Computer Animation Homework 2

112550184 陳璽安

1 Introduction

Inverse Kinematics (IK) enables determining joint angles required to position an end effector at a desired location - the opposite of Forward Kinematics (FK). This project implements FK and IK algorithms with obstacle avoidance for skeletal animation, focusing on the Jacobian-based approach. The goal is to develop a system that can accurately move a character's limb to touch a target while navigating around obstacles, demonstrating practical animation control techniques.

2 Fundamentals

2.1 Forward Kinematics

FK calculates bone positions from joint angles by traversing the skeleton from root to leaves. For a bone i with parent j:

$R_i = R_j * R_{local\_i}$

$T_i = T_j + R_j * (l_i * dir_i)$

Where R_i is the global rotation, T_i is the global position, R_local_i is the local rotation, l_i is the bone length, and dir_i is the bone direction vector.

2.2 Inverse Kinematics

2.2.1 Jacobian Metrix

The Jacobian(J) maps joint angle changes(deltatheta) to end effector position changes:

$\Delta p = J * \Delta\theta$

Where Δp is the end effector position change and $\Delta\theta$ represents joint angle changes.

2.2.2 Inverse Jacobian Method

To find joint angle changes needed for a desired position:

$\Delta\theta = J^+ * \Delta p$

Where J^+ is the pseudoinverse of J, providing a least-squares solution. The algorithm works iteratively:

1. Calculate current end effector position (FK)
2. Compute error vector (desired - current position)
3. Compute Jacobian matrix
4. Calculate joint angle changes via pseudoinverse
5. Update joint angles
6. Repeat until convergence

2.2.3 Obstacle Avoidance

When bones approach obstacles, repulsion forces modify the target direction:

```
obsDiff = bone_position - obstacle_center
distance = |obsDiff|
if distance < threshold:
    repulsion = normalize(obsDiff) * (threshold - distance)
```

This repulsion is added to the desired position vector before computing angle changes, biasing the solution away from obstacles.

3 Implementation

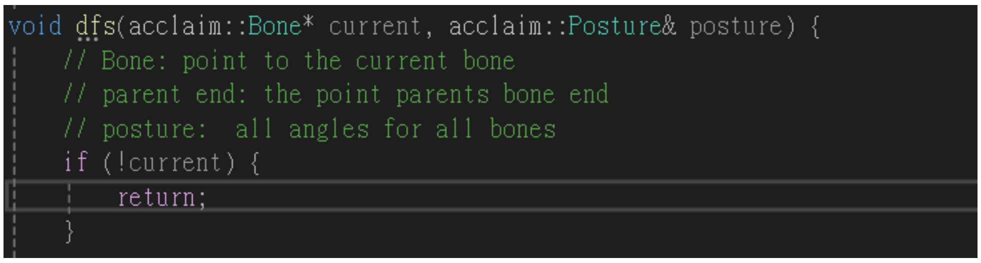
3.1 forwardSolver

In this function, we need to traverse each skeleton and compute each bone's global rotation and global position. I defined an external function dfs to traverse all the skeleton by using depth first search.

3.1.1 dfs

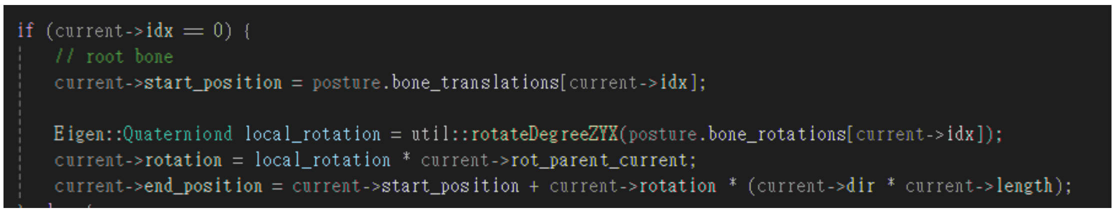
Takes two parameters: current is the pointer point to the current bone being processed, posture contains rotation and translation for all bones.

Handling the ending case, if current point to null -> return.



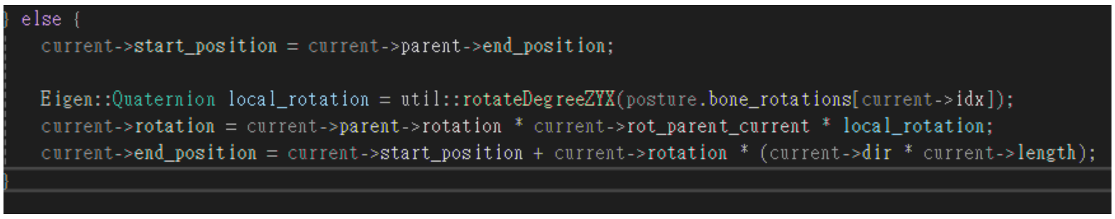
```
void dfs(acclaim::Bone* current, acclaim::Posture& posture) {
    // Bone: point to the current bone
    // parent end: the point parents bone end
    // posture:  all angles for all bones
    if (!current) {
        return;
    }
```

Root bone case: Set the start position from translation data, and calculate rotation by combining local rotation with parent-current rotation. Finally, compute the end position $_{i+1}T = {}^{i}_{0}RV_{i} + {}_{i}T$



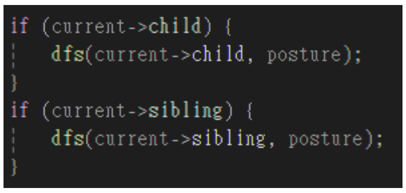
```
if (current->idx == 0) {
    // root bone
    current->start_position = posture.bone_translations[current->idx];

    Eigen::Quaterniond local_rotation = util::rotateDegreeZYX(posture.bone_rotations[current->idx]);
    current->rotation = local_rotation * current->rot_parent_current;
    current->end_position = current->start_position + current->rotation * (current->dir * current->length);
```

Child bone case: set the start position to parents' end position and calculate rotation by combining parent rotation with local rotation. Finally, compute the end position.



```
} else {
    current->start_position = current->parent->end_position;

    Eigen::Quaternion local_rotation = util::rotateDegreeZYX(posture.bone_rotations[current->idx]);
    current->rotation = current->parent->rotation * current->rot_parent_current * local_rotation;
    current->end_position = current->start_position + current->rotation * (current->dir * current->length);
}
```

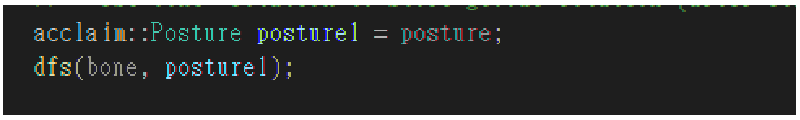
Recursively calls dfs function to traverse the child bones first, and traverse the sibling bones.



```
if (current->child) {
    dfs(current->child, posture);
}
if (current->sibling) {
    dfs(current->sibling, posture);
}
```

3.1.2 forwardSolver

Make a copy of posture(since the original parameter is const and cannot plugin dfs function), and call the dfs function to traverse the skeletons.



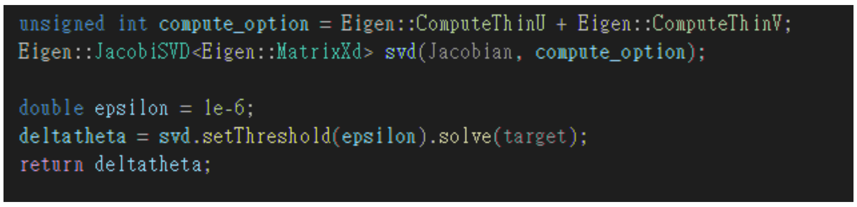
```
acclaim::Posture posture1 = posture;
dfs(bone, posture1);
```

3.2 psuedoInverseLinearSolver

Decompose the Jacobian matrix into U, S and V. Jacobian = U * S * V^T

Solve deltatheta = Jacobian^{-1}*target

By computing the pseudoinverse of Jacobian matrix(J^{+}), and multiplying J^{+} with target(Use svd.setThreshold().solve())

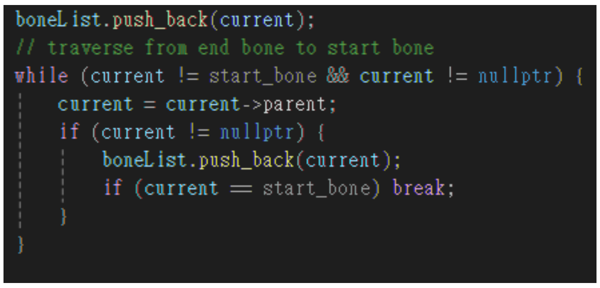


```
unsigned int compute_option = Eigen::ComputeThinU + Eigen::ComputeThinV;
Eigen::JacobiSVD<Eigen::MatrixXd> svd(Jacobian, compute_option);

double epsilon = 1e-6;
deltatheta = svd.setThreshold(epsilon).solve(target);
return deltatheta;
```

3.3 InverseJacobianIKSolver

3.3.1 Store bones into boneList[] and store bone number in bone_num

Push the bones to the boneList.

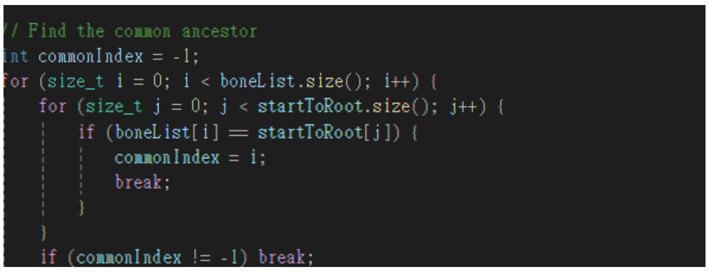


```
boneList.push_back(current);
// traverse from end bone to start bone
while (current != start_bone && current != nullptr) {
    current = current->parent;
    if (current != nullptr) {
        boneList.push_back(current);
        if (current == start_bone) break;
    }
}
```

If we reach the nullptr without reaching the start bone, we start over.

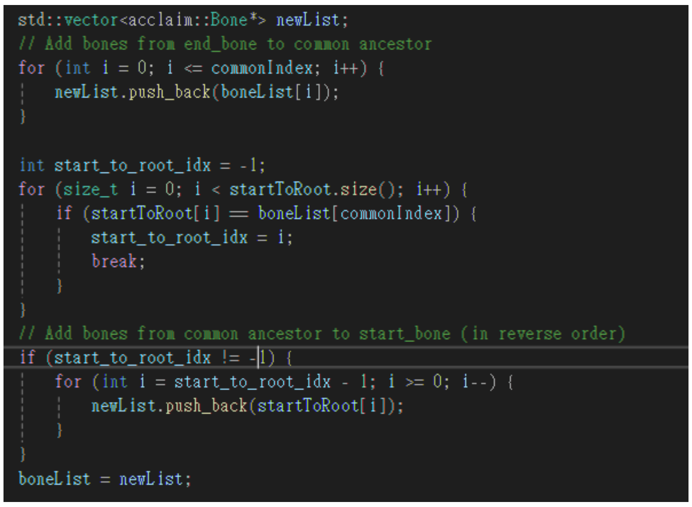
And store the bones in the boneList by

Get common index.



```
/ Find the common ancestor
nt commonIndex = -1;
or (size_t i = 0; i < boneList.size(); i++) {
   for (size_t j = 0; j < startToRoot.size(); j++) {
       if (boneList[i] == startToRoot[j]) {
           commonIndex = i;
           break;
       }
   }
   if (commonIndex != -1) break;
```

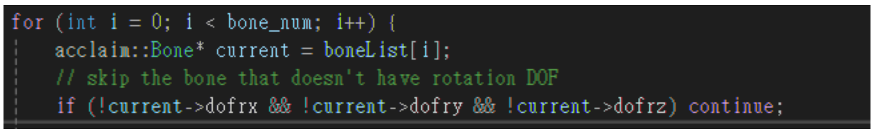
Store end_bone→common ancestor→start_bone in the boneList[].



```
std::vector<acclaim::Bone*> newList;
// Add bones from end_bone to common ancestor
for (int i = 0; i <= commonIndex; i++) {
    newList.push_back(boneList[i]);
}

int start_to_root_idx = -1;
for (size_t i = 0; i < startToRoot.size(); i++) {
    if (startToRoot[i] == boneList[commonIndex]) {
        start_to_root_idx = i;
        break;
    }
}
// Add bones from common ancestor to start_bone (in reverse order)
if (start_to_root_idx != -1) {
    for (int i = start_to_root_idx - 1; i >= 0; i--) {
        newList.push_back(startToRoot[i]);
    }
}
boneList = newList;
```

3.3.2 Compute Jacobian

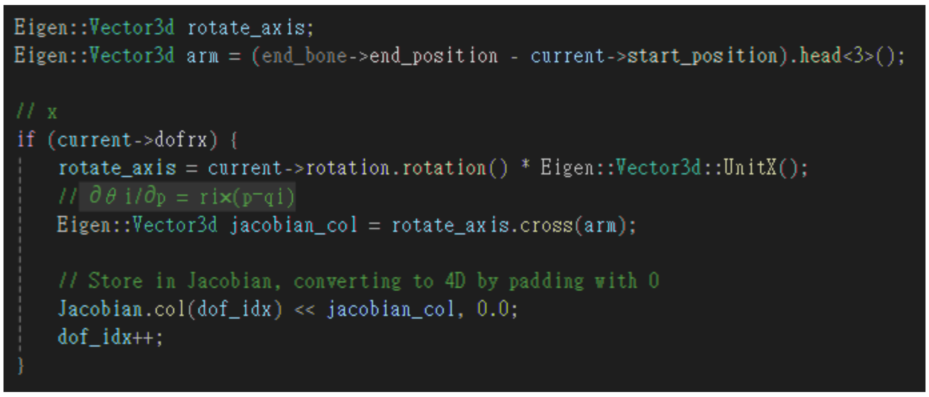
Iterate each bone in boneList to compute Jacobian matrix and skip the bone that doesn't have rotation DOF.



```
for (int i = 0; i < bone_num; i++) {
    acclaim::Bone* current = boneList[i];
    // skip the bone that doesn't have rotation DOF
    if (!current->dofrx && !current->dofry && !current->dofrz) continue;
```

Compute Jacobian matrix columns by ∂ θ i/∂p = r_i×(p−q_i)

Below is the code for x direction.



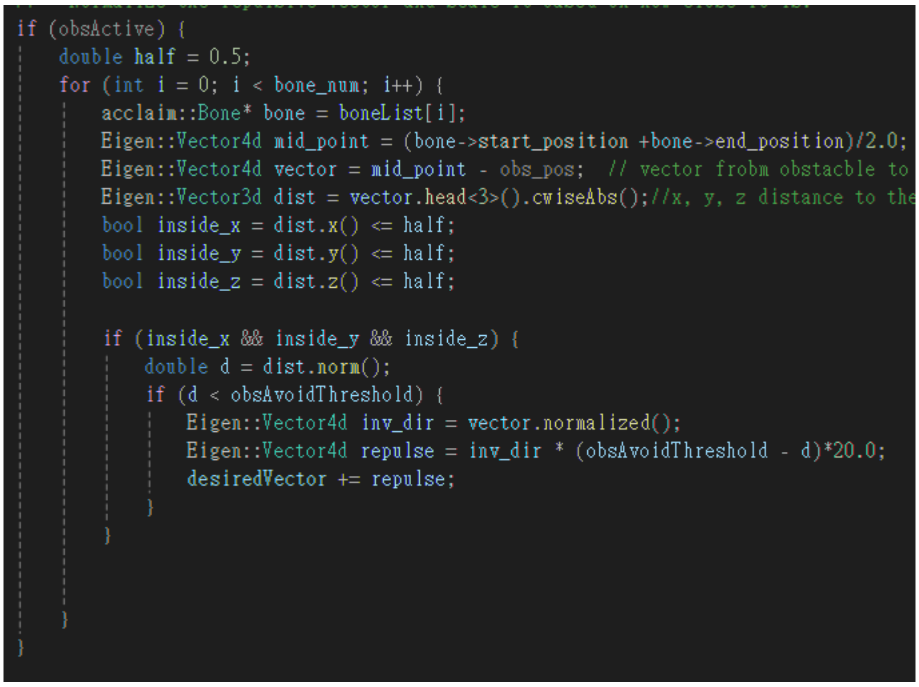
```
Eigen::Vector3d rotate_axis;
Eigen::Vector3d arm = (end_bone->end_position - current->start_position).head<3>();

// x
if (current->dofrx) {
    rotate_axis = current->rotation.rotation() * Eigen::Vector3d::UnitX();
    // ∂θ i/∂p = ri×(p-qi)
    Eigen::Vector3d jacobian_col = rotate_axis.cross(arm);

    // Store in Jacobian, converting to 4D by padding with 0
    Jacobian.col(dof_idx) << jacobian_col, 0.0;
    dof_idx++;
}
```

Repeat the same steps for y and z directions.

3.3.3 Obstacle avoidance

Check whether the mid point of the bone is in the cube. If yes, compute the repulse force by

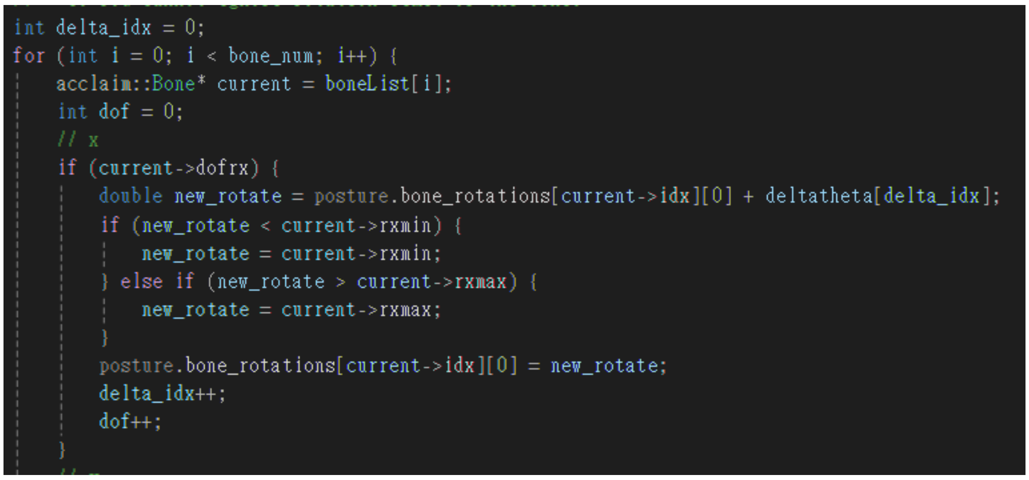
repulse = normalize(obsDiff) * (threshold - dist) *20.0



```
if (obsActive) {
    double half = 0.5;
    for (int i = 0; i < bone_num; i++) {
        acclaim::Bone* bone = boneList[i];
        Eigen::Vector4d mid_point = (bone->start_position +bone->end_position)/2.0;
        Eigen::Vector4d vector = mid_point - obs_pos;  // vector frobm obstacble to
        Eigen::Vector3d dist = vector.head<3>().cwiseAbs();//x, y, z distance to the
        bool inside_x = dist.x() <= half;
        bool inside_y = dist.y() <= half;
        bool inside_z = dist.z() <= half;

        if (inside_x && inside_y && inside_z) {
            double d = dist.norm();
            if (d < obsAvoidThreshold) {
                Eigen::Vector4d inv_dir = vector.normalized();
                Eigen::Vector4d repulse = inv_dir * (obsAvoidThreshold - d)*20.0;
                desiredVector += repulse;
            }
        }


    }
}
```

3.3.4 Update rotation

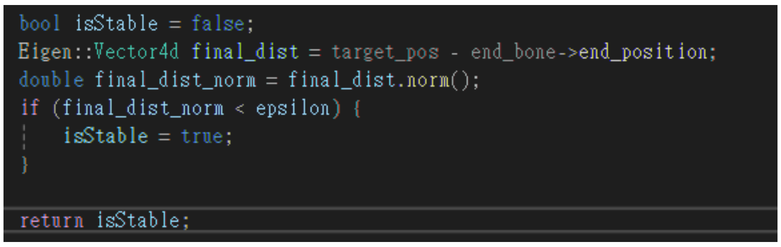
For each bone, updating the rotation angle by adding the current rotation to deltatheta(computed in the pseudoInverseLinearSolver). If the rotation exceeds the max rotation of the bone, set the rotation angle to the max rotation angle. Else if the rotation angle is less than the min rotation angle, set the rotation angle to the min rotation angle.



```
int delta_idx = 0;
for (int i = 0; i < bone_num; i++) {
    acclaim::Bone* current = boneList[i];
    int dof = 0;
    // x
    if (current->dofrx) {
        double new_rotate = posture.bone_rotations[current->idx][0] + deltatheta[delta_idx];
        if (new_rotate < current->rxmin) {
            new_rotate = current->rxmin;
        } else if (new_rotate > current->rxmax) {
            new_rotate = current->rxmax;
        }
        posture.bone_rotations[current->idx][0] = new_rotate;
        delta_idx++;
        dof++;
    }
```

3.3.5 Check whether IK is stable

If the distance between ball and end bone is less than epsilon, return stable. Else, return not stable.



```
bool isStable = false;
Eigen::Vector4d final_dist = target_pos - end_bone->end_position;
double final_dist_norm = final_dist.norm();
if (final_dist_norm < epsilon) {
    isStable = true;
}

return isStable;
```

4 Results and Discussion

During testing, the skeleton will penetrate the cube slightly before being pushed away although the implementation of object avoidance. Besides, obstacle avoidance appears to work more effectively in the Y direction compared to the Z direction.

Factors above are observed during implementation, but I don’t know the specific reasons that may lead to those phenomena.

5 Bonus

1. I updated the rotation angle of each bone while considering the rotation limit of each bone.
2. Return whether IK is stable.

6 Conclusion

This project successfully implemented forward and inverse kinematics using the Jacobian method. The forward kinematics correctly computes global bone positions, while the inverse kinematics effectively moves the end effector toward targets. And the obstacle avoidance mechanism functions but could be improved to address the penetration issues and inconsistent effectiveness across different axes.