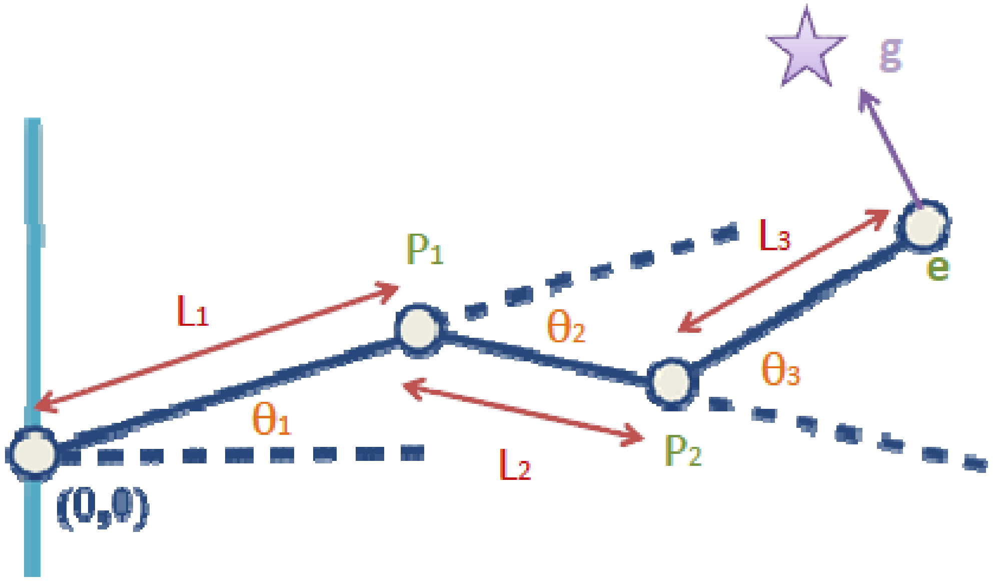
r'$_i$: position of joint pivot in world space

e: end effector position in world space

Computing the Jacobian columns

- For a rotational joint, the linear change in the end effector is the cross product of the axis of revolution and a vector from the joint to the end effector.

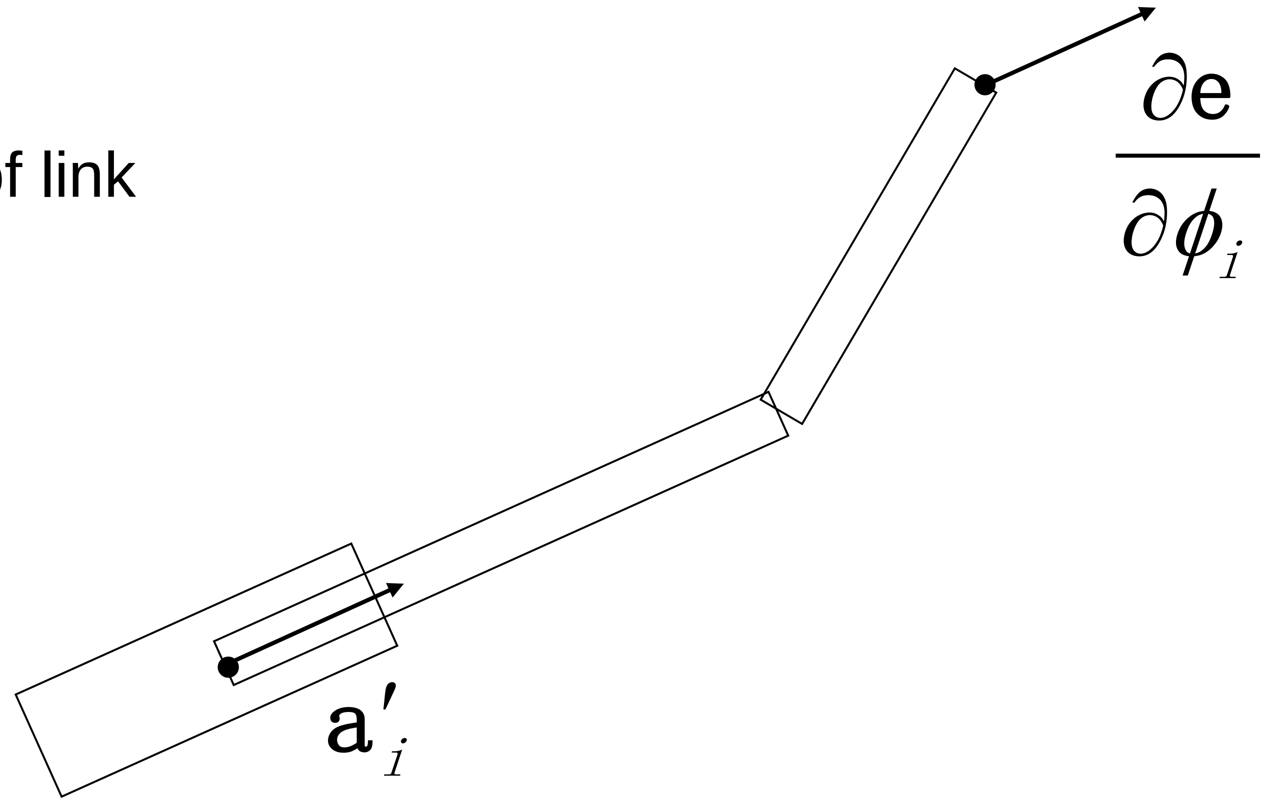
$$\frac{\partial \mathbf{e}}{\partial \theta_1} = \left[\frac{\partial \mathbf{e}_x}{\partial \theta_1}\ \frac{\partial \mathbf{e}_y}{\partial \theta_1}\ \frac{\partial \mathbf{e}_z}{\partial \theta_1}\right]^T = \left(\mathbf{a}_1' \times (\mathbf{e} - \mathbf{r}_1')\right)$$



$$J = \begin{bmatrix} ((0,0,1)\times \mathbf{e})_x & ((0,0,1)\times(\mathbf{e}-\mathbf{P}_1))_x & ((0,0,1)\times(\mathbf{e}-\mathbf{P}_2))_x \\ ((0,0,1)\times \mathbf{e})_y & ((0,0,1)\times(\mathbf{e}-\mathbf{P}_1))_y & ((0,0,1)\times(\mathbf{e}-\mathbf{P}_2))_y \\ ((0,0,1)\times \mathbf{e})_z & ((0,0,1)\times(\mathbf{e}-\mathbf{P}_1))_z & ((0,0,1)\times(\mathbf{e}-\mathbf{P}_2))_z \end{bmatrix}$$

Computing the Jacobian columns

- For a translational joint:
  - $\partial f(\boldsymbol{\theta})/\partial\theta_j$ = vector in direction of link



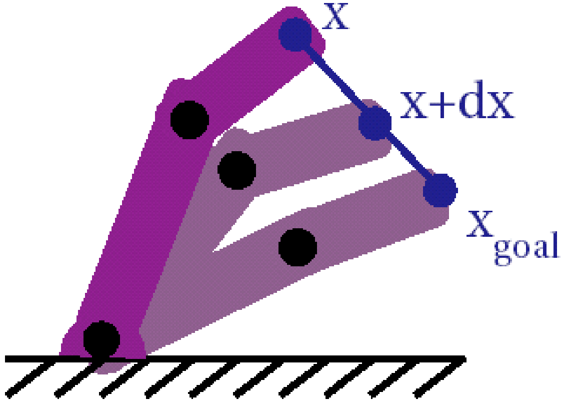
- Notes:
  - Remember to compute in world space!
  - I’ve assumed one degree of freedom per joint
  - If there are multiple DOFs per joint, refer to CSE169_12.ppt of CSE 169: Computer Animation @ UCSD winter 2004

Building the Jacobian

- To build the entire Jacobian matrix, we just loop through each DOF and compute a corresponding column in the matrix
- If we wanted, we could use more elaborate joint types (scaling, translation along a path, shearing…) and still compute an appropriate derivative
- If absolutely necessary, we could always resort to computing a numerical approximation to the derivative

IK Algorithm

```
solve()
  {
    Vector θ = getLinkParameters();
    Vector E = target - computeEndPoint();
    for(k=0; k<max && E.norm() > eps; k++){
      Matrix J = computeJacobian();
      Matrix J⁺ = J.pseudoinverse();
      Vector Δ = J⁺ E;
      if (max(Δ)>2) Δ *= 2/max(Δ);
      θ = θ + Δ;
      putLinkParameters(θ);
      E = target - computeEndPoint();
    }
  }
```



What’s left for IK?

- Joint limits
- When to stop the iterations

Joint limits

- Each joint may have limited range.
- Modify algorithm:
  - After finding *Δ,* test each joint:

    $$\boldsymbol{\theta min_i} < (\boldsymbol{\theta}+\Delta)_i < \boldsymbol{\theta max_i}$$

  - If it would go out of range
    - set column *i* of J to 0
    - claims “this parameter has no effect”
  - Recompute $\mathbf{J^+}$
    - Least-squares solution will make $\Delta_i\, z\, 0$
    - For robustness, you may want to force $\Delta_i = 0$
  - Find *Δ,* repeat

Note on numerical algorithms

- Various algorithms for non-linear multidimensional root-finding…this one works for us
- Root-finding is related to *optimization*:
  - $F(\boldsymbol{\theta})=X \Leftrightarrow$ minimize $||F(\boldsymbol{\theta})-X||^2$
- Many computer animation problems are optimization problems
- Many algorithms have solving AX = B at their core.

When to Stop

- There are three main stopping conditions we should account for
  - Finding a successful solution (or close enough)
  - Getting stuck in a condition where we can’t improve (local minimum)
  - Taking too long (for interactive systems)
- All three of these are fairly easy to identify by monitoring the progress of $\boldsymbol{\Phi}$
- These rules are just coded into the while() statement for the controlling loop

Finding a Successful Solution

- We really just want to get close enough within some tolerance
- If we’re not in a big hurry, we can just iterate until we get within some floating point error range
- Alternately, we could choose to stop when we get within some tolerance measurable in pixels
- For example, we could position an end effector to 0.1 pixel accuracy
- This gives us a scheme that should look good and automatically adapt to spend more time when we are looking at the end effector up close (level-of-detail)

Local Minima

- If we get stuck in a local minimum, we have several options
  - Don’t worry about it and just accept it as the best we can do
  - Switch to a different algorithm (CCD…)
  - Randomize the pose vector slightly (or a lot) and try again
  - Send an error to whatever is controlling the end effector and tell it to try something else
- Basically, there are few options that are truly appealing, as they are likely to cause either an error in the solution or a possible discontinuity in the motion

Taking Too Long

- In a time critical situation, we might just limit the iteration to a maximum number of steps
- Alternately, we could use internal timers to limit it to an actual time in seconds

Iteration Stepping

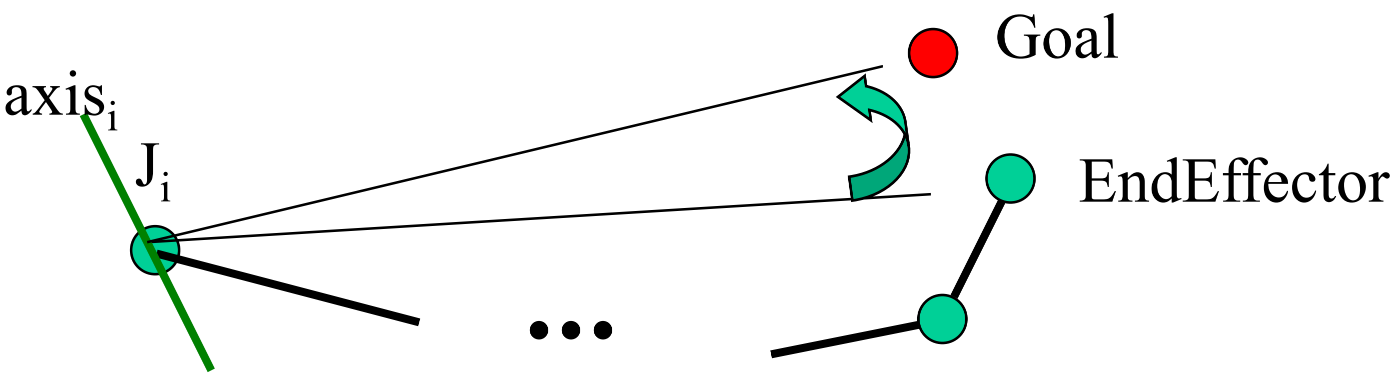
- Step size
- Stability
- Performance

IK – cyclic coordinate descent

Heuristic solution

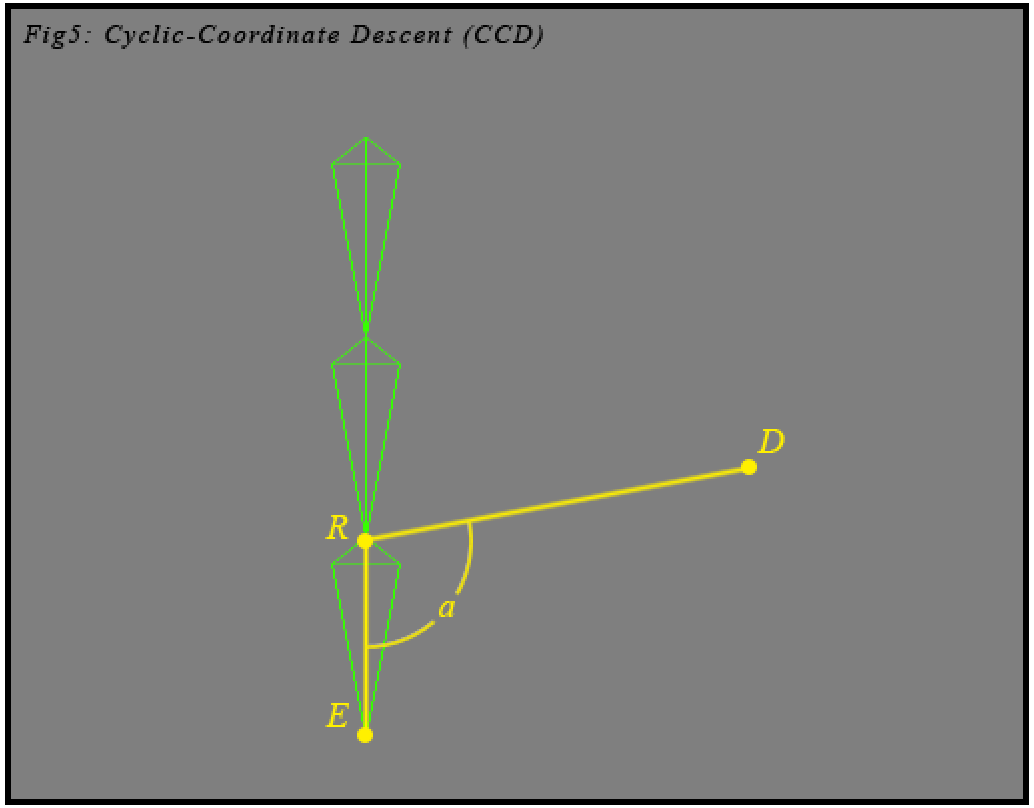
Consider one joint at a time, from outside in
At each joint, choose update that best gets end effector to goal position

In 2D – pretty simple



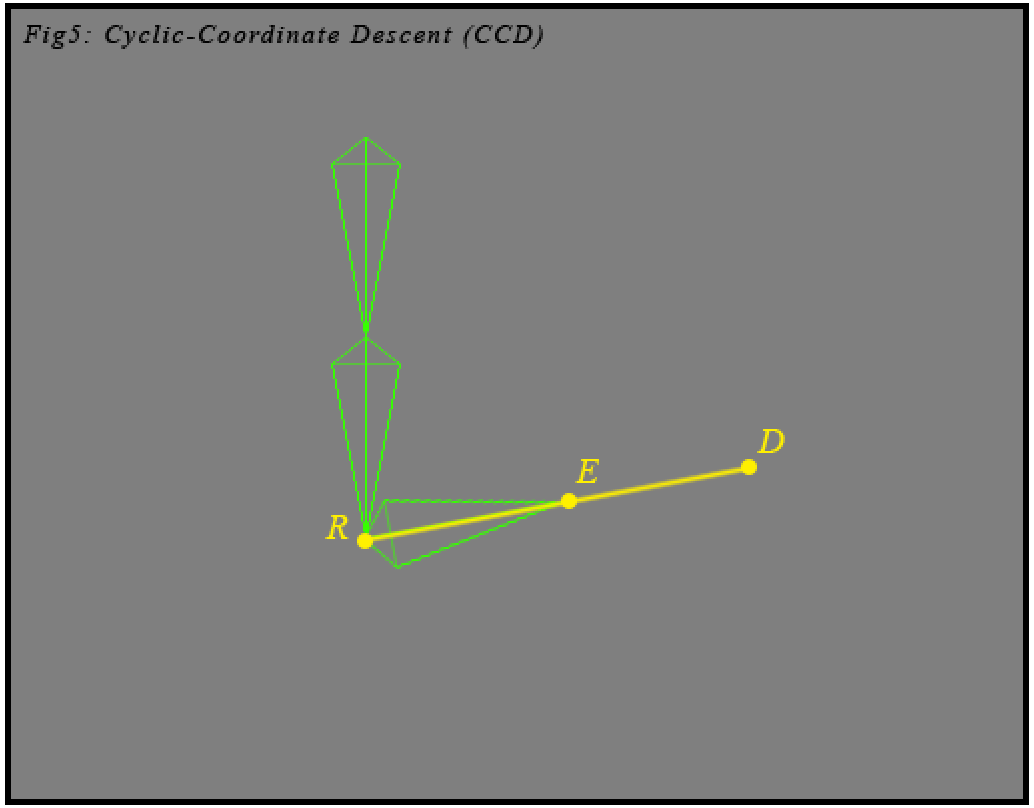
Cyclic-Coordinate Descent

- Starting with the root of our effector, R, to our current endpoint, E.
- Next, we draw a vector from R to our desired endpoint, D
- The inverse cosine of the dot product gives us the angle between the vectors: cos(a) = RD ● RE



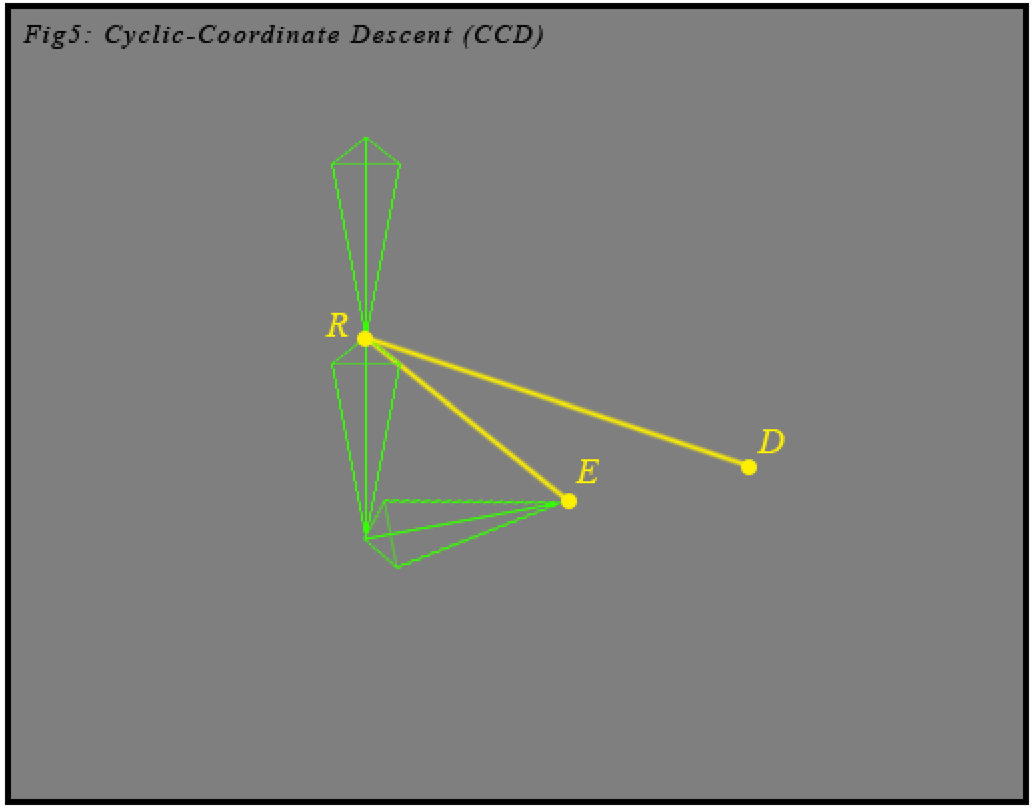
Cyclic-Coordinate Descent

Rotate our link so that RE falls on RD



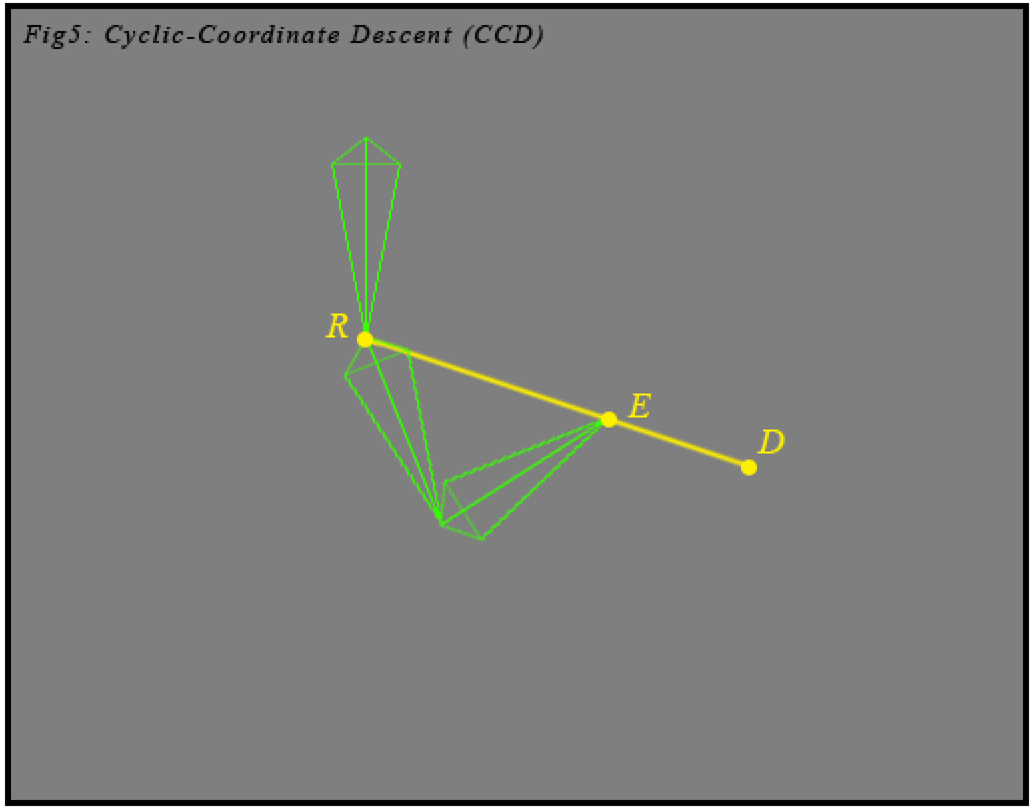
Cyclic-Coordinate Descent

Move one link up the chain, and repeat the process

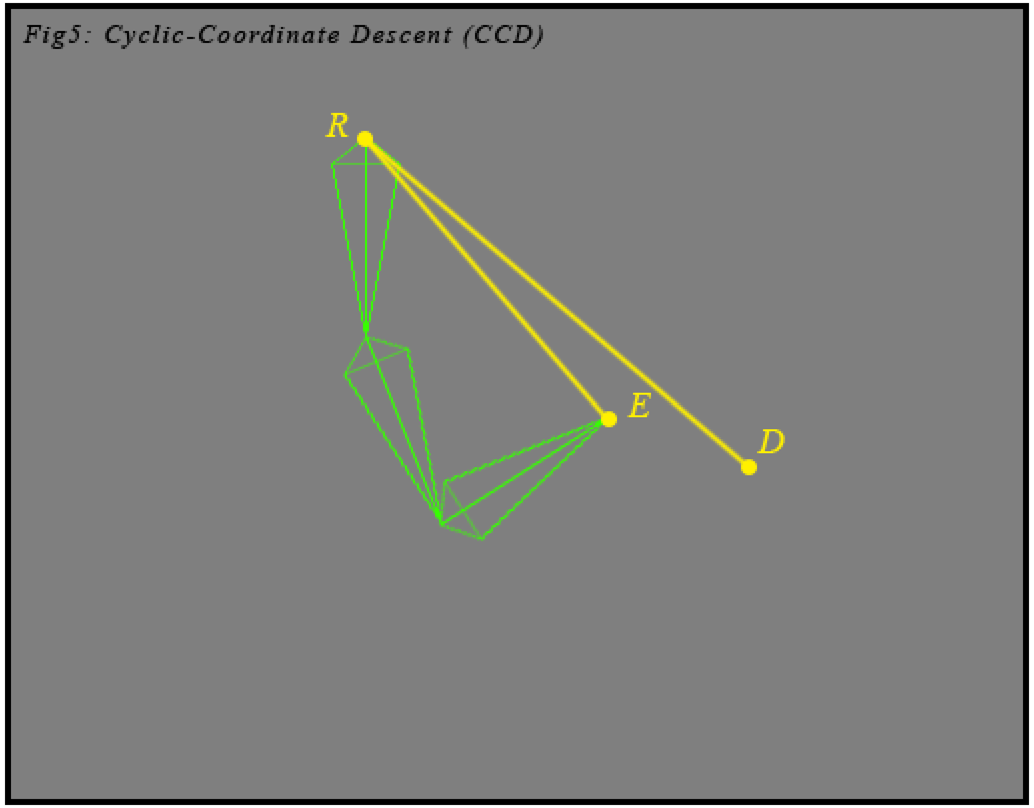


Cyclic-Coordinate Descent

The process is basically repeated until the root joint is reached. Then the process begins all over again starting with the end effector, and will continue until we are close enough to D for an acceptable solution.

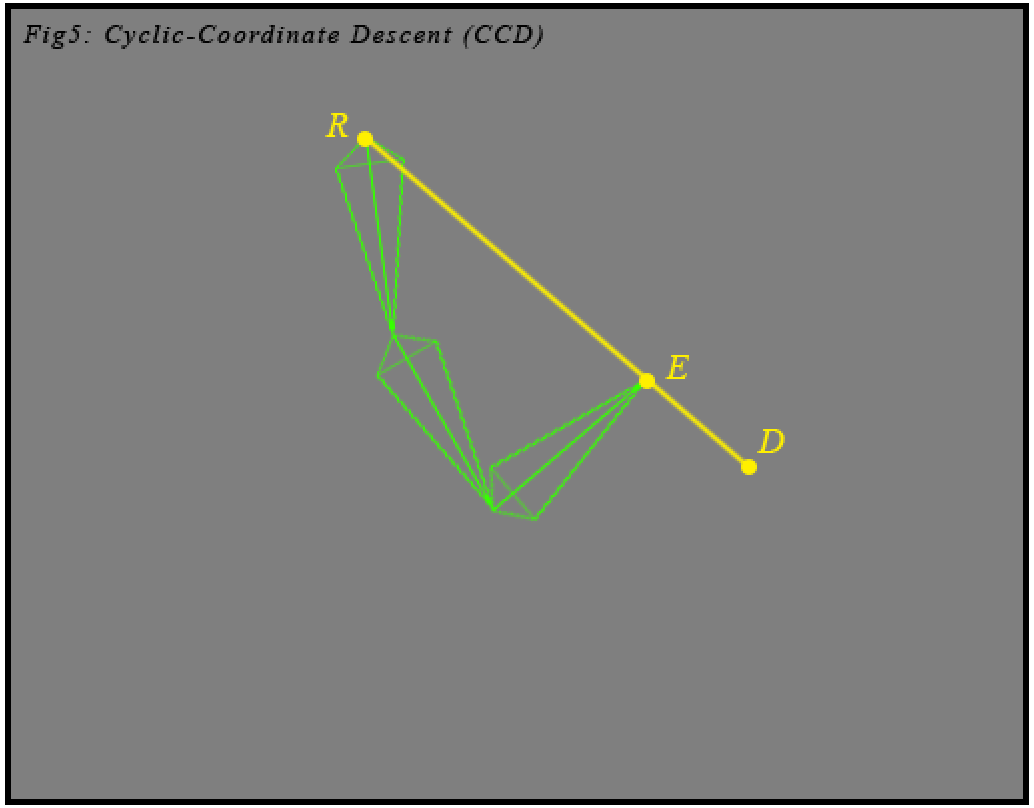


Cyclic-Coordinate Descent

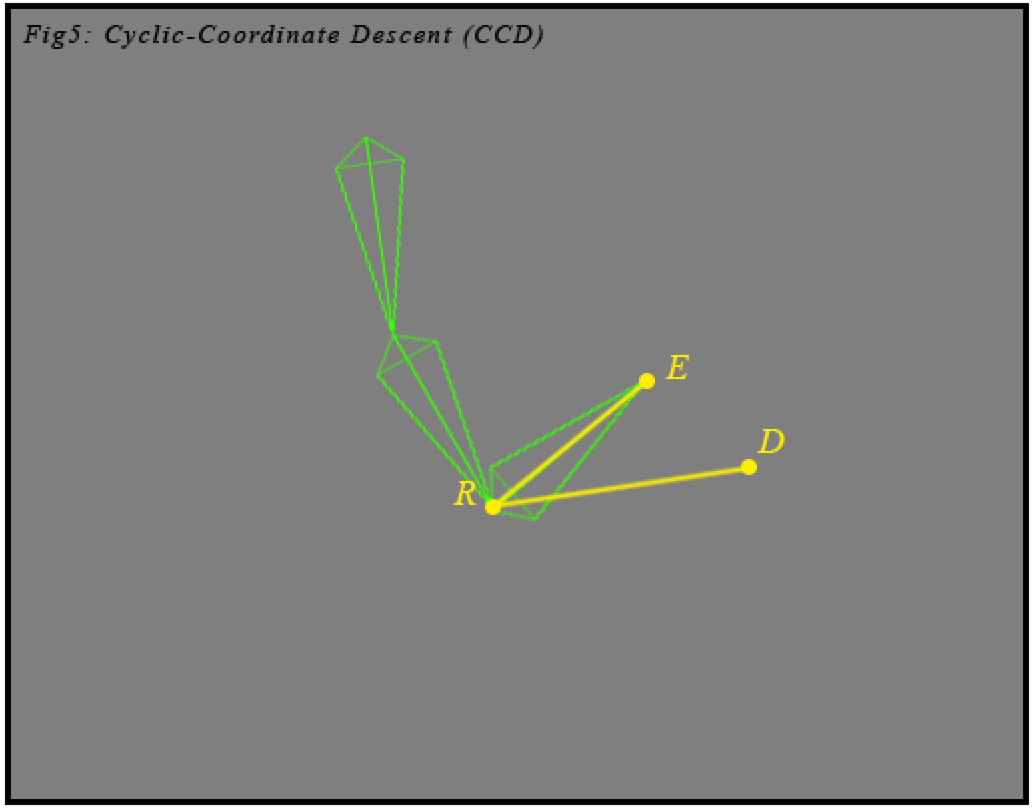


Cyclic-Coordinate Descent

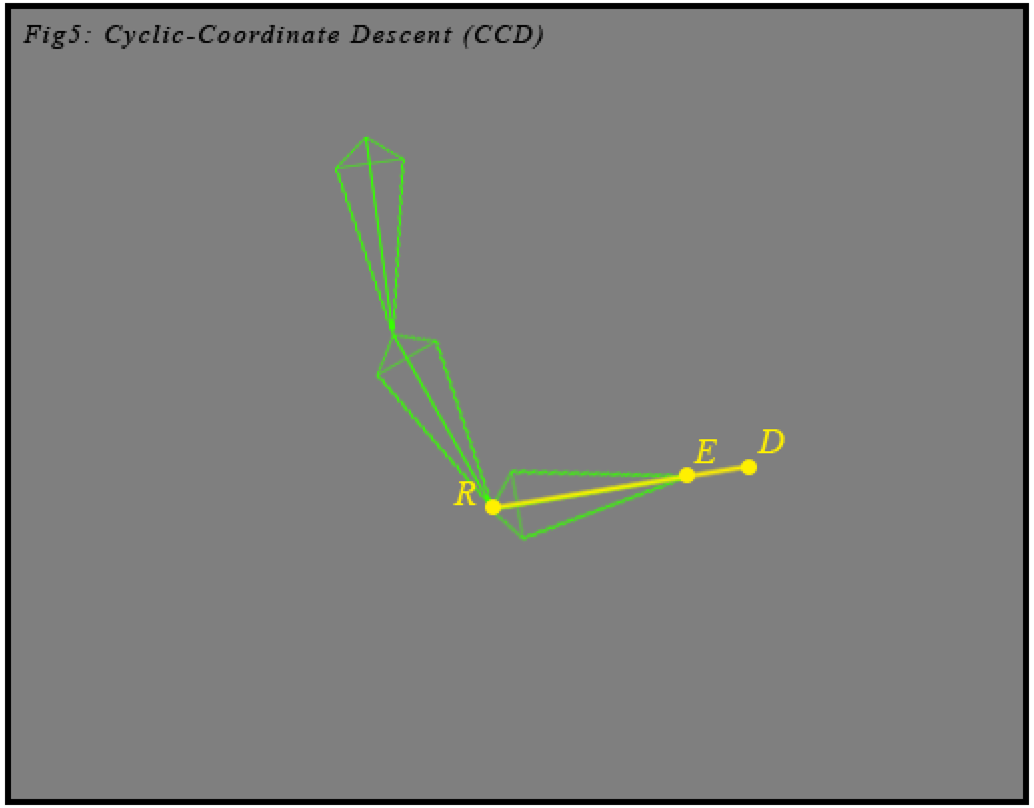
We’ve reached the root. Repeat the process



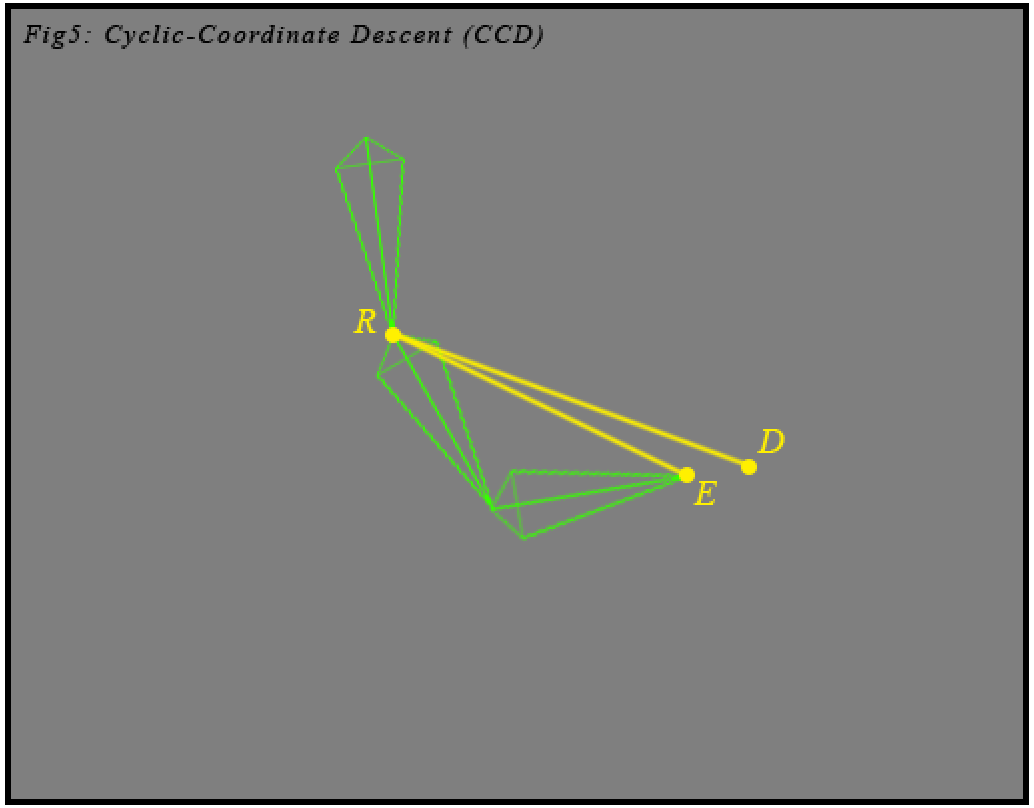
Cyclic-Coordinate Descent



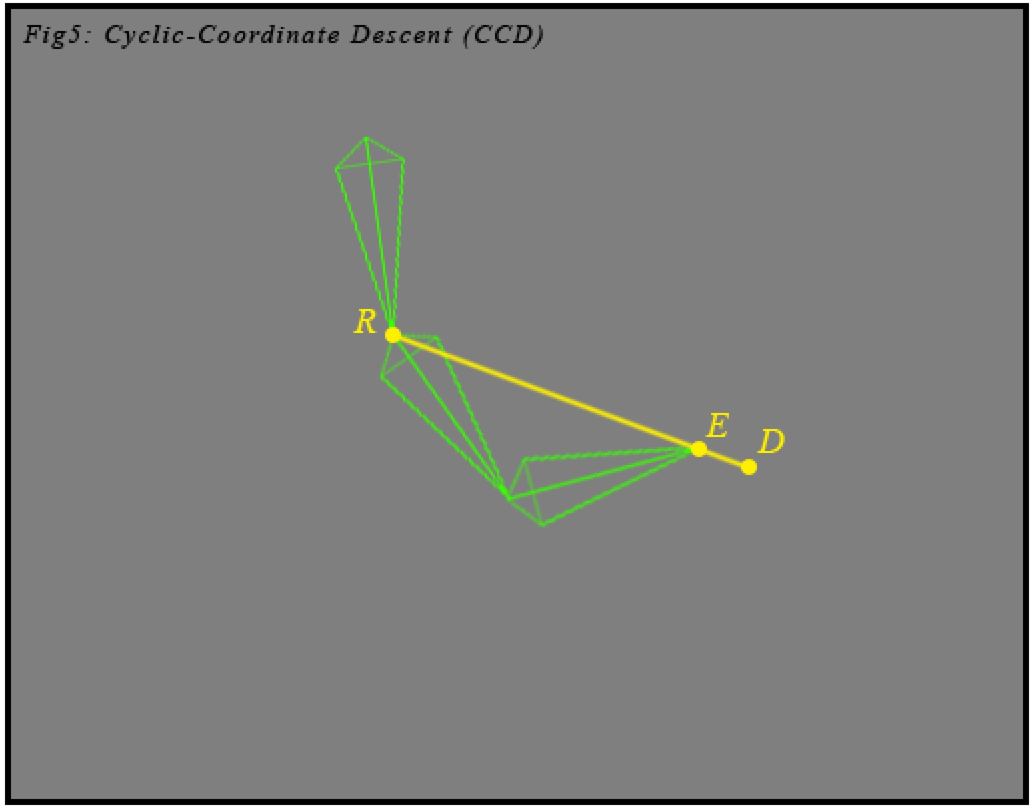
Cyclic-Coordinate Descent



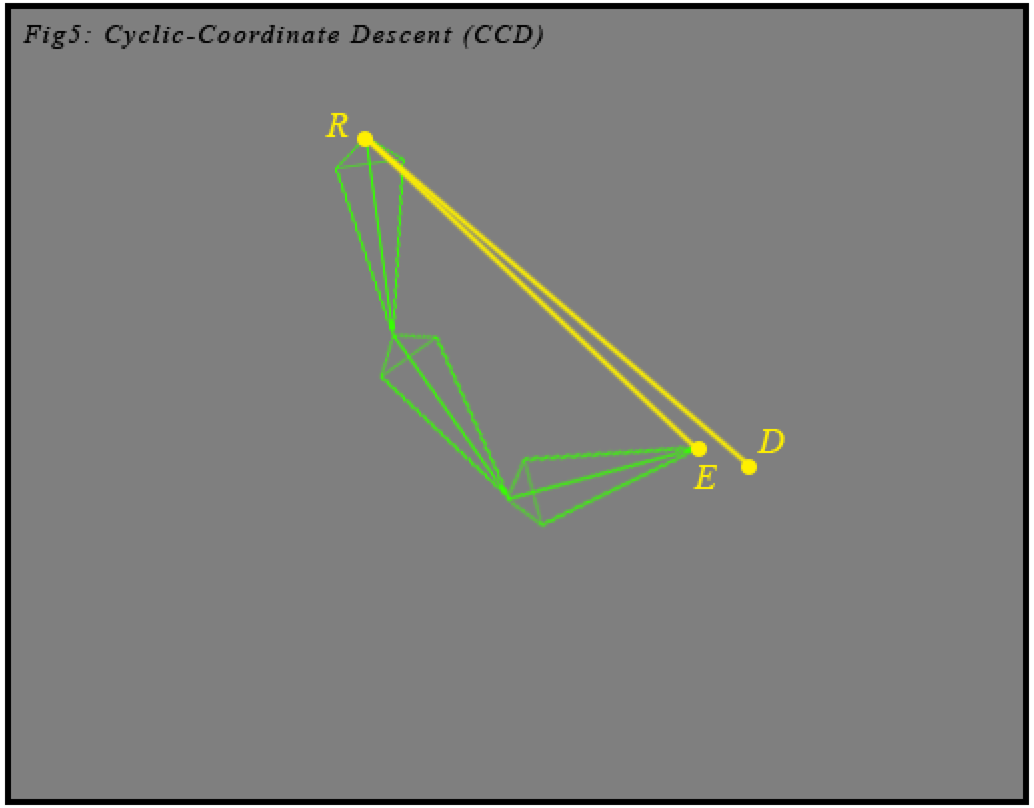
Cyclic-Coordinate Descent



Cyclic-Coordinate Descent

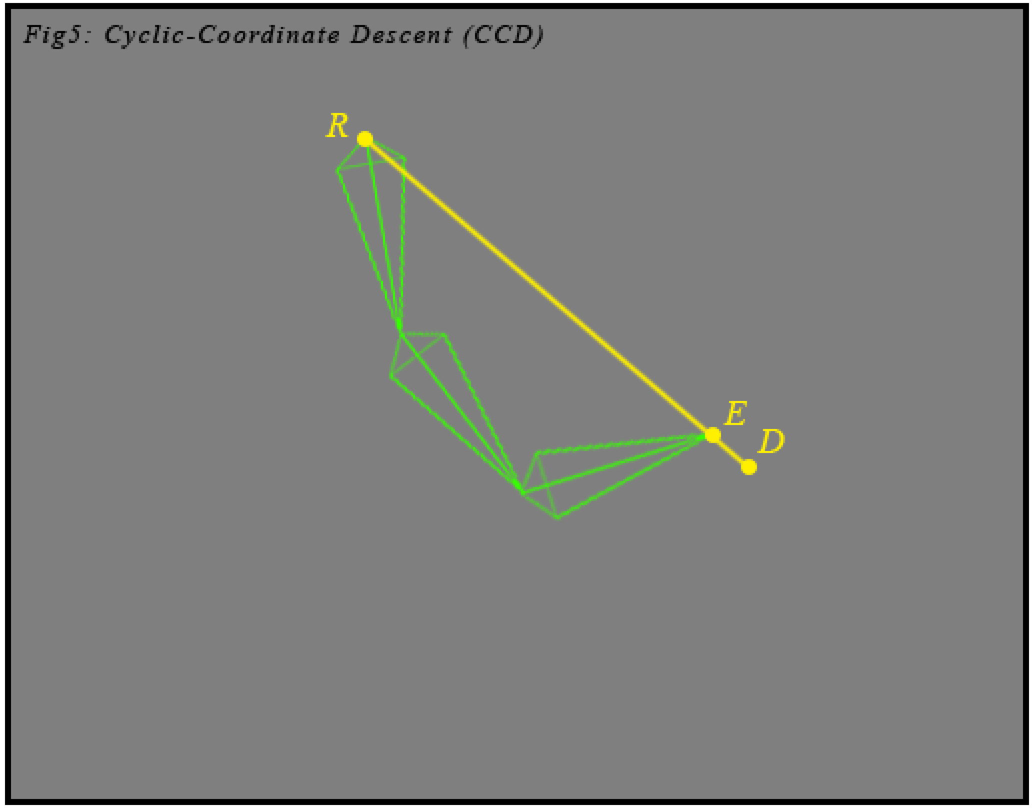


Cyclic-Coordinate Descent

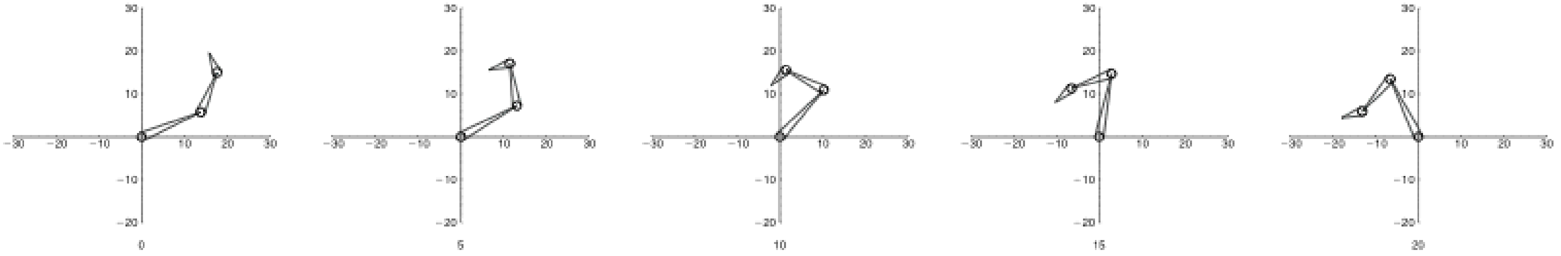


Cyclic-Coordinate Descent

We’ve reached the root again. Repeat the process until solution reached.



IK – cyclic coordinate descent



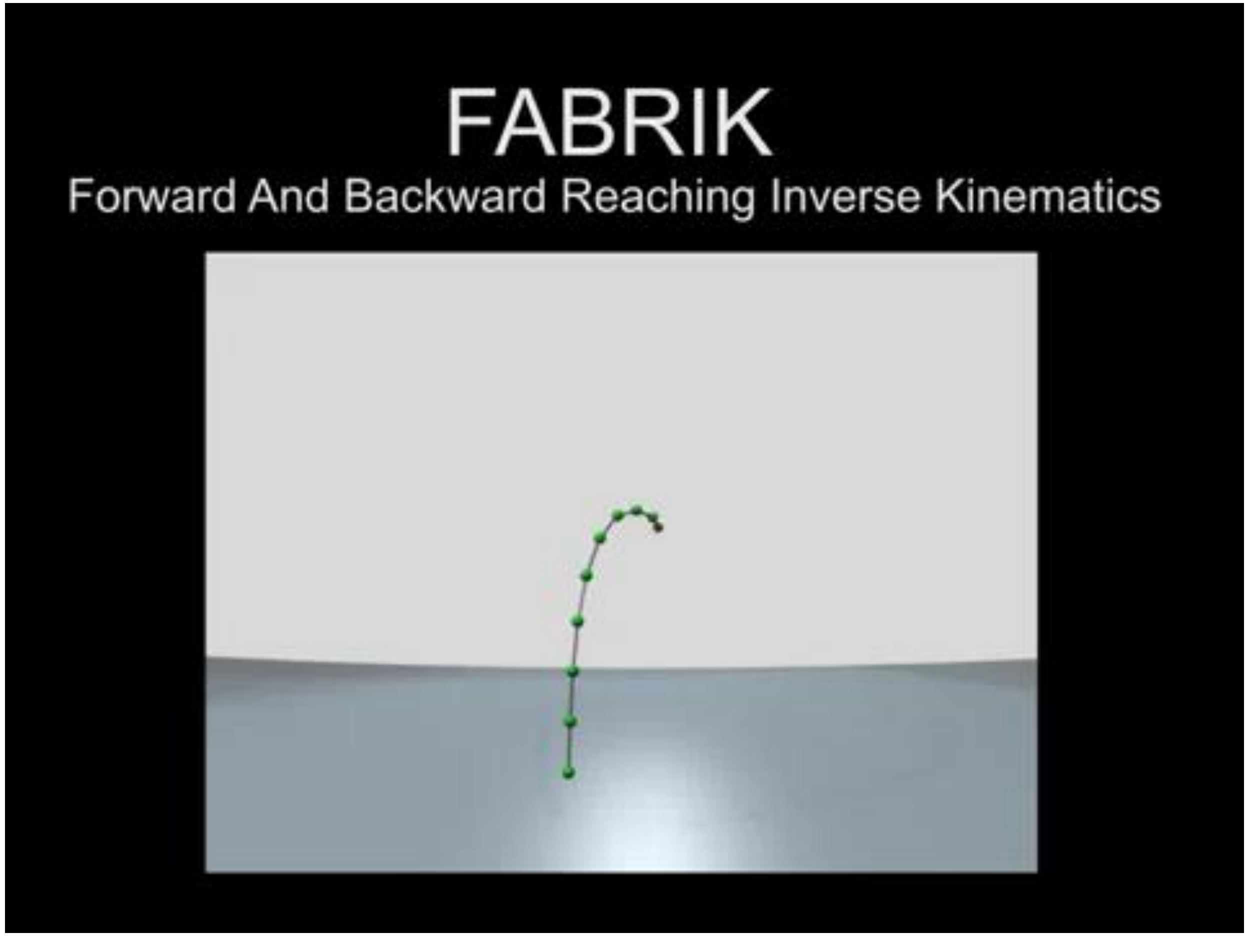
Other orderings of processing joints are possible

Because of its procedural nature

- **Lends itself to enforcing joint limits**
- **Easy to clamp angular velocity**

Dr. Andreas Aristidou

- FABRIK: a fast, iterative solver for the inverse kinematics problem, 2011
- Extending FABRIK with Model Constraints, 2016



http://wiki.roblox.com/index.php?title=Inverse_kinematics

Thank you