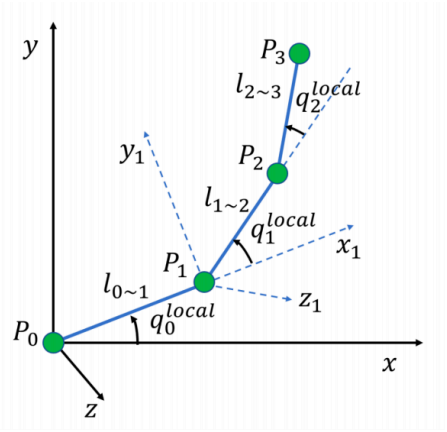
Lab4 Report

2300012929 尹锦润

Task 1: Inverse Kinematics

Subtask 1

对于前向运动学，我们可以参考 讲义上图示，



全局旋转就是前一个的全局旋转 * 相对旋转。

全局位置就是前一个全局位置+相对位移在全局旋转下的全局位移。

进而，写出如下代码：

```
for (int i = StartIndex; i < ik.JointLocalOffset.size(); i++) {
    ik.JointGlobalRotation[i] = ik.JointGlobalRotation[i - 1] *
ik.JointLocalRotation[i];
    auto transRotation = glm::mat4_cast(ik.JointGlobalRotation[i - 1]) *
glm::vec4(ik.JointLocalOffset[i], 1.0f);
    ik.JointGlobalPosition[i] = ik.JointGlobalPosition[i - 1] +
glm::vec3(transRotation.x, transRotation.y, transRotation.z) / transRotation.w;
}
```

Subtask 2

`InverseKinematicsCCD` 参考讲义上的流程，我们在每个 `iteration` 中进行如下操作：

```cpp
for (int i = ik.NumJoints() - 2; i >= 0; i--) {
    ik.JointLocalRotation[i] =
        glm::rotation(
            glm::normalize(ik.EndEffectorPosition() - ik.JointGlobalPosition[i]),
            glm::normalize(EndPosition              - ik.JointGlobalPosition[i])
        ) * ik.JointLocalRotation[i];
    ForwardKinematics(ik, i);
}
```

也就是每次通过旋转线确定旋转角度，进而就是相对 `rotation`。

Subtask 3

`InverseKinematicsFABR`在讲义上没有详细流程，我们请出 GPT 老师为我们介绍算法逻辑：

步骤 1：前向传递 (Forward Reaching)

1. 将末端效应器 P_n 移动到目标位置 T：
$$P'_n = T$$
2. 从末端效应器开始，逐个调整每个关节的位置，保持关节之间的长度不变：
$$P'_{i-1} = P'_i + \frac{L_i}{\|P'_i - P_{i-1}\|}(P_{i-1} - P'_i)$$
其中 P'_{i-1} 是调整后的上一个关节位置，$\|P'_i - P_{i-1}\|$ 是当前关节之间的距离。
3. 重复此操作，直到调整到根节点 P_0。

步骤 2：后向传递 (Backward Reaching)

1. 将根节点 P_0 重新固定到初始位置（或其他约束位置）：
$$P'_0 = P_0$$
2. 从根节点开始，逐个调整每个关节的位置，保持关节之间的长度不变：
$$P'_i = P'_{i-1} + \frac{L_i}{\|P_i - P'_{i-1}\|}(P_i - P'_{i-1})$$
其中 P'_i 是调整后的下一个关节位置。
3. 重复此操作，直到调整到末端效应器 P_n。

再结合已经有的代码框架，写出：

```cpp
glm::vec3 next_position         = EndPosition;
backward_positions[nJoints - 1] = EndPosition;

for (int i = nJoints - 2; i >= 0; i--) {
    auto r = glm::normalize(ik.JointGlobalPosition[i] - next_position);
    next_position = next_position + r * ik.JointOffsetLength[i + 1];
    backward_positions[i] = next_position;
```

```
}

// forward update
glm::vec3 now_position = ik.JointGlobalPosition[0];
forward_positions[0] = ik.JointGlobalPosition[0];
for (int i = 0; i < nJoints - 1; i++) {
    auto r =  glm::normalize(backward_positions[i + 1] - now_position);
    now_position = now_position + r * ik.JointOffsetLength[i + 1];
    forward_positions[i + 1] = now_position;
}
ik.JointGlobalPosition = forward_positions;
```

需要注意的是关节 Offsetlength 放在了标号大的一端点。

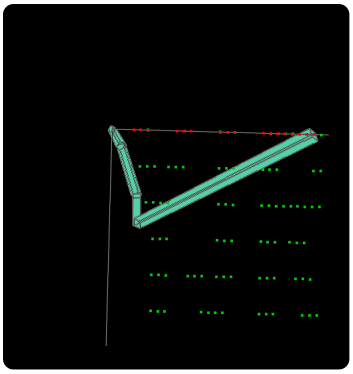
Subtask 4

绘制自定义曲线，这里通过转换 字符画网站获取了笔者姓名首字母：

```
std::string C = R"(
  ...    ...    ...    ..........
   ...  ...     ...    ...     ...
    ....        ...    .........
     ...        ...    ...  ...
     ...   ...  ...    ...   ...
     ...     ....     ...    ...
)";
```

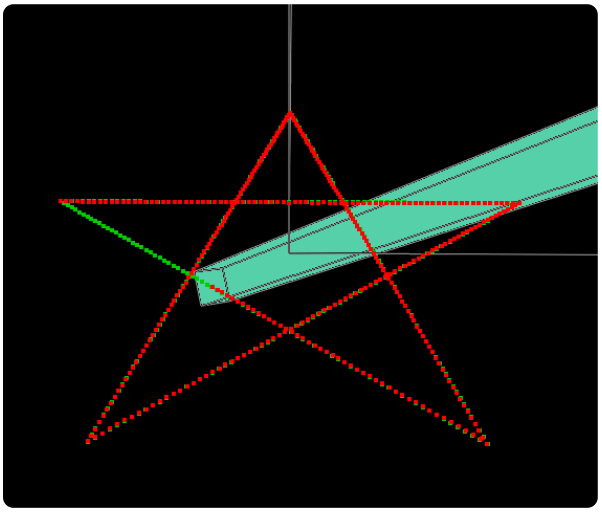
然后开始绘制

```
using Vec3Arr = std::vector<glm::vec3>;
Vec3Arr custom;
for (int i = 0; i < 6; i++) for(int j = 0; j < 30; j++) if(C[i * 31 + j] == '.')
    custom.emplace_back(glm::vec3(i / 6.f, 0.f, j / 30.f));
std::shared_ptr<Vec3Arr> custom_ptr(new Vec3Arr(custom.size()));
std::copy(custom.begin(), custom.end(), custom_ptr->begin());
return custom_ptr;
```

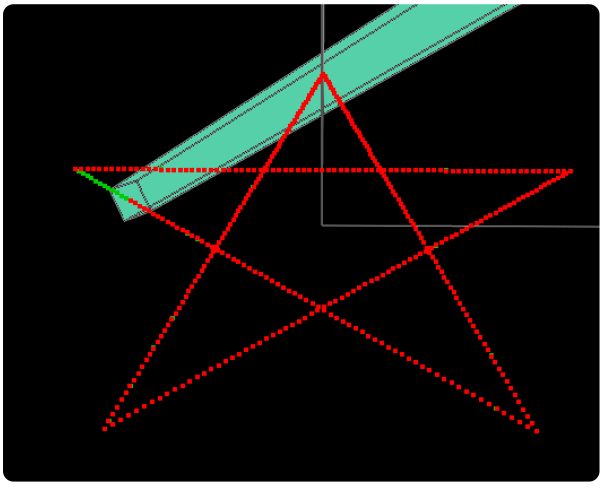


其余效果

CCD_IK:



FABR_IK:



其余问题

> 1. 如果目标位置太远，无法到达，IK 结果会怎样?

会导致机械臂在长度范围内笔直地伸向目标节点。

> 2. 比较 CCD IK 和 FABR IK 所需要的迭代次数。

FABR IK 需要的迭代次数比较少，在迭代次数到 3 这种级别表现还行。

而 CCD IK 通常需要 30～50 次才能表现不错。

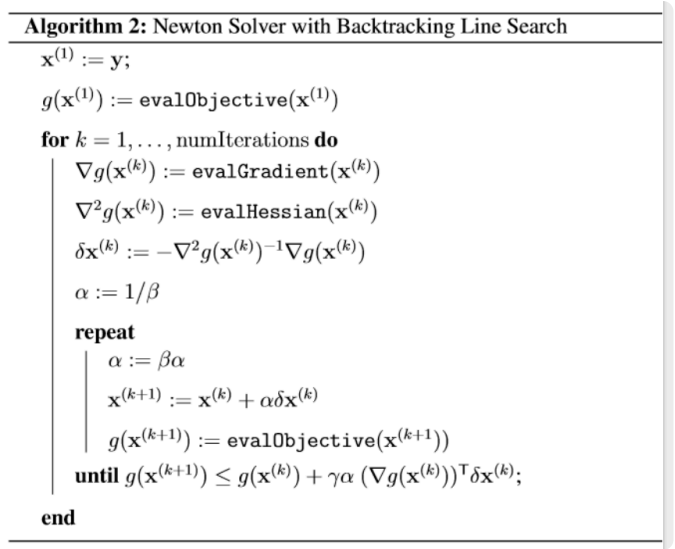
因此 FABR IK 更快。

> 3. 由于 IK 是多解问题，在个别情况下，会出现前后两帧关节旋转抖动的情况。怎样避免或是缓解这种情况?

可能可以通过引入旋转变化率来减少抖动情况?

Task 2: Mass-Spring System

核心算法可以参考课件上的图片：



进而写出类似框架，需要注意的是 `y` 是**每次更新的常量**，以及需要注意上面写的是 `until`，只要场景没有更新，`y`就不更新：

```
Eigen::VectorXf x_origin = glm2eigen(system.Positions);
Eigen::VectorXf y = calc_y(system, glm2eigen(system.Positions), dt);
Eigen::VectorXf x = y;
float g = calc_g(system, x, y, dt);
int numIter = 5;
for(int k = 1; k < numIter; k++) {
    Eigen::VectorXf delta_g = calc_dg(system, x, y, dt);
    Eigen::SparseMatrix delta_g2 = calc_ddg(system, glm2eigen(system.Positions),
y, dt);
    Eigen::VectorXf delta_x = ComputeSimplicialLLT(delta_g2, -delta_g);
    float beta = 0.8, alpha = 1 / beta, g_new = g, gamma = 0.001;
    Eigen::VectorXf x_new = x; //, y_new = y;
    do {
        alpha *= beta;
        x_new = x + alpha * delta_x;
        // y_new = calc_y(system, x_new, dt);
        g_new = calc_g(system, x_new, y, dt);
    } while(g_new > g + alpha * gamma * delta_g.dot(delta_x));
    x = x_new;
    g = g_new;
    // y = y_new;
}
```

之后是信息更新，仿照之前的代码，注意 Fixed 的节点即可，可以直接通过位移来计算速度不需要重新代入式子中。

```
std::vector<glm::vec3> newV = eigen2glm((x - x_origin) / dt);
std::vector<glm::vec3> newX = eigen2glm(x);
for(int i = 0; i < system.Positions.size(); i++) {
    if(system.Fixed[i]) continue;
    system.Positions[i] = newX[i];
    system.Velocities[i] = newV[i];
}
```

对于具体函数，参照讲义上的公式敲上去：

```
Eigen::VectorXf calc_y(MassSpringSystem &system, Eigen::VectorXf const Positions,
float const dt) {
    return  Positions + dt * glm2eigen(system.Velocities) +
            dt * dt / system.Mass * glm2eigen(
```

```
                std::vector<glm::vec3>(
                    system.Positions.size(),
                    glm::vec3(0, -system.Gravity, 0)
                )
            );
}

glm::vec3 getVec3(Eigen::VectorXf x, int pos) {
    return glm::vec3({ x[pos * 3], x[pos * 3 + 1], x[pos * 3 + 2] });
}
```

```
float calc_g(MassSpringSystem &system, Eigen::VectorXf const Positions,
Eigen::VectorXf const y, float const dt) {
    Eigen::VectorXf tmp = Positions - y;
    float ret = tmp.dot(tmp) * system.Mass / (2 * dt * dt);
    for(const auto spring : system.Springs) {
        auto const p0 = spring.AdjIdx.first;
        auto const p1 = spring.AdjIdx.second;
        glm::vec3 const x01 = getVec3(Positions, p1) - getVec3(Positions, p0);
        // glm::vec3 const e01 = glm::normalize(x01);
        ret += 0.5 * system.Stiffness * glm::pow(glm::length(x01) -
spring.RestLength, 2);
    }
    return ret;
}
```

```
Eigen::VectorXf calc_dg(MassSpringSystem &system, Eigen::VectorXf const Positions,
Eigen::VectorXf const y, float const dt) {
    Eigen::VectorXf ret = (Positions - y) * system.Mass / (dt * dt);
//Eigen::VectorXf::Zero(Positions.size());
    for(const auto spring : system.Springs) {
        auto const p0 = spring.AdjIdx.first;
        auto const p1 = spring.AdjIdx.second;
        glm::vec3 const x01 = getVec3(Positions, p1) - getVec3(Positions, p0);
        glm::vec3 const e01 = glm::normalize(x01);
        glm::vec3 f = system.Stiffness * (glm::length(x01) - spring.RestLength) *
e01;
        for(int i = 0; i < 3; i++) {
            ret[p0 * 3 + i] -= f[i];
            ret[p1 * 3 + i] += f[i];
        }
    }
    return ret;
}
```

```cpp
Eigen::SparseMatrix<float> calc_ddg(MassSpringSystem &system, Eigen::VectorXf const Positions, Eigen::VectorXf const y, float const dt) {
    std::vector<Eigen::Triplet<float>> triplets;
    for(const auto spring : system.Springs) {
        auto const p0 = spring.AdjIdx.first;
        auto const p1 = spring.AdjIdx.second;
        glm::vec3 const x01 = getVec3(Positions, p1) - getVec3(Positions, p0);
        glm::vec3 const e01 = glm::normalize(x01);
        glm::mat3 f(0);
        for(int i = 0; i < 3; i++) for(int j = 0; j < 3; j++)
            f[i][j] += system.Stiffness * (
                x01[i] * x01[j] / glm::dot(x01, x01) +
                (1 - spring.RestLength / glm::length(x01)) * ((i == j) - x01[i] * x01[j] / glm::dot(x01, x01))
            );
        for(int i = 0; i < 3; i++) for(int j = 0; j < 3; j++) {
            triplets.emplace_back(p0 * 3 + i, p0 * 3 + j, f[i][j]);
            triplets.emplace_back(p0 * 3 + i, p1 * 3 + j, -f[i][j]);
            triplets.emplace_back(p1 * 3 + i, p0 * 3 + j, -f[i][j]);
            triplets.emplace_back(p1 * 3 + i, p1 * 3 + j, +f[i][j]);
        }
    }
    for(int i = 0; i < Positions.size() * 3; i++) {
        triplets.emplace_back(i, i, system.Mass / (dt * dt));
    }
    return CreateEigenSparseMatrix(Positions.size() * 3, triplets);
}
```

以及要注意方向问题，这里比较难调试，需要多试错，对照讲义。