Task1 Inverse Kinematics

Sub-Task1 Forward Kinematic

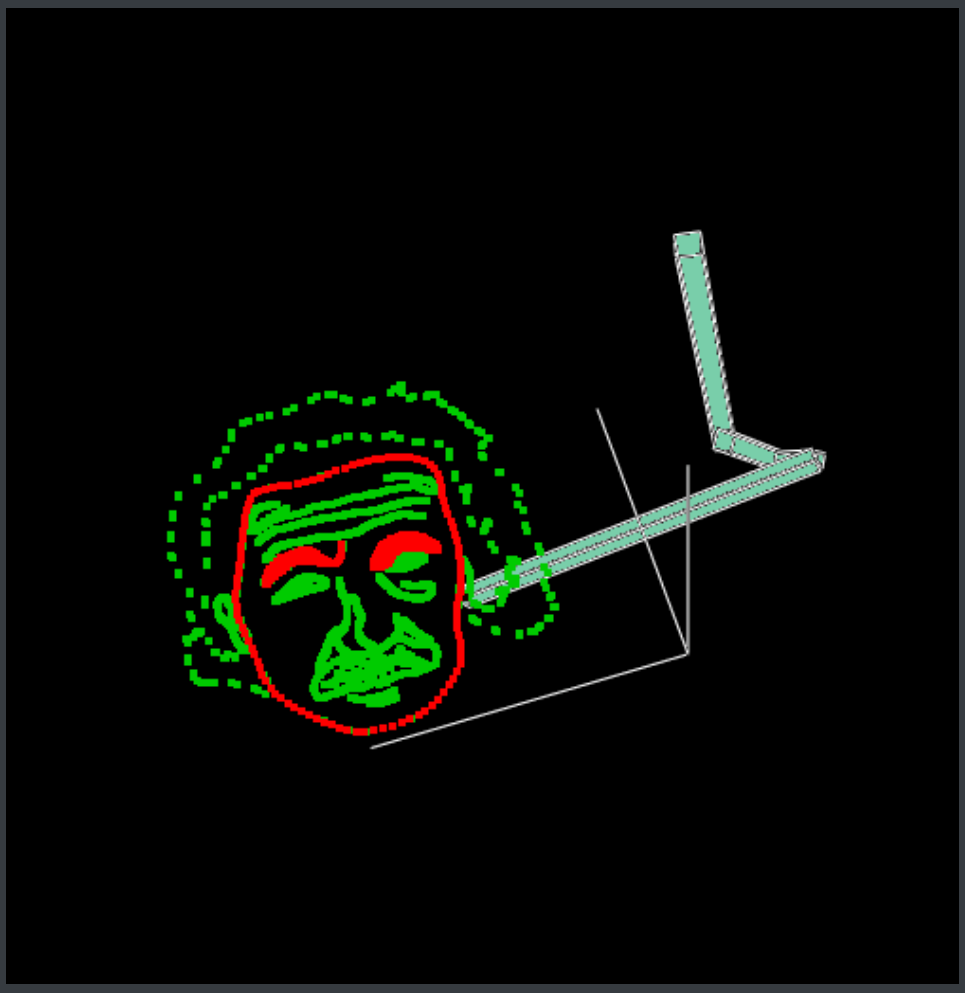
这一部分比较简单，就是根据我们前向运动的公式，先获取当前节点在局部坐标系中相对于父节点的位置，然后将其旋转到全局坐标系中获取全局坐标系的相对位置，然后加上父节点的全局位置以获取当前节点的全局位置，最后更新系统变量。这里由于旋转满足$q_n = q_{n-1}q_n^{local}$，所以每次更新全局旋转的时候直接用父节点的全局旋转乘以local旋转即可。实现上的细节要注意，由于框架是用四元数实现的，所以需要函数 `glm::mat4_cast` 把四元数旋转转化成4×4的旋转矩阵。同时我们计算的时候也要用齐次坐标，转换回普通坐标的时候不要忘了归一。代码如下：

```
glm::vec4 pos=glm::vec4(ik.JointLocalOffset[i], 1.0f);
pos=glm::mat4_cast(ik.JointGlobalRotation[i-1])*pos;
ik.JointGlobalPosition[i]=ik.JointGlobalPosition[i-1]+glm::vec3(pos.x,pos.y,pos
.z)/pos.w;
ik.JointGlobalRotation[i]=ik.JointGlobalRotation[i-1]*ik.JointLocalRotation[i];
```

Sub-Task2 CCD IK

这一部分的实现思路也很直接，就是从最后一根棒子（也就是倒数第二个点）开始往回走，每次计算从当前点到尖端的方向 `to_end` 以及当前点到目标点的方向 `to_target`，得出旋转数 `glm::quat rot=glm::rotation(glm::normalize(to_end),glm::normalize(to_target));`，然后在局部旋转中乘上这个旋转数更新即可。最后用前向运动更新位置。

效果图如下,*以60fps进行录制*:



可以看到，机械臂能够对位置的变化做出正确的反应，且在同一片点之间，它的运动是连贯的。

但是相比参考图，机械臂的动作似乎快了很多，窗口的fps也非常高，达到了600以上，我不清楚这是否是wsl的问题

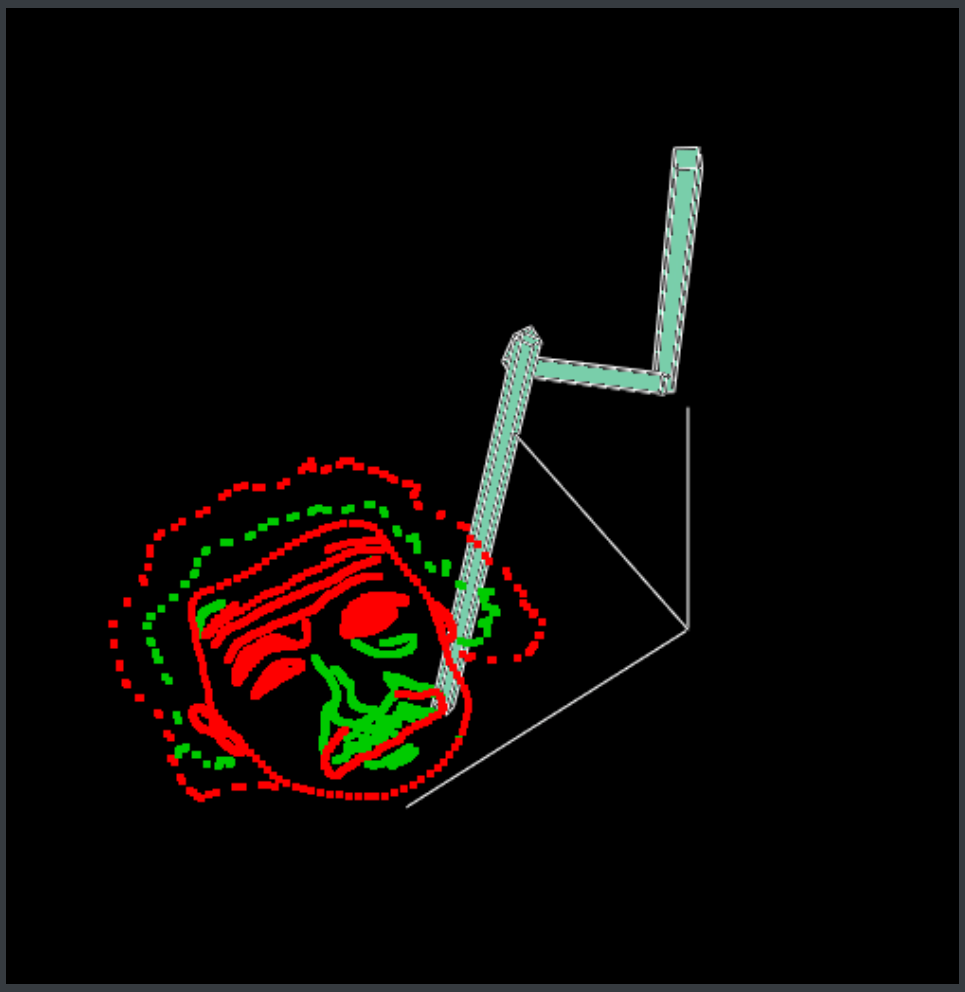
Sub-Task3 FABR IK

这一部分，我参考了知乎上的一篇文章里面的写法。简单来说，FABR IK的思想就是每次按骨节长度把端点沿着目标方向前后拉，每次只移动点的位置以满足长度约束，逐步逼近目标位置。例如在反向阶段：

```
glm::vec3 dir = glm::normalize(ik.JointGlobalPosition[i] - next_position);
backward_positions[i]=next_position+dir*ik.JointOffsetLength[i+1];
next_position=backward_positions[i];
```

我们每次对齐当前点和子节点，然后在同一个方向上把当前点拉到和子节点的距离为杆的长度即可。前向也是类似。

效果图如下，*以60fps进行录制*:



Sub-Task4 Custom

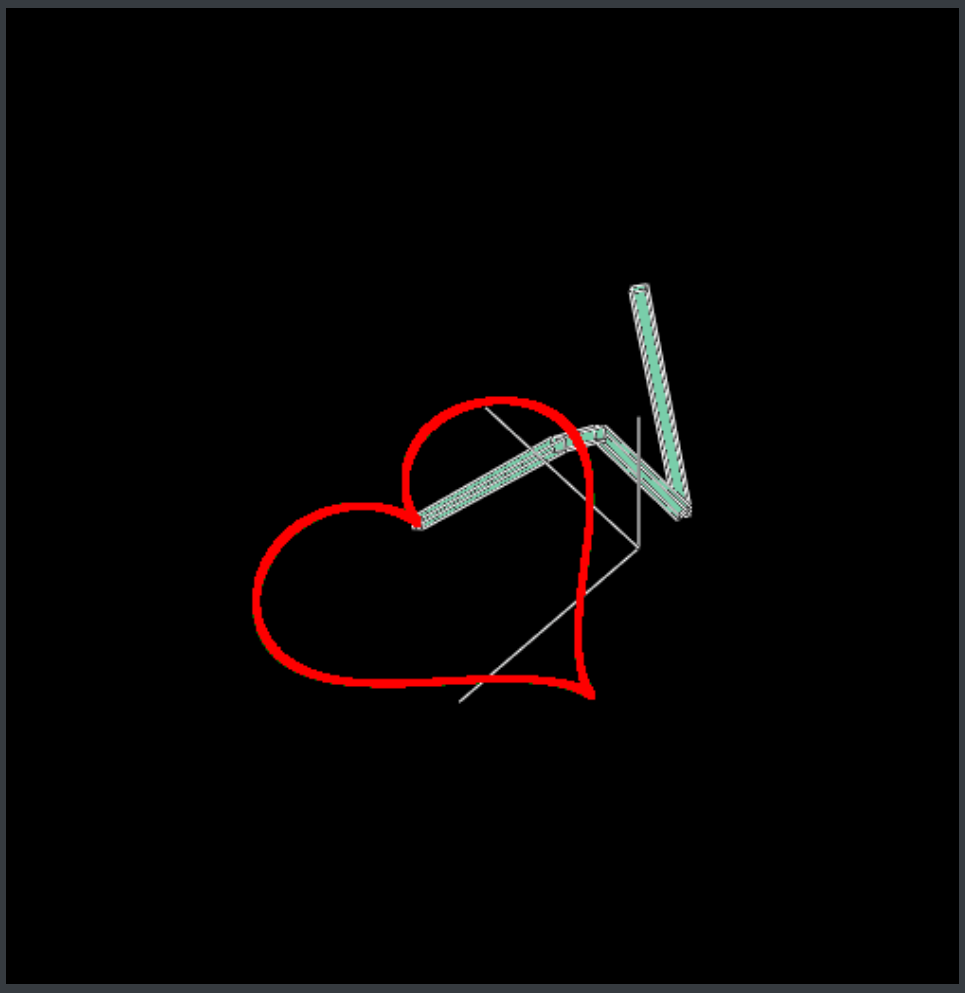
这里我选用了一个经典的心形作为图形。它的参数方程是

$$\begin{cases} x(t) = 16\sin^3(t) \\ y(t) = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t) \end{cases} \tag{1}$$

其中$t \in [0, 2\pi]$。经过一番调整（我们需要调整到让这个图案的中心和大小都合适such that机械臂能够够到所有的点），经过调整，一个可行的参数是

```
float scale=0.05f;
x_val*=scale;
y_val*=scale;
(*custom)[index++]=glm::vec3(0.6f-x_val,0.0f,y_val+0.5f);
```

这样得到的效果图如下：



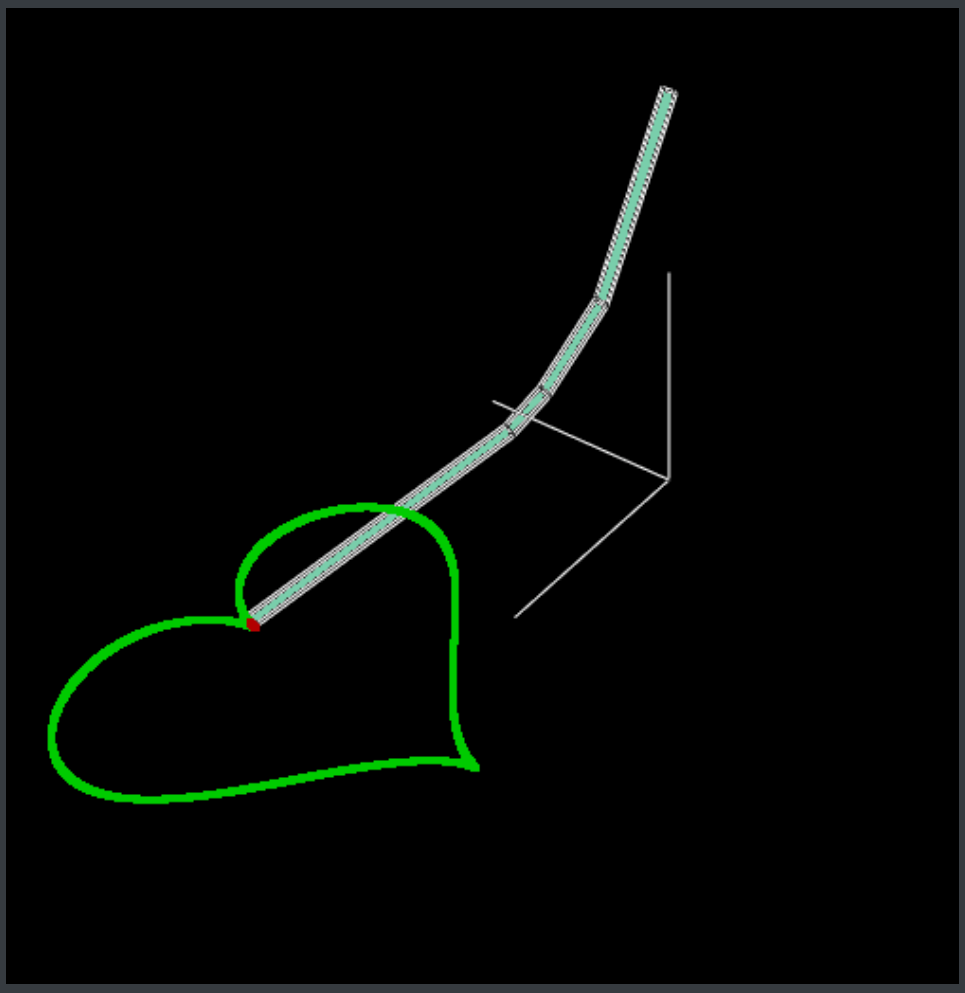
Sub-Task 4.1

可以降低采样参数的步长，这样就可以保证有更多的采样点，相当于做插值了。

Questions

> 1. 如果目标位置太远，无法到达，IK结果会怎样?

会伸直所有的关节去够到目标点，最后画出来以总长为半径的圆弧。比如这种情况：



在特定情况下，可能会有一些抖动

> 2. 比较 CCD IK 和 FABR IK 所需要的迭代次数。

在CCD函数中我添加了一行

```
printf("maxiter=%d,iter=%d,dist=%f\n", maxCCDIKIteration, CCDIKIteration,
glm::l2Norm(ik.EndEffectorPosition() - EndPosition));
```

用于观察迭代的情况。FABR IK同理。

- 对于CCD IK，观察到 `iter` 常常达到了 `maxiter` 还未收敛，输出类似：

  ```
  maxiter=100,iter=100,dist=0.002009
  ```

  查阅 `CaseInverseKinematics.cpp` 可以得知，设置的收敛误差 `eps=1e-4f` ，说明还远远没有达到收敛。
- 对于FABR IK，观察到几乎不需要多少次就能收敛，且误差能够达到接近零的程度。输出类似：

```
maxiter=100,iter=1,dist=0.000002
```

可以看到FABR IK的收敛性远比CCD IK好。

> 3. （选做，只提供大概想法即可）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?

可以设置一个约束，计算相邻两帧之间所有顶点的位置之差的平方和，然后把这个函数加入待优化的函数之中

Task2 Mass-Spring System

要实现隐式欧拉，就是要求解$\boldsymbol{x_{n+1}} = \boldsymbol{y} + h^2 M^{-1} \boldsymbol{f}_{int}(x_{n+1})$，其中$\boldsymbol{y} = \boldsymbol{x_n} + h(\boldsymbol{v_n} + hM^{-1}\boldsymbol{f_{ext}})$。要实现牛顿迭代法求解这个方程，最核心的步骤就是算出我们的待优化函数$g(\boldsymbol{x}) = \frac{1}{2h^2}|\boldsymbol{x} - \boldsymbol{y}|_M^2 + E(\boldsymbol{x})$对$\boldsymbol{x}$的梯度$\nabla g$以及Hessian Matrix $H(g)$。

由于隐式欧拉的稳定性比较强，所以我们并不需要把时间步压得很细，甚至只需要几步就可以了。这里设置 `int const steps = 6;` *主要是为了把fps控制在60附近*。

我们把整个函数分成几个部分：

1. 计算$\boldsymbol{y} = \boldsymbol{x_n} + h(\boldsymbol{v_n} + hM^{-1}\boldsymbol{f_{ext}})$，随后进入牛顿迭代
2. 计算$g(\boldsymbol{x})$的梯度∇g
3. 计算$g(\boldsymbol{x})$的Hessian Matrix $H(g)$
4. 执行牛顿迭代$\boldsymbol{x_{n+1}} = \boldsymbol{x_k} - H(g)^{-1}\nabla g$

分步骤来看我是如何实现的：

计算y

在循环中，我们这样来计算：

```
int n = static_cast<int>(system.Positions.size());
Eigen::VectorXf x0=glm2eigen(system.Positions);
Eigen::VectorXf v0=glm2eigen(system.Velocities);
Eigen::VectorXf gravity(n * 3);
gravity.setZero();
for (int i = 0; i < n; i++)
        if (!system.Fixed[i])
                gravity[3*i+1]=-system.Gravity;
Eigen::VectorXf y=x0+h*v0+h*h*gravity;
```

其中 `x0,v0` 分别是每一时间步开始的时候系统的状态。同时我们还定义了重力加速度向量以便计算。

在我们原始的定义中，$\boldsymbol{y}=\boldsymbol{x_n}+h(\boldsymbol{v_n}+hM^{-1}\boldsymbol{f_{ext}})$，其中$M$为质量矩阵。但是由于$\boldsymbol{f_{ext}}M^{-1}$正好就是每个质点的加速度，因此这一项直接就化为了我们的 `gravity` ，于是直接写 `Eigen::VectorXf y=x0+h*v0+h*h*gravity;` 即可。

计算∇g

$\boldsymbol{y}$是一项和$\boldsymbol{x}_{n+1}$无关的项，因此

$$\nabla g=\boldsymbol{x}-\boldsymbol{y}-h^2M^{-1}\boldsymbol{f}_{int}(\boldsymbol{x}) \tag{2}$$

其中$\boldsymbol{f}_{int}(\boldsymbol{x})$是系统的内力，也是弹簧总势能$E(\boldsymbol{x})$的负梯度。为此，我们需要计算系统内力组成的向量 `f_int` 。需要注意的是，这里系统的内力**不只有弹簧的弹力一项，还有一项粘滞力**（通过阅读显式欧拉的代码可以得知，这一项是存在的）

弹簧的弹力部分实际上可以直接参考给出的显式欧拉的代码，这里略去不讲。粘滞力的大小$f_d=-k_d(v\cdot e)e$，其中e为弹簧两端端点构成的向量标准化后得到的单位向量。因此，对每根弹簧 `for(auto const & spring:system.Springs)` ，我们做：

```
int p0=spring.AdjIdx.first;
int p1=spring.AdjIdx.second;
Eigen::Vector3f p1_pos=x_next.segment<3>(3*p1);
Eigen::Vector3f p0_pos=x_next.segment<3>(3*p0);
Eigen::Vector3f x01=p1_pos-p0_pos;
float length=x01.norm();
Eigen::Vector3f e01=x01.normalized();
Eigen::Vector3f p1_pos_old=x0.segment<3>(3*p1);
```

```
Eigen::Vector3f p0_pos_old=x0.segment<3>(3*p0);
Eigen::Vector3f v01=((p1_pos-p1_pos_old)-(p0_pos-p0_pos_old))/h;
float damping_force=system.Damping*e01.dot(v01);
Eigen::Vector3f f=(system.Stiffness*(length-
spring.RestLength)+damping_force)*e01;
if (!system.Fixed[p0])f_int.segment<3>(3*p0)+=f;
if (!system.Fixed[p1])f_int.segment<3>(3*p1)-=f;
```

在实现中，对于计算相对速度，我采用的方法是先计算端点在当前迭代与时间步开始的时候的位置差，也就是粒子经过一段时间的位移；再对两个端点的位移作差并除以时间步长，这样就得到了相对速度。再代入公式即可。注意，**这里得到的f_{int}是弹簧能量的负梯度**，符号的正负会直接影响整个系统能不能跑起来。

构建完内力 `f_int` 后，其他的项就比较简单了，直接代入公式即可，也就是在循环完所有的弹簧之后，计算

```
Eigen::VectorXf nabla_g=x_next-y-h*h*f_int/system.Mass;//∇g(x)
```

就可以了。

计算$H(g)$

讲义中给出了弹性势能$E(\boldsymbol{x})$的计算方式：

对弹簧(i,j)，由于我们之前已经定义了 `e01`，也就是$e=\dfrac{\boldsymbol{x}_i-\boldsymbol{x}_j}{||\boldsymbol{x}_i-\boldsymbol{x}_j||}$，因此其弹性势能对两端点$\boldsymbol{x}_1,\boldsymbol{x}_2$的Hessian Matrix可以写为

$$\boldsymbol{H}_e=\frac{\partial^2 E_{ij}(\boldsymbol{x}^k)}{\partial \boldsymbol{x}_i^2} \tag{3}$$

$$=k_{ij}\frac{(\boldsymbol{x}_i-\boldsymbol{x}_j)(\boldsymbol{x}_i-\boldsymbol{x}_j)^T}{||\boldsymbol{x}_i-\boldsymbol{x}_j||^2}+k_{ij}(1-\frac{l_{ij}}{||\boldsymbol{x}_i-\boldsymbol{x}_j||^2})(\boldsymbol{I}-\frac{(\boldsymbol{x}_i-\boldsymbol{x}_j)(\boldsymbol{x}_i-\boldsymbol{x}_j)^T}{||\boldsymbol{x}_i-\boldsymbol{x}_j||^2}) \tag{4}$$

$$=k_{ij}\left(ee^T+(1-\frac{l_{ij}}{||\boldsymbol{x}_i-\boldsymbol{x}_j||^2})(\boldsymbol{I}-ee^T)\right) \tag{5}$$

同时，我们有

$$\begin{aligned}\frac{\partial^2 E_{ij}(\boldsymbol{x}^k)}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j} &= -\boldsymbol{H}_e \\ \frac{\partial^2 E_{ij}(\boldsymbol{x}^k)}{\partial \boldsymbol{x}_j^2} &= \boldsymbol{H}_e\end{aligned} \tag{6}$$

（以上讨论均在两点不被固定的前提下，若有一点被固定，那么它自己以及交叉项的Hessian Matrix全部为0）

然后我们来看粘滞力。粘滞力$f_d = -k_d(\boldsymbol{v}\cdot\boldsymbol{e})\boldsymbol{e}$，而我们的$\boldsymbol{v} = \dfrac{\boldsymbol{x}-\boldsymbol{x}_{old}}{\boldsymbol{h}}$，于是$\dfrac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}} = \dfrac{1}{h}\boldsymbol{I}$。

我们假定弹簧的方向向量$\boldsymbol{e}$在一个时间步内不变，于是

$$\frac{\partial f_d}{\partial \boldsymbol{x}} = -k_d \boldsymbol{e}\left(\frac{\partial (\boldsymbol{v}^T \boldsymbol{e})}{\partial \boldsymbol{x}}\right) \tag{7}$$

$$= -k_d \boldsymbol{e}\left(\boldsymbol{e}^T \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{x}}\right) \tag{8}$$

$$= -k_d \boldsymbol{e}\left(\boldsymbol{e}^T \frac{1}{h}\boldsymbol{I}\right) \tag{9}$$

$$= -\frac{k_d}{h}\boldsymbol{e}\boldsymbol{e}^T \tag{10}$$

由于这是粘滞力对$\boldsymbol{x}$的导数，而我们要计算的是势能的二阶导数矩阵，也就是内力的负梯度，因此这一项反号后加入我们的Hessian Matrix的计算即可。

这段公式翻译成代码也就是

```
Eigen::Matrix3f I=Eigen::Matrix3f::Identity(),
                eeT=e01*e01.transpose();
assert(length>1e-6);
Eigen::Matrix3f H_e=system.Stiffness*((1 - spring.RestLength/length)*(I - eeT)
+ eeT);
H_e+=(system.Damping/h)*eeT;//Damping term
if(!system.Fixed[p0])
    for(int i=0;i<3;++i)
        for(int j=0;j<3;++j)
            triplets.emplace_back(3*p0+i,3*p0+j,H_e(i,j));
if(!system.Fixed[p1])
    for(int i=0;i<3;++i)
        for(int j=0;j<3;++j)
            triplets.emplace_back(3*p1+i,3*p1+j,H_e(i,j));
if (!system.Fixed[p0]&&!system.Fixed[p1]) {
```

```
        for(int i=0;i<3;++i)
        for(int j=0;j<3;++j) {
            triplets.emplace_back(3*p0+i,3*p1+j,-H_e(i,j));
            triplets.emplace_back(3*p1+i,3*p0+j,-H_e(i,j));
            }
    }
```

循环完所有弹簧，得到包含 `H_e` 的 `triplets` 之后，我们构建整体弹簧的Hessian Matrix：

```
    auto H_E=CreateEigenSparseMatrix(n * 3, triplets);
```

然后对于整体函数的Hessian Matrix，由于$\nabla g = \boldsymbol{x} - \boldsymbol{y} + h^2 M^{-1} \nabla E(\boldsymbol{x})$，所以$H(g) = \boldsymbol{I} + h^2 M^{-1} \nabla^2 E(\boldsymbol{x})$，其中$\nabla^2(\boldsymbol{x})$就是我们刚刚构建的 `H_E` 。这样就得到了我们的 `nabla2_g` 。

执行牛顿迭代

$-H(g)^{-1}\nabla(g)$这一项可以直接用提供的函数 `Eigen::VectorXf dx=ComputeSimplicialLLT(nabla2_g,-nabla_g);`

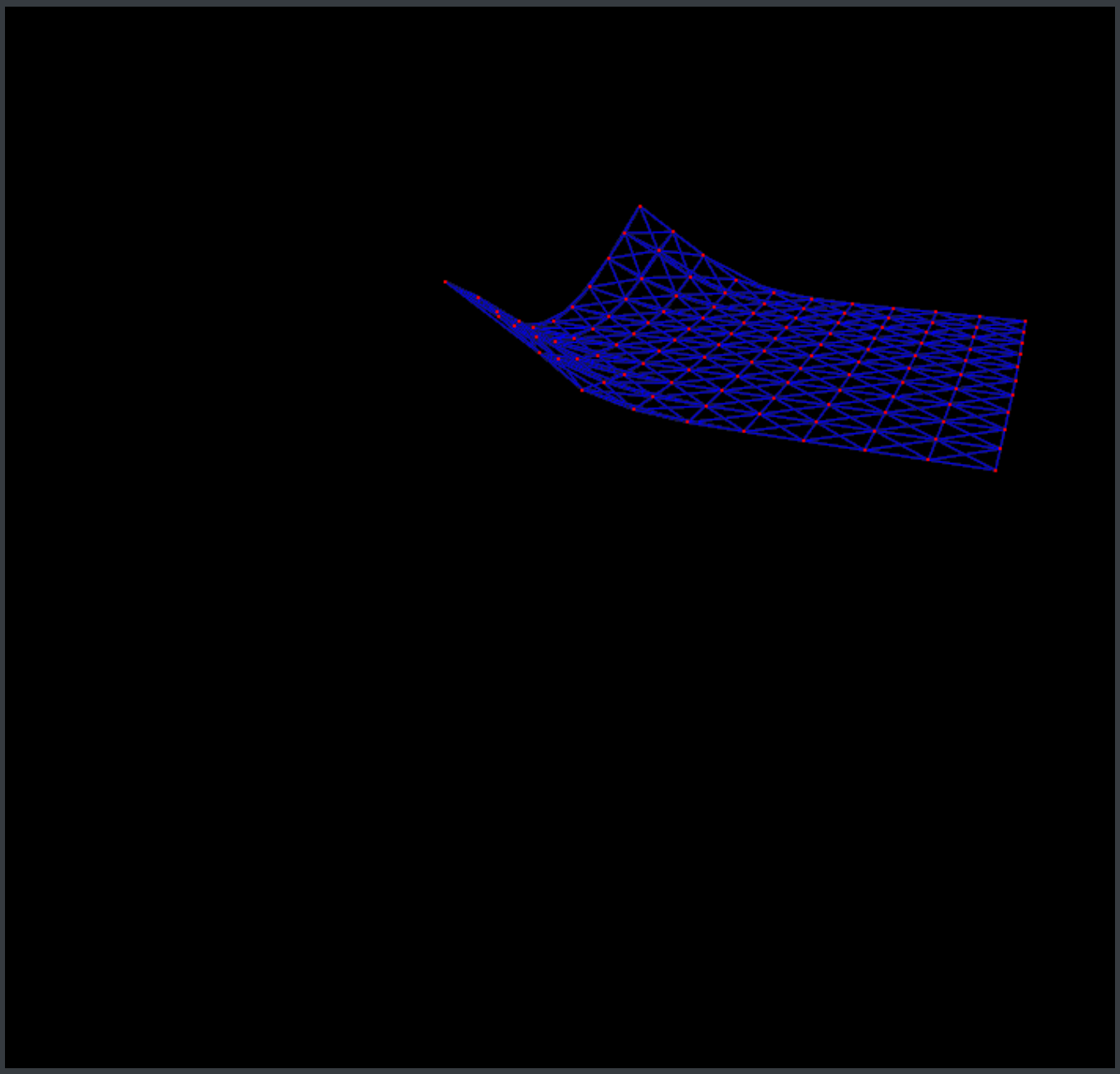
进行求解。计算得到了这一项之后，**我们需要把固定点的部分置零**。即

```
    Eigen::VectorXf dx=ComputeSimplicialLLT(nabla2_g,-nabla_g);
    for(int i=0;i<n;i++)
        if(system.Fixed[i])
            dx.segment<3>(3*i).setZero();
```

然后只要 `x_next+=dx;` 即可，最后更新系统变量：

```
    Eigen::VectorXf v_next=(x_next - x0)/h;
    system.Positions=eigen2glm(x_next);
    system.Velocities=eigen2glm(v_next);
```

这样就完成了我们整个牛顿迭代法求解隐式欧拉的过程。效果图如下：



其中参数为默认参数，以60fps进行录制。