HW2

111550143 林彥佑

- ## Introduction

  本次作業的目標是在給定的骨架上，實作一套基於 Jacobian 的逆向運動學系統。目標是讓 bones 可以透過旋轉、移動，觸碰到指定的球體位置。當骨骼接近障礙物時，需要自動避開，以避免穿越障礙物。

- ## Fundamentals

  Forward Kinematics:

  給定骨架上每一個關節的位置、旋轉角度以及 Bones 之間的層級結構，推導出每個 bone 在 global 座標系下的起點位置（start_position）、終點位置（end_position）以及旋轉矩陣（rotation）。mapping from joint space to cartesian space

  Inverse Kinematics:

  逆向運動學的目標是，給定一個目標位置（target），反推各個關節應該做出哪些旋轉變化，才能使末端骨骼接近或到達該目標。mapping from cartesian space to joint space

  inverse-Jacobain method:

  每次利用 Jacobian 矩陣，計算目前姿勢下的小角度微分，通過連續小步驟調整骨架，讓末端骨骼逐漸逼近目標。由於 Jacobian 本身可能不可逆，因此在實務上，我們使用的是其偽逆矩陣（Pseudo-Inverse）來進行近似求解。

  Acclaim Skeleton:

  本作業所使用的骨架模型，來源於 Acclaim 動作捕捉系統。Acclaim 是一種早期廣泛使用於電腦動畫與虛擬角色中的動作資料格式。ASF（Acclaim Skeleton File）記錄了每個骨骼（Bone）的初始層級結構、骨骼之間的連接關係、骨骼的方向（dir）、長度（length）、以及每個骨骼的旋轉自由度（DOF）設定。

- ## Implementation

  forwardSolver:

  從 root 出發，設置 bone 的 start, end, rotation，再透過寫一個 dfs 去 traverse 每一個 bone。

```
void forwardSolver(const acclaim::Posture& posture, acclaim::Bone* bone) {
    // TODO#1: Forward Kinematic
    // Hint:
    // - Traverse the skeleton tree from root to leaves.
    // - Compute each bone's global rotation and global position.
    // - Use local rotation (from posture) and bone hierarchy (parent rotation, offset, etc).
    // - Remember to update both bone->start_position and bone->end_position.
    // - Use bone->rotation to store global rotation (after combining parent, local, etc).
    std::vector<bool> visited(31, false);
    bone->start_position = posture.bone_translations[0];
    bone->rotation = bone->rot_parent_current * util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);
    bone->end_position = bone->start_position + bone->rotation * bone->dir.normalized() * bone->length;
    visited[bone->idx] = true;
    if (bone->child) {
        dfs(bone->child, posture, visited);
    }
}
```

```cpp
void dfs(acclaim::Bone* bone, const acclaim::Posture& posture, std::vector<bool>& visited) {
    if (visited[bone->idx]) {
        return;
    }

    visited[bone->idx] = true;

    if (bone->parent) {
        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.normalized() * bone->length;
    }

    if (bone->child && !visited[bone->child->idx]) {
        dfs(bone->child, posture, visited);
    }

    for (acclaim::Bone* sibling = bone->sibling; sibling != nullptr; sibling = sibling->sibling) {
        if (!visited[sibling->idx]) {
            dfs(sibling, posture, visited);
        }
    }
}
```

Traverse 每個 bone 時，看他是否有 parent，若有那他的 start 就是 parent 的 end，接著計算該 bone 的 global 旋轉矩陣，公式為:

$$ {}_{i}R_{asf} = {}_{i}^{i+1}R = {}_{i}^{0}R \cdot {}_{0}^{i+1}R $$

且結尾位置為:

$$ V_i = \hat{V}_i \cdot l_i $$

$$ {}_{i}T = {}_{0}^{i-1}RV_{i-1} + {}_{i-1}T $$

若該 bone 有 child 則繼續執行 dfs，以及拜訪每一個尚未處理過的 siblings。

pseudoInverseLinearSolver:

使用 Eigen 函式庫提供的 JacobiSVD，對 Jacobian 矩陣進行奇異值分解（Singular Value Decomposition, SVD）

```cpp
Eigen::VectorXd pseudoInverseLinearSolver(const Eigen::Matrix4Xd& Jacobian, const Eigen::Vector4d& target) {
    Eigen::VectorXd deltatheta;
    // TODO#2: Inverse linear solver (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::JacobiSVD<Eigen::Matrix4Xd> svd(Jacobian, Eigen::ComputeThinU | Eigen::ComputeThinV);
    deltatheta = svd.solve(target);

    return deltatheta;
}
```

使用 SVD 分解後的結果，svd.solve(target)計算 pseudo-inverse，得到最適合的關節角度微小變化向量

inverseJacobianIKSolver

#todo 3-1

找出從起點 start_bone 到終點 end_bone 之間，所有需要被調整角度的 boneList，並記錄 bone 數量。這份清單中的 bone，將是之後建立 Jacobian 矩陣與更新角度的對象。

```
// 3-1 start
while (current != nullptr) {
    boneList.push_back(current);
    bone_num++;
    if (current == start_bone) break;
    current = current->parent;
}
// 3-1 end
```

#todo 3-2

```
// 3-2 start
Jacobian.setZero();
for (int i = 0; i < bone_num; i++) {
    acclaim::Bone* bone = boneList[i];
    Eigen::Vector3d arm = end_bone->end_position.head<3>() - bone->start_position.head<3>();

    if (bone->dofrx) {
        Jacobian.col(i * 3).head<3>() = ((bone->rotation * Eigen::Vector4d(1, 0, 0, 0)).head<3>()).cross(arm);
        Jacobian.col(i * 3)(3) = 0;
    }
    if (bone->dofry) {
        Jacobian.col(i * 3 + 1).head<3>() = ((bone->rotation * Eigen::Vector4d(0, 1, 0, 0)).head<3>()).cross(arm);
        Jacobian.col(i * 3 + 1)(3) = 0;
    }
    if (bone->dofrz) {
        Jacobian.col(i * 3 + 2).head<3>() = ((bone->rotation * Eigen::Vector4d(0, 0, 1, 0)).head<3>()).cross(arm);
        Jacobian.col(i * 3 + 2)(3) = 0;
    }
}
// 3-2 end
```

遍歷剛剛建立好的 boneList，處理每一根需要移動的骨骼，Arm 是從目前骨骼的起點位置到 end effector 位置的向量，它代表骨骼若在此處旋轉，會怎麼影響末端的槓桿效果基準。

針對每個自由度建立 Jacobian 列，每個骨骼可能有三個旋轉自由度（rx、ry、rz），分別對應 X、Y、Z 軸旋轉。對每一個 DOF-rotation*(x, y, z, 0)把 X/Y/Z 軸轉換到 global 座標下的方向，再對 arm vector 做 cross product，表示繞 X/Y/Z 軸微小旋轉，會讓 end effector 產生什麼瞬時速度方向。計算出來的向量填入 Jacobian 的對應 column。

#todo 3-3

```
if (obsActive) {
    for (int i = 0; i < bone_num; i++) {
        acclaim::Bone* bone = boneList[i];
        Eigen::Vector4d center = (bone->start_position + bone->end_position) / 2;
        Eigen::Vector4d obs_vector = center - obs_pos;
        double dist = obs_vector.head<3>().norm();

        if (dist < obsAvoidThreshold && dist > 1e-6) {
            Eigen::Vector3d d = obs_vector.head<3>().normalized();
            desiredVector.head<3>() += d * (obsAvoidThreshold - dist);
        }
    }
}
```

當骨架的某根骨骼靠近障礙物時，系統會產生一個排斥向量推開骨骼以避免穿越障礙物。

遍歷 boneList 中的每一根骨骼，取骨骼起點與終點位置的平均，作為骨骼的中心點代表這根骨骼在空間中的主要位置，計算中心到障礙物中心的向量 obs_vector，dist 是這個向量在三維空間中的長度，當骨骼中心到障礙物的距離小於設定的 obsAvoidThreshold，且大於極小值，就

需要進行排斥處理，先將 obs_vector 正規化得到排斥方向（單位向量），然後將排斥向量乘上一個強度比例 (threshold−dist)，距離越近，推力越大。最後，將這個排斥向量加到 desiredVector 的前三維（位置變化量）中。

#todo 3-4

```
for (int iter = 0; iter < max_iteration; ++iter) {
    for (int i = 0; i < bone_num; i++) {
        acclaim::Bone* bone = boneList[i];
        if (bone->dofrx) {
            posture.bone_rotations[bone->idx][0] += deltatheta(i * 3) * 180 / PI;
        }
        if (bone->dofry) {
            posture.bone_rotations[bone->idx][1] += deltatheta(i * 3 + 1) * 180 / PI;
        }
        if (bone->dofrz) {
            posture.bone_rotations[bone->idx][2] += deltatheta(i * 3 + 2) * 180 / PI;
        }

        // Bonus
        if (bone->dofrx) {
            if (posture.bone_rotations[bone->idx][0] < bone->rxmin) {
                posture.bone_rotations[bone->idx][0] = bone->rxmin;
            } else if (posture.bone_rotations[bone->idx][0] > bone->rxmax) {
                posture.bone_rotations[bone->idx][0] = bone->rxmax;
            }
        }
        if (bone->dofry) {
            if (posture.bone_rotations[bone->idx][1] < bone->rymin) {
                posture.bone_rotations[bone->idx][1] = bone->rymin;
            } else if (posture.bone_rotations[bone->idx][1] > bone->rymax) {
                posture.bone_rotations[bone->idx][1] = bone->rymax;
            }
        }
        if (bone->dofrz) {
            if (posture.bone_rotations[bone->idx][2] < bone->rzmin) {
                posture.bone_rotations[bone->idx][2] = bone->rzmin;
            } else if (posture.bone_rotations[bone->idx][2] > bone->rzmax) {
                posture.bone_rotations[bone->idx][2] = bone->rzmax;
            }
        }
```

遍歷每一根需要調整的骨骼，對每個骨骼，如果該骨骼允許某個旋轉自由度（dofrx, dofry, dofrz），則將對應的 deltatheta 加到 posture.bone_rotations 中。

Bonus: 骨架的每個旋轉自由度都有一個允許的最小角度（rxmin, rymin, rzmin）與最大角度（rxmax, rymax, rzmax），更新完角度後，必須檢查是否超出範圍，若超出則修正到邊界值。

#todo 3-5

```
// 3-5 start
forwardSolver(posture, root_bone);
Eigen::Vector4d finalDesiredVector = target_pos - end_bone->end_position;
if (finalDesiredVector.head<3>().norm() < epsilon) {
    return true;
} else {
    posture = original_posture;
    forwardSolver(posture, root_bone);
    return false;
}
// 3-5 end
```

在 IK 計算完畢後，首先重新執行一次 Forward Kinematics，用最新的角度更新所有骨骼的位置與旋轉資訊，計算目前 end_bone 的 end_position 與 target_pos 之間的向量差，計算其歐式距離，如果這個距離小於事先設定好的 epsilon，代表骨架已經成功接觸到目標，return true。如果誤差超過 epsilon，代表無法靠近目標，此時需要回復到原本的姿勢，避免因錯誤的 IK 更新造成骨架異常，再次呼叫 forwardSolver，讓骨架根據 original_posture 更新世界位置，最後回傳 false，代表 IK 失敗，球不可觸及。

Bonus: Return whether IK is stable so that the skeleton would not swing its hand in the air

- Result and Discussion

在皆未改變的情況下，骨架能黏著球進行運動，且遇到障礙物時不會穿越，較不合理的彎曲也因為我們有限制最大最小值而不會出現。

Different step:

極小(0.001)->骨架無法跟著球移動，待在原地沒有動作

小(0.01)->原始情況不會找到球，但經移動求到特定方位十，又能跟著球運動

原始(0.1)->動作平滑

大(1)->看不出與 0.1 之差別

極大(5)->動作明顯斷斷續續，較瞬間，不穩定，動作大且不自然

step 小：穩定但較慢，自然平滑

step 大：快速但可能震盪、不自然

Different epsilon:

epsilon 控制著 IK 需要多接近目標，才算達成成功

原始(1E-3)->正常

大(1E-1)->骨頭末端球會有偏上偏下的情況

極大(1)->球沒在目標 bone 上，移動球，骨架仍會跟著運動

極小(1E-10)->碰球的 bone 精準的跟著球

能否成功觸碰目標主要受到以下因素影響：

骨架最大可達範圍：骨架從起點到末端骨骼的最長延伸距離，決定了能夠觸碰多遠的目標。

目標位置：若目標距離超出骨架的最大可達範圍，則即使多次迭代也無法觸碰成功。

初始姿勢：若初始骨架姿勢離目標過遠，且每次調整步伐（step）過小，可能需要非常多次迭代才能成功收斂，甚至失敗。

如果目標距離骨架太遠，系統會自動判斷 IK 不可收斂，並恢復原始姿勢，避免產生錯誤的骨架變形。

此次作業選擇採用 Least Squares Method，透過 SVD 計算 Jacobian 的 Pseudo-Inverse，從而穩定地求解出最佳近似解。

使用 Least Squares 的好處包括：

適用於非方陣系統：即使自由度數量多於或少於空間維度，也能找到最適合的解。

避免奇異問題：即使 Jacobian 存在接近奇異的情況，SVD 也能提供數值穩定的解。

誤差最小化：最小化末端骨骼與目標之間的距離誤差，使運動更加合理與自然。

- Bonus

  Return whether IK is stable so that the skeleton would not swing its hand in the air

  Take joint limits into account in bool inverseJacobianIKSolver

  皆有完成且在前面提及

```
// Bonus
if (bone->dofrx) {
    if (posture.bone_rotations[bone->idx][0] < bone->rxmin) {
        posture.bone_rotations[bone->idx][0] = bone->rxmin;
    } else if (posture.bone_rotations[bone->idx][0] > bone->rxmax) {
        posture.bone_rotations[bone->idx][0] = bone->rxmax;
    }
}
if (bone->dofry) {
    if (posture.bone_rotations[bone->idx][1] < bone->rymin) {
        posture.bone_rotations[bone->idx][1] = bone->rymin;
    } else if (posture.bone_rotations[bone->idx][1] > bone->rymax) {
        posture.bone_rotations[bone->idx][1] = bone->rymax;
    }
}
if (bone->dofrz) {
    if (posture.bone_rotations[bone->idx][2] < bone->rzmin) {
        posture.bone_rotations[bone->idx][2] = bone->rzmin;
    } else if (posture.bone_rotations[bone->idx][2] > bone->rzmax) {
        posture.bone_rotations[bone->idx][2] = bone->rzmax;
    }
}
```

```
// 3-5 start
forwardSolver(posture, root_bone);
Eigen::Vector4d finalDesiredVector = target_pos - end_bone->end_position;
if (finalDesiredVector.head<3>().norm() < epsilon) {
    return true;
} else {
    posture = original_posture;
    forwardSolver(posture, root_bone);
    return false;
}
// 3-5 end
```

- Conclusion

  本次作業中，我成功實作了基於 Jacobian 的逆向運動學，並在骨架上進行了目標觸碰與障礙物避開的測試。整體過程涵蓋了 Forward Kinematics、Jacobian 建立、角度更新與範圍限制，以及最終的收斂性判定。

  從實驗結果可以觀察到，調整 step 大小與 epsilon 值對收斂速度與精度有明顯影響：

  也驗證了骨架是否能成功觸碰到目標，並分析了在目標位置超出可達範圍時，系統正確回復原始姿勢的情況，提升了整體穩定性。

  我學到了骨架模擬相關的知識及上課內容在程式上的實作，也讓我清楚在電腦動畫中如何處理障礙物，對 Jacobian 在此堂課的使用有更深一步的了解。