Computer Animation
Algorithms and Techniques

Kinematic Linkages
Hierarchical Modeling

Relative motion → Parent-child relationship
Simplifies motion specification

Constrains motion → Reduces dimensionality
Modeling & animating hierarchies

3 aspects
1. Linkages & Joints – the relationships
2. Data structure – how to represent such a hierarchy
3. Converting local coordinate frames into global space
Some terms

**Joint** – allowed relative motion & parameters
**Joint Limits** – limit on valid joint angle values
**Link** – object involved in relative motion
**Linkage** – entire joint-link hierarchy
**Armature** – same as linkage
**End effector** – most distant link in linkage
**Articulation variable** – parameter of motion associated with joint
**Pose** – configuration of linkage using given set of joint angles
**Pose vector** – complete set of joint angles for linkage

**Arc** – of a tree data structure – corresponds to a joint
**Node** – of a tree data structure – corresponds to a link
Use of hierarchies in animation

**Forward Kinematics (FK)**
- animator specifies values of articulation variables
- global transform for each linkage is computed

**Inverse Kinematics (IK)**
- animator specifies final desired global transform for end effector (and possibly other linkages)
- Values of articulation variables are computed
Joints – relative movement

Revolute joint

Prismatic joint
**Complex Joints**

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

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

- **Zero-length linkages**
Hierarchical structure

Articulated figure

Abstract hierarchical representation

Tree structure

root node

root arc

link

joint
Tree structure

Node $i$ contains
- a transformation to be applied to object data to position it so its point of rotation is at the origin (optional)
- object data

Arc $i$ contains
- a constant transformation of Link$_i$ to its neutral position relative to Link$_{i-1}$
- a variable transformation responsible for articulating Link$_i$
Tree structure

Original definition of root object \( (Link_0) \)

Root object \( (Link_0) \) transformed (translated and scaled) by \( T_0 \) to some known location in global space

Original definition of \( Link_1 \)

\( Link_1 \) transformed by \( T_1 \) to its position relative to untransformed \( Link_0 \)

Original definition of \( Link_{1,1} \)

\( Link_{1,1} \) transformed by \( T_{1,1} \) to its position relative to untransformed \( Link_1 \)
Tree structure

$T_0$ (global position and orientation)

Data for Link$_0$ (the root)

$T_1$ (transformation of Link$_1$ relative to Link$_0$)

Data for Link$_1$

$T_{1.1}$ (transformation of Link$_{1.1}$ relative to Link$_1$)

Data for Link$_{1.1}$
Relative movement
Relative movement

Data for \( \text{Link}_0 \) (the root)

\[ T_0 \]

\[ T_1 \]
\[ R_1(\theta_1) \]

Data for \( \text{Link}_1 \)

\[ T_{1,1} \]
\[ R_{1,1}(\theta_{1,1}) \]

Data for \( \text{Link}_{1,1} \)
Tree structure
Tree structure

Data for Link₀ (the root)

\[ T₀ \]

\[ T₂ \]

\[ R₂(θ₂) \]

Data for Link₂

\[ T₂₁ \]

\[ R₂₁(θ₂₁) \]

Data for Link₂₁

\[ T₁ \]

\[ R₁(θ₁) \]

Data for Link₁

\[ T₁₁ \]

\[ R₁₁(θ₁₁) \]

Data for Link₁₁
Implementation note

Nodes & arcs

NODE
Pointer to data
Data transformation
Pointer to arcs

ARC
Transform of one next node relative to parent node
Articulation transform
Pointer to node
Implementation note
Representing arbitrary number of children with fixed-length data structure

Use array of pointers to children
In node, arcPtr[]

Node points to first child
Each child points to sibling
Last sibling points to NULL
In node: arcPtr for 1st child
In arc: arcPtr for sibling
traverse (arcPtr, matrix)
{
    // concatenate arc matrices
    matrix = matrix * arcPtr->Lmatrix;
    matrix = matrix * arcPtr->Amatrix;

    // get node and transform data
    nodePtr = arcPtr->nodePtr;
    push (matrix);
    matrix = matrix * nodePtr->matrix;
    aData = transformData(matrix, dataPtr);
    draw(aData);
    matrix = pop();

    // process children
    if (nodePtr->arcPtr != NULL) {
        nextArcPtr = nodePtr->arcPtr;
        while (nextArcPtr != NULL) {
            push(matrix);
            traverse(nextArcPtr, matrix);
            matrix = pop();
            nextArcPtr = nextArcPtr->arcPtr;
        }
    }
}
OpenGL Single linkage

```c
glPushMatrix();
  For (i=0; i<NUMDOFS; i++) {
    glRotatef(a[i],axis[i][0], axis[i][1], axis[i][2]);
    if (linkLen[i] != 0.0) {
      draw_linkage(linkLen[i]);
      glTranslatef(0.0,linkLen[i],0.0);
    }
  }
glPopMatrix();
```

OpenGL concatenates matrices

A[i] – joint angle
Axis[i] – joint axis
linkLen[i] – length of link

Computer Animation
Rick Parent
Inverse kinematics

Given goal position (and orientation) for end effector

Compute internal joint angles

If simple enough => analytic solution
Else => numeric iterative solution
Inverse kinematics - spaces

Configuration space
Reachable workspace
Dextrous workspace
Analytic inverse kinematics

\[
\cos(\theta_T) = \frac{X}{\sqrt{x^2 + y^2}}
\]

\[
\theta_T = \cos^{-1}\left(\frac{X}{\sqrt{x^2 + y^2}}\right)
\]

\[
\cos(\theta_1 - \theta_T) = \frac{L_1^2 + x^2 + y^2 - L_2^2}{2L_1\sqrt{x^2 + y^2}} \quad \text{(cosine rule)}
\]

\[
\theta_1 = \cos^{-1}\left(\frac{L_1^2 + x^2 + y^2 - L_2^2}{2L_1\sqrt{x^2 + y^2}}\right) + \theta_T
\]

\[
\cos(180 - \theta_2) = -\cos(\theta_2) = \frac{L_1^2 + L_2^2 - (x^2 + y^2)}{2L_1L_2} \quad \text{(cosine rule)}
\]

\[
\theta_2 = \cos^{-1}\left(-\frac{L_1^2 + L_2^2 - (x^2 + y^2)}{2L_1L_2}\right)
\]
IK - numeric

If linkage is too complex to solve analytically
E.g., human arm is typically modeled as 3-1-3 or 3-2-2 linkage

Solve iteratively – numerically solve for step toward goal

Desired change from this specific pose
Compute set of changes to the pose to effect that change
IK math notation

\[ y_1 = f_1(x_1, x_2, x_3, x_4, x_5, x_6) \]
\[ y_2 = f_2(x_1, x_2, x_3, x_4, x_5, x_6) \]
\[ y_3 = f_3(x_1, x_2, x_3, x_4, x_5, x_6) \]
\[ y_4 = f_4(x_1, x_2, x_3, x_4, x_5, x_6) \]
\[ y_5 = f_5(x_1, x_2, x_3, x_4, x_5, x_6) \]
\[ y_6 = f_6(x_1, x_2, x_3, x_4, x_5, x_6) \]

\[ Y = F(X) \]
IK - chain rule

\[
\frac{dy_i}{dt} = \frac{\partial f_i}{\partial x_1} \frac{dx_1}{dt} + \frac{\partial f_i}{\partial x_2} \frac{dx_2}{dt} + \frac{\partial f_i}{\partial x_3} \frac{dx_3}{dt} + \frac{\partial f_i}{\partial x_4} \frac{dx_4}{dt} + \frac{\partial f_i}{\partial x_5} \frac{dx_5}{dt} + \frac{\partial f_i}{\partial x_6} \frac{dx_6}{dt}
\]

\[
\dot{Y} = \frac{\partial F}{\partial X} \dot{X}
\]
Inverse Kinematics - Jacobian

\[ \dot{Y} = \frac{\partial F}{\partial X} \dot{X} \]

\[ V = J(\theta) \dot{\theta} \]

Desired motion of end effector

Unknown change in articulation variables

The Jacobian is the matrix relating the two: it’s a function of current variable values.
Inverse Kinematics - Jacobian

\[ V = J(\theta) \dot{\theta} \]

\[ V = \begin{bmatrix} v_x, v_y, v_z, \omega_x, \omega_y, \omega_z \end{bmatrix} \]

\[ \dot{\theta} = \begin{bmatrix} \dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, \dot{\theta}_4, \dot{\theta}_5, \dot{\theta}_6 \end{bmatrix} \]

\[ J = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_1} & \frac{\partial p_x}{\partial \theta_2} & \cdots & \frac{\partial p_x}{\partial \theta_6} \\ \frac{\partial p_y}{\partial \theta_1} & \frac{\partial p_y}{\partial \theta_2} & \cdots & \frac{\partial p_y}{\partial \theta_6} \\ \frac{\partial p_z}{\partial \theta_1} & \frac{\partial p_z}{\partial \theta_2} & \cdots & \frac{\partial p_z}{\partial \theta_6} \\ \frac{\partial \alpha_x}{\partial \theta_1} & \frac{\partial \alpha_x}{\partial \theta_2} & \cdots & \frac{\partial \alpha_x}{\partial \theta_6} \\ \frac{\partial \alpha_y}{\partial \theta_1} & \frac{\partial \alpha_y}{\partial \theta_2} & \cdots & \frac{\partial \alpha_y}{\partial \theta_6} \\ \frac{\partial \alpha_z}{\partial \theta_1} & \frac{\partial \alpha_z}{\partial \theta_2} & \cdots & \frac{\partial \alpha_z}{\partial \theta_6} \end{bmatrix} \]

Change in position
Change in orientation

Computer Animation
Rick Parent
IK – computing the Jacobian

(need to convert to global coordinates)

a) Angular velocity, $\omega_j$

b) Linear velocity, $Z_i \times (E - J_i)$

$E$ — end effector
$J_i$ — $i$th joint
$Z_i$ — $i$th joint axis
$\omega_j$ — angular velocity of $i$th joint

Change in orientation

Change in position

Only valid instantaneously
IK - configuration
IK – compute positional change vectors induced by changes in joint angles

Instantaneous positional change vectors

Desired change vector

One approach to IK computes linear combination of change vectors that equal desired vector
IK – compute position and axis of joints

Set identity matrix

for (i=0; i<NUMDOFS; i++) {
    record_transformed_joint(i)
    glRotate(angle[i],axis[i][0],axis[i][1],axis[i][2]);
    append_rotation(angle[i],axis[i][0],axis[i][1],axis[i][2]);
    if (linkLen[i] != 0) {
        draw_linkage(linkLen[i]);
        glTranslatef(0.0,linkLen[i],0.0);
        append_translation(0,linkLen[i],0);
    }
}
record_endEffector();
**IK – append rotation**

If joint axis is:

one of major axes: 3 cases of simple rotation

Arbitrary axis – angle-axis to matrix conversion

**IK – append translation**

Form translation matrix

**Matrix**

Transformed coordinate system
Position
Transforms axis of rotation
IK – record joint information

Joint position – last column of matrix

Joint coordinate system – upper left 3x3 submatrix

Joint axis – transform local joint axis vector by matrix
IK - singularity

Some singular configurations are not so easily recognizable.
Near singular configurations are also problematic – why?
Inverse Kinematics - Numeric

Given
• Current configuration
• Goal position/orientation

Determine
• Goal vector
• Positions & local coordinate systems of interior joints (in global coordinates)
• Jacobian

\[ V = J(\theta)\dot{\theta} \]

Is in same form as more recognizable:

\[ Ax = b \]

Solve & take small step – or clamp acceleration or clamp velocity

Repeat until:
• Within epsilon of goal
• Stuck in some configuration
• Taking too long
Solving

If J square, compute inverse, $J^{-1}$

If J not square, usually under-constrained: more DoFs than constraints
Requires use of pseudo-inverse of Jacobian

\[
V = J \dot{\theta}
\]

\[
J^T V = J^T J \dot{\theta}
\]

\[
(J^T J)^{-1} J^T V = (J^T J)^{-1} J^T J \dot{\theta}
\]

\[
J^+ V = \dot{\theta}
\]
Solving

Avoid direct computation of inverse by substitution solving $Ax=B$ form, then substituting back

$V = J\dot{\theta}$

$J^T \beta = \dot{\theta}$

$V = JJ^T \beta$

$J^T \beta = \dot{\theta}$
IK - Jacobian solution
IK - Jacobian solution - problem

When goal is out of reach
Bizarre undulations can occur
As armature tries to reach the unreachable

Add a damping factor
**IK – Jacobian w/ damped least squares**

Undamped form: \[ \dot{\theta} = \left( J^T J \right)^{-1} J^T V \]

Damped form with user parameter:

\[ \dot{\theta} = J^T \left( J J^T + \lambda^2 I \right)^{-1} V \]
**IK – Jacobian w/ control term**

Physical systems (i.e. robotics) and synthetic character simulation (e.g., human figure) have limits on joint values.

IK allows joint angle to have any value.

Difficult (computationally expensive) to incorporate hard constraints on joint values.

Take advantage of redundant manipulators - Allow user to set parameter that urges DOF to a certain value.

Does not enforce joint limit constraints, but can be used to keep joint angles at mid-range values.
IK - Jacobian w/ control term

\[
\dot{\theta} = J^+ V + (J^+ J - I)^{-1} z
\]

\[
z = \alpha_i (\theta_i - \theta_{ci})^2
\]

\[
\begin{align*}
V &= J \dot{\theta} \\
V &= J (J^+ J - I) z \\
V &= (JJ^+ J - J) z \\
V &= 0z \\
V &= 0
\end{align*}
\]

Change to the pose parameter in the form of the control term adds nothing to the velocity
IK - Jacobian w/ control term

All bias to 0
Top gains = \{0.1, 0.5, 0.1\}
Bottom gains = \{0.1, 0.1, 0.5\}

\[
\dot{\theta} = J^+ V + (J^+ J - I)^{-1} z
\]

\[
z = \alpha_i (\theta_i - \theta_{ci})^2
\]
IK - alternate Jacobian

Jacobian formulated to pull the goal toward the end effector

Use same method to form Jacobian but use goal coordinates instead of end-effector coordinates
IK - Transpose of the Jacobian

Compute how much the change vector contributes to the desired change vector:

Project joint change vector onto desired change vector

Dot product of joint change vector and desired change vector => Transpose of the Jacobian
**IK - Transpose of the Jacobian**

\[ J^T V = \dot{\theta} \]

\[
J^T = \begin{bmatrix}
\frac{\partial p_x}{\partial \theta_1} & \frac{\partial p_y}{\partial \theta_1} & \cdots & \frac{\partial \alpha_z}{\partial \theta_1} \\
\frac{\partial p_x}{\partial \theta_2} & \cdots & & \\
\cdots & & & \\
\frac{\partial p_x}{\partial \theta_6} & \frac{\partial \alpha_z}{\partial \theta_6}
\end{bmatrix}
\]

\[
V = \begin{bmatrix}
v_x \\
v_y \\
v_z \\
\omega_x \\
\omega_y \\
\omega_z
\end{bmatrix}
\]
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
IK - cyclic coordinate descent

In 3D, a bit more computation is needed
IK - 3D cyclic coordinate descent

First – goal has to be projected onto plane defined by axis (normal to plane) and EF

Second – determine angle at joint
IK – cyclic coordinate descent – 3D

Other orderings of processing joints are possible

Because of its procedural nature
• Lends itself to enforcing joint limits
• Easy to clamp angular velocity
Inverse kinematics - review

Analytic method
Forming the Jacobian
Numeric solutions
  Pseudo-inverse of the Jacobian
  $J^+$ with damping
  $J^+$ with control term
  Alternative Jacobian
  Transpose of the Jacobian
  Cyclic Coordinate Descent (CCD)
Inverse kinematics - orientation

Change in orientation at end-effector is same as change at joint
Inverse kinematics – orientation

How to represent orientation (at goal, at end-effector)?
How to compute difference between orientations?
How to represent desired change in orientation in V vector?
How to incorporate into IK solution?

Matrix representation: \( M_g, M_{ef} \)

Difference \( M_d = M_{ef}^{-1} M_g \)

Use scaled axis of rotation: \( \theta(a_x, a_y, a_z) \):
  - Extract quaternion from \( M_d \)
  - Extract (scaled) axis from quaternion

E.g., use Jacobian Transpose method:
Use projection of scaled joint axis onto extracted axis