Introduction to Computer Animation HW1 Report

資工系 0716234 蕭彧

Introduction/Motivation

In this homework, we need to use particle system to simulate a simple cloth, and simulate the collision of balls and cloth. The cloth in the homework are constructed by lots of particles, which have internal force like damper force and spring force. What we need to do is to construct those springs, calculate the correct forces, handle the collision, and complete four types of integrators.

Fundamentals

Internal Force

The internal force in springs can sperate to spring force and damper force. The main functionality of spring force is to store the vibration energy when the vibration happens, and the main functionality of damper force is to decrease the vibration energy. So, in implementation, we need to calculate all the internal forces of each springs and add the force they suffer now to be the final force.

Collision

- **Ball / Ball**

  The collision between balls can see as the collision of two particles. We calculate the relative velocity and position to determine whether they collide, and further calculate the tangent velocity and normal velocity of each ball.

- **Ball / cloth**

  The collision between ball and cloth can see as a particle collide with several particles. Just like the collision of ball and ball, we calculate the relative velocity and position to determine the collision, and further calculate the tangent velocity and normal velocity of ball and cloth.

- Integration
  - **Explicit Euler Method**

    Euler Method is an easy method in solving differential equations, which use delta time and derivative to approximate.
  - **Implicit Euler Method**

    Implicit Euler is similar to Explicit Euler. The main difference between them is Explicit Euler use current derivative whether Implicit Euler uses future derivative.
  - **Midpoint Euler Method**

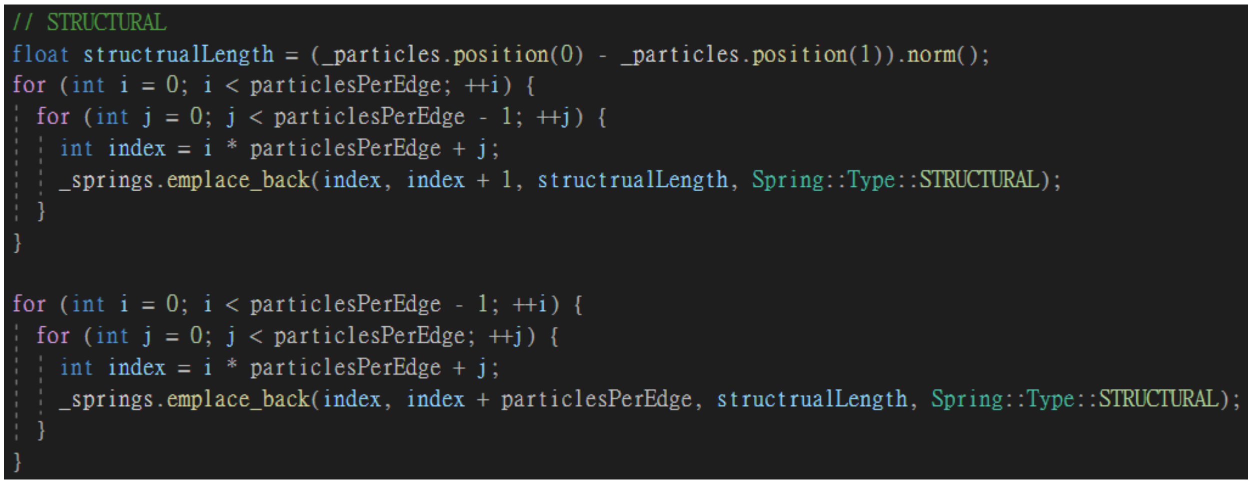
    Midpoint Euler method first use Explicit Euler to walk half step, calculate the derivative there, and use that derivative to do Explicit Euler again.
  - **Runge Kutta 4th Method**

    Similar to midpoint method, this method uses different derivatives to do Explicit Euler, and calculate the derivative there. Finally, add these all derivatives with weights and use explicit method again to calculate the final answer

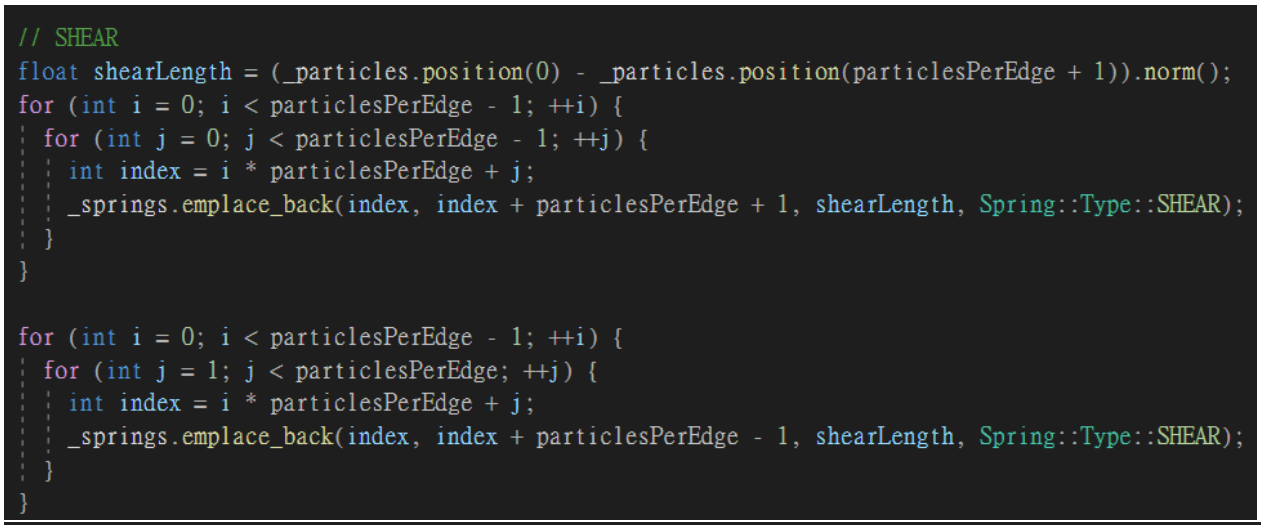
Implementation

- Initialize Spring

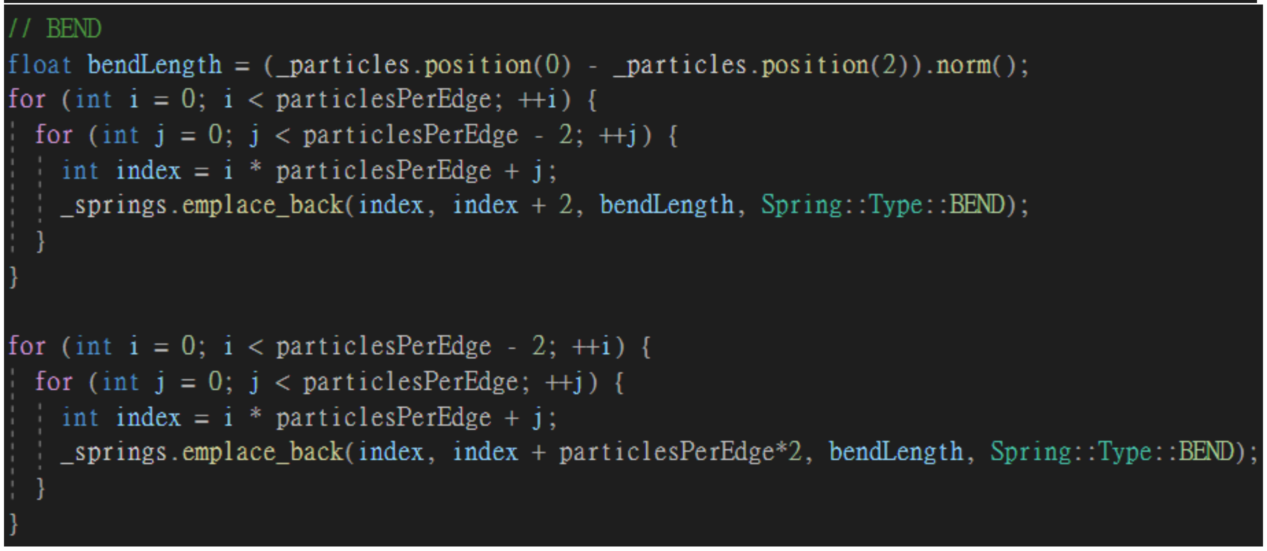
  The main mission in this function is to find the pattern of each spring. I let each particle be responsible for half of the directions. By this kind of distribution, the pattern can be easily found and can be coded tidy.



```
// STRUCTURAL
float structrualLength = (_particles.position(0) - _particles.position(1)).norm();
for (int i = 0; i < particlesPerEdge; ++i) {
  for (int j = 0; j < particlesPerEdge - 1; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + 1, structrualLength, Spring::Type::STRUCTURAL);
  }
}

for (int i = 0; i < particlesPerEdge - 1; ++i) {
  for (int j = 0; j < particlesPerEdge; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + particlesPerEdge, structrualLength, Spring::Type::STRUCTURAL);
  }
}
```



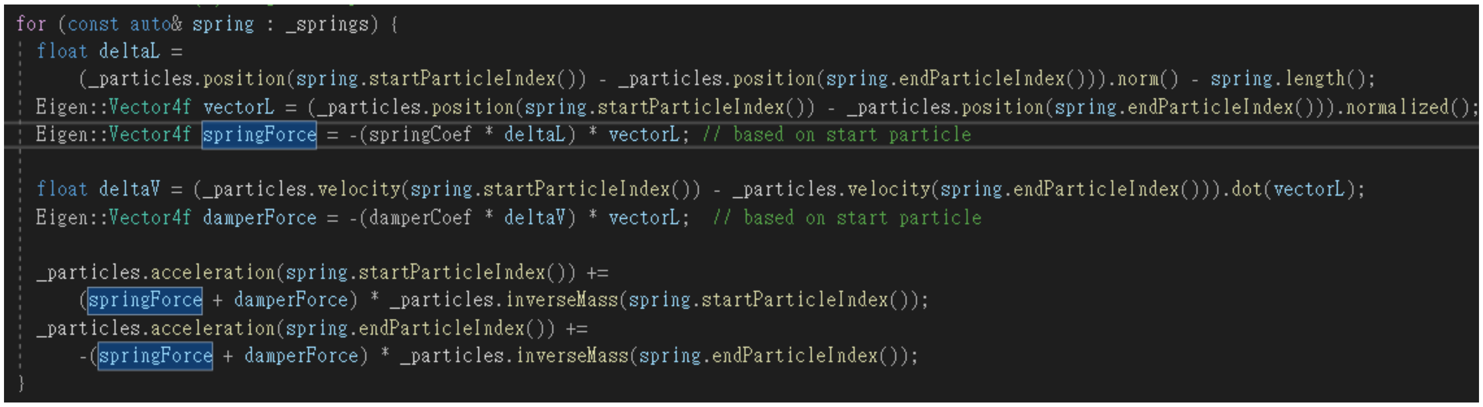
```
// SHEAR
float shearLength = (_particles.position(0) - _particles.position(particlesPerEdge + 1)).norm();
for (int i = 0; i < particlesPerEdge - 1; ++i) {
  for (int j = 0; j < particlesPerEdge - 1; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + particlesPerEdge + 1, shearLength, Spring::Type::SHEAR);
  }
}

for (int i = 0; i < particlesPerEdge - 1; ++i) {
  for (int j = 1; j < particlesPerEdge; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + particlesPerEdge - 1, shearLength, Spring::Type::SHEAR);
  }
}
```



```
// BEND
float bendLength = (_particles.position(0) - _particles.position(2)).norm();
for (int i = 0; i < particlesPerEdge; ++i) {
  for (int j = 0; j < particlesPerEdge - 2; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + 2, bendLength, Spring::Type::BEND);
  }
}

for (int i = 0; i < particlesPerEdge - 2; ++i) {
  for (int j = 0; j < particlesPerEdge; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + particlesPerEdge*2, bendLength, Spring::Type::BEND);
  }
}
```

- Internal Force

  I use the formula to calculate the corresponding spring force and damper force of each spring, use the well-known formula F=ma to calculate the acceleration of each particle, and add the acceleration to each particle.



```
for (const auto& spring : _springs) {
  float deltaL =
      (_particles.position(spring.startParticleIndex()) - _particles.position(spring.endParticleIndex())).norm() - spring.length();
  Eigen::Vector4f vectorL = (_particles.position(spring.startParticleIndex()) - _particles.position(spring.endParticleIndex())).normalized();
  Eigen::Vector4f springForce = -(springCoef * deltaL) * vectorL; // based on start particle

  float deltaV = (_particles.velocity(spring.startParticleIndex()) - _particles.velocity(spring.endParticleIndex())).dot(vectorL);
  Eigen::Vector4f damperForce = -(damperCoef * deltaV) * vectorL;  // based on start particle

  _particles.acceleration(spring.startParticleIndex()) +=
      (springForce + damperForce) * _particles.inverseMass(spring.startParticleIndex());
  _particles.acceleration(spring.endParticleIndex()) +=
      -(springForce + damperForce) * _particles.inverseMass(spring.endParticleIndex());
}
```

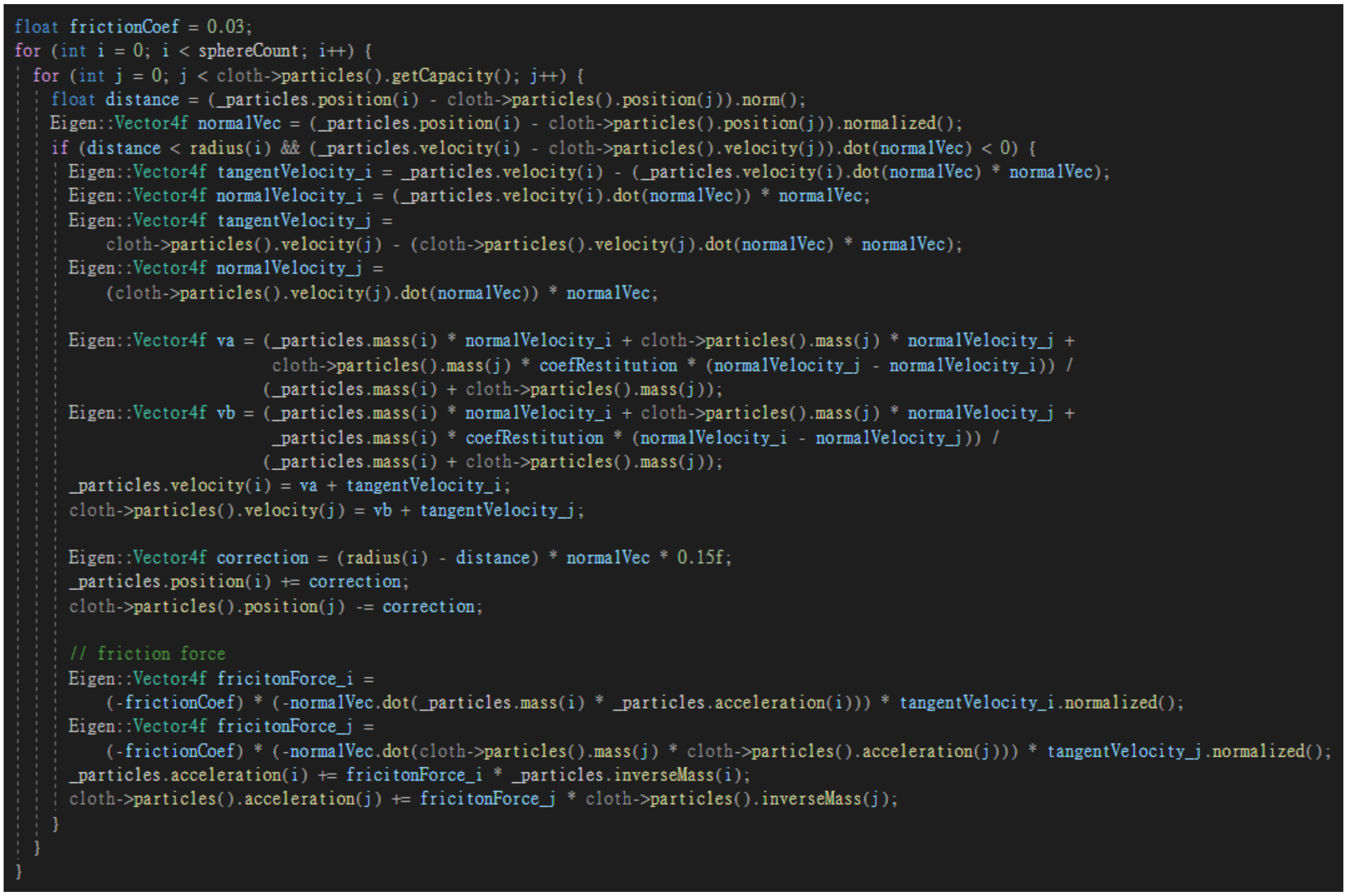
- Collision
  - Ball/Cloth & Ball/Ball

    For each particle, we judge the relationship between them in two terms:

    1. Whether the distance between the two is less than the sum of their radius
    2. Whether the two particles are close to each other (one particle is fixed in the two particles, so that the inner product of the relative velocity and the position vector of one particle is less than zero)

    If the above conditions are met, it means that a collision occurs, and the tangent velocity and the normal velocity are calculated.

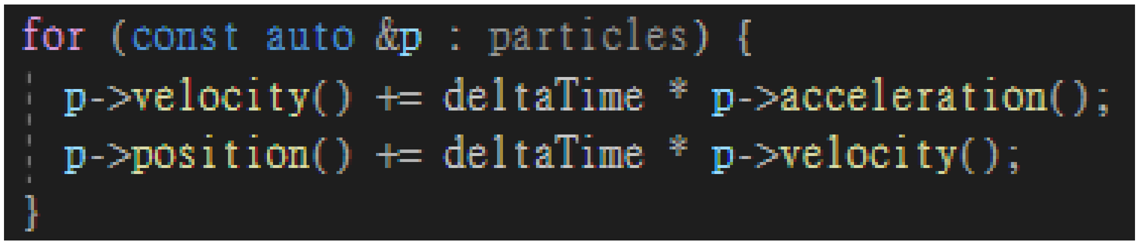
(the below picture is Ball/Cloth, since they are similar so I post one of them)



```
float frictionCoef = 0.03;
for (int i = 0; i < sphereCount; i++) {
  for (int j = 0; j < cloth->particles().getCapacity(); j++) {
    float distance = (_particles.position(i) - cloth->particles().position(j)).norm();
    Eigen::Vector4f normalVec = (_particles.position(i) - cloth->particles().position(j)).normalized();
    if (distance < radius(i) && (_particles.velocity(i) - cloth->particles().velocity(j)).dot(normalVec) < 0) {
      Eigen::Vector4f tangentVelocity_i = _particles.velocity(i) - (_particles.velocity(i).dot(normalVec) * normalVec);
      Eigen::Vector4f normalVelocity_i = (_particles.velocity(i).dot(normalVec)) * normalVec;
      Eigen::Vector4f tangentVelocity_j =
          cloth->particles().velocity(j) - (cloth->particles().velocity(j).dot(normalVec) * normalVec);
      Eigen::Vector4f normalVelocity_j =
          (cloth->particles().velocity(j).dot(normalVec)) * normalVec;

      Eigen::Vector4f va = (_particles.mass(i) * normalVelocity_i + cloth->particles().mass(j) * normalVelocity_j +
                            cloth->particles().mass(j) * coefRestitution * (normalVelocity_j - normalVelocity_i)) /
                           (_particles.mass(i) + cloth->particles().mass(j));
      Eigen::Vector4f vb = (_particles.mass(i) * normalVelocity_i + cloth->particles().mass(j) * normalVelocity_j +
                            _particles.mass(i) * coefRestitution * (normalVelocity_i - normalVelocity_j)) /
                           (_particles.mass(i) + cloth->particles().mass(j));
      _particles.velocity(i) = va + tangentVelocity_i;
      cloth->particles().velocity(j) = vb + tangentVelocity_j;

      Eigen::Vector4f correction = (radius(i) - distance) * normalVec * 0.15f;
      _particles.position(i) += correction;
      cloth->particles().position(j) -= correction;

      // friction force
      Eigen::Vector4f fricitonForce_i =
          (-frictionCoef) * (-normalVec.dot(_particles.mass(i) * _particles.acceleration(i))) * tangentVelocity_i.normalized();
      Eigen::Vector4f fricitonForce_j =
          (-frictionCoef) * (-normalVec.dot(cloth->particles().mass(j) * cloth->particles().acceleration(j))) * tangentVelocity_j.normalized();
      _particles.acceleration(i) += fricitonForce_i * _particles.inverseMass(i);
      cloth->particles().acceleration(j) += fricitonForce_j * cloth->particles().inverseMass(j);
    }
  }
}
```

- Integration
    - Explicit Euler Method

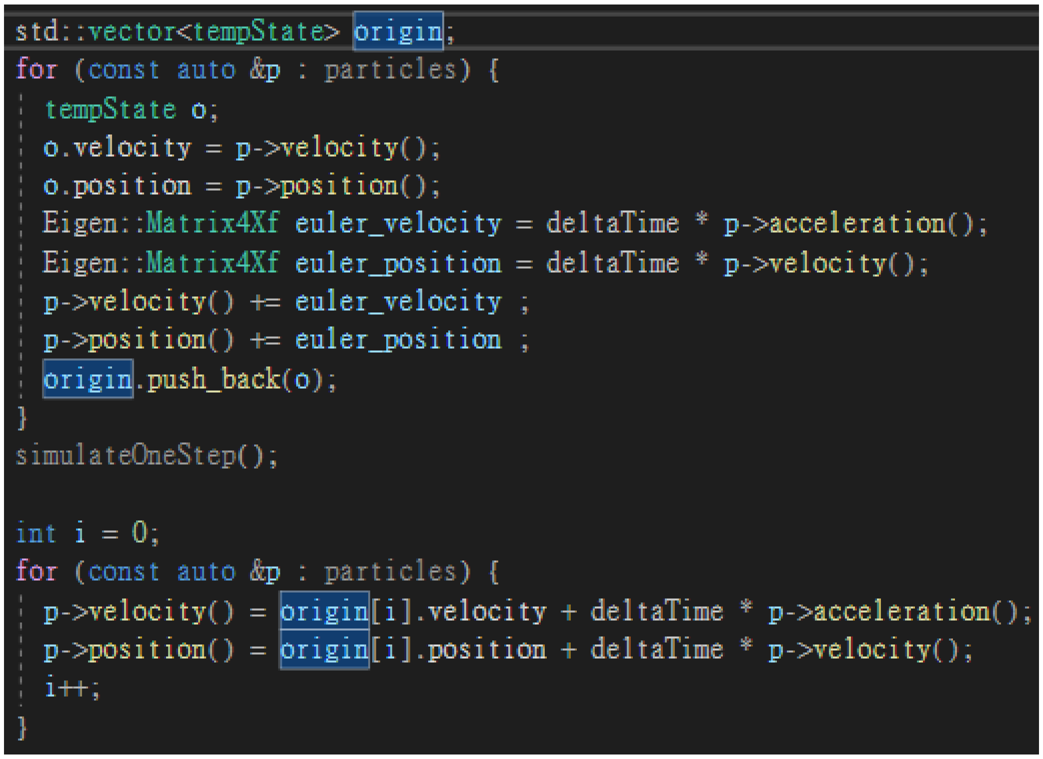
      The position and velocity of the next time point are calculated for each particle. Take the appropriate deltaT = 0.001, multiply by the amount of change, that is the differential value, plus the original value, it is the solution after deltaT time.



```
for (const auto &p : particles) {
  p->velocity() += deltaTime * p->acceleration();
  p->position() += deltaTime * p->velocity();
}
```

    - Implicit Euler Method

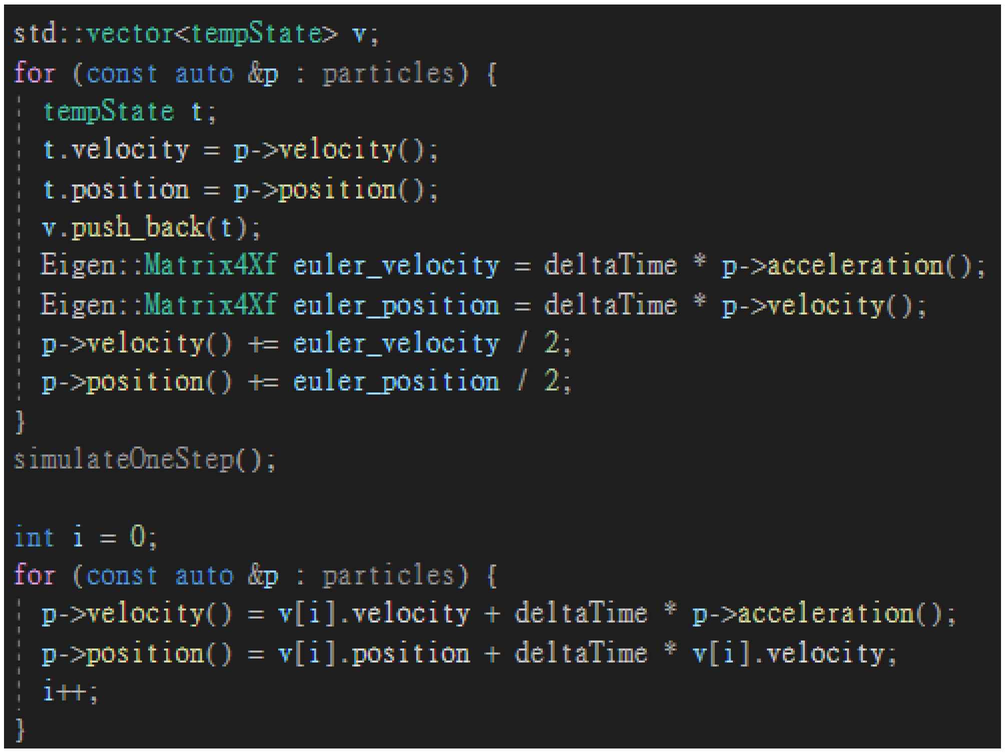
      I use a vector of self-defined struct to back up the original particles data. Since we need future information, I use simulateOneStep to get them. Then use Explicit Euler to calculate the final result by the future differential value.



```
std::vector<tempState> origin;
for (const auto &p : particles) {
  tempState o;
  o.velocity = p->velocity();
  o.position = p->position();
  Eigen::Matrix4Xf euler_velocity = deltaTime * p->acceleration();
  Eigen::Matrix4Xf euler_position = deltaTime * p->velocity();
  p->velocity() += euler_velocity ;
  p->position() += euler_position ;
  origin.push_back(o);
}
simulateOneStep();

int i = 0;
for (const auto &p : particles) {
  p->velocity() = origin[i].velocity + deltaTime * p->acceleration();
  p->position() = origin[i].position + deltaTime * p->velocity();
  i++;
}
```

    - Midpoint Euler Method

      The code of Midpoint Euler and Implicit Euler are nearly the same. The only difference is that in Midpoint Euler, we only need to add half of the Euler step when doing first Explicit Euler.

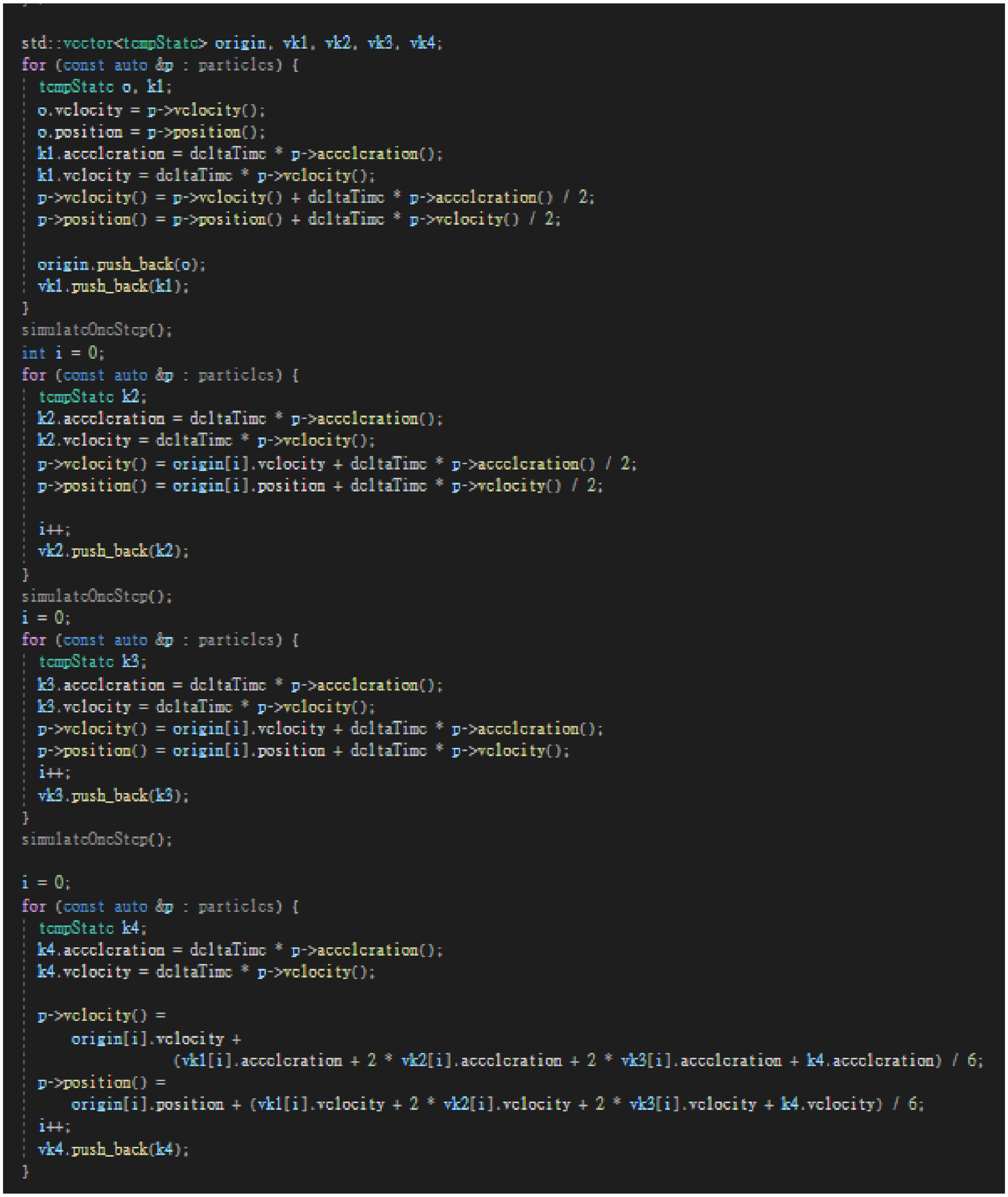


```
std::vector<tempState> v;
for (const auto &p : particles) {
  tempState t;
  t.velocity = p->velocity();
  t.position = p->position();
  v.push_back(t);
  Eigen::Matrix4Xf euler_velocity = deltaTime * p->acceleration();
  Eigen::Matrix4Xf euler_position = deltaTime * p->velocity();
  p->velocity() += euler_velocity / 2;
  p->position() += euler_position / 2;
}
simulateOneStep();

int i = 0;
for (const auto &p : particles) {
  p->velocity() = v[i].velocity + deltaTime * p->acceleration();
  p->position() = v[i].position + deltaTime * v[i].velocity;
  i++;
}
```

- Runge Kutta 4th

  This method is to make the desired value composed of more half-step combinations of different styles, including the slope of the start point and the slope of the end point. where k1 is the slope of the function at the beginning of time; k2 is the slope in the middle of time, and the Euler Method uses the slope k1 to determine the value of y at the point t(n) + h/2; k3 is also the slope at the midpoint, but uses the slope k2 Determines the y value; k4 is the slope at the end of the time period, and its y value is determined by k3.

  In implementation, I use a lots of vectors to store k1,k2,k3,k4,origin data. Then, add them all with weights at the end with Explicit Euler to get the final result.



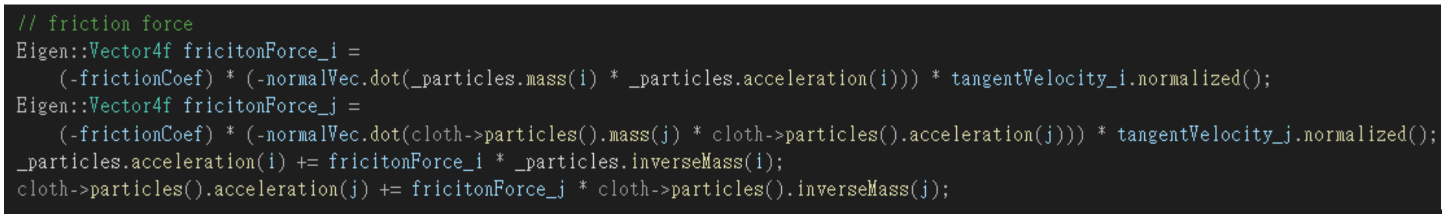
```cpp
std::vector<tempState> origin, vk1, vk2, vk3, vk4;
for (const auto &p : particles) {
  tempState o, k1;
  o.velocity = p->velocity();
  o.position = p->position();
  k1.acceleration = deltaTime * p->acceleration();
  k1.velocity = deltaTime * p->velocity();
  p->velocity() = p->velocity() + deltaTime * p->acceleration() / 2;
  p->position() = p->position() + deltaTime * p->velocity() / 2;

  origin.push_back(o);
  vk1.push_back(k1);
}
simulateOneStep();
int i = 0;
for (const auto &p : particles) {
  tempState k2;
  k2.acceleration = deltaTime * p->acceleration();
  k2.velocity = deltaTime * p->velocity();
  p->velocity() = origin[i].velocity + deltaTime * p->acceleration() / 2;
  p->position() = origin[i].position + deltaTime * p->velocity() / 2;

  i++;
  vk2.push_back(k2);
}
simulateOneStep();
i = 0;
for (const auto &p : particles) {
  tempState k3;
  k3.acceleration = deltaTime * p->acceleration();
  k3.velocity = deltaTime * p->velocity();
  p->velocity() = origin[i].velocity + deltaTime * p->acceleration();
  p->position() = origin[i].position + deltaTime * p->velocity();
  i++;
  vk3.push_back(k3);
}
simulateOneStep();

i = 0;
for (const auto &p : particles) {
  tempState k4;
  k4.acceleration = deltaTime * p->acceleration();
  k4.velocity = deltaTime * p->velocity();

  p->velocity() =
      origin[i].velocity +
                  (vk1[i].acceleration + 2 * vk2[i].acceleration + 2 * vk3[i].acceleration + k4.acceleration) / 6;
  p->position() =
      origin[i].position + (vk1[i].velocity + 2 * vk2[i].velocity + 2 * vk3[i].velocity + k4.velocity) / 6;
  i++;
  vk4.push_back(k4);
}
```

- ## Result and Discussion
  - ### The difference between intergrators

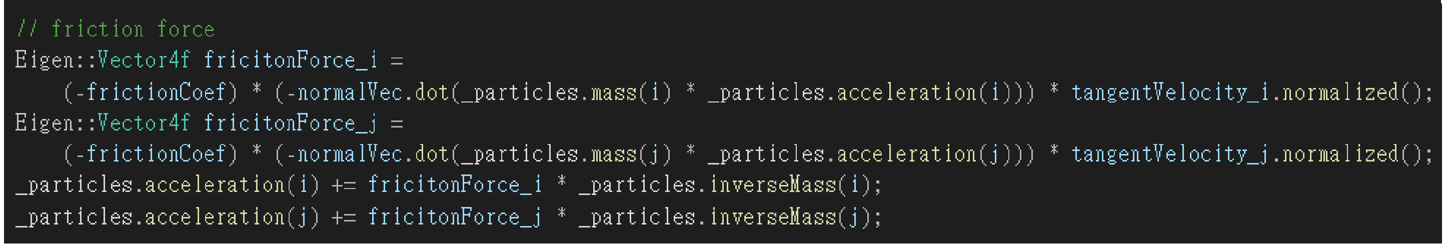
    Runge Kutta Method and Midpoint Euler can be seen as application of median value of Explicit Euler. Since Explicit Euler calculate one step by one step, Runge Kutta Method walk multiple steps in one round and calculate the final step by these steps. Normally, Explicit Euler will have larger error compared to Midpoint Euler, and Midpoint Euler will have larger error compared to Runge Kutta.

    Implicit Euler Method is like we use future status to determine the current move. In realistic circumstances, we need to solve the root of the differential equation.
  - ### Effect of parameters
    1. If the spring coefficient is lower and the damping coefficient is higher, the net will not be stable, which will lead to a crash after pressing start. If the spring coefficient is higher and the damping coefficient is lower, no problem occurs.
    2. If the deltaT is too large when calculating the integration, it is easy to generate errors; if the deltaT is too small, the overflow problem is easy to occur, or the calculation amount is too high.
    3. If the spring Restlength is too short, the spring will be too tight. After pressing start, the ball might bounce really high and leave the screen.
- ## Bonus

  First, I set the frictionCoef to 0.03. Since the formula of friction force is F =uN, where u is the friction coefficient and N is the normal force. I calculate the normal force of the particle, use the formula to calculate the friction force, use F=ma to get acceleration, and add the acceleration to the negative side of the tangent velocity.



```
// friction force
Eigen::Vector4f fricitonForce_i =
    (-frictionCoef) * (-normalVec.dot(_particles.mass(i) * _particles.acceleration(i))) * tangentVelocity_i.normalized();
Eigen::Vector4f fricitonForce_j =
    (-frictionCoef) * (-normalVec.dot(cloth->particles().mass(j) * cloth->particles().acceleration(j))) * tangentVelocity_j.normalized();
_particles.acceleration(i) += fricitonForce_i * _particles.inverseMass(i);
cloth->particles().acceleration(j) += fricitonForce_j * cloth->particles().inverseMass(j);
```



```
// friction force
Eigen::Vector4f fricitonForce_i =
    (-frictionCoef) * (-normalVec.dot(_particles.mass(i) * _particles.acceleration(i))) * tangentVelocity_i.normalized();
Eigen::Vector4f fricitonForce_j =
    (-frictionCoef) * (-normalVec.dot(_particles.mass(j) * _particles.acceleration(j))) * tangentVelocity_j.normalized();
_particles.acceleration(i) += fricitonForce_i * _particles.inverseMass(i);
_particles.acceleration(j) += fricitonForce_j * _particles.inverseMass(j);
```

- ## Conclusion
  - After observing the FPS, it is found that when the Integration method is Euler, the FPS is higher, but when using other integrators, the FPS is lower. It can be seen that the calculation amounts of other integrators is indeed larger than that of Euler. However, the

accuracy cannot be clearly identified by the naked eye, so it cannot be proved that the error of other integrators is smaller than that of Euler.

- The value of the parameter should be selected carefully, deltaT should not be too large or too small; the resistance coefficient, spring coefficient and damping coefficient will all affect the stability of the system.
- The number of particles of cloth will significantly affect the calculation amount. If we add more particles per edge, the fps will go down extremely.