# Introduction to Computer Animation HW3 Report

資工系 0716234 蕭彧

- Introduction/Motivation

What we want to implement in this assignment is also to simulate the action of the character. The difference from the 2nd assignment is that the method used in the second assignment is Forward Kinematics, and this time we use Inverse Kinematics. In the homework, we will calculate the Skeleton Pose through the Forward Kinematics that we did last time, and use the Rotation matrix, Axis, Euler angle to calculate the Jacobian Matrix we need. Through Pseudo Inverse, we can find the change in orientation, and then update the rotation angle of each Bone to simulate the entire human shape, so that the End position of the arm is closer to the Target Position.

- Fundamentals

  - Iterative IK

The goal of Inverse Kinematics is to calculate the joint DOFs that affect the end effector so that the end effector can reach the target position we set. In other words, Forward Kinematics is, after angle being given, then calculate the point to reach; on the contrary, Inverse Kinematics is to calculate the angle that should be changed after the point we want to reach being given. Since we hope to reach the target position step by step, we need to adjust the orientation to make the end effector close to the target position through continuous calculation and correction. So, we use the iterative method to calculate, through each step with small correction, and finally reach the goal.

    - effects of step:

      If the step is too large, the convergence will be fast, but relatively inaccurate; and if the step is small, the convergence will be slow, but relatively accurate.

    - distance_epsilon:

      This distance is the acceptable range for the error of the distance between the target position and the end effector. If it is set too large, the distance between the two will be relatively far. If it is set too small, it is easy for the end effector to be calculated all the time. Therefore, it is more careful to choose a suitable epsilon.

  - Jacobian Matrix

A Jacobian Matrix can be represented as the best linear approximation of a

multivariate vector function. In our homework, it represents the partial derivatives of the system we want to calculate. In other words, it defines how the end effector changes relatively instantaneously.

- Pseudo Inverse

  After calculating the above Jacobian Matrix, we know how the end effector affects the relationship of each angle in the x axis, y axis, z axis, and in order to find the change in orientation, we need to calculate the inverse matrix of the Jacobian Matrix and multiply it on V (the distance between target and end effector). However, it is very difficult to calculate the inverse Jacobian. Although we can use transpose, the method we choose to do is to adapt pseudo inverse and calculate J+ to replace the originally more difficult inverse.

- Orientation

  We can know the rotation angle of each bone (in the form of Euler angle), but in order to let the end effector reach the target position, we need to change these orientations and calculate our final orientation step by step in small steps.

- Implementation
    - leastSquareSolver

      In this home, I choose SVD to calculate the pseudo inverse. A matrix M can be decomposed to UΣV*, and the pseudo inverse can be constructed as $V\Sigma^{+}U^{*}$, where $\Sigma^{+}$ is the transpose of Σ with all of the elements in it become reciprocal. Use this pseudo inverse multiply by target to get the least square answer.

```
Eigen::VectorXf leastSquareSolver(const Eigen::Matrix3Xf& jacobian, const Eigen::Vector3f& target) {
  // TODO (find x which min(| jacobian * x - target |))
  // Hint:
  //   1. Linear algebra - least squares solution
  //   2. https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse#Construction
  // Note:
  //   1. SVD or other pseudo-inverse method is useful
  //   2. Some of them have some limitation, if you use that method you should check it.
  Eigen::VectorXf solution(jacobian.cols());
  solution.setZero();
  Eigen::JacobiSVD<Eigen::MatrixXf> svd(jacobian, Eigen::ComputeFullV | Eigen::ComputeFullU);
  const auto& singularValues = svd.singularValues();
  Eigen::MatrixXf singularValuesInv(jacobian.cols(), jacobian.rows());
  singularValuesInv.setZero();
  double toler = 1.e-6;
  for (int i = 0; i < singularValues.size(); i++) {
    if (singularValues(i) > toler)
      singularValuesInv(i, i) = 1.0f / singularValues(i);
    else
      singularValuesInv(i, i) = 0.f;
  }
  Eigen::MatrixXf jacobianInv = svd.matrixV() * singularValuesInv * svd.matrixU().transpose();
  solution = jacobianInv * target;
  return solution;
}
```

    - Iterative IK
        - Traverse the skeleton

          First, we need to traverse the skeleton from end to start to get a list of bones which we need to modify their euler angle later. There is a little tricky part in this process. When we use parent to traverse the skeleton, we might traverse to the root and then stop the process since the parent of root is nullptr. To cope with it, I first traverse the skeleton to check whether end bone can reach start bone without passing through root and set a flag to distinguish. By this pre-process I can successfully put the bones between end and start into the list.

```
bool flag = false;
// test whether end can reach start
for (Bone* itr = end; itr != nullptr; itr = itr->parent) {
    if (itr->idx == start->idx) flag = true;
}

if (flag) {
    for (Bone* itr = end; itr->idx != start->parent->idx; itr = itr->parent) {
        boneList.emplace_back(itr);
    }
} else {
    for (Bone* itr = end; itr != nullptr; itr = itr->parent) {
        boneList.emplace_back(itr);
    }
    for (Bone* itr = start; itr->idx != root->idx; itr = itr->parent) {
        boneList.emplace_back(itr);
    }
}
```

- Jacobian Matrix & dTheta

$$\frac{\partial \mathbf{p}}{\partial \theta_i} = \mathbf{a}_i \times (\mathbf{p} - \mathbf{r}_i)$$

The formula above is all you need to calculate Jacobian matrix. P is the end position of the end bone, $r_i$ is the start position of the current bone, $a_i$ is the axis of the rotation, which I get it from rotation matrix of current bone. The first column of the rotation matrix is the x axis, second column is the y axis, third column is the z axis. The last thing is to put rotation in jacobian only when the bone has that DOF. DOF can be accessed by dofrx, dofry, dofrz in the bone class.

$$\dot{\theta} = J^{-1}(\theta)V$$

dTheta can be calculated by this formula. We just need to put the jacobian we calculated above and the V (target minus end position of end bone) into leastSquareSolution and then we can get dTheta.

```
for (int j = 0; j < boneNum; j++) {
  int colIndex = j * 3;
  Eigen::Vector3f dist = end->endPosition - boneList[j]->startPosition;
  if (boneList[j]->dofrx) {
    jacobian.col(colIndex) = posture.rotations[boneList[j]->idx].toRotationMatrix().col(0).normalized().cross(dist);
  }
  if (boneList[j]->dofry) {
    jacobian.col(colIndex + 1) =
        posture.rotations[boneList[j]->idx].toRotationMatrix().col(1).normalized().cross(dist);
  }
  if (boneList[j]->dofrz) {
    jacobian.col(colIndex + 2) =
        posture.rotations[boneList[j]->idx].toRotationMatrix().col(2).normalized().cross(dist);
  }
}

Eigen::VectorXf dTheta = leastSquareSolver(jacobian, target - end->endPosition);
```

- Orientation

After getting the change of orientation, we can convert delta x, delta y, delta z to Euler angle plus the original angular vector to get the new orientation.

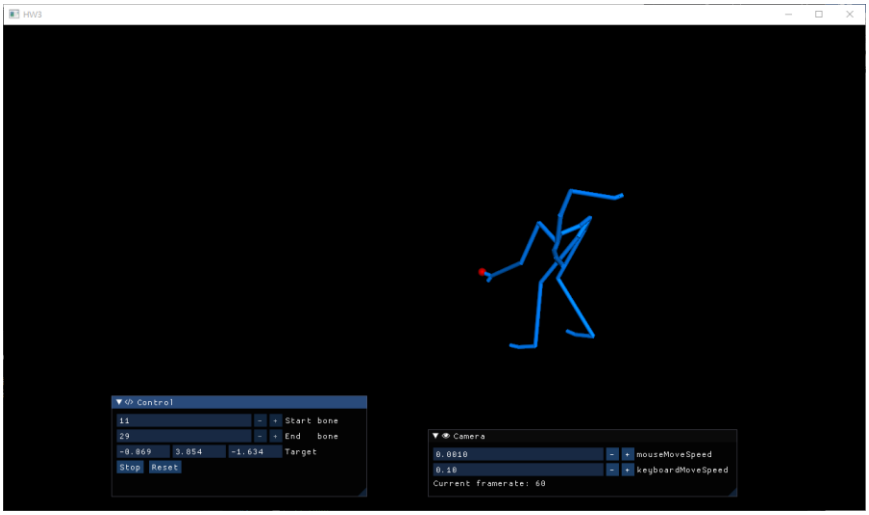
```
for (int j = 0; j < boneNum; j++) {
  const auto& bone = *boneList[j];
  // TODO (update rotation)
  //   1. Update posture's eulerAngle using deltaTheta
  // Hint:
  //   1. Use posture.eulerAngle to get posture's eulerAngle
  //   2. All angles are in radians.
  //   3. You can ignore rotation limit of the bone.
  // Bonus:
  //   1. You cannot ignore rotation limit of the bone.
  int colIndex = j * 3;
  posture.eulerAngle[bone.idx][0] += step * dTheta(colIndex);
  posture.eulerAngle[bone.idx][1] += step * dTheta(colIndex + 1);
  posture.eulerAngle[bone.idx][2] += step * dTheta(colIndex + 2);

  posture.rotations[bone.idx] = Eigen::AngleAxisf(posture.eulerAngle[bone.idx][2], Eigen::Vector3f::UnitZ()) *
                                Eigen::AngleAxisf(posture.eulerAngle[bone.idx][1], Eigen::Vector3f::UnitY()) *
                                Eigen::AngleAxisf(posture.eulerAngle[bone.idx][0], Eigen::Vector3f::UnitX());
}
```

- Result & Discussion
    - How different step and epsilon affect the result:
        - Step

          If the step is too large, the convergence will be fast, but relatively inaccurate; and if the step is small, the convergence will be slow, but relatively accurate.

          Step: 0.001f:


          Step: 0.1f:




        - Epsilon

          This distance is the acceptable range for the error of the distance between the target position and the end effector. If it is set too large, the distance between the two will be relatively far. If it is set too small, it is easy for the end effector to be calculated all the time. Therefore, it is more careful to choose a suitable epsilon.

- Touch the target or not:

  When the ball is near the skeleton, the skeleton can touch the ball most of the time. But if the ball is far away from the skeleton, the skeleton cannot touch the ball.


- Least square solver:

  The way you calculate the pseudo inverse can affect the result, since computer have limit accuracy in floating point. You may lose the accuracy when using some pseudo inverse method. My solution is to set a tolerance when calculating in SVD. If the singular value is higher than the tolerance, then calculate its reciprocal. If its not, set it as 0.

## Conclusion

- We need to take the rotation matrix of the pose after FK, and multiply it by (1,0,0), (0,1,0), (0,0,1) to get relative to the x, y, z axis angle of rotation to get the axis of rotation.
- If the step is too large, the convergence will be fast, but relatively inaccurate; and if the step is small, the convergence will be slow, but relatively accurate. So, the answer obtained by Small increments will be more correct.
- One need to be careful when calculating pseudo inverse. Be aware of the constrain of method.


HW3
Control
11
Start bone
29
End bone
-0.069
5.000
-1.634
Target
Stop
Reset
Camera
0.0010
mouseMoveSpeed
0.10
keyboardMoveSpeed
Current framerate: 60