Introduction to Computer Animation HW3 Report

109350008 張詠哲

• Introduction/Motivation

這次要在這個作業中實作的也是模擬角色的動作。但是與 HW2 不同的是，HW2 使用的方法是 Forward Kinematics，而這次我們改為使用 Inverse Kinematics。相較於 Forward Kinematics 是根據關節的角度或變換矩陣計算末端骨頭的位置和姿態。而 Inverse Kinematics 則是根據期望的末端骨頭的位置和姿態，計算出關節的角度或變換矩陣，以實現所需的運動。作業中我們會利用上次做的 Forward Kinematics 計算出骨架 Pose，接著計算需要的 Jacobian Matrix。通過 Pseudo Inverse 找到 orientation 的變化，然後更新每個骨頭的旋轉角度來模擬整個骨架的形狀，讓目標骨頭的 End position 更接近 Target。

• Fundamentals

✧ Inverse Kinematics

Inverse Kinematics 的目標是計算影響 end effector 的關節自由度，使其能夠到達我們設定的目標位置。相比於 Forward Kinematics 是在給定角度後，計算到達的點；Inverse Kinematics 是計算給定我們要到達的點後應該改變的角度。由於我們希望逐步到達目標位置，因此需要通過不斷的計算和修正來調整方向，使 end effector 接近目標位置。採用迭代的方法計算，通過每一次小修正，最終達到目標。

✧ Jacobian Matrix

Jacobian Matrix 可以表示為最佳的 linear approximation of a multivariate vector function。其中每個元素表示 end effector 位置或姿態的變化對關節角度的變化的敏感度。可知當調整關節角度時，end effector 的位置和姿態如何變化。這次作業中代表了我們要計算的系統的偏導數。用於定義 end effector 的相對瞬時變化。

✧ Pseudo Inverse

假設有一個線性系統 $Ax = b$，其中 A 是一個 $m \times n$ 的矩陣，x 和 b 是 n 維的向量。當這個系統無法有精確解時，我們可以使用最小平方法

（Least Squares Method）來尋找一個近似解。而 Pseudo Inverse 就是在最小平方法中使用的一個特殊矩陣。

在經過計算上面的 Jacobian Matrix，可以知道 end effector 是如何影響 x、y、z 軸各個角度的關係，接著為了找到方向的變化，我們需要計算 Jacobian Matrix 的 inverse，並將其乘以 V（target 和 end effector 之間的距離）。但是 Inverse Jacobian Matrix 很難計算。因此改用 Pseudo Inverse 來計算 J+來代替。

✧ **Orientation**

我們可以知道每個骨骼的旋轉角度，但是為了讓 end effector 可以到達目標位置，我們需要改變骨頭的方向，逐步計算我們最終的方向.

• Implementation

✧ **forwardSolver**

這部分和 HW2 一樣。

```
bone->start_position = posture.bone_translations[bone->idx];
bone->end_position = posture.bone_translations[bone->idx];
q.push(bone);

while (!q.empty()) {
    acclaim::Bone* temp = q.top();
    q.pop();
    if (temp->idx != 0) {
        temp->start_position = temp->parent->end_position;
        Eigen::Quaternion rota =
            util::rotateDegreeZYX(posture.bone_rotations[temp->idx].x(), posture.bone_rotations[temp->idx].y(),
                                  posture.bone_rotations[temp->idx].z());

        Eigen::Affine3d R = temp->rot_parent_current * rota;

        for (acclaim::Bone* i = temp->parent; i != nullptr; i = i->parent) {
            Eigen::Quaternion i_rota =
                util::rotateDegreeZYX(posture.bone_rotations[i->idx].x(), posture.bone_rotations[i->idx].y(),
                                      posture.bone_rotations[i->idx].z());

            R = i->rot_parent_current * i_rota * R;
        }

        Eigen::Vector4d dir = temp->dir * temp->length;
        temp->end_position = R * dir + temp->start_position;
        temp->rotation = R;

    }

    else {
        temp->rotation = Eigen::Affine3d::Identity();
        temp->start_position = posture.bone_translations[temp->idx];
        temp->end_position = temp->start_position;
    }

    for (acclaim::Bone* i = temp->child; i != nullptr; i = i->sibling) {
        if (vis.find(i->idx) == vis.end()) {
            vis[i->idx] = true;
            q.push(i);
        }
    }
}
```

✧ **pseudoInverseLinearSolver**

這次使用 SVD 來計算 pseudo inverse。矩陣 M 可以分解為 $U\Sigma V^*$，而 pseudo inverse 為$V\Sigma^+U^*$，其中 Σ+ 是 Σ 的轉置，所有元素都變成倒數。在程式碼中，使用 bdcSvd() 函數來進行奇異值分解，並使用 solve() 函數計算 Jacobian 的 pesudo inverse matrix (pinvJacobian)。然後，然後把 target 乘以 pinvJacobian，得到 deltatheta (最小化 | Jacobian * deltatheta – target | 的解)。 最後返回。

```
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();

    // Compute the pseudo-inverse of Jacobian using SVD
    Eigen::MatrixXd pinvJacobian = Jacobian.bdcSvd(Eigen::ComputeThinU | Eigen::ComputeThinV)
                                           .solve(Eigen::MatrixXd::Identity(Jacobian.rows(), Jacobian.rows()));

    // Compute deltatheta that minimizes |jacobian * deltatheta - target|
    deltatheta = pinvJacobian * target;

    return deltatheta;
}
```

✧ **inverseJacobianIKSolver**

■ **Travel the skeleton**

我們需要遍歷骨架以獲得 boneList。這個過程中當我們使用 parent 遍歷骨架時，我們可能會遍歷到 root 然後因為根的 parent 是 nullptr 而被迫中止。因此我首先遍歷骨骼檢查結束骨骼是否可以不通過根到達起始骨骼。

```
// TODO
// Calculate number of bones need to move to perform IK, store in `bone_num`
// a.k.a. how may bones from end_bone to its parent then to start_bone (include both start_bone and end_bone)
// Store the bones need to move to perform IK into boneList
// Hint:
//   1. Traverse from end_bone to start_bone is easier than start to end (since there is only 1 parent)
//   2. If start bone is not reachable from end. Go to root first.
// Note:
//   1. Both start_bone and end_bone should be in the list
acclaim::Bone* current = end_bone;

bool flag = false;
for (acclaim::Bone* itr = current; itr != nullptr; itr = itr->parent) {
    if (itr->idx == start_bone->idx) flag = true;
}

if (flag) {
    for (acclaim::Bone* itr = current; itr->idx != start_bone->parent->idx; itr = itr->parent) {
        boneList.push_back(itr);
    }
} else {
    for (acclaim::Bone* itr = current; itr != nullptr; itr = itr->parent) {
        boneList.push_back(itr);
    }
    for (acclaim::Bone* itr = start_bone; itr->idx != root_bone->idx; itr = itr->parent) {
        boneList.push_back(itr);
    }
}
```

■ **Jacobian**

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

上面的公式為計算 Jacobian Matrix 的內容。P 是結束骨骼的結束位置，r_i是當前骨骼的起始位置，a_i是旋轉軸。旋轉矩陣的第一列是 x 軸，第二列是 y 軸，第三列是 z 軸。最後是僅當骨骼具有 DOF 時才將須計算 jacobian。

```
int column;
for (long long i = 0; i < bone_num; i++) {
    Eigen::Vector4d dis = end_bone->end_position - boneList[i]->start_position;
    Eigen::Vector3d distance = Eigen::Vector3d(dis[0], dis[1], dis[2]);
    Eigen::Affine3d rota = boneList[i]->rotation;
    column = i * 3;

    if (boneList[i]->dofrx) {
        Eigen::Vector4d unit = rota.matrix().col(0);
        Eigen::Vector3d unit_rota = Eigen::Vector3d(unit[0], unit[1], unit[2]);
        Jacobian.col(column) =
            Eigen::Vector4d(unit_rota.cross(distance)[0], unit_rota.cross(distance)[1], unit_rota.cross(distance)[2], 0.0);
    }

    if (boneList[i]->dofry) {
        Eigen::Vector4d unit = rota.matrix().col(1);
        Eigen::Vector3d unit_rota = Eigen::Vector3d(unit[0], unit[1], unit[2]);
        Jacobian.col(column + 1) =
            Eigen::Vector4d(unit_rota.cross(distance)[0], unit_rota.cross(distance)[1], unit_rota.cross(distance)[2], 0.0);
    }

    if (boneList[i]->dofrz) {
        Eigen::Vector4d unit = rota.matrix().col(2);
        Eigen::Vector3d unit_rota = Eigen::Vector3d(unit[0], unit[1], unit[2]);
        Jacobian.col(column + 2) =
            Eigen::Vector4d(unit_rota.cross(distance)[0], unit_rota.cross(distance)[1], unit_rota.cross(distance)[2], 0.0);
    }

}
```

■ **orientation**

得到方向變化後，我們可以將 delta x, delta y, delta z 轉換為 Degree 加上原來的 rotation 得到新的方向。這裡也順便做了 bouns，若是新的方向會超過最大或最小的可旋轉角度(碰不到 target)則為 unstable，而其旋轉角度就設為最大或最小的可旋轉角度。

```
for (long long i = 0; i < bone_num; i++) {
    acclaim::Bone bone = *boneList[i];
    Eigen::Vector4d deg = posture.bone_rotations[bone.idx] + util::toDegree(deltatheta.segment(i * 3, 3));
    bool exceedsLimit = false;

    if (deg[0] < bone.rxmin) {
        deg[0] = bone.rxmin;
        exceedsLimit = true;
    } else if (deg[0] > bone.rxmax) {
        deg[0] = bone.rxmax;
        exceedsLimit = true;
    }

    if (deg[1] < bone.rymin) {
        deg[1] = bone.rymin;
        exceedsLimit = true;
    } else if (deg[1] > bone.rymax) {
        deg[1] = bone.rymax;
        exceedsLimit = true;
    }

    if (deg[2] < bone.rzmin) {
        deg[2] = bone.rymin;
        exceedsLimit = true;
    } else if (deg[2] > bone.rzmax) {
        deg[2] = bone.rymax;
        exceedsLimit = true;
    }

    if (exceedsLimit) {
        stable = false;
    }

    posture.bone_rotations[bone.idx] = deg;

}
```

• Result & Discussion

✧ How different step and epsilon affect the result

因為 Inverse Kinematic 是採用迭代的方法計算，通過每一次小修正，最終達到目標。因此 step 和 epsilon 會影響。

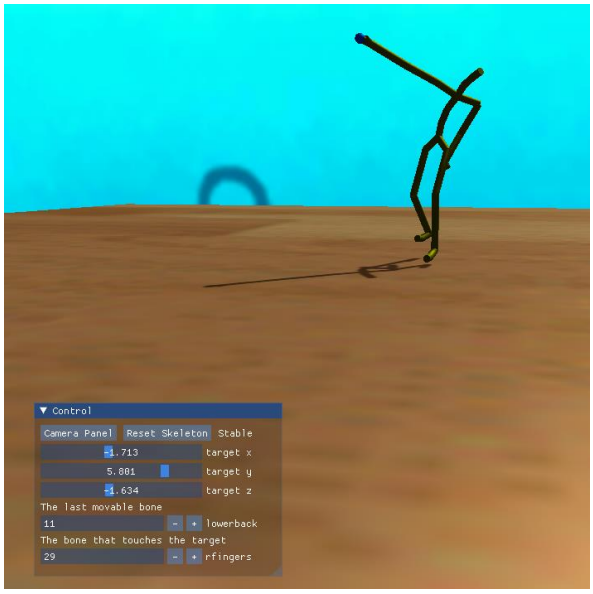
▪ **step affect**：step 太大會導致收斂快，但相對不准確；相反的，如果 step 很小會導致收斂很慢，但相對準確。

▪ **epsilon affect**：distance epsilon 是 target 與 end effector 之間距離誤差的可接受範圍。如果設置太大，兩者之間的距離會比較遠。如果設置得太小，很容易使 end effector 一直被計算。

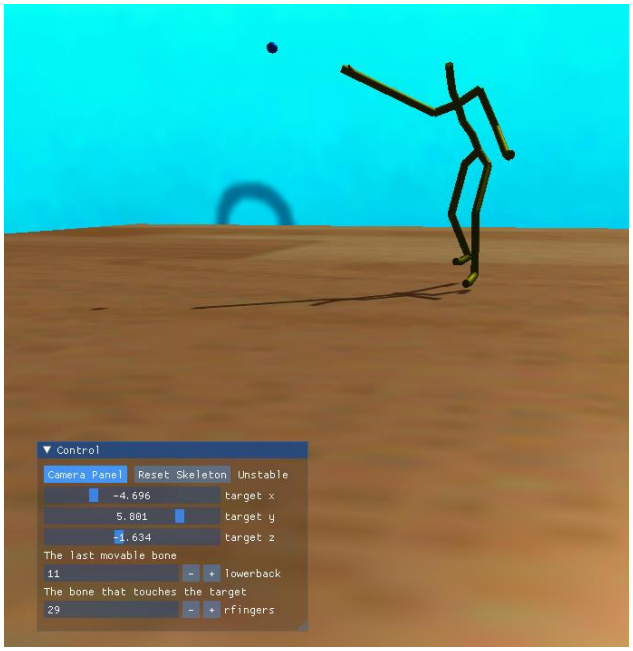
✧ Touch the target or not

當球靠近骨架時，骨架都可以接觸到球。但如果球離骨架很遠，骨架就不能碰到球。

touch



Don’t touch



✧ **Least square solve(LSS)**

解 Least square solve 的方法有多種:

Normal Equations：

透過把 LSS 轉化為求解線性方程組來求解。將系統的 jacobian matrix 成上目標向量，然後解析求解線性方程組來獲得 LSS 解。這種方法在計算上比較簡單，但可能存在數值穩定性的問題。

QR decomposition：

將 jacobian matrix 分解為一個正交矩陣和一個上三角矩陣的乘積。具有較好的數值穩定性。通過 QR 分解，可以使用 Backward Substitution 求解 LSS。

Singular Value Decomposition(SVD)：

SVD 是最廣泛使用的方法之一，可以適用於非方正矩陣。通過 SVD，可以將矩陣分解為三個部分：$V\Sigma U$。可以用於計算矩陣的 pseudo inverse。具有良好的數值穩定性和 robust。這次的作業中採用的即是 SVD 來求解。

而不同的算法對於骨架移動呈現的效果造成的影響可能原因為: 數值精度、收斂速度、平滑度、robust。

- **Bonus**

```
bool exceedsLimit = false;

if (deg[0] < bone.rxmin) {
    deg[0] = bone.rxmin;
    exceedsLimit = true;
} else if (deg[0] > bone.rxmax) {
    deg[0] = bone.rxmax;
    exceedsLimit = true;
}

if (deg[1] < bone.rymin) {
    deg[1] = bone.rymin;
    exceedsLimit = true;
} else if (deg[1] > bone.rymax) {
    deg[1] = bone.rymax;
    exceedsLimit = true;
}

if (deg[2] < bone.rzmin) {
    deg[2] = bone.rymin;
    exceedsLimit = true;
} else if (deg[2] > bone.rzmax) {
    deg[2] = bone.rymax;
    exceedsLimit = true;
}

if (exceedsLimit) {
    stable = false;
}
```

解釋在上面的 orientation

• Conclusion

在本次作業中，我們實作了角色動作的模擬，與上次的 HW2 不同是這次採用了 Inverse Kinematics 的方法。根據期望的末端骨頭位置和姿態，計算出關節的角度或變換矩陣，以實現所需的運動。在本次作業中，我們首先利用上次實作的 Forward Kinematics 方法計算出骨架的 Pose，然後進一步計算所需的 Jacobian Matrix。透過 Pseudo Inverse 找到 orientation 的變化量，並使用這些變化量來更新每個骨頭的旋轉角度，以模擬整個骨架的形狀。這樣做可以使目標骨頭的末端位置更接近期望的目標位置。