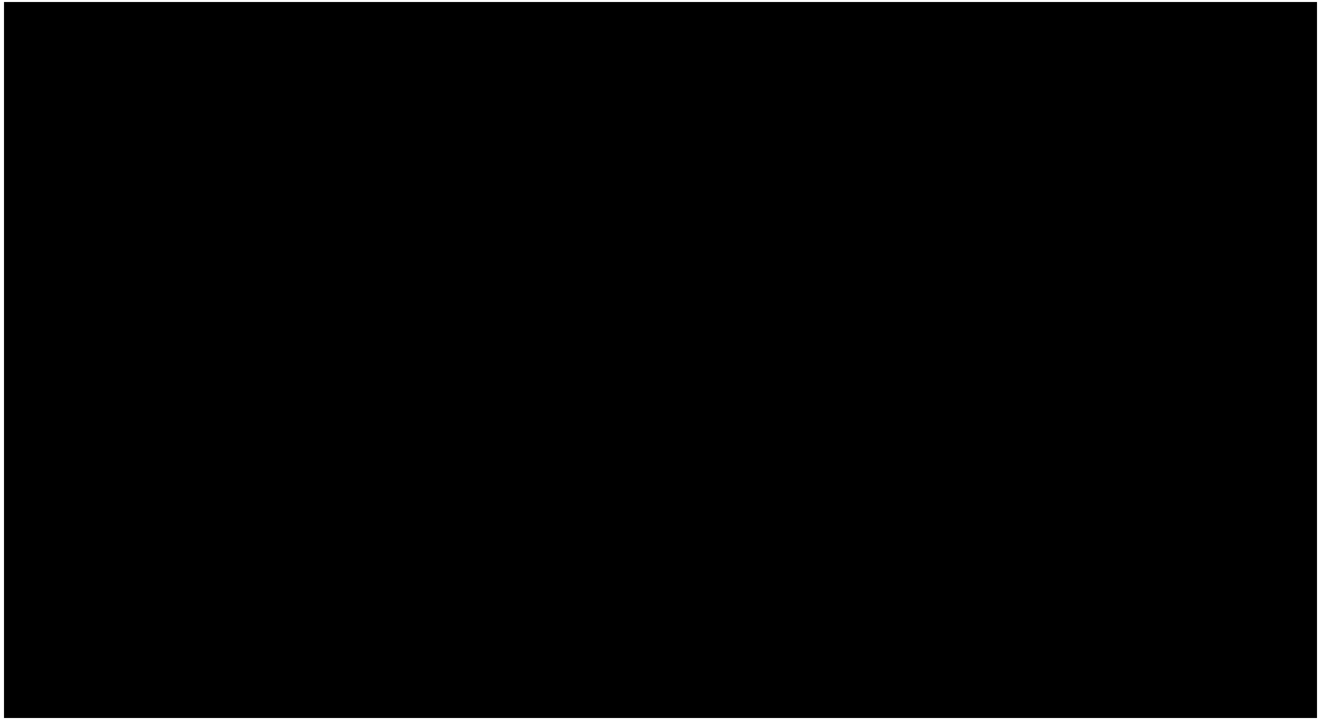
Math for Game Programmers: Inverse Kinematics

Gino van den Bergen
gino@dtecta.com
Twitter: @dtecta

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 17–21, 2014
EXPO DATES: MARCH 19–21
2014

Uhhh… Inverse Kinematics?



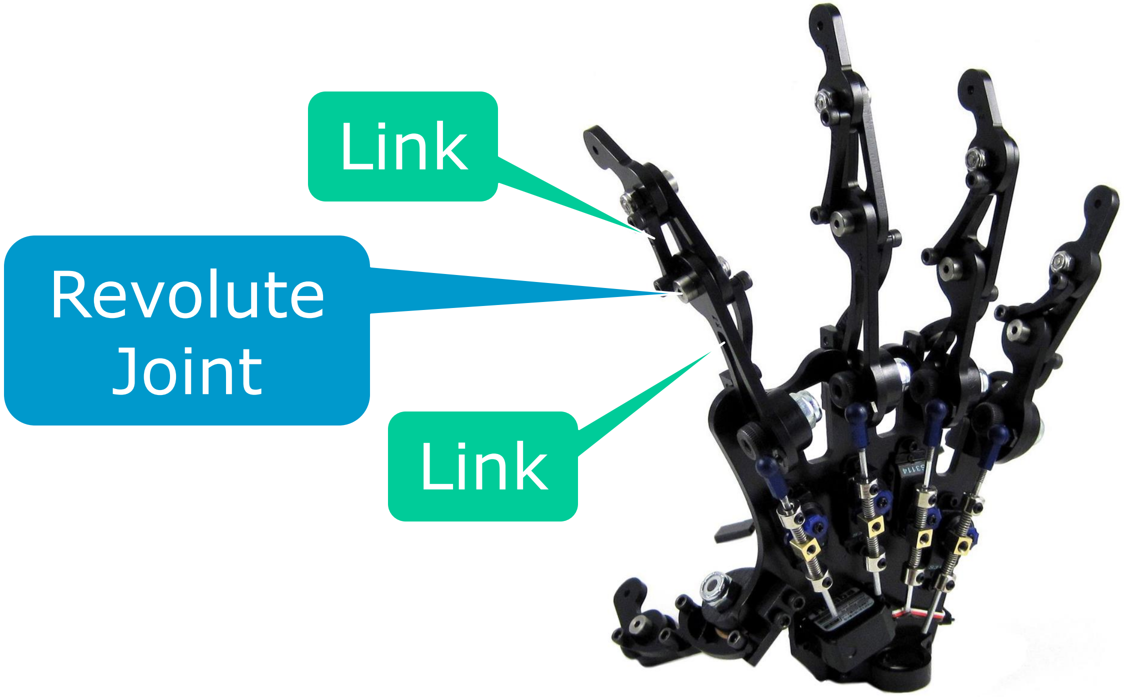
Problem Description

- We have a bunch of rigid bodies aka *links* (aka *bones*).
- Pairs of links are connected by *joints*.
- A joint limits the *degrees of freedom* (DoFs) of one link relative to the other.
- Connection graph is a tree. No loops!

Problem Description (cont’d)

- Let’s consider 1-DoF joints only:
    - *Revolute*: single-axis rotation aka *hinge*.
    - *Prismatic*: single-axis translation aka *slider*.
- Positions and velocities of links are defined by the values and speeds of the scalar joint parameters (angles, distances).

Problem Description (cont'd)



Problem Description (cont’d)

- Given some constraints on the poses and velocities of one or more links, compute a vector of joint parameters that satisfies the constraints.
- The constrained links are called *end-effectors*, and are usually (but not per se) the end-links of a linkage.

Free vs. Fixed Joints

- Usually, only a few joints are free. Free joints are available for constraint resolution.
- The other joints are controlled by forward kinematics. Their positions and velocities are fixed at a given instance of time.

Position and Orientation

- Each link maintains a *pose, i.e.* position and orientation, relative to its parent.
- Position is a 3D vector. Orientation is a rotation matrix or a quaternion.
- Position and orientation can be combined into a single entity as a *dual quaternion.*

Dual Quaternions

- Quaternion algebra is extended by introducing a dual unit ε, for which $\varepsilon^2 = 0$.
- Elements are 1, *i*, *j*, *k*, ε, $i\varepsilon$, $j\varepsilon$, and $k\varepsilon$.
- A dual quaternion is expressed as:

$$\hat{\mathbf{q}} = \mathbf{q} + \mathbf{q}'\varepsilon$$

We call $\mathbf{q}$ the *real* part and $\mathbf{q}'$ the *dual* part.

Dual Quaternions (cont'd)

- Multiplication gives: $(\mathbf{q}_1 + \mathbf{q}_1'\varepsilon)(\mathbf{q}_2 + \mathbf{q}_2'\varepsilon)$

$$= \mathbf{q}_1\mathbf{q}_2 + (\mathbf{q}_1\mathbf{q}_2' + \mathbf{q}_1'\mathbf{q}_2)\varepsilon + 0$$

- Real part is the product of real parts only; it does not depend on dual parts!

Dual Quaternions (cont'd)

- Unit dual quaternions represent poses.
- Given an orientation represented by a unit (real) quaternion $\mathbf{q}$, and a position by a 3D vector $\mathbf{c}$, the pose is represented by:

$$\mathbf{q} + \frac{1}{2}\mathbf{c}\mathbf{q}\varepsilon$$

$\mathbf{c}$ is considered a *pure* imaginary quaternion (zero scalar part).

Dual Quaternions (cont’d)

- The *conjugate* of a dual quaternion:

$$\hat{\mathbf{q}}^* = (\mathbf{q} + \mathbf{q}'\varepsilon)^* = \mathbf{q}^* + \mathbf{q}'^*\varepsilon$$

- The *inverse* of a unit dual quaternion is its conjugate: $(\mathbf{q} + \mathbf{q}'\varepsilon)(\mathbf{q} + \mathbf{q}'\varepsilon)^* =$

$$\mathbf{q}\mathbf{q}^* + \left(\mathbf{q}\mathbf{q}'^* + \mathbf{q}'\mathbf{q}^*\right)\varepsilon = 1 + 0\varepsilon$$

Dual Quaternions (almost done)

- Given a pose $\hat{\mathbf{q}} = \mathbf{q} + \mathbf{q}'\varepsilon$,
- The orientation is simply $\mathbf{q}$ (the real part).
- The position is given by $2\mathbf{q}'\mathbf{q}^*$.
- Exercise: Prove that for unit dual quaternions, $2\mathbf{q}'\mathbf{q}^*$ has a zero scalar part.

Hint: $\mathbf{q}\mathbf{q}^* + \left(\mathbf{q}\mathbf{q}'^* + \mathbf{q}'\mathbf{q}^*\right)\varepsilon = 1 + 0\varepsilon$

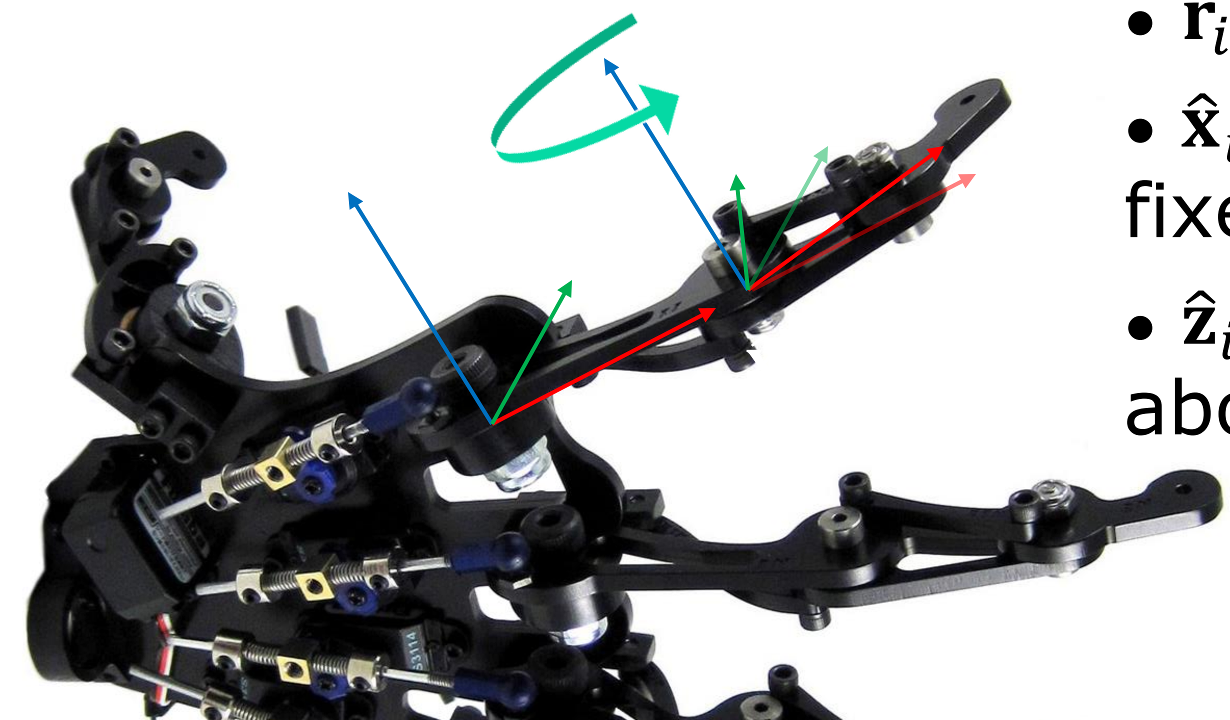
Kinematic Chain

- In a chain of links, $\hat{\mathbf{r}}_i$ is the relative pose from link *i* to its parent link *i* – 1.
- The pose from a link *i* to the world frame is simply $\hat{\mathbf{q}}_i = \hat{\mathbf{r}}_1 \cdots \hat{\mathbf{r}}_i$, the product of all relative poses in the chain up to link *i*.
- The pose from link *i* to link *j* is: $\hat{\mathbf{q}}_j{}^* \hat{\mathbf{q}}_i$ (even if *i* and *j* are on different chains).

Relative Pose

- The relative pose is the product of a fixed pose and a variable pose: $\hat{\mathbf{r}}_i = \hat{\mathbf{x}}_i \hat{\mathbf{z}}_i$
- $\hat{\mathbf{x}}_i$ fixes the joint axis relative to the parent’s frame.
- $\hat{\mathbf{z}}_i$ represents the joint’s degree of freedom.

Relative Pose (cont’d)



- $\hat{\mathbf{r}}_i = \hat{\mathbf{x}}_i \hat{\mathbf{z}}_i$
- $\hat{\mathbf{x}}_i$ (transparent) fixes joint axis.
- $\hat{\mathbf{z}}_i$ rotation about z-axis.

Relative Pose (cont'd)

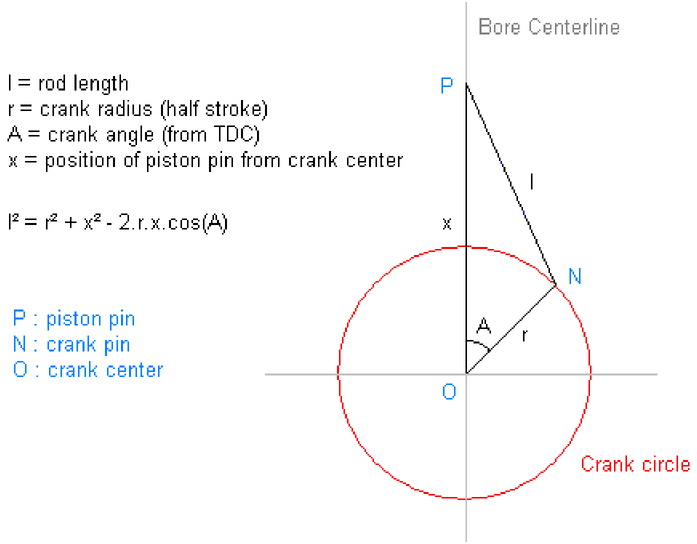
- W.l.o.g., we choose $\hat{\mathbf{x}}_i$ such that the joint axis is the z-axis of the new frame.
- For a revolute: $\hat{\mathbf{z}}_i = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)k$, rotating θ radians about the local z-axis.
- For a prismatic: $\hat{\mathbf{z}}_i = 1 + \frac{d}{2}k\varepsilon$, translating d units along the local z-axis.

Positional Constraints

- Find a vector of joint parameters that satisfies constraints on the poses of the end-effectors. Examples:
    - The feet of a character land firmly on an irregular terrain without interpenetration.
    - The gaze of an NPC follows some target.
    - The fingertip of a character presses a button.

Analytical Approach

- Sometimes joint parameters can be solved analytically, e.g. the position of a piston is determined by the crank angle.



Analytical Approach

- However, polynomials of degree 5 and up can generally not be solved analytically.
- Moreover, analytical solvers often yield multiple solutions which is less practical.
- Can’t get a closest-fit solution if a solution does not exists.

Iterative Approach

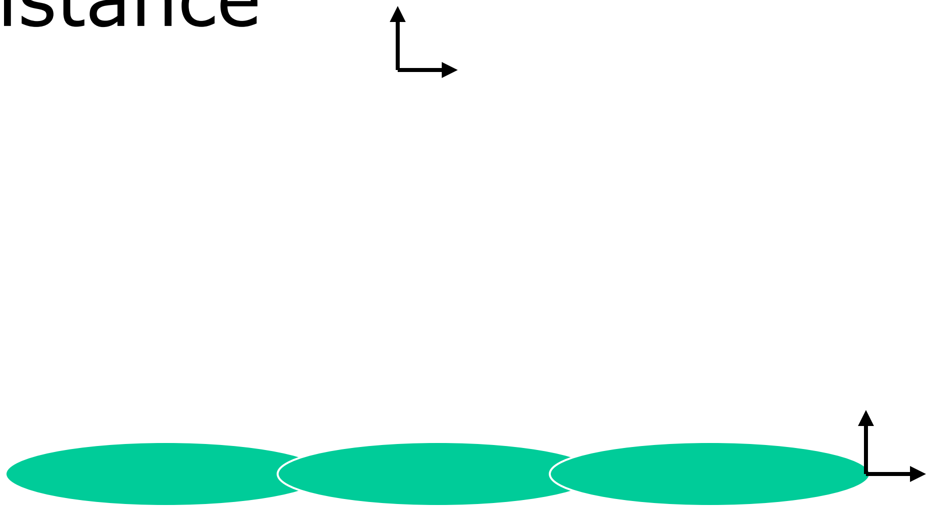
- A constraint solution is approximated by taking many steps towards reducing the constraint error.
- Converges to the nearest local minimum, which may not be a proper solution (should one exist).

Cyclic Coordinate Descent (CCD)

- Iteratively solve each joint while keeping relative poses between other joints fixed.
- “Solving” means minimizing some error.
- Different strategies: Repeatedly
  - Work from end-effector to base.
  - Work from base to end-effector.

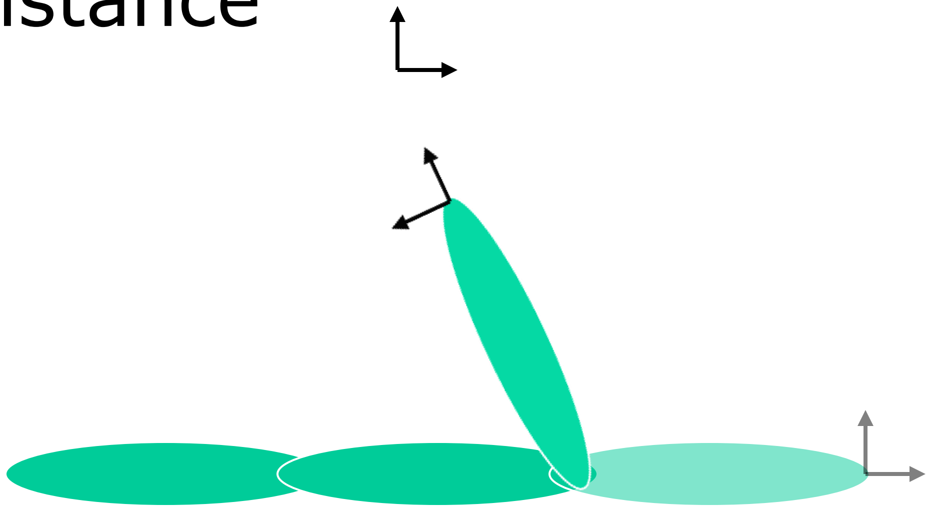
Cyclic Coordinate Descent

- Minimize distance



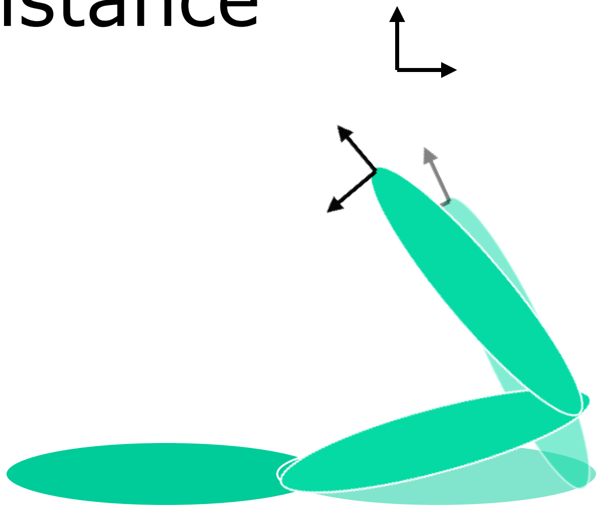
Cyclic Coordinate Descent

- Minimize distance



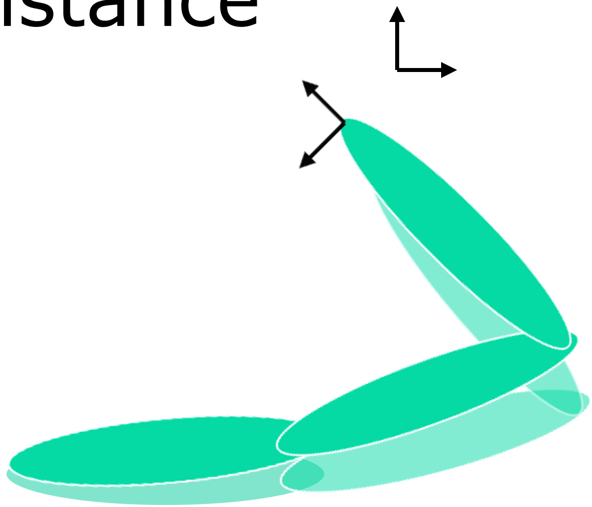
Cyclic Coordinate Descent

- Minimize distance



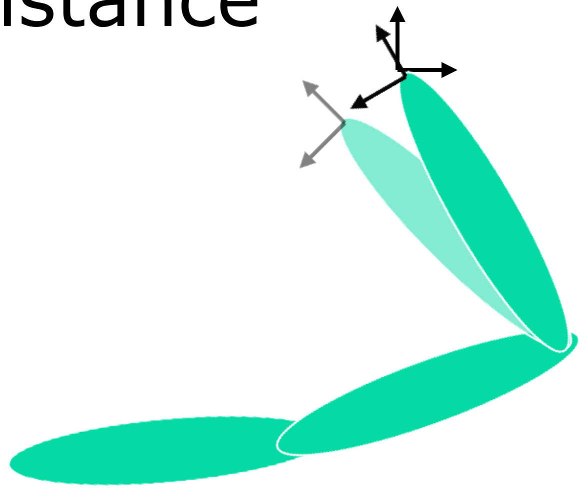
Cyclic Coordinate Descent

- Minimize distance



Cyclic Coordinate Descent

- Minimize distance



Cyclic Coordinate Descent

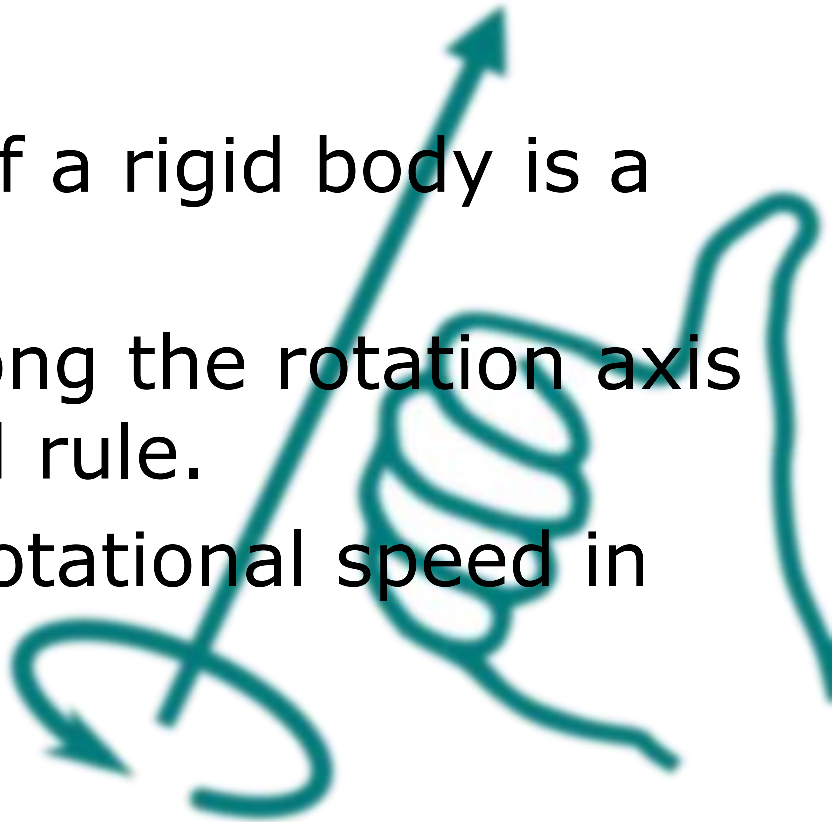
- Pros:
  - Easy to implement
  - Linear time complexity ($O(n)$ for n DoFs)
- Cons:
  - May converge violently (requires relaxation).
  - Not fit for multiple simultaneous constraints.

Velocity-based IK

- Satisfy positional constraints by solving joint speeds that move the end-effectors towards their desired poses.
- Best solution for interactive animation:
  - Offers control over jerkiness.
  - Ideal for following a moving target.

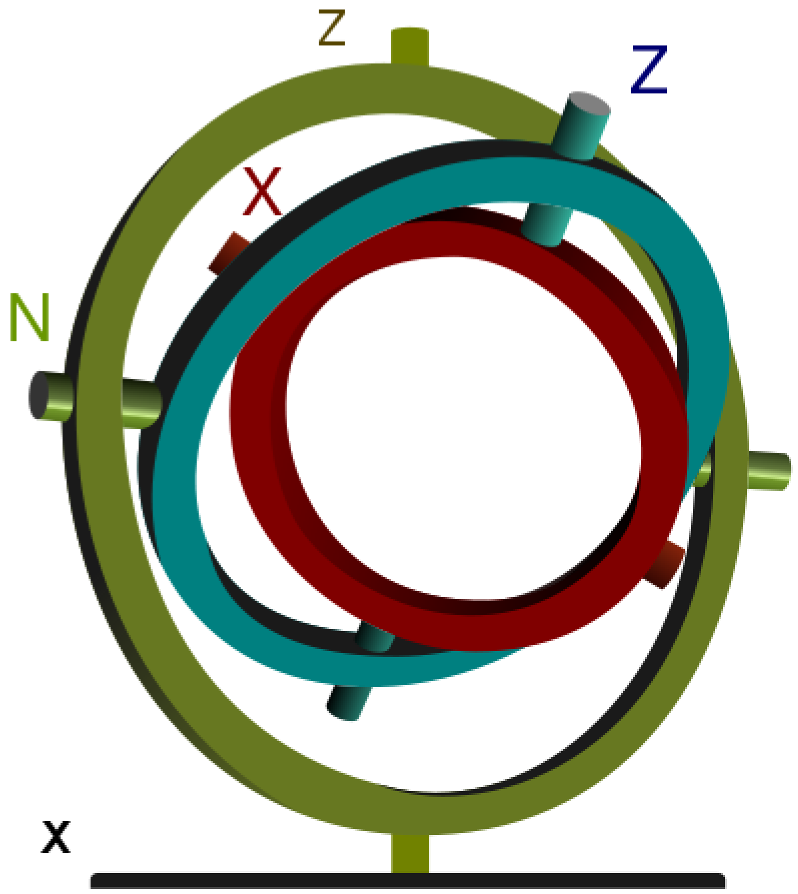
Angular Velocity

- The *angular velocity* of a rigid body is a 3D vector.
- Its *direction* points along the rotation axis following the right-hand rule.
- Its *magnitude* is the rotational speed in radians per second.



Angular Velocity

- Angular velocity is a proper vector:
- The angular velocity of a link is the sum of all joint velocities along the chain.



Joint Velocity

- The directions of the joint axes $\mathbf{a}_i$ form a vector space for the angular velocity $\boldsymbol{\omega}$ of an end-effector:

$$\boldsymbol{\omega} = \mathbf{a}_1 \dot{\theta}_1 + \cdots + \mathbf{a}_n \dot{\theta}_n$$

- Here, $\dot{\theta}_i$ are the joint speeds in radians per second.

Joint Velocity

- In matrix notation this looks like

$$\boldsymbol{\omega} = \begin{pmatrix} \vdots & & \vdots \\ \mathbf{a}_1 & \cdots & \mathbf{a}_n \\ \vdots & & \vdots \end{pmatrix} \begin{pmatrix} \dot{\theta}_1 \\ \vdots \\ \dot{\theta}_n \end{pmatrix}$$

- The matrix columns are the *n* joint axes.

Joint Axis Direction

- For $\hat{\mathbf{q}}_i = \mathbf{q}_i + \mathbf{q}_i{}'\varepsilon$, link i's pose expressed in the world frame, the direction of the joint axis is the local z-axis in world coordinates:

$$\mathbf{a}_i = \mathbf{q}_i \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \mathbf{q}_i{}^*$$

Free & Fixed Joint Parameters

- Move the fixed joint parameters over to the left-hand side

$$\boldsymbol{\omega} - (\mathbf{a}_1\dot{\theta}_1 + \cdots + \mathbf{a}_l\dot{\theta}_l) = \begin{pmatrix} \vdots & & \vdots \\ \mathbf{a}_{l+1} & \cdots & \mathbf{a}_n \\ \vdots & & \vdots \end{pmatrix} \begin{pmatrix} \dot{\theta}_{l+1} \\ \vdots \\ \dot{\theta}_n \end{pmatrix}$$

- Here, only $\dot{\theta}_{l+1}$ to $\dot{\theta}_n$ are variables.

Jacobian Matrix

- The remaining matrix expresses the influence of changing joint speeds on the angular velocity of the end-effector (link *n*).
- This is in fact the *Jacobian* matrix.
- #rows = #constrained DoFs.
- #colums = #free joint parameters.

No Inverse

- The Jacobian matrix generally does not have an inverse.
- Often the matrix is not square, and thus not invertible.
- Square Jacobians may not be invertible, since they can have dependent columns.

Too Few Variables

- The constraints fix more DoFs than there are variables:

$$J = \begin{pmatrix} \vdots & \vdots \\ \mathbf{a}_{n-1} & \mathbf{a}_n \\ \vdots & \vdots \end{pmatrix}$$

- Likely, no solution exists. We settle for a best-fit solution.

Too Many Variables

- The constraints fix fewer DoFs than there are variables:

$$J = \begin{pmatrix} \vdots & & \vdots \\ \mathbf{a}_{n-3} & \cdots & \mathbf{a}_n \\ \vdots & & \vdots \end{pmatrix}$$

- Infinitely many solutions may exist. We seek the one with the lowest joint speeds.

Jacobian Transpose

- Quick-and-dirty solver:

$$\begin{pmatrix} \dot{\theta}_{l+1} \\ \vdots \\ \dot{\theta}_n \end{pmatrix} \cong J^{\mathrm{T}} \left(\boldsymbol{\omega} - \left(\mathbf{a}_1 \dot{\theta}_1 + \cdots + \mathbf{a}_l \dot{\theta}_l \right) \right)$$

- Good for getting the right trend, but no best-fit and no lowest joint speeds.

Jacobian Transpose (cont'd)

- Needs a relaxation factor β to home in on the sweet spot:

$$\begin{pmatrix} \dot{\theta}_{l+1} \\ \vdots \\ \dot{\theta}_n \end{pmatrix} = \beta J^{\mathrm{T}} \left(\boldsymbol{\omega} - \left(\mathbf{a}_1 \dot{\theta}_1 + \cdots + \mathbf{a}_l \dot{\theta}_l \right) \right)$$

- Still, convergence is slow and unpredictable.

Pseudoinverse

- The Moore-Penrose *pseudoinverse* J^+ is defined as

$$J^+ = \left(J^{\mathrm{T}}J\right)^{-1}J^{\mathrm{T}}$$

- Giving: $\begin{pmatrix} \dot{\theta}_{l+1} \\ \vdots \\ \dot{\theta}_n \end{pmatrix} = J^+\left(\boldsymbol{\omega} - \left(\mathbf{a}_1\dot{\theta}_1 + \cdots + \mathbf{a}_l\dot{\theta}_l\right)\right)$

Pseudoinverse (cont'd)

- If no solution exists, returns a best-fit (least-squares) solution.
- If infinitely many solutions exist, returns the least-norm (lowest speed) solution.
- If an inverse exists, the pseudoinverse is the inverse.

Computing the Pseudoinverse

- J^+can be computed using open-source linear-algebra packages (Eigen, Armadillo+ LAPACK).
- Cubic complexity! ($O(n^3)$ for *n* variables)
- Decimate into smaller Jacobians, rather than solve one huge Jacobian.

Orientation Alignment

- End-effector's world frame $\hat{\mathbf{q}}_n$ is constrained to align with a target frame $\hat{\mathbf{q}}_t$.
- For moving targets, end-effector's angular velocity equals the target frame's: $\boldsymbol{\omega}_n = \boldsymbol{\omega}_t$.
- Correct the alignment error by adding a correcting angular velocity to the target's.

Orientation Alignment (cont'd)

- For aligning an end-effector's orientation to a moving target, solve:

Corrects error

$$\begin{pmatrix} \dot{\theta}_{l+1} \\ \vdots \\ \dot{\theta}_n \end{pmatrix} = J^+\left(\boldsymbol{\omega}_t - \left(\mathbf{a}_1\dot{\theta}_1 + \cdots + \mathbf{a}_l\dot{\theta}_l\right) - \boldsymbol{\omega}_e\right)$$

Orientation Alignment (cont'd)

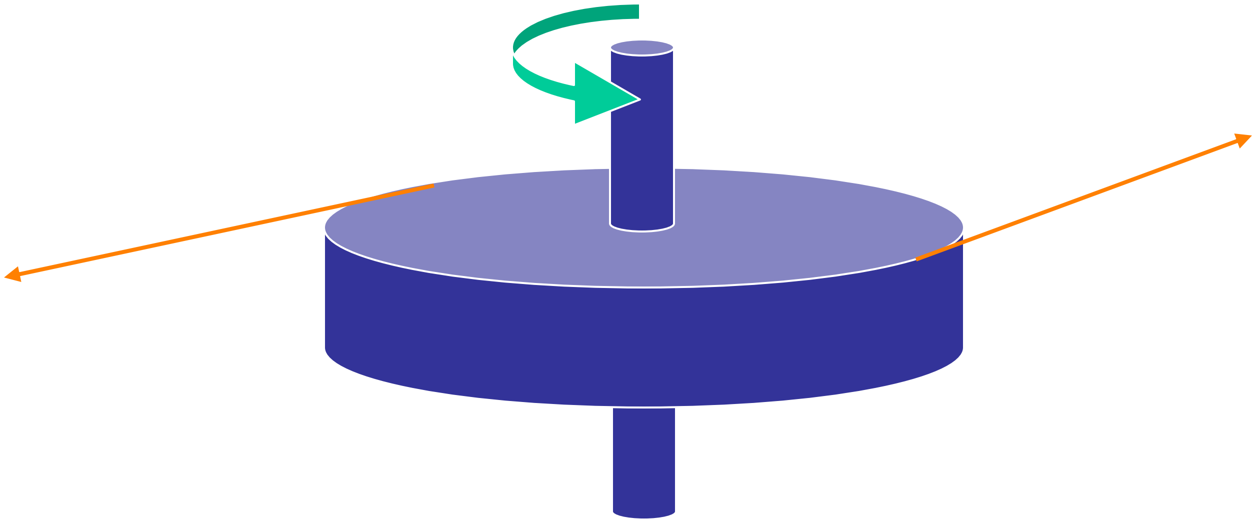
- As target velocity $\boldsymbol{\omega}_e$, we choose the vector part of $\beta\frac{2}{h}\mathbf{q}_n{\mathbf{q}_t}^*$.
- Here, quaternions $\mathbf{q}_n$ and $\mathbf{q}_t$ are the orientations of resp. end-effector and target, and h is the time interval.
- Factor $\beta(< 1)$ relaxes correction speed.

There's a Twist…

- Quaternions $\mathbf{q}$ and $-\mathbf{q}$ represent the same orientation.
- For computing $\boldsymbol{\omega}_e$, make sure that $\mathbf{q}_n$ and $\mathbf{q}_t$ point in the same direction ($\mathbf{q}_n \cdot \mathbf{q}_t > 0$).
- If not, then negate either $\mathbf{q}_n$ or $\mathbf{q}_t$ to take the shortest way home.

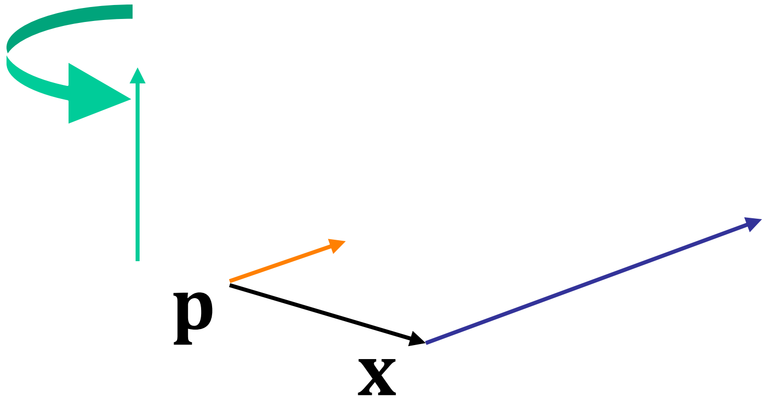
Linear Velocity

- Linear velocity, unlike angular velocity, is bound to a point in space:



Linear Velocity (cont'd)

- Given angular velocity $\boldsymbol{\omega}$, and linear velocity $\mathbf{v}$ at point $\mathbf{p}$, the linear velocity at an arbitrary point $\mathbf{x}$ is $\mathbf{v} + \boldsymbol{\omega} \times (\mathbf{x} - \mathbf{p})$.



Plücker Coordinates

- Angular and linear velocity of a link are combined into a single entity represented by a dual vector (aka *Plücker coordinates*):

$$\hat{\mathbf{v}} = \boldsymbol{\omega} + \mathbf{v}^{O}\varepsilon$$

- Here, $\mathbf{v}^{O}$ is the linear velocity at the origin of the coordinate frame.

Transforming Plücker Coordinates

- Plücker coordinates are transformed from one coordinate frame to another using the dual quaternion "sandwich" product:

$$\hat{\mathbf{q}}\hat{\mathbf{v}}\hat{\mathbf{q}}^*$$

- Returns the image of velocity $\hat{\mathbf{v}}$ after *rigid transformation* by unit dual quaternion $\hat{\mathbf{q}}$.

Deja Vu?

- The (combined) velocity of a link is the sum of all joint velocities along the chain.
- The joint axes $\hat{\mathbf{a}}_i$ form a vector space for the velocity $\hat{\mathbf{v}}$ of an end-effector:

$$\hat{\mathbf{v}} = \hat{\mathbf{a}}_1\dot{\theta}_1 + \cdots + \hat{\mathbf{a}}_n\dot{\theta}_n$$

- Here, $\dot{\theta}_i$ are the revolute and prismatic joint speeds.

Deja Vu? (cont'd)

- For $\hat{\mathbf{q}}_i = \mathbf{q}_i + \mathbf{q}_i{}'\varepsilon$, link *i*'s pose expressed in the world frame, the joint axis is the local z-axis in world coordinates:
- For a revolute:

$$\hat{\mathbf{a}}_i = \hat{\mathbf{q}}_i \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \hat{\mathbf{q}}_i{}^*$$

For a prismatic:

$$\hat{\mathbf{a}}_i = \hat{\mathbf{q}}_i \begin{pmatrix} 0 \\ 0 \\ \varepsilon \end{pmatrix} \hat{\mathbf{q}}_i{}^*$$

Deja Vu? (cont'd)

- End-effector's world frame $\hat{\mathbf{q}}_n$ is constrained to lock onto a target frame $\hat{\mathbf{q}}_t$.
- For moving targets, end-effector's velocity equals the target frame's: $\hat{\mathbf{v}}_n = \hat{\mathbf{v}}_t$.
- To correct the error, we add the dual vector part of $\beta\frac{2}{h}\hat{\mathbf{q}}_n\hat{\mathbf{q}}_t{}^*$ to the target velocity.

Emotion FX Demo

References

- K. Shoemake. *Plücker Coordinate Tutorial.* Ray Tracing News, Vol. 11, No. 1
- R. Featherstone. *Spatial Vectors and Rigid Body Dynamics.* http://royfeatherstone.org/spatial.
- L. Kavan et al. Skinning with dual quaternions. *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2007.
- G. van den Bergen. *Math for Game Programmers: Dual Numbers*. GDC 2013 Tutorial.

Open-Source Code

- *Eigen: A C++ Linear Algebra Library*. http://eigen.tuxfamily.org. License: MPL2
- *Armadillo: C++ Linear Algebra Library*. http://arma.sourceforge.net. License: MPL2
- *LAPACK – Linear Algebra PACKage*. http://www.netlib.org/lapack. License: BSD
- *MoTo C++ template library* (dual quaternion code) https://code.google.com/p/motion-toolkit/. License: MIT

Thank You!

My pursuits can be traced on:

- Web: http://www.dtecta.com
- Twitter: @dtecta
- Or just mail me: gino@dtecta.com