Inverse Kinematics of human model (Using Jacobian pseudoinverse)

Jeehoon Hyun, Computer Science (2016-14251)

1. About the human model

The human model is the same structure of PA2's 'Trial001.bvh' file. For this model, I implemented IK where the end effector is the left hand of the human body. I implemented two cases of IK, where the first case is when the shoulder location is fixed, and the IK is implemented just on the arm. The second case is when the location of the thorax bone(3rd child of the pubic bone) is fixed. For each cases, the forward kinematics looks like this.

- First Case (5 DOFs)

```
(x,y,z) = (Root to Shoulder transformation) * Shoulder_T *
Shoulder_Rx(q_0) * Shoulder_Ry(q_1) *
Shoulder_Rz(q_2) * Elbow_T * Elbow_Rx(q_3) * Hand_T *
Hand_Rx(q_4) * Position
```

-Second Case (8 DOFs)

```
(x,y,z) =(Root to thorax transformation) * Thorax_T *
Thorax_Rx(q_0) * Thorax_Ry(q_1) *Thorax_Rz(q_2) *
Shoulder_T(q_3) Shoulder_Rx(q_4) * Shoulder_Ry(q_5) *
Shoulder_Rz(q_6) * Elbow_T * Elbow_Rx(q_7) * Hand_T *
Hand_Rx(q_8) * Position
```

2. Jacobian and its Pseudoinverse

$$J = \begin{pmatrix} \omega_1 \times p_1 & \omega_2 \times p_2 & \cdots & \omega_n \times p_n \\ \omega_1 & \omega_2 & \cdots & \omega_n \end{pmatrix}$$

For calculating the Jacobian matrix for each cases, I used the equation above that we learned in class. Each w_i represents the i_th axis of rotation in the global coordinates. p_i represents the vector from the i_th axis position to the goal position. Therefore, the row of each J was 6 in all cases. The number of columns was equal to the DOF of each case.

For overdetermined systems, the pseudoinverse of the jacobian matrix looks like this. This was used in my first case, where the Jacobian matrix was a 6*5 matrix.

J_***plus = ((J.transpose() * J).inverse()) * J.transpose();***

For underdetermined systems, the pseudoinverse of the jacobian matrix looks like this. This was used in my second case, where the Jacobian matrix was a 6*8 matrix.

J_plus = J.transpose() * (J * J.transpose()).inverse();

However, the professor told us that these types of pseudoinverses were really vulnerable to singularities. Instead, he recommended us to use the damped version of these.

Thus, instead of using ***(J * J.transpose()).inverse()*** or ***(J * J.transpose()).inverse()***, I used the damped version ***(J * J.transpose() + λ *λ *I).inverse()***. If we use the damped version, it is said that the singularity problem is avoidable. In my case, the damping constants were set as 0.1.

3. IK solving process

$$\mathbf{J}\dot{\boldsymbol{\theta}} = \dot{\mathbf{x}}$$

Because of the equation on the left, as long as the Jacobian pseudo inverse is obtainable, theta_dot is obtainable as well. Now, we add theta_dot iteratively to the theta we have until the end effector reaches the target position. The judge whether the end effector is close enough to the target position, I used 2 objective functions, which were positional errors and orientational errors. (Based on the slides of the presentation)

This is the most important part of my code, where the IK problem is solved. Without loss of generality, we will only analyze the code of the first case, where the shoulder location is fixed and only the arm is moving. This is the solveIKArm() function, located in the BoneRig class.

```
bool BoneRig::solveIKArm(const glm::vec3& target, const glm::quat& ori_d) {
   … (OMITTED)
   while(true){
      //We get the end effector position here
      endEffectorP = getEndEffectorP();

      //We get the end effector orientation here
      endEffectorOri = getEndEffectorQ();

      //Calculate the positional error
      errorP = target - endEffectorP;
      x_dot = getx_dot(deltaTime, errorP, endEffectorOri, ori_d);

      //Get the Jacobian matrix of current state
      J = getJacobianArm(endEffectorP);

      //Pseudo inverse with damped constants to prevent singularity
      J_plus = ((J.transpose() * J + 0.1*0.1*Eigen::Matrix<float,5,5>::Identity()).inverse()) * J.transpose();
      deltaQ = J_plus * x_dot;
      oldQ_desired = Q_desired;
      Q_desired += deltaQ;

      //Compute the arm location with the updated theta vector
      setOrientationArm(Q_desired);

      //I used quaternions for representing the end effector orientation
      glm::quat qd= glm::inverse(ori_d) * endEffectorOri;

      //Calculate the orientational error
      errorQ = glm::length(glm::log(qd));

      //Calculate total error by adding those two errors.
      error = glm::length(errorP) * glm::length(errorP) + errorQ * errorQ;
      iter++;
      if(iter>= MaxIter){
         return false;
      }
      if(error < 0.1){
         return true;
      }
   }
}
```

4. Configuration
Just as PA2, I used glfw and the glad library(OpenGL ver 3.3) for programming. Furthermore, the math libraries were glm and Eigen. I used both because I read that glm was more stable the Eigen, but Eigen offered vectors and matrices whose sizes were bigger than 4.

I will not upload the makefiles this time since I have to present my program to my TA within the next couple of days.

Also, I would like to state the fact that my Shader.h and Camera.h were an opensource from the website learnopengl.com