Inverse Kinematics

Animating Characters

Many editing techniques rely on either:

- Interactive posing
- Putting constraints on bodyparts' positions and orientations (includes mapping sensor positions to body motion)
- Optimizing over poses or sequences of poses

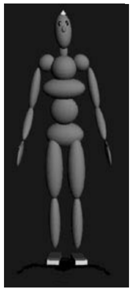
All three tasks require inverse kinematics

Goal

Several different approaches to IK, varying in capability, complexity, and robustness

We want to be able to choose the right kind for any particular motion editing task/tool

IK Problem Definition



1) Create a handle on body
   - position or orientation

2) Pull on the handle

3) IK figures out how joint angles should change

More Formally

Let:

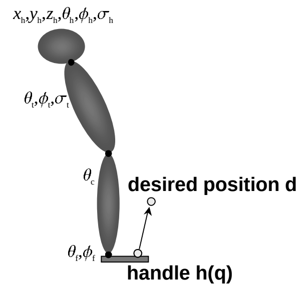
- **q** — **actor *state vector* (joint bundle)**
- **C(q)** — **constraint functions that pull handles**

Then:

solve for q such that C(q) = 0

What's a Constraint?

q=[$x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f$ **]**



Can be rich, complicated

But most common is very simple:

Position constraint just sets difference of two vectors to zero:

C(q) = h(q) - d = 0

The *Real* problem & Approaches

The IK problem is usually very underspecified

- many solutions
- most bad (unnatural)
- how do we find a good one?

Two main approaches:

- Geometric algorithms
- Optimization/Differential based algorithms

Geometric

Use geometric relationships, trig, heuristics

Pros:

- fast, reproducible results

Cons:

- proprietary; no established methodology
- hard to generalize to multiple, interacting constraints
- cannot be integrated into dynamics systems

Optimization Algorithms

Main Idea: use a numerical metric to specify which solutions are good

metric - a function of state q (and/or state velocity) that measures a quantity we'd like to minimize

Example

Some commonly used metrics:

- joint stiffnesses
- minimal power consumption
- minimal deviation from "rest" pose

Problem statement:

Minimize metric G(q)
subject to satisfying C(q) = 0

An Approach to Optimization

If G(q) is quadratic, can use Sequential Quadratic Programming (SQP)

- original problem highly non-linear, thus difficult
- SQP breaks it into sequence of quadratic subproblems
- iteratively improve an initial guess at solution
- How?

Search and Step

Use constraints and metric to find direction Δq that moves joints closer to constraints

Then $q_{new} = q + a\,\Delta q$ where

$$\min_{a} \; C(q + a\,\Delta q)$$

Iterate whole process until C(q) is minimized

Breaking it Down

Performing the integration $q_{new} = q + a\,\Delta q$ is easy (Brent's alg. to find a)

Finding a good Δq is much trickier

Enter Derivatives.

What Derivatives Give Us

We want:

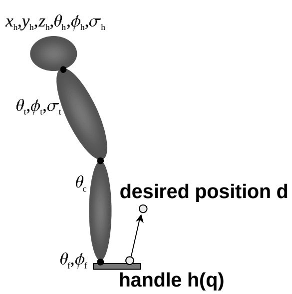
- a direction in which to move joints so that constraint handles move towards goals

Constraint Derivatives tell us:

- in which direction constraint handles move if joints move

Constraint derivatives

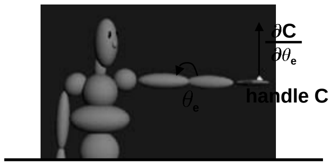
q=[$x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f$ **]**



C(q) = h(q) - d = 0

$$\frac{\partial C(\mathbf{q})}{\partial \mathbf{q}} = \frac{\partial h(\mathbf{q})}{\partial \mathbf{q}}$$

Jacobian Matrix



$$\frac{\partial \mathbf{C}}{\partial \mathbf{q}}$$

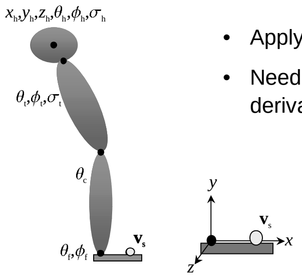
		θ_e	
		0	
		1	
		0	

Can compute Jacobian for each constraint / handle

Value of Jacobian depends on current state

Jacobian linearly relates joint angle velocity to constraint velocity

Computing Derivatives



- Apply the chain rule
- Need to know how to compute derivatives for each transformation

$$\mathbf{v}_w = \mathbf{T}(x_h, y_h, z_h)\mathbf{R}(\theta_h, \phi_h, \sigma_h)\,\mathbf{TR}(\theta_t, \phi_t, \sigma_t)\;\mathbf{T}\mathbf{R}(\theta_c)\mathbf{TR}(\theta_f, \phi_f)\,\mathbf{v}_s$$

$$\frac{\partial \mathbf{v}_w}{\partial \theta_c} = \mathbf{T}(x_h, y_h, z_h)\mathbf{R}(\theta_h, \phi_h, \sigma_h)\,\mathbf{TR}(\theta_t, \phi_t, \sigma_t)\;\mathbf{T}\,\frac{\partial \mathbf{R}(\theta_c)}{\partial \theta_c}\,\mathbf{TR}(\theta_f, \phi_f)\,\mathbf{v}_s$$

Jacobian Matrix

Have efficient techniques for computing Jacobians

But how do we use it to compute Δq ?

- Constrained optimization
- Unconstrained optimization

Constrained Optimization

- Many formulations (*e.g.* Lagrangian, Lagrange Multipliers)
- All involve solving a linear system comprised of Jacobians, the quadratic metric, and other quantities

$$\begin{array}{ll} \underset{\mathbf{q}}{\text{minimize}} & G(\mathbf{q}) \\ \text{subject to} & \mathbf{C}(\mathbf{q}) \end{array}$$

Result: constraints satisfied (if possible), metric minimized subject to constraints

Constrained Performance

Pros:

- Enforces constraints exactly
- Has a good “feel” in interactive dragging
- Quadratic convergence

Cons:

- A Dark Art to master
- near-singular configurations cause instability

Unconstrained Optimization

Main Idea: treat each constraint as a separate metric, then just minimize combined sum of all individual metrics, plus the original

- Many names: penalty method, soft constraints, Jacobian Transpose
- physical analogy: placing damped springs on all constraints
  – *each spring pulls on constraint with force proportional to violation*

$$G'(q) = G(q) + \sum C(q)^2$$

Unconstrained Performance

Pros:

- Simple, no linear system to solve, each iteration is fast
- near-singular configurations less of problem

Cons:

- Constraints fight against each other and original metric
- sloppy interactive dragging (can't maintain constraints)
- linear convergence

Why Does Convergence Matter?

Trying to drive C(q) to zero:

# Iterations	1	2	3	4	5
quadratic **C(q)**					
linear **C(q)**					
linear/quadratic					

Recap and Conclusions

Inverse Kinematics

- Geometric algorithms
  – *fast, predictable for special purpose needs*
  – *don't generalize to multiple constraints or physics*
- Optimization-based algorithms
  – *Constrained vs. unconstrained methods*

Recap and Conclusions

Constrained optimization

- achieves true constrained minimum of metric
- solid feel and fast convergence
- involves arcane math
- near-singular configurations must be tamed

Recap and Conclusions

Unconstrained optimization

- near-singular configurations manageable
- spongy feel
- poor convergence
- easy to get penalty method up and running