# Assignment 2: Inverse Kinematics

Release Date: Tuesday, September 15, 2009
Due Date: Thursday, October 1, 2009, 11:59pm
Grading Value: 15%

**Overview:**
Kinematics describes the motion of a hierarchical skeleton structure, animations can be thought of as driving the kinematics in a game. While forward kinematics computes world space geometric descriptions based on joint DOF values, inverse kinematics could help us compute the vector of joint DOFs that will cause the end effector to reach some desired goal state.

In this assignment, you are required to implement two ways (Cyclic-Coordinate Descent and Pseudoinverse of Jacobian) of doing inverse kinematics for an articulated character and create a demonstration that clearly shows the differences between the two techniques. We will also examine how IK works in Maya.

**Requirement:**
Implement Cyclic-Coordinate Descent Method.
Implement Pseudoinverse of Jacobian Method.
Compare those two methods and report the difference you observe in your presentation slides/website.
Load .asf file into Maya and try to add IK to the skeleton's legs and/or arms. Play with the IK handles you added.
Report your observation of the difference of Maya's IK to the methods of your implementation.

**Deliverables:**
You are required to submit your code and documentation of how to run it, the features you have implemented and the ones you have not. Please make sure your code could run on one of the 5336 machines.
You need to submit one video of your animation (two methods of IK from your code). The video should contain the incremental movement the character makes to a target.
You need to submit a .mb file with the skeleton you load from .asf file and the IKs you add to it.
You also need to prepare a 10 minutes presentation and submit your slides or website to describe your goals, techniques you used, as well as your results.
Your handin directory is at /afs/cs.cmu.edu/academic/class/15464-f09-users/andrewid/Assign2.
Note that you will be randomly selected to demo/discuss at least 1 of your 3 projects in class during the semester.

**Implementation:**
We will use the vector: $\boldsymbol{\theta} = [\boldsymbol{\theta}_1 \ \boldsymbol{\theta}_2 \ \cdots \ \boldsymbol{\theta}_M]$ to represent the array of M joint DOF values, and the vector $\mathbf{e} = [\mathbf{e}_1 \ \mathbf{e}_2 \ \cdots \ \mathbf{e}_N]$ to represent an array of N DOFs that describes the end effector in world space. If you are using .asf skeleton file, the size of $\boldsymbol{\theta}$ will depends on the joints you choose, and the size of $\mathbf{e}$ will be 3 since we are only concerned with the end effector position.

The forward kinematics f() computes the world space end effector DOFs from the joint DOFS:

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

Our goal of inverse kinematics is to compute the inverse of it:

$$\boldsymbol{\theta} = \mathbf{f}^{-1}(\mathbf{e})$$

**1. Cyclic-Coordinate Descent (CCD)**

CCD is a simple way to solve inverse kinematics. CCD deals with each joint individually. Although it is not as mathematically grouned as Jacobian, it's much simpler to implement and less expensive.

You can read these two articles "Oh My God, I Inverted Kine!" and "Making Kine More Flexible" for a thorough explanation of this method.

**2. Pseudoinverse of the Jacobian.**

A Jacobian is a vector derivative respect to another vector.

Assume $\boldsymbol{\theta}$ represents the current joint DOFs and $\mathbf{e}$ represents the current end effector DOFs, the forward kinematics function is

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

It maps a vector to another vector. Forward kinematics is nonlinear; it involves sin's and cos's of the input variables. Jacobian is a linear approximation of f(), it is a matrix of partial derivatives—one partial derivative for each combination of components of the vectors. We can only use the Jacobian as an approximation that is valid near the current configuration. So we must repeat the process of computing a Jacobian and taking a small step towards the goal until we get close enough.

Let's use $\mathbf{g}$ to represent the goal DOFs, here is the algorithm we can use to compute inverse kinematics:

while ($\mathbf{e}$ is too far from $\mathbf{g}$){

compute the Jacobian matrix J

compute the pseudoinverse of the Jacobian matrix— $\mathbf{J}^{+}$

compute change in joint DOFs: $\Delta\boldsymbol{\theta} = \mathbf{J}^{+} \cdot \Delta\mathbf{e}$

apply the change to DOFs, move a small step of $\boldsymbol{\alpha}\Delta\boldsymbol{\theta}$: $\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{\alpha}\Delta\boldsymbol{\theta}$

}

**Computing the Jacobian Matirx:**

For any given joint pose vector $\boldsymbol{\theta}$, we can explicitly compute the individual components of the Jacobian matrix.

We are just concerned with the end effector position. Therefore, $\mathbf{e}$ is just a 3D vector representing the end effector position in world space. The Jacobian will be a 3×N matrix (N is the number of DOFs).

Let's examine one column in the Jacobian matrix.

$$\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}_\mathbf{i}} = \left[\frac{\partial \mathbf{e}_\mathbf{x}}{\partial \boldsymbol{\theta}_\mathbf{i}} \frac{\partial \mathbf{e}_\mathbf{y}}{\partial \boldsymbol{\theta}_\mathbf{i}} \frac{\partial \mathbf{e}_\mathbf{z}}{\partial \boldsymbol{\theta}_\mathbf{i}}\right]^\mathbf{T}$$

We can fill this column numerically:

Add small $\Delta\boldsymbol{\theta}$ to $\boldsymbol{\theta}_\mathbf{i}$. (Don't forget to call angles.to_pose() after this change.)

Get the change of the end effector in world space:

$\Delta\mathbf{e} = \mathbf{e}' - \mathbf{e}$. (You can use the WorldBone class now.)

So we have:

$$\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}_\mathbf{i}} \approx \frac{\Delta\mathbf{e}}{\Delta\boldsymbol{\theta}} = \left[\frac{\Delta\mathbf{e}_\mathbf{x}}{\Delta\boldsymbol{\theta}} \frac{\Delta\mathbf{e}_\mathbf{y}}{\Delta\boldsymbol{\theta}} \frac{\Delta\mathbf{e}_\mathbf{z}}{\Delta\boldsymbol{\theta}}\right]^\mathbf{T}$$

Note: The columns of the Jacobian matrix could also be solved analytically. See more about this in the Extra Credit section.

**Computing the Pseudoinverse of the Jacobian Matrix:**

Now we have our Jacobian matrix and the equation $\Delta\mathbf{e} = \mathbf{J} \cdot \Delta\boldsymbol{\theta}$

If we can have the inverse of the Jacobian matrix, we can solve $\Delta\boldsymbol{\theta}$ from $\Delta\mathbf{e}$ : $\mathbf{J}^{-1} \cdot \Delta\mathbf{e} = \Delta\boldsymbol{\theta}$

But sometimes the Jacobian matrix is not invertable, so we need to use the Pseudoinverse of the Jacobian Matrix $\mathbf{J}^+$:

$$\Delta\mathbf{e} = \mathbf{J} \cdot \Delta\boldsymbol{\theta}$$

$$\mathbf{J}^\mathbf{T} \cdot \Delta\mathbf{e} = \mathbf{J}^\mathbf{T}\mathbf{J} \cdot \Delta\boldsymbol{\theta}$$

$$(\mathbf{J}^\mathbf{T}\mathbf{J})^{-1}\mathbf{J}^\mathbf{T} \cdot \Delta\mathbf{e} = (\mathbf{J}^\mathbf{T}\mathbf{J})^{-1}(\mathbf{J}^\mathbf{T}\mathbf{J}) \cdot \Delta\boldsymbol{\theta}$$

$$(\mathbf{J}^\mathbf{T}\mathbf{J})^{-1}\mathbf{J}^\mathbf{T} \cdot \Delta\mathbf{e} = \Delta\boldsymbol{\theta}$$

$$\mathbf{J}^+ \cdot \Delta\mathbf{e} = \Delta\boldsymbol{\theta}$$

$$\mathbf{J}^+ = (\mathbf{J}^\mathbf{T}\mathbf{J})^{-1}\mathbf{J}^\mathbf{T}$$

Now we should be able to follow the algorithm provided at the beginning to solve inverse kinematics.

There is also some explanation in Rick Parent's "Computer Animation" book, Section 5.3.

You will need to find some way of inverting a matrix. You can use the safe matrix inversion code in this directory: /afs/cs.cmu.edu/academic/class/15464-f09/asst2/

Here's an example of how to use it:

```
#include "matrix.hpp"

matrix *m1 = new matrix(2, 3);
m1->setValue(1.0, 0, 0);
m1->setValue(2.0, 0, 1);
m1->setValue(3.0, 0, 2);
m1->setValue(4.0, 1, 0);
m1->setValue(5.0, 1, 1);
m1->setValue(6.0, 1, 2);

matrix *m1_pinv = new matrix(3, 2);
```

```
int r = m1->invertMatrix(m1_pinv, SVD_TOL);

matrix *res = new matrix(2, 2);
m1->computeMatrixMul(m1_pinv, res);

res->printMatrix();
std::cout << std::endl;

delete m1;
m1 = NULL;
delete m1_pinv;
m1_pinv = NULL;
delete res;
res = NULL;
```

**One Last Thing:**

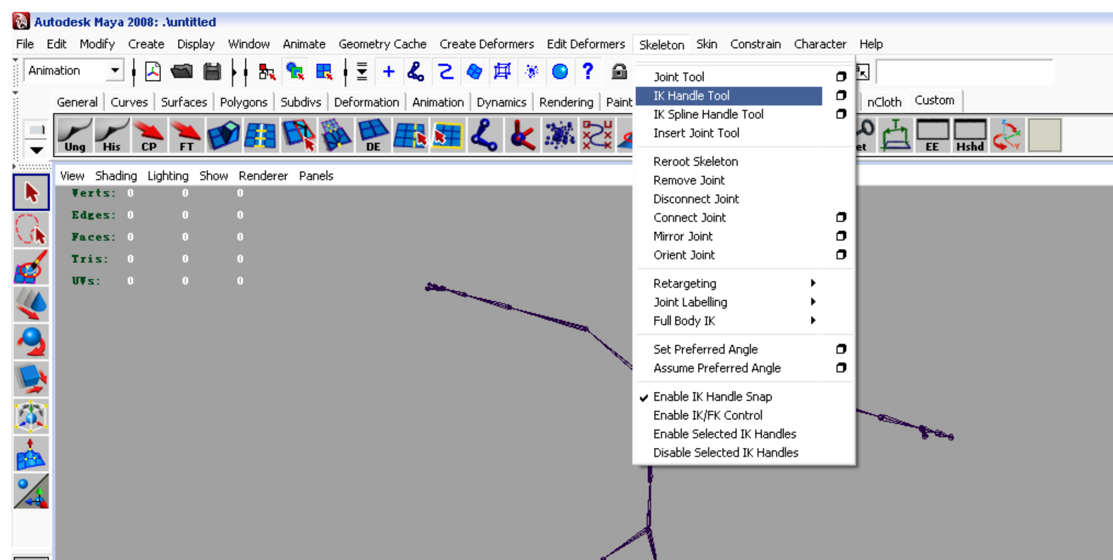
A numerical solution to IK is not very desirable in any implementation as it should be much slower than an analytical solution. However, it is always a good first cut to make sure that you have the math right in an analytical approach.

## 3. IK in Maya

Set Maya back to Y-up.

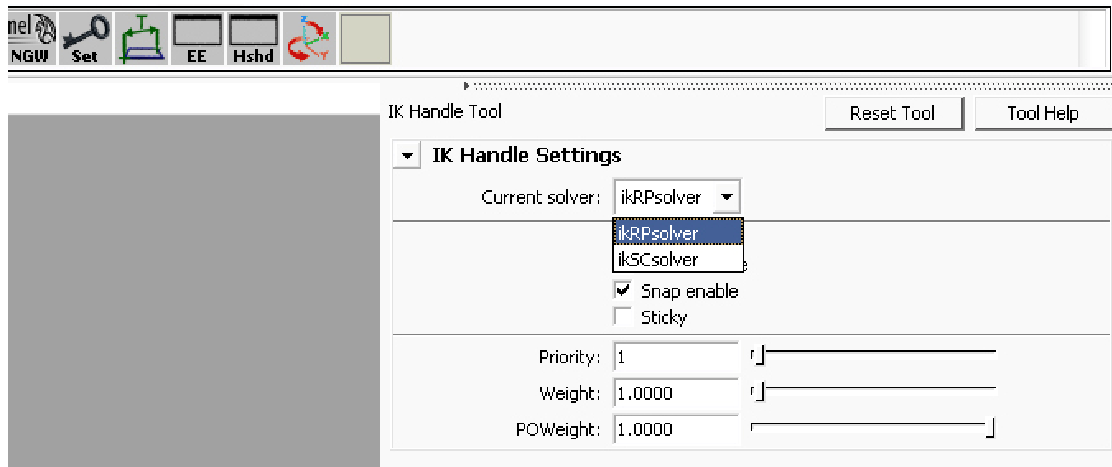
You can follow the instruction from this link http://mocap.cs.cmu.edu/asfamcmel/asfamcmel.php to load the skeleton from .asf file into Maya.

Click the box at the right side of “IK Handle Tool”



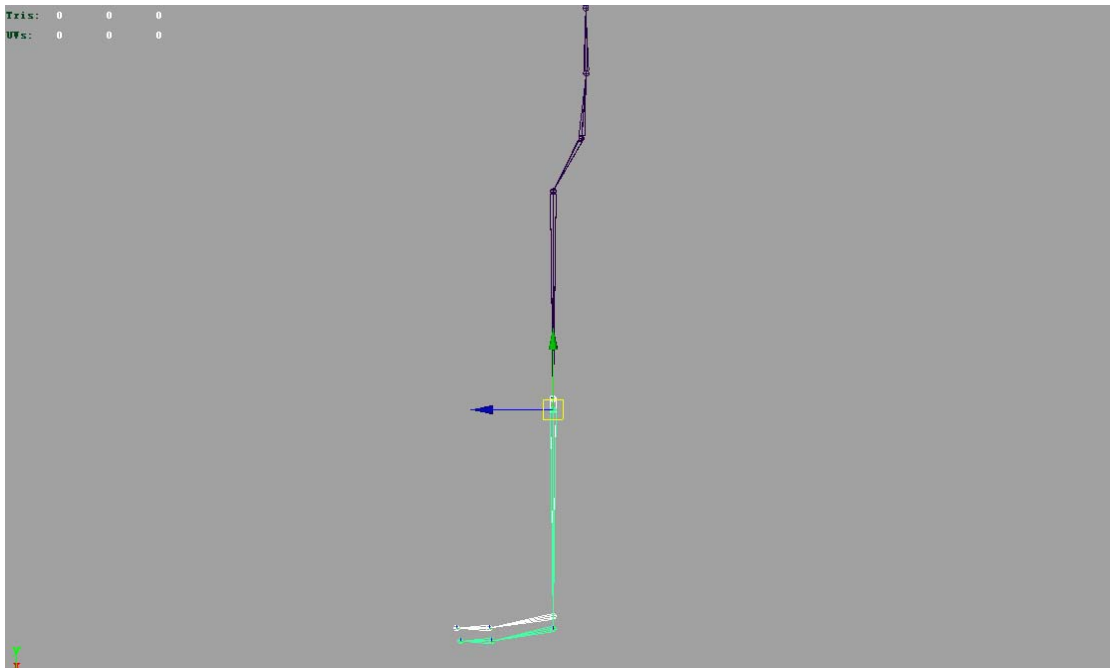

Change Current solver to “ikRPsolver” if it’s not your default solver. (The single chain IK handle's end effector tries to reach the position and the orientation of its IK handle, whereas the rotate

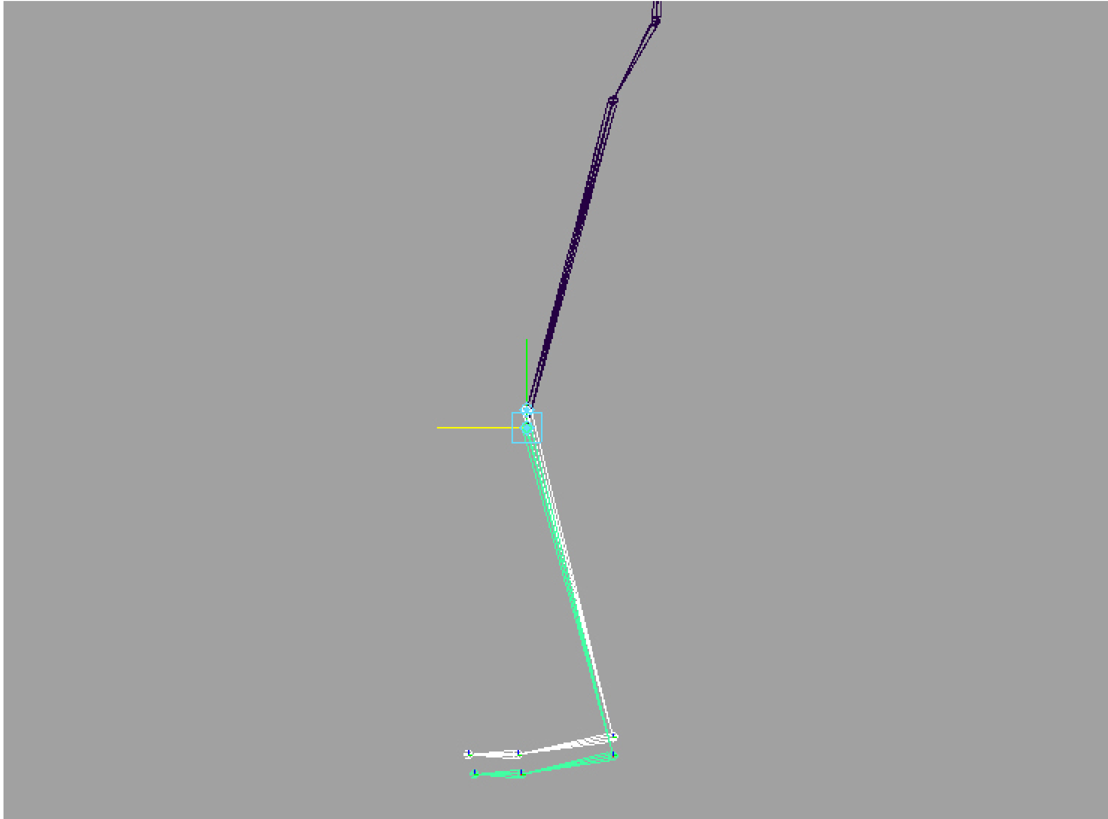
plane IK handle's end effector only tries to reach the position of its IK handle.)




Here's an example of how to set IK to a skeleton:

Go to side view from the viewport, select the joints of the knees, use the move tool to move them forward before we set IK. (We need to bend the knees a little bit so Maya could know which way it should bend). You can press "d" while moving, so you can move only the knee joints other than the whole hierarchy under those two joints.





You can go back to perspective view and select "IK Handle Tool", click one thigh joint, then click its corresponding ankle joint. Now you should have your IK handle set up for one leg.

**Grading:**

This assignment will be graded more on your implementation. Please make sure you are doing the right math.

Your video, report and presentation will also be a part of your grade.

**Tips:**

Start early. This assignment is much more code and math-intensive than our 1st assignment.

**Extra Credit:**

1. Interactive interface: Create a click-and-drag environment so the user can drag the target points. The character's pose updates continuously to match the targets.

2. Compute the Jacobian Matrix analytically:

$$\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}_\mathbf{i}} = \left[\frac{\partial \mathbf{e}_\mathbf{x}}{\partial \boldsymbol{\theta}_\mathbf{i}} \; \frac{\partial \mathbf{e}_\mathbf{y}}{\partial \boldsymbol{\theta}_\mathbf{i}} \; \frac{\partial \mathbf{e}_\mathbf{z}}{\partial \boldsymbol{\theta}_\mathbf{i}}\right]^\mathbf{T} = \left(\mathbf{a}_\mathbf{i}' \times (\mathbf{e} - \mathbf{r}_\mathbf{i}')\right)$$

$\mathbf{a}_\mathbf{i}'$ is the rotation axis of this DOF in world space, $\mathbf{r}_\mathbf{i}'$ is the DOF's world space offset. The "axis" along with the "dof" in the asf file specify the number of joints and their axis directions. To compute the rotation axis in world space you will need this information. You need to transform them by their parent joint's world matrix.

3. Allow user to click on any part of the character.

4. Anything else that makes posing the character more intuitive!