HW2

110550085 房天越

1. Introduction

In this assignment, we need to implement the animation for the human-like creature to do something, like running, throwing the ball, gymnastics, etc..., we accomplish this part by implementing the forward kinematics.
Also, we need to implement the actions for the parts of human to touch the ball, we need to traverse each bone chain from end bone to start bone, so we need Inverse kinematics and use the Jacobian Matrix to accomplish the goal.

2. Fundamentals

I. Local Coordinate

For local coordinate, we only use the coordinate of the root bone, every root has its own local coordinate, which would be changed when we apply translation or rotation on it.
Compared to the global coordinate, it would be easier for us to manipulate different objects, and do not need to worry about the effect which other objects apply on it.

II. Global Coordinate

Global Coordinate is the coordinate of the whole scene, we can observe the relative position more easily here compared to the local coordinate.
However, we cannot use global coordinate to distinguish which bone is there, we need to use local coordinate first, then paste the structure to the global coordinate.

III. Forward Kinematics

When a bone in the structure is moved, the bones it connects to would also move. So, in forward kinematics, we

first get the translation and rotation of the root bone, and calculate the translation and rotation of the next bone it connects to, then, traverse every bone to do the things above.

IV. Inverse Kinematics

In inverse kinematics, we use endpoint to calculate the difference in the process, also, we use step to fix the bone's end position to approach the target, so it is an iterative method.

V. Pseudo Inverse of Jacobian

The Jacobian matrix is used to represent the best linear approximation of a multiple variable vectors function. In this assignment, it is use to represent the instant difference of the end effector.

After we get the Jacobian, we get its inverse, and multiply it by the vector from the end effector to the target to get the delta theta, and the reason for "Pseudo" is because not all the Jacobian matrices can be inverted.

The process is to traverse from the target to the start bone(the 0^{th} bone in the boneChain), for every rotation of the bone, we need to calculate the column of the Jacobian matrix, and get the partial difference of the x, y, z dimensions in every rotation.

3. Implementation

I. forwardSolver (FK)

In FK part, we first set the start_position, end_position, and rotation to 0, then, use rotationDegreeZYX to alter the vector bone_rotations[bone->idx] to Quaterniond.

Then, we divide the situation into two cases, the first is the root, it would not be affected by the previous bones, ao we get its rotation, start_position and end_position directly. And for the other bones, we get its rotation, start_position, and

end_position with respect to their parents' vectors.
Finally, we go on traversing if the current bone has child or sibling with recursion.
The code is here:

```
bone->start_position = Eigen::Vector4d::Zero();
bone->end_position = Eigen::Vector4d::Zero();
bone->rotation = Eigen::Matrix4d::Zero();

Eigen::Quaterniond r = util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);

if (bone->name == "root") {
    bone->rotation = r.toRotationMatrix();
    bone->start_position = posture.bone_translations[bone->idx];
    bone->end_position = bone->start_position;
}
else {
    bone->rotation = bone->parent->rotation * bone->rot_parent_current * r.toRotationMatrix();
    bone->start_position = bone->parent->end_position + posture.bone_translations[bone->idx];
    bone->end_position = bone->start_position + bone->rotation * bone->dir * bone->length;
}

if (bone->child != nullptr) {
    forwardSolver(posture, bone->child);
}
if (bone->sibling != nullptr) {
    forwardSolver(posture, bone->sibling);
}
```

II. PseudoInverseLinearSolver

In this part, we apply SVD on the Jacobian matrix to find the deltatheta of the min(|Jacobian * deltatheta – target|), and solve it with respect to the target_pos.
The code is here:

```
Eigen::VectorXd deltatheta(Jacobian.cols());
// get pseudo inverse of the Jacobian with SVD
Eigen::JacobiSVD<Eigen::Matrix4Xd> svd(Jacobian, Eigen::ComputeThinU | Eigen::ComputeThinV);
deltatheta = svd.solve(target);
return deltatheta;
```

III. inverseJacobianIKSolver(IK)

First, we construct the Jacobian matrix, since we have the chains of the bones, we first get the arm vector with jointChains[chainIdx][0]->head<3>() - jointChains[chainIdx][i+1]->head<3>();
Then, we get its rotation.
After that, we go on to construct the Jacobian matrix, if the current bone has dofrx, dofry or dofrz, we calculate the corresponding results of the part difference and store the results into the corresponding Jacobian column.
The code for this part is here:

```
for (long long i = 0; i < bone_num; i++) {
    Eigen::Vector3d arm =
        jointChains[chainIdx][0]->head<3>() - jointChains[chainIdx][i+1]->head<3>();
    Eigen::Affine3d rotation = boneChains[chainIdx][i]->rotation;
    if (boneChains[chainIdx][i]->dofrx) {
        Eigen::Vector3d unit_rotation = rotation.matrix().col(0).head<3>();
        Eigen::Vector3d J = unit_rotation.cross(arm);
        Jacobian.col(3 * i) = Eigen::Vector4d(J[0], J[1], J[2], 0);
    }
    if (boneChains[chainIdx][i]->dofry) {
        Eigen::Vector3d unit_rotation = rotation.matrix().col(1).head<3>();
        Eigen::Vector3d J = unit_rotation.cross(arm);
        Jacobian.col(3 * i + 1) = Eigen::Vector4d(J[0], J[1], J[2], 0);
    }
    if (boneChains[chainIdx][i]->dofrz) {
        Eigen::Vector3d unit_rotation = rotation.matrix().col(2).head<3>();
        Eigen::Vector3d J = unit_rotation.cross(arm);
        Jacobian.col(3 * i + 2) = Eigen::Vector4d(J[0], J[1], J[2], 0);

    }
}
```

Then, we update the rotation using deltatheta, which is got by step multiplied by the result if the pseudoInverseLinearSolver. We then update the rotation from end_bone to its connected bone, get the direction the bone should rotate to and update the bone rotation with the current dimension of x, y, and z.

Also, we check the rotation limit of the bone, if the rotation exceeds the limit of the bone, we set it to the rotation limit (minimum or maximum).

The code for this part is here:

```
for (long long i = 0; i < bone_num; i++) {
    acclaim::Bone curr = *boneChains[chainIdx][i];
    Eigen::Vector3d delta = deltatheta.segment(i * 3, 3);
    posture.bone_rotations[curr.idx] += util::toDegree(Eigen::Vector4d(delta[0], delta[1], delta[2], 0));


    if (posture.bone_rotations[curr.idx][0] < curr.rxmin) {
        posture.bone_rotations[curr.idx][0] = curr.rxmin;
    }
    else if (posture.bone_rotations[curr.idx][0] > curr.rxmax) {
        posture.bone_rotations[curr.idx][0] = curr.rxmax;
    }
    if (posture.bone_rotations[curr.idx][1] < curr.rymin) {
        posture.bone_rotations[curr.idx][1] = curr.rymin;
    }
    else if (posture.bone_rotations[curr.idx][1] > curr.rymax) {
        posture.bone_rotations[curr.idx][1] = curr.rymax;
    }
    if (posture.bone_rotations[curr.idx][2] < curr.rzmin) {
        posture.bone_rotations[curr.idx][2] = curr.rzmin;
    }
    else if (posture.bone_rotations[curr.idx][2] > curr.rzmax) {
        posture.bone_rotations[curr.idx][2] = curr.rzmax;
    }


}
```

4. Result and Discussion

I. The effect of step

The step is kind of like the learning rate in deep learning, if we use bigger step, we would make the speed of

converging faster, but we would get a relatively not accurate result. In contrast, if we use a smaller step, we would get the speed of convergence slower, but the result would be more accurate.

II. The effect of epsilon

If we use a larger epsilon, we would make the final distance between the end_bone and the target_pos farther. While if we use a smaller epsilon, we would need more time for calculation and requires more computation.

III. Problems that I met

When implementing the FK part, I first use rotationDegreeXYZ, which is mentioned in the hint. However, the action of the creature is then very strange. Then, after modifying it to rotationZYX, I finally got a better result.

When implementing the IK part, I first used boneChains when computing the arm vector, and get the a result that most stretching are unstable, and then I tried to use the i^{th} joint, the result is even worse, after realizing more about the notations, I used the $i+1^{th}$ joint, and the result finally became relative normal.

5. Conclusion

In this assignment, I learned how to implement the FK and IK, and apply the Jacobian matrix to solve the rotation and position problems. I had quite a difficult time dealing with the IK part, but I would say that I learned a lot and had a nice feeling of accomplishment after fixing the problems and see that the creature could be relatively better stretched.

6. Demo link

https://www.youtube.com/watch?v=9Zkpyw7-Yuo