Cornell University
CS 4620 Program #2: Character Animation
Forward and Inverse Kinematics

out: Tuesday 28 September 2010
due: Tuesday 19 October 2010

Author: Prof. Doug James

In this second programming assignment, you will implement forward and inverse kinematic methods to animate and pose 3D articulated systems. You should be able to animate the joint angles to demonstrate the character moving (forward kinematics), as well as pose the character it by dragging points on the screen (inverse kinematics).

1 Getting Started

The mathematical foundations of forward and inverse kinematics were discussed in lectures, and we refer you to that material. For starter code, please use that of the first programming assignment (PA#1 "Hello OpenGL"), or a stripped-down version of your submission. You are free to use any 3rd-party library to help you solve square full-rank linear systems. In Java, you may use the `vecmath` library for 3D matrix/vector algebra, and a LU-based matrix solver will be provided.

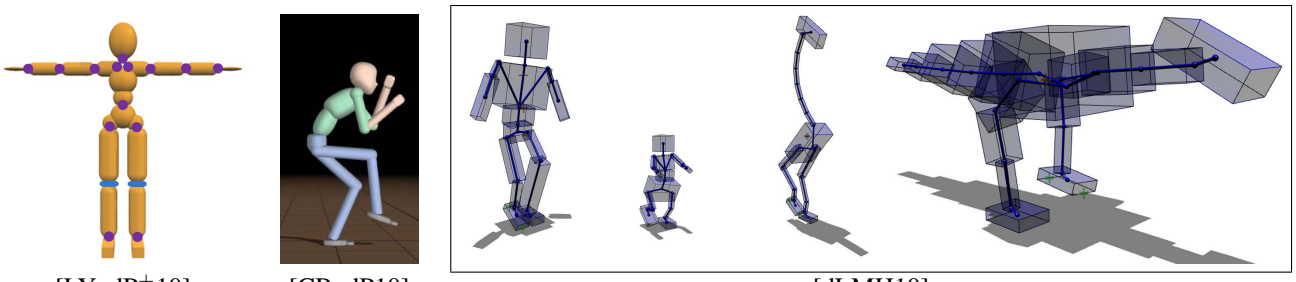
2 What To Do

The assignment has the following steps:

2.1 Geometric and kinematic modeling

You need only use simple geometric representations for your 3D "characters." For example, you can make simple models using colored boxes, ellipsoids, cylinders, etc. (see Figure 1—feel free to model more interesting shapes (or import them from Turbo Squid or model them in Blender) once you finish the core parts of the assignment. The first step is to model each rigid-body of the character in the rest pose ($\theta_j = 0$). One you have defined each rigid link's geometry, you can define each joint's center position, $\boldsymbol{c}_j$, and rotation axis $\boldsymbol{u}_j$. You need only use revolute joints in this assignment. You should also infer the tree structure of the kinematic model, and define approach parent/child data structures needed for the kinematics algorithms, e.g., $\{\mu_i\}$ and $\{\lambda_i\}$. Since we assume that each joint is located at the origin in each link's own (body) frame, you can translate each link's geometry so that its joint center is at the origin of its body coordinate frame. You may do this modeling in whatever manner you wish, e.g., hard-coded numbers, text file input, via a custom mouse interface in "edit mode," or a separate program, etc.

What to model: You should build computer models of at least two "characters": (1) planar serial-chain manipulator(s), and (2) either a model of your hand or an articulated figure of your choosing. If you are working in a group, then you should make at least one other example. There is no limit on how many examples you provide, e.g., to debug your implementation you may wish to model a number of simple planar manipulators (1 link, 2 links, 12 links).



[LYvdP+10] [CBvdP10] [dLMH10]

Figure 1: **Simplified character geometry** is sufficient for the assignment, and is commonly used in the animation community for proof-of-concept demonstrations of kinematics and locomotion algorithms.

2.2 Forward kinematics

Once you have a tree-structured model defined you can start animating it using forward kinematics. At each frame, after you adjust the joint angles, $\{\theta_j\}$, you should update the body-to-world transformation matrices (using the recursive root-to-tip evaluation algorithm described in class), then draw the articulated geometry. Build a simple joint-angle animation to demonstrate a functioning forward kinematics implementation, while in "forward-kinematics mode."

2.3 Inverse kinematics solver

As discussed in class, you should implement an inverse kinematics solver based on a damped least-squares solver (a.k.a. regularized least-squares, or ridge regression) [Wam86, Bus04]. At its heart your implementation will solve least-squares problems of the form,

$$\mathbf{J}\,\Delta\boldsymbol{\theta} = \Delta\mathbf{p} \tag{1}$$

where $\Delta\boldsymbol{\theta} \in \mathbb{R}^N$ are the changes in N joint angles, $\Delta\mathbf{p} \in \mathbb{R}^{M=3m}$ are the changes in m constraint positions (in the world frame), and $\mathbf{J} = \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{\theta}^T} \in \mathbb{R}^{M\times N}$ is the Jacobian matrix. Your implementation should use appropriate over-determined ($M > N$) and under-determined ($M < N$) solvers depending on the number of position constraints, e.g., your damped least-squares solver (for $M < N$) will generate angle updates of the form

$$\Delta\boldsymbol{\theta} = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T + \delta\boldsymbol{I})^{-1}\Delta\mathbf{p}, \tag{2}$$

where $\delta > 0$ is a user-specified regularization parameter to avoid singular matrices. Note that for small values of δ, e.g., $\delta < 10^{-6}\|\mathbf{J}\mathbf{J}^T\|_F$, you will definitely require double-precision calculations. *What values of δ give the best results for your characters?*

2.3.1 Interactive character posing

You will implement a mouse-based interface which allows you to interactively (and intuitively) pose your character while in "inverse-kinematics mode." You should be able to add/remove position (pin) constraints on your character, and drag these constraints around to pose it. This involves a number of subtasks:

- *Picking* should be implemented so that given a mouse click you can determine which, if any, link has been selected and at what position. Added position constraints should be drawn on the screen.
- *Dragging* should allow you to move constraints around in the plane perpendicular to the view direction—the model-view matrix will help you find this direction and plane. In this way, it should be intuitive to fix a body in space, such as the feet, and move other parts around, e.g., to make a character wave.
- *IK solution* should proceed in an incremental manner by updating the joints a small amount at each time step, which involves recomputing/solving the Jacobian problem each time.

2.3.2 Keyboard interface requirements

To help evaluate your submissions, please ensure your application supports the following keystrokes:

- **Number keys (1,2,...) select the kinematic model:** For example, "1" might select a 2D serial-chain example with two links, "2" a chain with 10 links, "3" a hand model, and "4" a bug creature.
- **Spacebar toggles between forward and inverse kinematics:** In forward-kinematics mode, the model can be self-animating based on available animations, with mouse clicks ignored (except for view controls). In inverse-kinematics mode, the model only moves when the user clicks-and-drags on the object to adjust the joint angles.

2.4 More advanced things to try (if you are in a group)

Key-frame Animation: Use your IK implementation to produce keyframes, then play them back by interpolating key-framed joint angles. You can use linear interpolation for this assignment. Dump frames of your creation to make a cool video highlight.

Joint weights: You will notice that some IK solutions tend to produce somewhat unnatural motions/poses, which tend to move large parts of the body more than might be necessary for low-energy movements. As discussed in class, you can weight the joints so that certain joint changes have additional cost, so that instead of finding the min-norm solution with minimal $\|\Delta\boldsymbol{\theta}\|_2$, you will obtain one that minimizes $\|\boldsymbol{W}\Delta\boldsymbol{\theta}\|_2$ for some suitable weighting matrix, $\boldsymbol{W} = \mathsf{diag}(w_i)$.

Balance constraints: One annoying thing about your IK solver when posing characters is that they have no sense of balance. Fortunately, with the click of a button, you can use your solver to enable character balance. One approximation is that the center of mass should not move outside the support of the characters feet. A sufficient and really simple approximation is that the character's center of mass, when projected onto the floor, should not move. Given the center of mass position of each link, $\bar{\boldsymbol{p}}_i$ and approximate mass m_i, the center of mass of the character is

$$\bar{\boldsymbol{p}} = \sum_i w_i\, \bar{\boldsymbol{p}}_i.$$

The linearized condition that the center of mass not move at all is the 3 constraints,

$$\Delta\bar{\boldsymbol{p}} = \sum_i w_i\, \Delta\bar{\boldsymbol{p}}_i = \sum_i w_i\, \mathbf{J}_i\, \Delta\boldsymbol{\theta} \equiv \bar{\mathbf{J}}\, \Delta\boldsymbol{\theta},$$

where $\Delta\bar{\boldsymbol{p}}_i = \mathbf{J}_i\Delta\boldsymbol{\theta}$. If the floor is parallel to the xy-plane, then you only need to constrain the xy coordinates, by incorporating two equations,

$$\Delta\bar{\boldsymbol{p}}_{xy} = \bar{\mathbf{J}}_{xy}\, \Delta\boldsymbol{\theta},$$

into your existing linear system, i.e., solve the augmented least-squares problem,

$$\begin{bmatrix} \mathbf{J} \\ \bar{\mathbf{J}}_{xy} \end{bmatrix} \Delta\boldsymbol{\theta} = \begin{pmatrix} \Delta\mathbf{p} \\ \Delta\bar{\boldsymbol{p}}_{xy} \end{pmatrix}.$$

Build a walking controller: Using your FK/IK implementations, you should be able to animate some interesting locomotion behaviors. An alternative to key-frame animation, is to build a controller which when given user inputs, will execute specified motions. For example, try animating a multi-legged creature, such as a person or a spider robot, walking at a fixed rate.

Other joint types: Feel free to explore joints beyond the simple revolute type. For example, you may wish to use spherical or prismatic joints. In each case, the joint transformation matrix is different, but the overall kinematic/matrix structures are analogous.

Other IK solvers: The literature abounds with IK solvers. Try a different one, and compare it to the traditional damped least-squares solver.

Skinned characters: Soft character deformations are not always well approximated by piecewise rigid-body motions. The latter assume that mesh vertex i is transformed by a single bone transformation,

$$\boldsymbol{p}_i' = \boldsymbol{T}_b \, \boldsymbol{p}_i. \tag{3}$$

Linear blend skinning (a.k.a. skeletal subspace deformation (SSD), matrix palette skinning, weighted enveloping, etc.) approximates the deformation using a linear combination of bone transformations via

$$\boldsymbol{p}_i' = \sum_b w_{ib} \, \boldsymbol{T}_b \, \boldsymbol{p}_i \tag{4}$$

where the per-vertex bone weights, w_{ib}, typically only weight to a small number of bones, and also sum to one, $\sum_b w_{ib} = 1, \forall i$. In practice, non-negative weights are chosen that weight vertices heavily to their nearby bones. At runtime, when the joint angles change the bone transformations are updated, and used to deform the mesh vertices (often in a vertex shader) via (4).

3 Submission and FAQ

A FAQ page will be kept on the course website detailing any new questions and their answers brought to the attention of the course staff.

Working in Groups: You can work by yourself or with one other person. If you work in a group, it is expected that your assignment will in some way do a little more than if you just worked on your own.

Submission Checklist: Submit your assignment as a zip file via CMS. You need to submit:

- **Documented code:** Include all of your code and libraries, with usage instructions. Your code should be reasonably documented and be readable/understandable by the TAs. If the TAs are not able to run and use your program, they will be unable to grade it. Try to avoid using obscure packages or OS-dependent libraries, especially for C++ implementations.
- **Brief report:** Include a description of what you've attempt, special features you've implemented, and any instructions on how to run/use your program. In compliance with Cornell's Code of Academic Integrity, you must document and reference any external sources or discussions that were used to achieve your results.
- **Video highlight!** Include a video that highlights what you've achieved. The video footage should be in the default starter-code resolution, and be no longer than 10 seconds. We will concatenate all of the class videos together to provide a summary of your work.
- **Any additional results:** Please include additional information, pictures, videos, that you believe help the TAs understand what you've achived.

Evaluation: Your work will be evaluated on how well you demonstrate proficiency with the requested kinematics functionality, the quality of the submitted code and documentation, and on how exceptional your overall project is.

References

[Bus04] Samuel R. Buss. Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods. *University of California, San Diego, Typeset manuscript*, 2004.

[CBvdP10] Stelian Coros, Philippe Beaudoin, and Michiel van de Panne. Generalized biped walking control. *ACM Transctions on Graphics*, 29(4):Article 130, 2010.

[dLMH10] Martin de Lasa, Igor Mordatch, and Aaron Hertzmann. Feature-Based Locomotion Controllers. *ACM Transactions on Graphics*, 29(3), 2010.

[LYvdP+10] Libin Liu, KangKang Yin, Michiel van de Panne, Tianjia Shao, and Weiwei Xu. Sampling-based contact-rich motion control. *ACM Transctions on Graphics*, 29(4):Article 128, 2010.

[Wam86] C.W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):93–101, 1986.