HW1 report

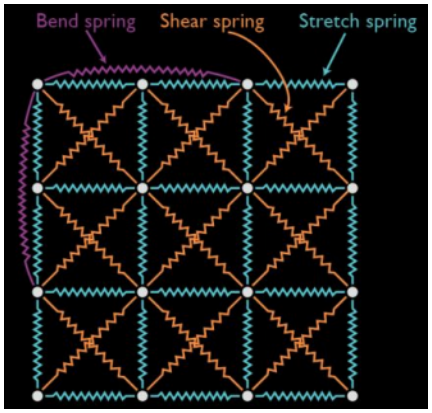
Introduction and motivation:

This homework assignment aims to simulate the movement of a cloth against a solid stationary sphere through various methods including: explicit euler, implicit euler, midpoint euler and Runge Kutta 4th order method. This is done by calculating the next position of each cloth particle per deltaTime interval, under the influence of forces such as: gravity, cloth inner spring forces (structural, shear, bending) and collision against the sphere.

Fundamentals:

This homework requires understanding of concepts of physics such as spring forces and particle collisions:

- First, initialize the different types of springs (structural, shear, bending) along with their corresponding spring arrangement



- Then compute the spring force and damper force for each spring and multiply it with a particle's inverse mass to calculate the particle's acceleration

$$\mathbf{f}_a = -k_s(|\mathbf{x}_a - \mathbf{x}_b| - l_0)\frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|}$$

$$-k_d\left(\frac{(\mathbf{v}_a - \mathbf{v}_b)\cdot(\mathbf{x}_a - \mathbf{x}_b)}{|\mathbf{x}_a - \mathbf{x}_b|}\right)\frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|}$$

- Next, detect whether cloth particles collide (penetrate) with the sphere. If collision occurs, update the cloth particle's velocity (rebounded) and rollback particle position

– Close to the wall

$\mathbf{N} \cdot (\mathbf{x} - \mathbf{p}) < \varepsilon$

– Heading in

$\mathbf{N} \cdot \mathbf{v} < 0$

$\mathbf{v}' = -k_r \, \mathbf{v}_N + \mathbf{v}_T$

coefficient of restitution

$$v_a = \frac{m_a u_a + m_b u_b + m_b C_R (u_b - u_a)}{m_a + m_b}$$

and

$$v_b = \frac{m_a u_a + m_b u_b + m_a C_R (u_a - u_b)}{m_a + m_b}$$

- Finally implement integrators by calculating the position and velocity of the next particle using the 4 methods mentioned above:
  - **Explicit Euler:**

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h \cdot f(\mathbf{x}, t)$$

  - **Implicit Euler:**

$$\mathbf{x}_{n+1} = \mathbf{x}_n + hf(\mathbf{x}_{n+1}, t_{n+1})$$

  - **Midpoint Euler:**

$$\Delta \mathbf{x} = h \cdot f(\mathbf{x}(t_0)) \qquad f_{mid} = f(\mathbf{x}(t_0) + \frac{\Delta \mathbf{x}}{2})$$

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h \cdot f_{mid}$$

  - **Runge Kutta 4th order:**

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5)$$

$$k_1 = hf(\mathbf{x}_0, t_0)$$

$$k_2 = hf(\mathbf{x}_0 + \frac{k_1}{2}, t_0 + \frac{h}{2})$$

$$k_3 = hf(\mathbf{x}_0 + \frac{k_2}{2}, t_0 + \frac{h}{2})$$

$$k_4 = hf(\mathbf{x}_0 + k_3, t_0 + h)$$

Implementation:

Cloth::initializeSpring

```
void Cloth::initializeSpring() {
  // TODO: Connect particles with springs.
  //   1. Compute spring length per type.
  //   2. Iterate the particles. Push spring objects into `_springs` vector
  // Note:
  //   1. The particles index:
  //   ===============================================
  //   0 1 2 3 ... particlesPerEdge - 1
  //   particlesPerEdge ... ...
  //   ... ... particlesPerEdge * particlesPerEdge - 1
  //   ===============================================
  // Here is a simple example which connects the horizontal structrual
springs.

  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);
      index = j * particlesPerEdge + i;
      _springs.emplace_back(index, index + particlesPerEdge,
structrualLength, Spring::Type::STRUCTURAL);
    }
  }

  // Shear spring
  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);
      _springs.emplace_back(index + 1, index + particlesPerEdge,
shearLength, Spring::Type::SHEAR);
    }
  }
```

```
  // Bending spring
  float bendingLength = (_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, bendingLength,
Spring::Type::BEND);
      index = j * particlesPerEdge + i;
      _springs.emplace_back(index, index + 2 * particlesPerEdge,
bendingLength, Spring::Type::BEND);
    }
  }
```

- Initialize the spring types and their corresponding length
- For each cloth particles, correctly connect springs to their corresponding arrangement

Cloth::computeSpringForce

```
void Cloth::computeSpringForce() {
  // TODO: Compute spring force and damper force for each spring.
  //   1. Read the start and end index from spring
  //   2. Use _particles.position(i) to get particle i's position.
  //   3. Modify particles' acceleration a = F / m;
  // Note:
  //   1. Use _particles.inverseMass(i) to get 1 / m can deal with m == 0.
Which will returns 0.

  int n = _springs.size();
  for (int i = 0; i < n; i++) {
    Spring* cur = &_springs[i];
    int startIndex = cur->startParticleIndex();
    int endIndex = cur->endParticleIndex();
    Eigen::Vector4f dist = _particles.position(startIndex) -
_particles.position(endIndex);
    float posDiff = (dist).norm();

    // spring force
    Eigen::Vector4f force = -springCoef * (posDiff - cur->length()) * (dist
/ posDiff);

    // damper force
    Eigen::Vector4f v_dist = _particles.velocity(startIndex) -
_particles.velocity(endIndex);
    force -= damperCoef * (v_dist.dot(dist) / posDiff) * (dist / posDiff);

    // change acceleration
    _particles.acceleration(startIndex) += force *
_particles.inverseMass(startIndex);
    _particles.acceleration(endIndex) -= force *
_particles.inverseMass(endIndex);
  }
}
```

- For each spring, force on particles equal spring force - damper force
- Using the formula to compute spring force and damper force mentioned in [Fundamentals], multiply the resulting force with inverse mass of particle to obtain new particle acceleration

Spheres::collide

```
void Spheres::collide(Cloth* cloth) {
  constexpr float coefRestitution = 0.0f;
  // TODO: Collide with particle (Simple approach to handle softbody
collision)

  // Detect collision
  for (int i = 0; i < particlesPerEdge * particlesPerEdge; i++) {
    // for each Sphere
    for (int k = 0; k < sphereCount; k++) {
      int particleIndex = i;
      Eigen::Vector4f dist = cloth->particles().position(particleIndex) -
_particles.position(k);

      // if penetrate
      if (dist.norm() < _radius[k]) {
        float ma = cloth->particles().mass(particleIndex);
        float mb = _particles.mass(k);
        Eigen::Vector4f ua = cloth->particles().velocity(particleIndex);
        Eigen::Vector4f ub = _particles.velocity(k);

        // update cloth particle velocity
        cloth->particles().velocity(particleIndex) = (ma * ua + mb * ub +
mb * coefRestitution * (ub - ua)) / (ma + mb);

        // rollback position
        cloth->particles().position(particleIndex) = _particles.position(k)
+ (dist).normalized() * (_radius[k] + 0.01f);
      }
    }
  }
}
```

- First, detect collision by determining whether cloth particle position has penetrated the sphere's surface. Using the if statement: cloth particle's position - sphere center position < sphere's radius.
- Using the formula mentioned in [Fundamentals] to update the cloth particle's velocity
- Rollback particle position back to 0.01f above sphere surface (as it initially penetrated the sphere). By normalizing the "dist" vector (direction from sphere center to cloth particle) to get the unit vector, then multiply with radius of sphere plus a little epsilon distance 0.01f (to prevent surface of sphere from touching with cloth particle, ultimately revealing itself in the animation)
- Finally, add this vector on top of sphere center position to obtain new cloth particle position

ExplicitEuler::integrate

```
void ExplicitEuler::integrate(const std::vector<Particles *> &particles,
std::function<void(void)>) const {
  // TODO: Integrate velocity and acceleration

  particles[0]->position() += deltaTime * particles[0]->velocity();
  particles[0]->velocity() += deltaTime * particles[0]->acceleration();
}
```

- Integrate velocity by multiplying it with deltaTime
- Adding integrated velocity to original position to obtain new position
- Integrate acceleration by multiplying it with deltaTime
- Adding integrated acceleration to original velocity to obtain new velocity

ImplicitEuler::integrate

```
void ImplicitEuler::integrate(const std::vector<Particles *> &particles,
std::function<void(void)> simulateOneStep) const {
  // TODO: Integrate velocity and acceleration

  Eigen::Matrix4Xf xn = particles[0]->position();
  Eigen::Matrix4Xf vn = particles[0]->velocity();

  particles[0]->position() += deltaTime * particles[0]->velocity();
  particles[0]->velocity() += deltaTime * particles[0]->acceleration();

  simulateOneStep();

  particles[0]->position() = xn + deltaTime * particles[0]->velocity();
  particles[0]->velocity() = vn + deltaTime * particles[0]->acceleration();
}
```

- Back up original particle's position and velocity
- Integrate velocity and acceleration to update particle's position and velocity (Xn+1)
- Call simulateOneStep() function to deal with compute force, handle collisions, etc.
- To obtain new particle velocity and acceleration
- Finally, update particle position and velocity by adding the new integrated velocity and acceleration with the original particle position and velocity

MidpointEuler::integrate

```
void MidpointEuler::integrate(const std::vector<Particles *> &particles,
std::function<void(void)> simulateOneStep) const {
  // TODO: Integrate velocity and acceleration

  Eigen::Matrix4Xf xn = particles[0]->position();
  Eigen::Matrix4Xf vn = particles[0]->velocity();

  particles[0]->position() += deltaTime * particles[0]->velocity() / 2;
  particles[0]->velocity() += deltaTime * particles[0]->acceleration() / 2;

  simulateOneStep();

  particles[0]->position() = xn + deltaTime * particles[0]->velocity();
  particles[0]->velocity() = vn + deltaTime * particles[0]->acceleration();
}
```

- Back up original particle's position and velocity
- Integrate velocity and acceleration and divide by 2 to obtain the midpoint. Using that to update the particle's position and velocity (Xn+1)
- Call simulateOneStep() function to deal with compute force, handle collisions, etc.
- To obtain new particle velocity and acceleration
- Finally, update particle position and velocity by adding the new integrated velocity and acceleration with the original particle position and velocity

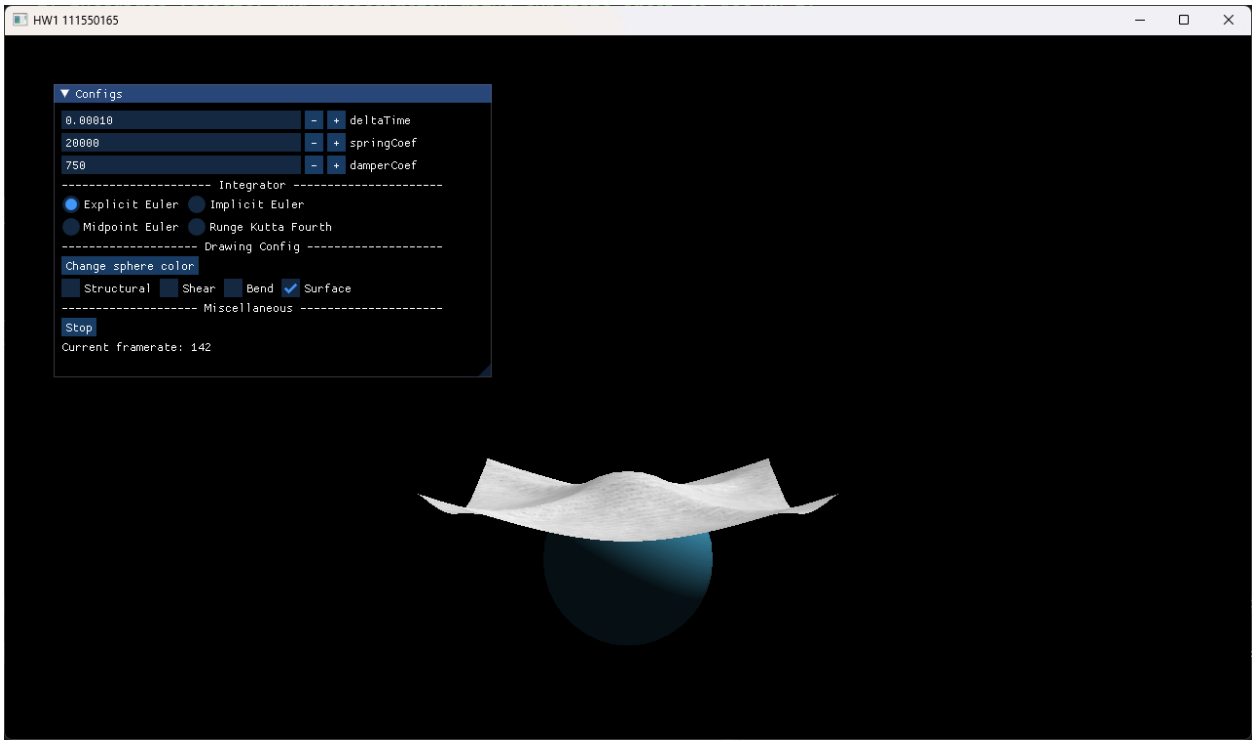
RungeKuttaFourth::integrate

```
void RungeKuttaFourth::integrate(const std::vector<Particles *> &particles,
                                 std::function<void(void)> simulateOneStep)
const {
  // TODO: Integrate velocity and acceleration

  Eigen::Matrix4Xf xn = particles[0]->position();
  Eigen::Matrix4Xf vn = particles[0]->velocity();

  Eigen::Matrix4Xf k1_pos = deltaTime * particles[0]->velocity();
  Eigen::Matrix4Xf k1_vel = deltaTime * particles[0]->acceleration();
  particles[0]->position() = xn + k1_pos / 2;
  particles[0]->velocity() = vn + k1_vel / 2;
  simulateOneStep();
```

```
  Eigen::Matrix4Xf k2_pos = deltaTime * particles[0]->velocity();
  Eigen::Matrix4Xf k2_vel = deltaTime * particles[0]->acceleration();
  particles[0]->position() = xn + k2_pos / 2;
  particles[0]->velocity() = vn + k2_vel / 2;
  simulateOneStep();

  Eigen::Matrix4Xf k3_pos = deltaTime * particles[0]->velocity();
  Eigen::Matrix4Xf k3_vel = deltaTime * particles[0]->acceleration();
  particles[0]->position() = xn + k3_pos;
  particles[0]->velocity() = vn + k3_vel;
  simulateOneStep();

  Eigen::Matrix4Xf k4_pos = deltaTime * particles[0]->velocity();
  Eigen::Matrix4Xf k4_vel = deltaTime * particles[0]->acceleration();

  particles[0]->position() = xn + (k1_pos + 2 * k2_pos + 2 * k3_pos +
k4_pos) / 6;
  particles[0]->velocity() = vn + (k1_vel + 2 * k2_vel + 2 * k3_vel +
k4_vel) / 6;
}
```

- Back up original particle's position and velocity
- Then calculate the k (4) orders' position and velocity using the formula mentioned in [Fundamentals], meanwhile using simulateOneStep() function to update the particle's velocity and acceleration.
- Finally, use weighted sum mentioned in [Fundamentals] to calculate next particle's position and velocity

Result and Discussion:



Integrators:

- Resulting animation of the 4 methods seemed more or less identical (though theoretically accuracy of Runge Kutta 4th order > Midpoint > implicit > explicit)
- However the there is a noticeable difference in **processing speed**:
  => **Explicit < Implicit < Midpoint < Runge Kutta 4th order**
- As shown in [Implementation], explicit euler method simply integrates velocity and acceleration to update new particle's position and velocity, doesn't need simulateOneStep() function to calculate Xn+1, hence it is the quickest method
- Implicit and midpoint method on the other hand, requires one simulateOneStep() function, which resulted in its slower processing speed
- Runge Kutta 4th order requires even more processing time as it uses 3 simulateOneStep() function

Effect of parameters (springCoef and damperCoef):

- Increasing the springCoef results in a **stiffer** spring, meaning that it requires more force to bend the spring and vice versa
- Increasing damperCoef results in the cloth becoming more **resilient**, meaning that the cloth dampens out forces better.

Conclusion:

In conclusion, this report detailed the implementation and analysis of various methods for simulating the movement of cloth interacting with a solid stationary sphere. Through the use of explicit Euler, implicit Euler, midpoint Euler, and Runge-Kutta 4th order integration methods, the behavior of the cloth was examined under the influence of gravitational forces, internal spring forces, and collisions with the sphere.

The implemented methods each demonstrated distinct performances in terms of computational efficiency and accuracy. Explicit Euler, being the simplest method, exhibited the fastest processing speed due to its direct integration of velocity and acceleration. On the other hand, implicit and midpoint Euler methods required an additional call of simulateOneStep function, resulting in slower processing speeds. Runge-Kutta 4th order, while providing the most accurate results, incurred the highest computational cost due to its multiple iterations and evaluations.

Furthermore, the report discussed the effect of parameters such as spring coefficient and damper coefficient on the behavior of the cloth. Adjusting these parameters allowed for control over the stiffness and resilience of the cloth, influencing its response to external forces.

Overall, this homework provides interesting insights into the dynamics of cloth simulation and the trade-offs between computational efficiency and accuracy in numerical methods.