Hw2 report

Introduction/Motivation

This homework consists of two parts: Forward Kinematics and Inverse Kinematics:

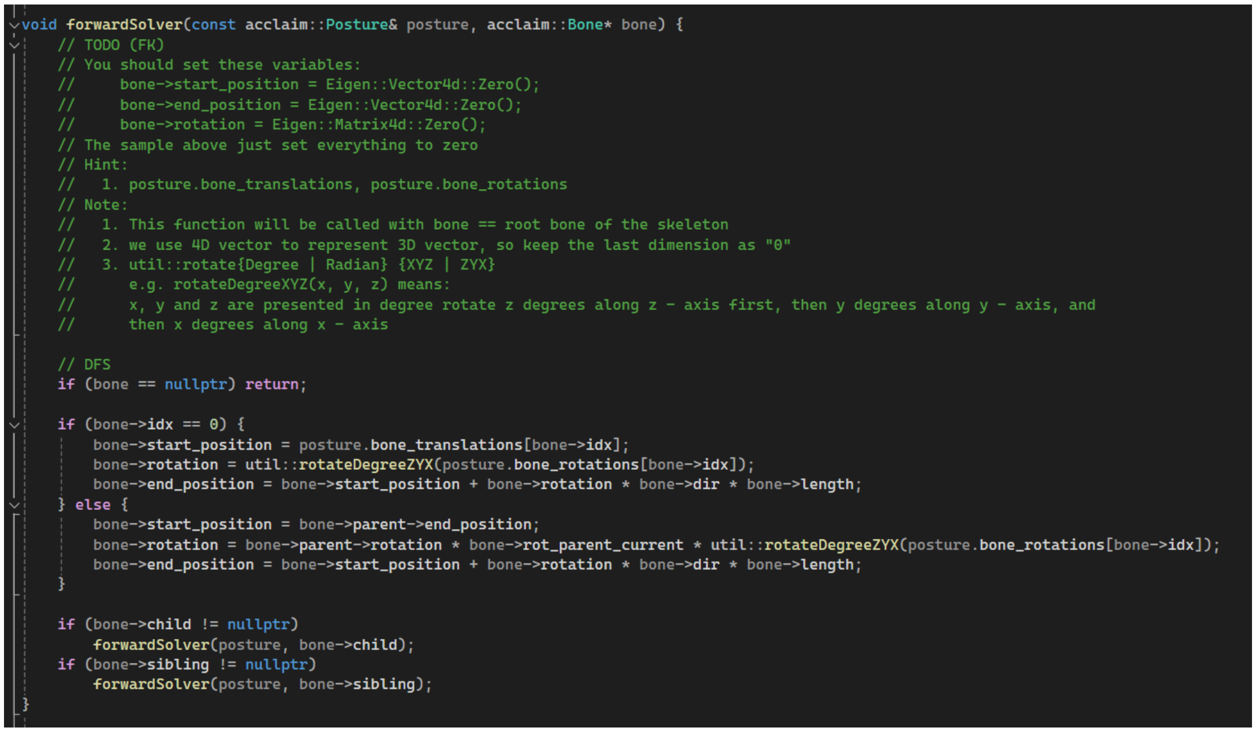
- Forward kinematics aims to traverse all bones in skeleton (BFS or DFS) and update start position, end position and rotation vector (rotation degree of dimensions x,y,z) of each bones in global position
- Inverse kinematics aims to make specific joints (jointChains[chainIdx][0]) touch the target ball (target_pos[chainIdx]). This is done by first traversing all bones in a bone chain to compute their rotation and place them to their corresponding jacobian columns. Then use this jacobian matrix to calculate least squares (x = A^+b, with b as vector from specified joint to target ball), I computed this using Moore-Penrose inverse. Finally, update bone rotation (posture.bone_rotations) of each bone in the bone chain, while keeping in mind the max/min rotation degree, if exceeded, set rotation degree as max/min degree .

Fundamentals

Local and Global coordinates:
Global coordinate often refers to the coordinate in regards to the whole setting/system. While local coordinate often refers to the coordinate in regards to the subcomponents of a subsystem, describing the relative position of the subcomponents. Direction between the axis of the whole system and its subsystem may not face the same way. Hence, we usually need a rotational matrix to convert between these systems.

Implementation

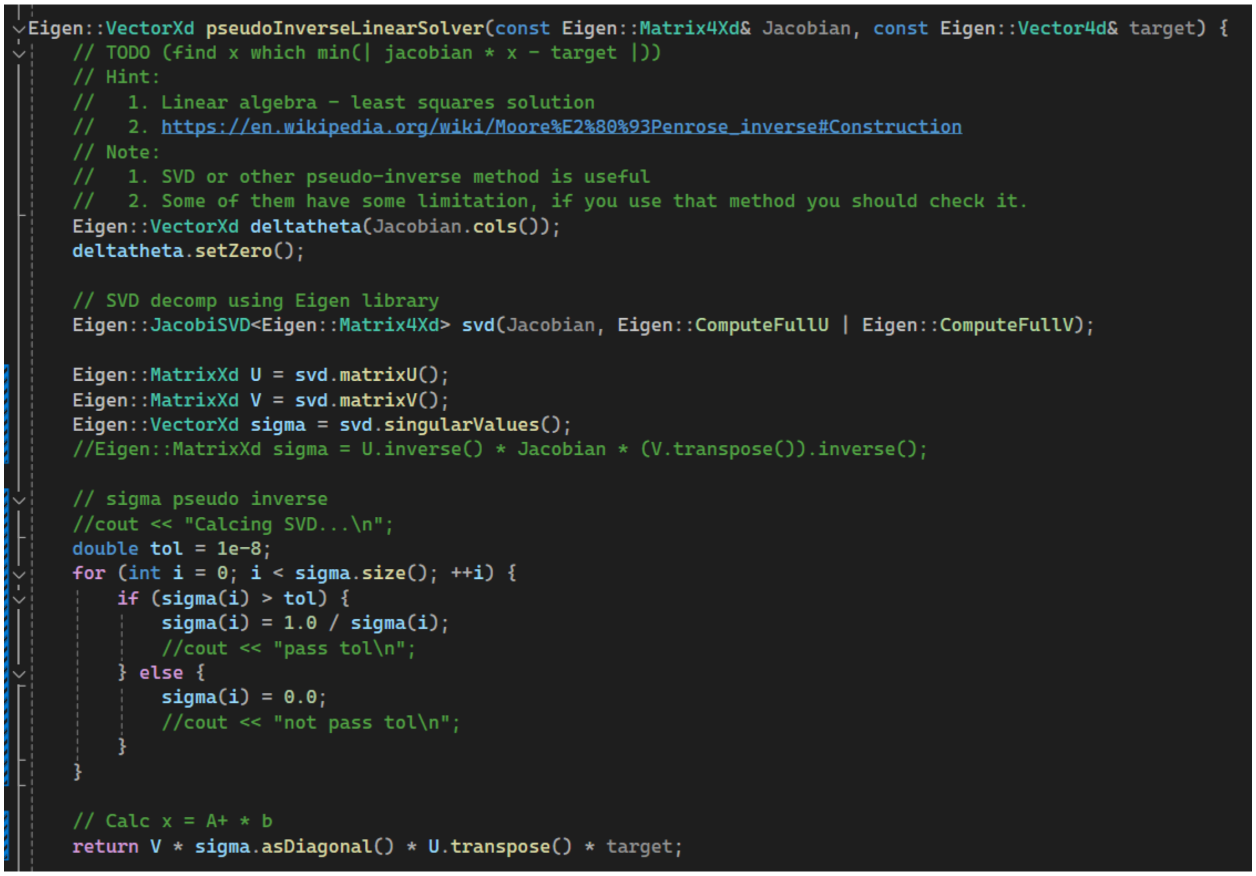
fowardSolver():



```cpp
void forwardSolver(const acclaim::Posture& posture, acclaim::Bone* bone) {
    // TODO (FK)
    // You should set these variables:
    //     bone->start_position = Eigen::Vector4d::Zero();
    //     bone->end_position = Eigen::Vector4d::Zero();
    //     bone->rotation = Eigen::Matrix4d::Zero();
    // The sample above just set everything to zero
    // Hint:
    //   1. posture.bone_translations, posture.bone_rotations
    // Note:
    //   1. This function will be called with bone == root bone of the skeleton
    //   2. we use 4D vector to represent 3D vector, so keep the last dimension as "0"
    //   3. util::rotate{Degree | Radian} {XYZ | ZYX}
    //      e.g. rotateDegreeXYZ(x, y, z) means:
    //      x, y and z are presented in degree rotate z degrees along z - axis first, then y degrees along y - axis, and
    //      then x degrees along x - axis

    // DFS
    if (bone == nullptr) return;

    if (bone->idx == 0) {
        bone->start_position = posture.bone_translations[bone->idx];
        bone->rotation = util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);
        bone->end_position = bone->start_position + bone->rotation * bone->dir * bone->length;
    } else {
        bone->start_position = bone->parent->end_position;
        bone->rotation = bone->parent->rotation * bone->rot_parent_current * util::rotateDegreeZYX(posture.bone_rotations[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);
}
```

- I choose to traverse all bones using DFS (recursive method)
- If current bone is root bone (bone->idx == 0), since it doesn't have a parent bone and won't be influenced by it: let start position be posture.bone_translations[0], rotation be posture.bone_rotations[0] (converted to global) and end position be start position + rotation * direction * bone length
- For other bones (would be influenced by parent bone): let start position be parent bone's end position, rotation be parent's rotation * parent to child's rotation matrix * posture.bone_rotations[bone->idx] (current bone rotation)
- Finally, for each bone, check whether its child or sibling is nullptr or not, if not, explore them. This way all surrounding bones would be traversed

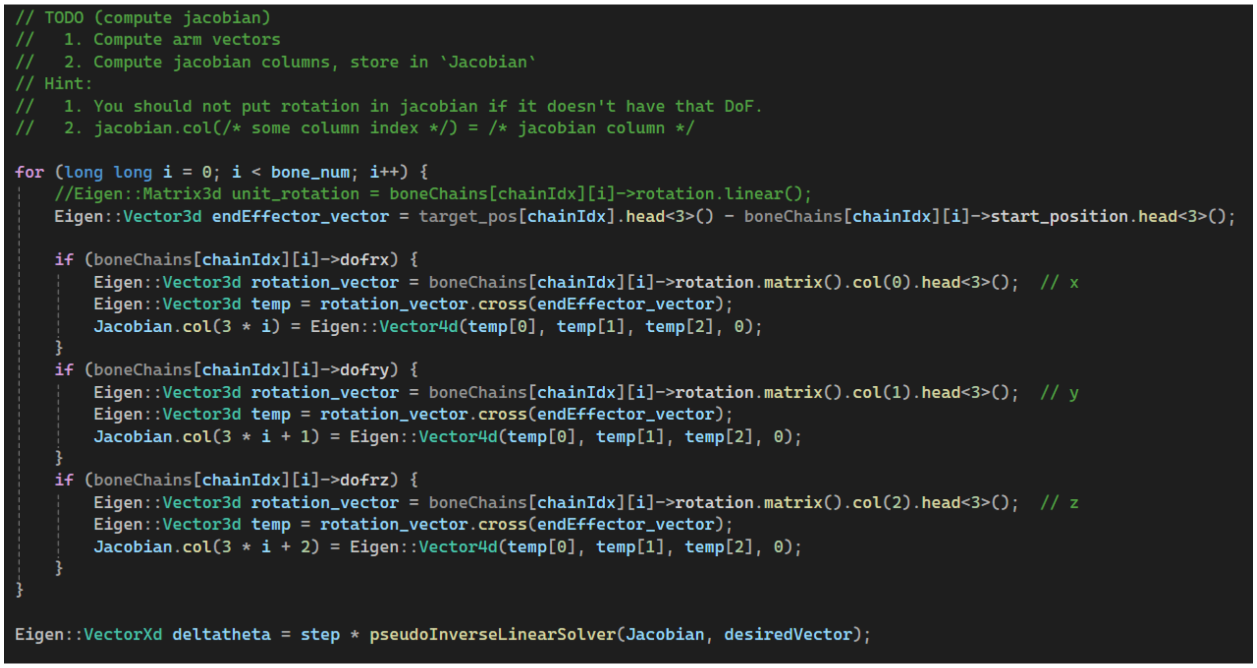
pseudoInverseLinearSolver():



```
Eigen::VectorXd pseudoInverseLinearSolver(const Eigen::Matrix4Xd& Jacobian, const Eigen::Vector4d& 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::VectorXd deltatheta(Jacobian.cols());
    deltatheta.setZero();

    // SVD decomp using Eigen library
    Eigen::JacobiSVD<Eigen::Matrix4Xd> svd(Jacobian, Eigen::ComputeFullU | Eigen::ComputeFullV);

    Eigen::MatrixXd U = svd.matrixU();
    Eigen::MatrixXd V = svd.matrixV();
    Eigen::VectorXd sigma = svd.singularValues();
    //Eigen::MatrixXd sigma = U.inverse() * Jacobian * (V.transpose()).inverse();

    // sigma pseudo inverse
    //cout << "Calcing SVD...\n";
    double tol = 1e-8;
    for (int i = 0; i < sigma.size(); ++i) {
        if (sigma(i) > tol) {
            sigma(i) = 1.0 / sigma(i);
            //cout << "pass tol\n";
        } else {
            sigma(i) = 0.0;
            //cout << "not pass tol\n";
        }
    }

    // Calc x = A+ * b
    return V * sigma.asDiagonal() * U.transpose() * target;
```

- First decompose Jacobian using SVD decomposition function in Eigen library
- Obtain matrix V, U and singular values as vector Sigma of SVD
- To solve for least squares, simply solve Ax=b where A is Jacobian matrix, B is target vector and x is the deltatheta we want
- To solve Ax=b, we first find the pseudo inverse of A (since A may not always be invertible). To do this, I used Moore-penrose inverse: x = A^+b -> x = V * Sigma_pseudo_inverse * U_transpose * b
- Calculate Sigma_pseudo_inverse by simply computing the reciprocal of the diagonal elements (with certain tolerance as sometimes some diagonal elements may be close to zero, leading to ill-condition), finally use the asDiagonal() function in Eigen library to convert it to diagonal matrix.

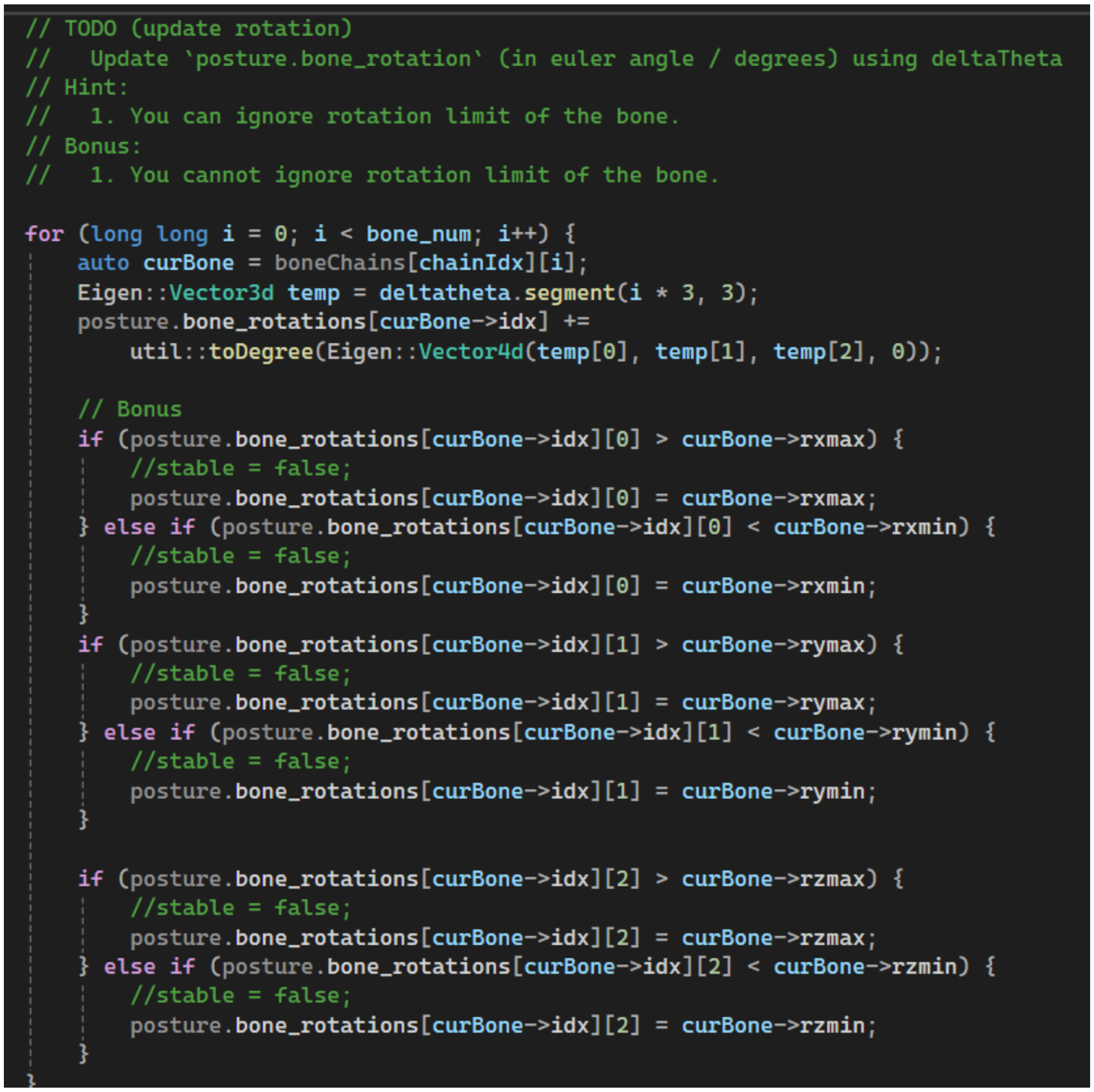
inverseJacobianIKSolver (outside of TODO remains the same):



```
// TODO (compute jacobian)
//   1. Compute arm vectors
//   2. Compute jacobian columns, store in `Jacobian`
// Hint:
//   1. You should not put rotation in jacobian if it doesn't have that DoF.
//   2. jacobian.col(/* some column index */) = /* jacobian column */

for (long long i = 0; i < bone_num; i++) {
    //Eigen::Matrix3d unit_rotation = boneChains[chainIdx][i]->rotation.linear();
    Eigen::Vector3d endEffector_vector = target_pos[chainIdx].head<3>() - boneChains[chainIdx][i]->start_position.head<3>();

    if (boneChains[chainIdx][i]->dofrx) {
        Eigen::Vector3d rotation_vector = boneChains[chainIdx][i]->rotation.matrix().col(0).head<3>();  // x
        Eigen::Vector3d temp = rotation_vector.cross(endEffector_vector);
        Jacobian.col(3 * i) = Eigen::Vector4d(temp[0], temp[1], temp[2], 0);
    }
    if (boneChains[chainIdx][i]->dofry) {
        Eigen::Vector3d rotation_vector = boneChains[chainIdx][i]->rotation.matrix().col(1).head<3>();  // y
        Eigen::Vector3d temp = rotation_vector.cross(endEffector_vector);
        Jacobian.col(3 * i + 1) = Eigen::Vector4d(temp[0], temp[1], temp[2], 0);
    }
    if (boneChains[chainIdx][i]->dofrz) {
        Eigen::Vector3d rotation_vector = boneChains[chainIdx][i]->rotation.matrix().col(2).head<3>();  // z
        Eigen::Vector3d temp = rotation_vector.cross(endEffector_vector);
        Jacobian.col(3 * i + 2) = Eigen::Vector4d(temp[0], temp[1], temp[2], 0);
    }
}

Eigen::VectorXd deltatheta = step * pseudoInverseLinearSolver(Jacobian, desiredVector);
```

- Traverse each bone in the bone chain
- Compute the (p - ri) vector by subtracting target_pos[chainIdx] with boneChains[chainIdx][i]->start_position. Note, extract only first 3 dimension as 4th one is always zero
- For each DoF (x,y,z) if the current bone has that DoF, compute its Jacobian column.
- First find ai rotation vector, boneChains[chainIdx][i]->rotation.matrix().col(DoF)
- Cross multiply ai with (p - ri) and store it in corresponding jacobian column.
- Finally, obtain the Jacobian matrix and put it in pseudoInverseLinearSolver to compute deltatheta

inverseJacobianIKSolver (update rotation and BONUS):



```
// TODO (update rotation)
//   Update `posture.bone_rotation` (in euler angle / degrees) using deltaTheta
// Hint:
//   1. You can ignore rotation limit of the bone.
// Bonus:
//   1. You cannot ignore rotation limit of the bone.

for (long long i = 0; i < bone_num; i++) {
    auto curBone = boneChains[chainIdx][i];
    Eigen::Vector3d temp = deltatheta.segment(i * 3, 3);
    posture.bone_rotations[curBone->idx] +=
        util::toDegree(Eigen::Vector4d(temp[0], temp[1], temp[2], 0));

    // Bonus
    if (posture.bone_rotations[curBone->idx][0] > curBone->rxmax) {
        //stable = false;
        posture.bone_rotations[curBone->idx][0] = curBone->rxmax;
    } else if (posture.bone_rotations[curBone->idx][0] < curBone->rxmin) {
        //stable = false;
        posture.bone_rotations[curBone->idx][0] = curBone->rxmin;
    }
    if (posture.bone_rotations[curBone->idx][1] > curBone->rymax) {
        //stable = false;
        posture.bone_rotations[curBone->idx][1] = curBone->rymax;
    } else if (posture.bone_rotations[curBone->idx][1] < curBone->rymin) {
        //stable = false;
        posture.bone_rotations[curBone->idx][1] = curBone->rymin;
    }

    if (posture.bone_rotations[curBone->idx][2] > curBone->rzmax) {
        //stable = false;
        posture.bone_rotations[curBone->idx][2] = curBone->rzmax;
    } else if (posture.bone_rotations[curBone->idx][2] < curBone->rzmin) {
        //stable = false;
        posture.bone_rotations[curBone->idx][2] = curBone->rzmin;
    }
}
```

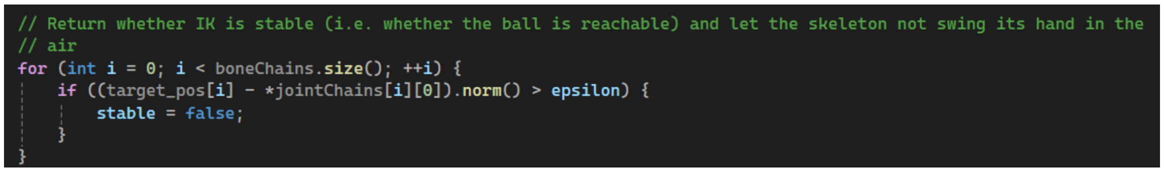
- Use the calculated deltatheta (converted to degree) to update posture.bone_rotations
- Bonus: Check whether the updated rotation exceeds the maximum/minimum rotation degree of each DoF, if so set it as max/min rotation.

Result and Discussion

FK:
Animation was able to run smoothly for each animation movement throughout the frames. It seems each bone of the skeleton was able to successfully update its start position, rotation and end position.

IK:
For most positions (upon not exceeding min/max rotation and bone length), the skeleton was able to adjust accordingly to the target ball.
Observations:

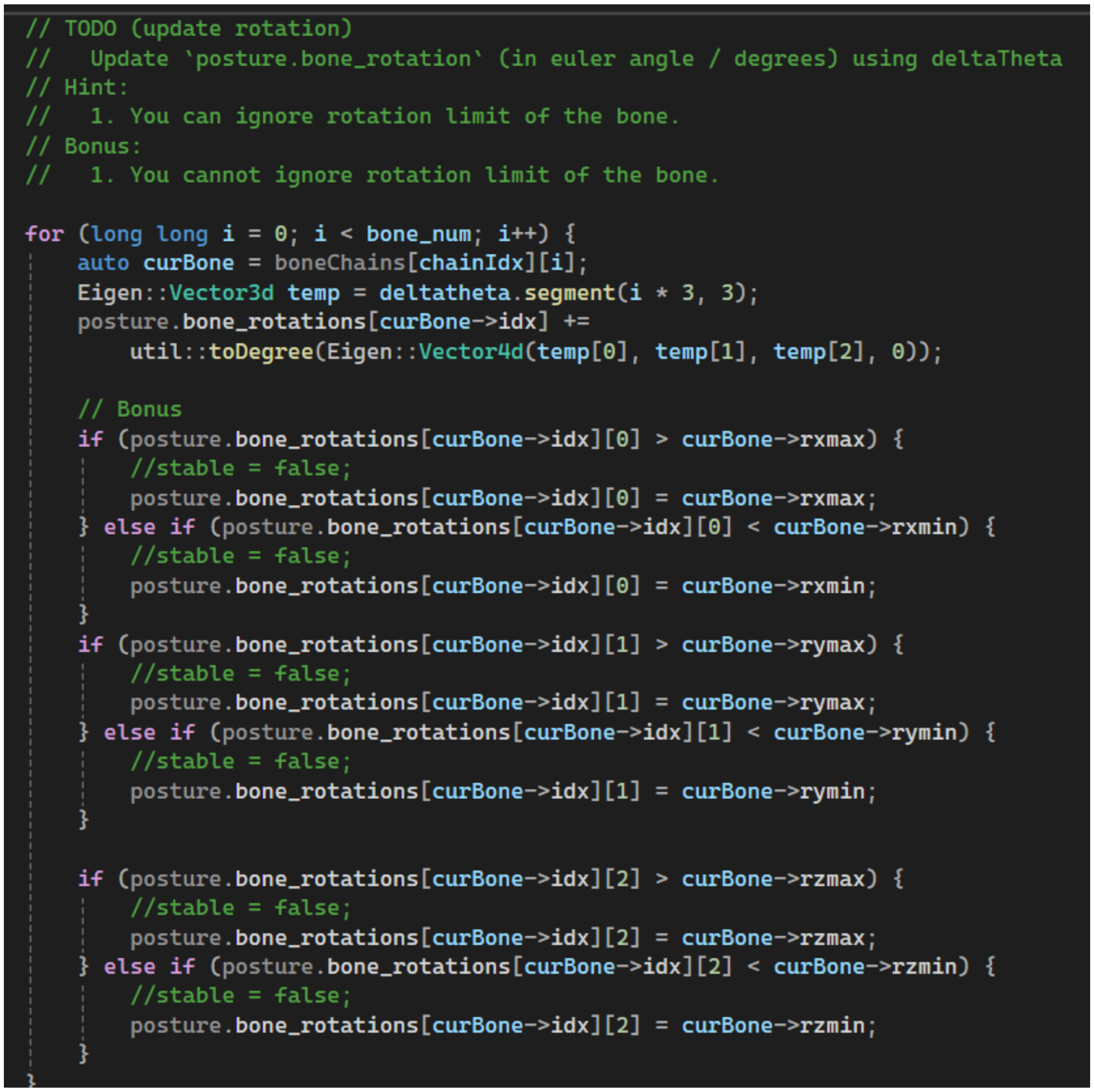
- After playing around with the parameter step, I observed that lowering step (below 1.0) would result in the skeleton's adjusting motion becoming smoother when dragging around the target ball. In other words, the motion won't have a sudden change. This is most likely due to bones adjusting its rotation with smaller increments, allowing for more recalculation (within max_iteration) to gradually converge towards the target, leading to a smoother motion. However, setting step too low might result in jointChain[chainIdx][0] not being within epsilon distance of the target ball after iterating max_iteration number of iterations, leading to unstable. Hence, we must increase max_iterations correspondingly. The reverse is true when increasing step…
- After playing around with the parameter epsilon, I observed that increasing epsilon would allow the skeleton to still move even though jointChain[chainIdx][0] is quite far from the target ball. This is due to:



```
// Return whether IK is stable (i.e. whether the ball is reachable) and let the skeleton not swing its hand in the
// air
for (int i = 0; i < boneChains.size(); ++i) {
    if ((target_pos[i] - *jointChains[i][0]).norm() > epsilon) {
        stable = false;
    }
}
```

  Causing variable stable to remain true, hence updating posture. The reverse is true when decreasing epsilon…

Bonus



```
// TODO (update rotation)
//   Update `posture.bone_rotation` (in euler angle / degrees) using deltaTheta
// Hint:
//   1. You can ignore rotation limit of the bone.
// Bonus:
//   1. You cannot ignore rotation limit of the bone.

for (long long i = 0; i < bone_num; i++) {
    auto curBone = boneChains[chainIdx][i];
    Eigen::Vector3d temp = deltatheta.segment(i * 3, 3);
    posture.bone_rotations[curBone->idx] +=
        util::toDegree(Eigen::Vector4d(temp[0], temp[1], temp[2], 0));

    // Bonus
    if (posture.bone_rotations[curBone->idx][0] > curBone->rxmax) {
        //stable = false;
        posture.bone_rotations[curBone->idx][0] = curBone->rxmax;
    } else if (posture.bone_rotations[curBone->idx][0] < curBone->rxmin) {
        //stable = false;
        posture.bone_rotations[curBone->idx][0] = curBone->rxmin;
    }
    if (posture.bone_rotations[curBone->idx][1] > curBone->rymax) {
        //stable = false;
        posture.bone_rotations[curBone->idx][1] = curBone->rymax;
    } else if (posture.bone_rotations[curBone->idx][1] < curBone->rymin) {
        //stable = false;
        posture.bone_rotations[curBone->idx][1] = curBone->rymin;
    }

    if (posture.bone_rotations[curBone->idx][2] > curBone->rzmax) {
        //stable = false;
        posture.bone_rotations[curBone->idx][2] = curBone->rzmax;
    } else if (posture.bone_rotations[curBone->idx][2] < curBone->rzmin) {
        //stable = false;
        posture.bone_rotations[curBone->idx][2] = curBone->rzmin;
    }
}
```

- As mentioned above
- Check whether the updated rotation exceeds the maximum/minimum rotation degree of each DoF, if so set it as max/min rotation.

Conclusion

To sum up, this report discusses how we make animated skeletons move realistically. We looked at two main parts: Forward Kinematics (FK) and Inverse Kinematics (IK). FK helps figure out where each bone should be and how it should rotate, making animations look smooth. IK lets us move the skeleton to touch a specific point, the target ball, which works well most of the time, as long as we don't push the bones too far.

We learned some important things by adjusting the parameters step and epsilon. Lowering the step parameter leads to smoother motion adjustments but requires careful consideration of the maximum iteration count to allow joint to reach the target ball (within epsilon) before running out of iterations. On the other hand, epsilon helps the skeleton know when to stop moving towards the target. We found that increasing this would let the skeleton stop moving towards the target earlier, causing there to be a gap (of epsilon length) between the target and joint.

Demo Link (Google Drive)

https://drive.google.com/file/d/1J-bHhfoAKFZ6Au71YiiTv_o5ANVOrA_O/view?usp=sharing