COMS W4167: Computer Animation

Programming Assignment 2
Due 10:00 PM Thur., Feb. 26th, 2024 (EST)

Academic Honesty Policy

You are permitted and encouraged to discuss your work with other students. Except where explicitly stated otherwise, you may work out equations in writing on paper or a whiteboard. You are encouraged to use Ed Discussion to converse with other students, the TAs, and the instructor.

HOWEVER, you must NOT share source code or hard copies of source code. Refrain from activities or the sharing materials that could cause your source code to APPEAR TO BE similar to another student's enrolled in this or previous years. We will be monitoring source code for individuality. Cheating will be dealt with severely. Source code should be yours and yours only. Do not cheat. For more details, please refer to the full academic honesty policy on the Engineering School's website.

1 Introduction

In this assignment, you will implement an implicit Euler integrator. But you will start with a simplified version of the implicit integrator, called linearized Implicit Euler, before implementing a full version of Implicit Euler. The implemention will involve the computation of the *force Jacobian* for each of the forces you used in previous assignment, as well as the solution of a linear system.

2 Newton's Method

Before discussing implicit Euler, we will briefly review the topics of root finding, Newton's method for univariate problems, and Newton's method for multivariate problems.

2.1 Univariate Root Finding

Recall that the root of an equation $f(x)$ is a value of x for which $f(x) = 0$. As a simple example, consider the linear function $f(x) = 3x + 6$. $f(x)$ has a single root, $x = -2$. For an example with multiple roots, consider the polynomial $f(x) = x^2 - x - 6$. You might recall from introductory algebra that we can solve for the roots of this equation in a number of a ways: we could employ the quadratic formula, we could factor the polynomial, etc. Taking the latter approach, we find that $f(x) = x^2 - x - 6 = (x - 3)(x + 2)$, from which we immediately read off the roots $x = 3$ and $x = -2$. Graphically, these roots correspond to points where the parabola intersects the x-axis. While similar methods exist for cubic and quartic polynomials, they do not exist in general for higher degree polynomials. Thus, one is forced to develop specialized algorithms to locate these roots.

Polynomial root finding is an important but specialized subclass of the more general non-linear root finding problem. For example, we might want to find the root(s) of the non-linear equation $f(x) = e^x - x - 1$. To do so, we will employ Newton's method, which is really just an application of the venerable Taylor's Theorem. By Taylor's Theorem, we know that we can approximate a sufficiently smooth function as a series of polynomials. If we truncate this series, Taylor's Theorem tells us that we make an error that increases with the distance from the point about which we compute the Taylor series. Practically speaking, this means we can approximate a function locally by a polynomial provided we don't stray too far from the point of interest.

Newton's method exploits Taylor's Theorem by assuming we have some educated estimate of a root. If this estimate is sufficiently close to the true root, then we are not making a large error by approximating

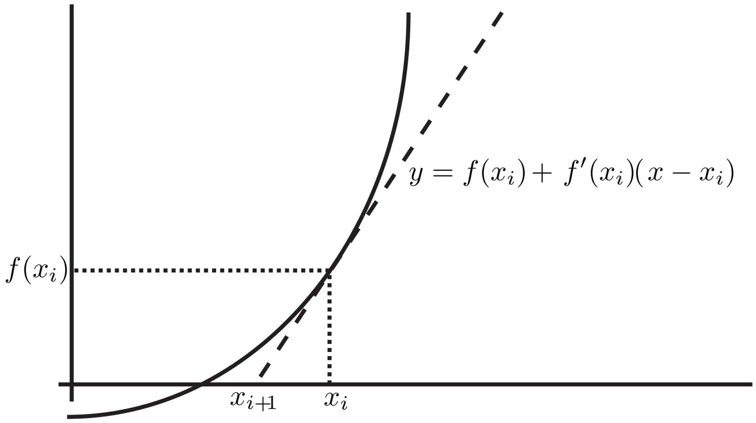

Figure 1: One Step of Newton's Method

the non-linear function as a line. Therefore, the root of this line, while not exactly equal to the root of the non-linear function, will be an improvement on our current guess. We can repeat this process with the improved estimate, and compute an even better approximation of the true root. Repeating this process, we have turned the problem of non-linear root finding into a sequence of linear root finding problems. Let's make this concrete with some math. Computing the Taylor expansion up to the linear term for some non-linear function $f(x)$ about our initial guess x_0, we find that

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(\xi)(x - x_0)^2$$

where $\xi \in (x_0, x)$. Assuming x_0 is close to the true root, we can safely neglect the higher order error term and equate with 0, giving

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) = 0.$$

Solving for the root, we find that

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Repeating this process until we are satisfied with our estimate, we obtain the iterative method

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

See Figure 1 for an illustration of one step of Newton's method.

2.2 Multivariate Root Finding

We would now like to find a root of the function $\mathbf{F}(\mathbf{x})$ where $\mathbf{F} : \mathbb{R}^N \to \mathbb{R}^N$ and $\mathbf{x} \in \mathbb{R}^N$. Let us follow the same prescription as in the univariate case; we will derive a linear approximation to $\mathbf{F}$ that does not give the exact solution, but that is 'easy' to solve and will hopefully give an improved estimate of the solution. Computing the Taylor series of $\mathbf{F}(\mathbf{x})$ about some estimate of the solution $\mathbf{x}_0$, we find that

$$\mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + H.O.T.$$

where $\nabla\mathbf{F}(\mathbf{x})$ is the gradient of the function $\mathbf{F}$. As $\mathbf{F}(\mathbf{x}) \in \mathbb{R}^N$, $\nabla\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{N\times N}$. The i, j entry of the gradient is given by $\frac{\partial \mathbf{F}_i}{\partial \mathbf{x}_j}$. If we neglect the higher order terms ($H.O.T.$) and equate the function with 0, we find that

$$\mathbf{F}(\mathbf{x}) \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) = 0.$$

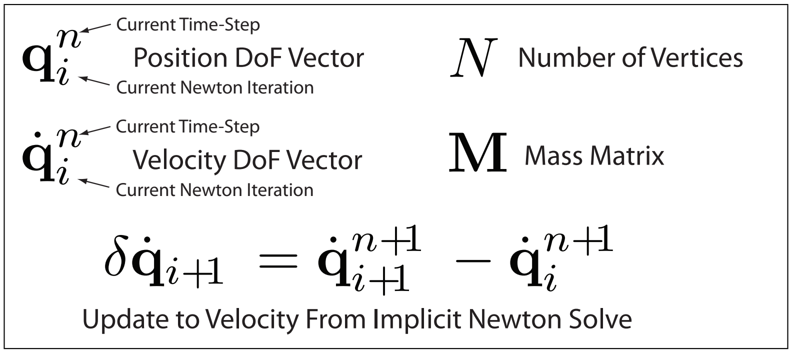

Figure 2: Some Useful Quantities

We can solve this system to obtain an improved estimate of the root. Turning this into an iterative process:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \nabla\mathbf{F}(\mathbf{x}_i)^{-1}\mathbf{F}(\mathbf{x}_i)$$

In both the univariate and the multivariate cases, the convergence of Newton's method depends on the quality of the initial estimate of the root. If the initial estimate is far from the root, there is no reason to expect a linear approximation to land us near a root, and Newton's method is unlikely to converge.

3 Implicit Euler

So far our discussion has been fairly abstract, and you might wonder how root finding relates to the topic of our course, physically based computer animation. Consider the following discretization of Newton's Second law:

$$\mathbf{q}^{n+1} = \mathbf{q}^n + h\dot{\mathbf{q}}^{n+1}$$
$$\dot{\mathbf{q}}^{n+1} = \dot{\mathbf{q}}^n + h\mathbf{M}^{-1}\mathbf{F}(\mathbf{q}^{n+1}, \dot{\mathbf{q}}^{n+1})$$

Let N denote the number of particles in the system. Here $\mathbf{M}$ is a $3N \times 3N$ mass matrix, in our case a diagonal matrix with masses on the diagonal (entries $[0,0]$, $[1,1]$ and $[2,2]$ are the mass of the first particle, entries $[3,3]$ $[4,4]$, and $[5,5]$ are the mass of the second particle, etc). Since the mass matrix is a diagonal matrix, we can store it as a vector. In our implementation, this vector is model.particle_mass= $[m_0, m_1, \ldots, m_N]$. The entry $[3i, 3i]$, $[3i+1, 3i+1]$, and $[3i+2, 3i+2]$ of the mass matrix is simply $\mathbf{m}[i]$.

Notice that the new position depends on the new velocity, and the new velocity depends on the new position—we are unable to solve one without the other! If this system were linear, we would only have to solve a linear system. $\mathbf{F}(\mathbf{q}^{n+1})$ could be non-linear, however. How do we solve a non-linear system of equations? By Newton's method!

Observe that we have $6N$ unknowns in our system, where N is the number of particles. $3N$ of these unknowns are positions, and $3N$ are the corresponding velocities. Concatenate these unknowns into one big vector, call it $\mathbf{y}^{n+1}$:

$$\mathbf{y}^{n+1} = \begin{pmatrix} \mathbf{q}^{n+1} \\ \dot{\mathbf{q}}^{n+1} \end{pmatrix}$$

Similarly, combine the above non-linear equations into a single vector of length $6N$:

$$G(\mathbf{y}^{n+1}) = G(\mathbf{q}^{n+1}, \dot{\mathbf{q}}^{n+1}) = \begin{bmatrix} P(\mathbf{q}^{n+1}, \dot{\mathbf{q}}^{n+1}) \\ Q(\mathbf{q}^{n+1}, \dot{\mathbf{q}}^{n+1}) \end{bmatrix} = \begin{bmatrix} \mathbf{q}^{n+1} - \mathbf{q}^n - h\dot{\mathbf{q}}^{n+1} \\ \dot{\mathbf{q}}^{n+1} - \dot{\mathbf{q}}^n - h\mathbf{M}^{-1}\mathbf{F}(\mathbf{q}^{n+1}, \dot{\mathbf{q}}^{n+1}) \end{bmatrix}$$

Computing the gradient for Newton's method, as discussed in class, we obtain a $6N \times 6N$ matrix:

$$\nabla G = \begin{bmatrix} \frac{\partial P}{\partial \mathbf{q}^{n+1}} & \frac{\partial P}{\partial \dot{\mathbf{q}}^{n+1}} \\ \frac{\partial Q}{\partial \mathbf{q}^{n+1}} & \frac{\partial Q}{\partial \dot{\mathbf{q}}^{n+1}} \end{bmatrix} = \begin{bmatrix} \mathbf{Id} & -h\mathbf{Id} \\ -h\mathbf{M}^{-1}\frac{\partial \mathbf{F}}{\partial \mathbf{q}^{n+1}} & \mathbf{Id} - h\mathbf{M}^{-1}\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}^{n+1}} \end{bmatrix}$$

A single step of Newton's method now involves solving the linear system:

$$\nabla G(\mathbf{y}_i^{n+1})(\mathbf{y}_{i+1}^{n+1} - \mathbf{y}_i^{n+1}) = -G(\mathbf{y}_i^{n+1}),$$

where we use subscript i to index the Newton iteraiton (see Fig. 2). This system is repeatedly solved until convergence is detected. Observe that we attach a subscript to each occurrence of $\mathbf{q}^{n+1}$ and $\dot{\mathbf{q}}^{n+1}$ to denote that we are computing a sequence of these values. In contrast, the solution from the last time step, $\mathbf{q}^n$ and $\dot{\mathbf{q}}^n$, is constant throughout this process and does not receive a subscript.

As for the initial guess, there are several different choices as discussed in class, but we'll use the simplest one - the state from the end of the last time step.

3.1 Reduction of System Size by Substitution

The above formulation is what we discussed in class. We could certainly proceed as described above, but we can reduce the size of our linear system with a bit of algebra (although this will modify the sparsity structure of the matrix). Let us expand a step of Newton's method. Multiplying out the blocks $\nabla G(\mathbf{y}_i^{n+1})(\mathbf{y}_{i+1}^{n+1} - \mathbf{y}_i^{n+1}) = -G(\mathbf{y}_i^{n+1})$, we obtain two linear equations of size $3N$:

$$(\mathbf{q}_{i+1}^{n+1} - \mathbf{q}_i^{n+1}) - h(\dot{\mathbf{q}}_{i+1}^{n+1} - \dot{\mathbf{q}}_i^{n+1}) = -(\mathbf{q}_i^{n+1} - \mathbf{q}^n - h\dot{\mathbf{q}}_i^{n+1})$$

$$\begin{aligned} -h\mathbf{M}^{-1}\frac{\partial \mathbf{F}(\mathbf{q}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})}{\partial \mathbf{q}}(\mathbf{q}_{i+1}^{n+1} - \mathbf{q}_i^{n+1}) + (\mathbf{Id} - h\mathbf{M}^{-1}\frac{\partial \mathbf{F}(\mathbf{q}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})}{\partial \dot{\mathbf{q}}})(\dot{\mathbf{q}}_{i+1}^{n+1} - \dot{\mathbf{q}}_i^{n+1}) \\ = -(\dot{\mathbf{q}}_i^{n+1} - \dot{\mathbf{q}}^n - h\mathbf{M}^{-1}\mathbf{F}(\mathbf{q}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})) \end{aligned}$$

The first equation simplifies to

$$\mathbf{q}_{i+1}^{n+1} = \mathbf{q}^n + h\dot{\mathbf{q}}_{i+1}^{n+1}$$

which we can use to eliminate $\mathbf{q}_{i+1}^{n+1}$ from the second equation. Carrying out this algebra, we find

$$\left[\mathbf{M} - \left(h^2\frac{\partial \mathbf{F}}{\partial \mathbf{q}} + h\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}\right)\right](\dot{\mathbf{q}}_{i+1}^{n+1} - \dot{\mathbf{q}}_i^{n+1}) = -\mathbf{M}(\dot{\mathbf{q}}_i^{n+1} - \dot{\mathbf{q}}^n) + h\mathbf{F}(\mathbf{q}^n + h\dot{\mathbf{q}}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})$$

or relabeling $\delta\dot{\mathbf{q}}_{i+1} = \dot{\mathbf{q}}_{i+1}^{n+1} - \dot{\mathbf{q}}_i^{n+1}$

$$\boxed{\left[\mathbf{M} - \left(h^2\frac{\partial \mathbf{F}}{\partial \mathbf{q}} + h\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}\right)\right]\delta\dot{\mathbf{q}}_{i+1} = -\mathbf{M}(\dot{\mathbf{q}}_i^{n+1} - \dot{\mathbf{q}}^n) + h\mathbf{F}(\mathbf{q}^n + h\dot{\mathbf{q}}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})} \tag{1}$$

where $\frac{\partial \mathbf{F}}{\partial \mathbf{q}}$ and $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$ are evaluated at $(\mathbf{q}^n + h\dot{\mathbf{q}}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})$. Note that this $3N \times 3N$ system is half the size of our original $6N \times 6N$ system. After solving this linear system for $\delta\dot{\mathbf{q}}_{i+1}$ and computing $\dot{\mathbf{q}}_{i+1}^{n+1} = \dot{\mathbf{q}}_i^{n+1} + \delta\dot{\mathbf{q}}_{i+1}$, we compute $\mathbf{q}_{i+1}^{n+1}$ by the simple rule:

$$\boxed{\mathbf{q}_{i+1}^{n+1} = \mathbf{q}^n + h\dot{\mathbf{q}}_{i+1}^{n+1}} \tag{2}$$

3.2 Linearized Implicit Euler

A common 'optimization' to implicit Euler employed in the graphics community is to not run Newton's method to convergence on the fully nonlinear problem, but to instead linearize about the previous time-step's solution and perform a single linear solve [1]. This is equivalent to performing one iteration of Newton's method with the initial iterate set to the previous time-step's solution, that is with $\dot{\mathbf{q}}_0^{n+1} = \dot{\mathbf{q}}^n$. This results in the system

$$\boxed{\left[\mathbf{M} - \left(h^2 \frac{\partial \mathbf{F}}{\partial \mathbf{q}} + h \frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}\right)\right] \delta\dot{\mathbf{q}} = h\mathbf{F}(\mathbf{q}^n + h\dot{\mathbf{q}}^n, \dot{\mathbf{q}}^n)} \tag{3}$$

where $\delta\dot{\mathbf{q}} = \dot{\mathbf{q}}^{n+1} - \dot{\mathbf{q}}^n$ and where $\frac{\partial \mathbf{F}}{\partial \mathbf{q}}$ and $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$ are evaluated at:

$$\frac{\partial \mathbf{F}}{\partial \mathbf{q}} = \frac{\partial \mathbf{F}(\mathbf{q}^n + h\dot{\mathbf{q}}^n, \dot{\mathbf{q}}^n)}{\partial \mathbf{q}}$$

$$\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = \frac{\partial \mathbf{F}(\mathbf{q}^n + h\dot{\mathbf{q}}^n, \dot{\mathbf{q}}^n)}{\partial \dot{\mathbf{q}}}$$

After solving for $\delta\dot{\mathbf{q}}$, we know $\dot{\mathbf{q}}^{n+1} = \dot{\mathbf{q}}^n + \delta\dot{\mathbf{q}}$. Once $\dot{\mathbf{q}}^{n+1}$ is known, $\mathbf{q}^{n+1}$ can be computed by:

$$\boxed{\mathbf{q}^{n+1} = \mathbf{q}^n + h\dot{\mathbf{q}}^{n+1}} \tag{4}$$

4 Note on *Local* and *Global* Indices

In many of the computations below, you are provided formulae for force Jacobians in 'local' coordinates. For example, the spring force involves two particles or 6 degrees of freedom, and the corresponding force Jacobian is a 6×6 matrix. You will need to place these components into the 'global' force Jacobian, however.

Consider a system with three particles, or 9 degrees of freedom. The force Jacobian will be a 9×9 matrix, but we can view it as a 3×3 ($\#particles \times \#particles$) block matrix with 3×3 ($x, y, z \times x, y, z$) blocks. At any given instant, let the force Jacobian be given by

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{C} \\ \mathbf{D} & \mathbf{E} & \mathbf{F} \\ \mathbf{G} & \mathbf{H} & \mathbf{I} \end{bmatrix}$$

where each letter denotes a 3×3 matrix. Consider a spring force connecting particles 0 and 2. This spring force will produce a force Jacobian given in 'local' indices by:

$$\begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix}$$

When we add this local force Jacobian into the global force Jacobian, we will obtain the matrix

$$\begin{bmatrix} \mathbf{A}+\mathbf{P} & \mathbf{B} & \mathbf{C}+\mathbf{Q} \\ \mathbf{D} & \mathbf{E} & \mathbf{F} \\ \mathbf{G}+\mathbf{R} & \mathbf{H} & \mathbf{I}+\mathbf{S} \end{bmatrix}.$$

5 Force Jacobians

5.1 Simple Gravity Force Jacobian

The simple gravity force has a constant force Jacobian, that is

$$\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = \frac{\partial \mathbf{F}}{\partial \mathbf{q}} = 0$$

[1] Note that our **Linearized Implicit Euler** is different from the one commonly used in the graphics community. If you are interested, you can read this paper: https://dl.acm.org/citation.cfm?id=280821.

5.2 Linear Drag Force Jacobian

For each particle, the linear drag force is defined as

$$\mathbf{F}(\dot{\mathbf{q}}) = -\beta\dot{\mathbf{q}}, \tag{5}$$

which has a 3 × 3 force Jacobian of:

$$\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = -\beta(\mathbf{Id})$$

The linear drag force has no position dependence, so $\frac{\partial \mathbf{F}}{\partial \mathbf{q}} = 0$. **Id** is the identity matrix.

In a YAML scene file, the drag force is enabled by specifying a positive drag coefficient:

```
# 1st particle
- pos: [0, 0, 1]
  vel: [0, 0, 0]
  fixed: false
  radius: 0.2     # optional
  mass: 0.2
  drag: 0.1  # optional. Enable Drag force
```

I've implemented the drag force Jacobian as an example. Please refer to the method sim.forces.eval_drag_force_vel_jacobians.

5.3 Spring Force Jacobian

For two particles in 3D interacting with a spring force, the force is

$$\mathbf{F}(\mathbf{q}_0, \mathbf{q}_1) = -\frac{\mathbf{q}_0 - \mathbf{q}_1}{\|\mathbf{q}_0 - \mathbf{q}_1\|} k \left(\|\mathbf{q}_0 - \mathbf{q}_1\| - l_0\right), \tag{6}$$

where l_0 denotes the rest length of the spring, and k denotes the spring stiffness. And the force Jacobian is a symmetric 6 × 6 matrix

$$\frac{\partial \mathbf{F}}{\partial \mathbf{q}} = \begin{pmatrix} \mathbf{K} & -\mathbf{K} \\ -\mathbf{K} & \mathbf{K} \end{pmatrix} \tag{7}$$

where $\mathbf{K}$ is a symmetric 3 × 3 matrix given by

$$\mathbf{K} = -k \left[\hat{\mathbf{n}}\hat{\mathbf{n}}^T + \frac{l - l_0}{l} \left(\mathbf{Id} - \hat{\mathbf{n}}\hat{\mathbf{n}}^T \right) \right].$$

where $\hat{\mathbf{n}} = (\mathbf{q}_0 - \mathbf{q}_1)/|\mathbf{q}_0 - \mathbf{q}_1|$ denotes a normalized vector pointing from $\mathbf{q}_1$ to $\mathbf{q}_0$; $l = |\mathbf{q}_0 - \mathbf{q}_1|$ denotes the distance between partial $\mathbf{q}_0$ and $\mathbf{q}_1$; and **Id** denotes the identity matrix. Recall that $\hat{\mathbf{n}}\hat{\mathbf{n}}^T$ is an example of an outer product. Our spring force has no velocity dependence, so $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = 0$.

5.4 Spring Damping Force Jacobian

The spring damping force depends on both its particles' positions and its particles' velocities, defined as

$$\mathbf{F}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{q}_1, \dot{\mathbf{q}}_1) = -\beta \frac{\mathbf{q}_0 - \mathbf{q}_1}{\|\mathbf{q}_0 - \mathbf{q}_1\|} \left(\frac{\mathbf{q}_0 - \mathbf{q}_1}{\|\mathbf{q}_0 - \mathbf{q}_1\|} \right)^T (\dot{\mathbf{q}}_0 - \dot{\mathbf{q}}_1). \tag{8}$$

This is the damping force applied on $\mathbf{q}_0$; and the force applied on $\mathbf{q}_1$ is $-\mathbf{F}(\mathbf{q}_0, \dot{\mathbf{q}}_0, \mathbf{q}_1, \dot{\mathbf{q}}_1)$.

Therefore, both $\frac{\partial \mathbf{F}}{\partial \mathbf{q}}$ and $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$ are nonzero. $\frac{\partial \mathbf{F}}{\partial \mathbf{q}}$ is given by:

$$\frac{\partial \mathbf{F}}{\partial \mathbf{q}} = \begin{pmatrix} \mathbf{K} & -\mathbf{K} \\ -\mathbf{K} & \mathbf{K} \end{pmatrix}$$

where $\mathbf{K}$ is a 3×3 matrix given by:

$$\mathbf{K} = -\frac{\beta}{l}\left[\hat{\mathbf{n}} \cdot (\dot{\mathbf{q}}_0 - \dot{\mathbf{q}}_1)\mathbf{Id} + \hat{\mathbf{n}}(\dot{\mathbf{q}}_0 - \dot{\mathbf{q}}_1)^T\right](\mathbf{Id} - \hat{\mathbf{n}}\hat{\mathbf{n}}^T)$$

$\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$ is given by:

$$\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = \begin{pmatrix} -\mathbf{B} & \mathbf{B} \\ \mathbf{B} & -\mathbf{B} \end{pmatrix}$$

where $\mathbf{B}$ is a symmetric 2×2 matrix given by:

$$\mathbf{B} = \beta\hat{\mathbf{n}}\hat{\mathbf{n}}^T$$

TODO [1]: Please implement methods that compute the spring and damping force Jacobians, namely eval_spring_force_pos_jacobians and eval_spring_force_vel_jacobians. The former method is to compute $\frac{\partial \mathbf{F}}{\partial \mathbf{q}}$ while the latter is to compute $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$.

5.5 Gravitational Force Jacobian

The Gravitational force, different from the Gravity force defined above, is defined to attract to particles

$$\mathbf{F}(\mathbf{q}_0, \mathbf{q}_1) = -G\frac{m_1 m_2}{\|\mathbf{q}_0 - \mathbf{q}_1\|^3}(\mathbf{q}_0 - \mathbf{q}_1),$$

where G denotes the gravitational constant, which will be specified in scene YAML file, and m_1 and m_2 denote the masses of the first and second particles. This is the force applied on particle $\mathbf{q}_0$, and the force on particle $\mathbf{q}_1$ is $-\mathbf{F}(\mathbf{q}_0, \mathbf{q}_1)$. The force Jacobian is a symmetric 6×6 matrix

$$\frac{\partial \mathbf{F}}{\partial \mathbf{q}} = \begin{pmatrix} \mathbf{K} & -\mathbf{K} \\ -\mathbf{K} & \mathbf{K} \end{pmatrix}$$

where $\mathbf{K}$ is a symmetric 3×3 matrix given by

$$\mathbf{K} = -\frac{Gm_1 m_2}{l^3}\left(\mathbf{Id} - 3\hat{\mathbf{n}}\hat{\mathbf{n}}^T\right).$$

$\hat{\mathbf{n}} = (\mathbf{q}_0 - \mathbf{q}_1)/|\mathbf{q}_0 - \mathbf{q}_1|$ denotes a normalized vector pointing from $\mathbf{q}_0$ to $\mathbf{q}_1$, and $l = |\mathbf{q}_0 - \mathbf{q}_1|$ denotes the distance between the two particles. The gravitational force has no velocity dependence, so $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = 0$.

In Nemo, Gravitational forces can be enabled on the basis of particle pairs, just like spring forces. So you can disable gravitational forces between certain pairs of particles. Here is an example:

```
gravitational:
  - particle_ids: [0, 1]
    G: 100.  # Newton's constant for gravitational force
```

where G is the gravitational constant (a.k.a. Newton's constant). In this way, you can set different gravitational constants for different particle pairs.

TODO [2]: Please implement method that compute the gravitational force Jacobians, namely eval_gravitational_force_pos_jacobians. Because here $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$ is always zero, you don't need a method that computes velocity force Jacobian.

Validation: The Jacobian implementation is prone to error. Before you go ahead implementing implicit Euler integrator, MAKE SURE YOUR JACOBIAN COMPUTATION IS BUG FREE! To this end, I provide a set of unit tests that compares your Jacobian against finite difference estimation of derivatives. As a sanity check, if your implemention is correct, you code should pass the following unit tests (by running the following command in your terminal):

```
pytest src/tests
```

and output the following lines

```
src/tests/test_builder.py .....
src/tests/test_forces.py .
src/tests/test_jacobians.py .........
```

6 Required Features

6.1 Linearized Implicit Euler

You will implement linearized implicit Euler as detailed in the preceding *Linearized Implicit Euler* section, using the state from the end of the last time step as your initial guess. This will involve computing the force Jacobians for the forces. In addition, you will have to solve a linear system using numpy. You construct a matrix A and its right-hand-side vector b, and use

```
import numpy as np
x = np.linalg.solve(A, b) # such that A @ x = b
```

to get the solution x.

TODO [3]: Please implement linearized implicit Euler in the provided source file solvers/linearized_implicit.py; Read the comments carefully in that file.

7 Fixed Particles

To fix a particle, we want an iteration of Newton's method to produce no change in that particle's position or velocity. One way to accomplish this is to set each of the particle's degrees of freedom in the right hand side of (3) to 0, to set the row and column corresponding to the particle's degrees of freedom in the left hand side of (3) to 0, and to set the diagonal entry corresponding to the particle's degrees of freedom in the left hand side of (3) to 1. At the end of an iteration of Newton's method (just one iteration in the case of linearized implicit Euler), the net result will be that the change in the particle's velocity, and thus position, is 0.

For example, say we want to fix particle 5. We would clear entries 15, 16, and 17 on the right hand side of (3). We would also clear rows 15, 16, and 17 and columns 15, 16, and 17 in the left hand side of (3). We would finally set entry $(15, 15)$, $(16, 16)$ and entry $(17, 17)$ of the left hand matrix of (3) to 1. When we now solve this linear system, the result will be that the x, y, and z coordinate of particle 5 will remain unchanged.

8 Full Implicit Euler

Recall that linearized implicit Euler is equivalent to taking a single step of Newton's method. We can run Newton's method to convergence if desired, however. One additional complication this introduces is the question of how to know when our system has reached convergence. There are a number of choices. We could monitor the magnitude of the residual, we could monitor the change in the magnitude of the residual between time steps, we could monitor the magnitude of step size, the options go on.

For our purposes, it will suffice to monitor the absolute magnitude of the step size. That is, at the end of each iteration of Newton's method, if $|\dot{\mathbf{q}}_{i+1}^{n+1} - \dot{\mathbf{q}}_i^{n+1}| < 10^{-5}$, we declare that Newton's method has converged.

TODO [4]: Implement implicit Euler as detailed in equation (1) using the source file implicit_euler.py. In the constructor of ImplicitEulerSolver, you can set the maxinum number of Newton iterations (default: 5) and the tolerace for checking for termination. In other words, you terminate your Newton's iteration when either the ieration number reaches self.maxits or the absolute magnitude of the step size is smaller than self.tol.

8.1 Bonus: Creative Scenes

Like PA1, while we provide a set of testing scenes, we encourage students to create their own scenes or even extend the starter code with more interesting features (e.g., you can consider to extent `src/nemo/sim/forces.py`) to add other kinds of forces (such as wind force or magnetic force) and use them in your implicit integrators.

This part is not *required*. But if you feel interested and motivated to do that, we will give you up to 5 bonus points (out of 100) based on the effort you need to devote to create the creative scenes and extensions.

When you submit, please include in the `README.md` file a list of scenes and features that you added and point to the files and commands we should run to see your effects.