Animation & CGI Motion: Mass-Spring Systems

Theme 1 Milestone 3

Academic Honesty Policy

You are permitted and encouraged to discuss your work with other students. You may work out equations in writing on paper or a whiteboard. You are encouraged to use the discussion boards to converse with other students, the TA, and the instructor.

HOWEVER, you may NOT share source code or hardcopies of source code. Refrain from activities or the sharing of 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. Source code should be yours and yours only. Do not cheat.

1 Introduction

In Milestone 3 of Theme 1, you will implement a new integrator, linearized implicit Euler. This will involve the computation of the *force Jacobian* for each of the forces you implemented in Milestone 2, as well as the solution of a linear system. The grading and lateness policies remain the same as in previous milestones; you will have access to an 'oracle' and roughly half of the testing scenes.

You will also have the opportunity to earn extra credit by implementing a fully nonlinear version of implicit Euler, as well as by experimenting with a new vortex-generating type of force.

2 New XML Features

In addition to the xml tags from Milestones 1 and 2, Milestone 3 introduces a new required feature:

- The *integrator* node now accepts the type "linearized-implicit" which is used in the feature linearized implicit Euler discussed below:

  ```
  <integrator type="linearized-implicit" dt="0.01"/>
  ```

Milestone III introduces a new extra credit feature:

- The *integrator* node now accepts the type "implicit" which is used in the extra-credit feature implicit Euler discussed below:

  ```
  <integrator type="implicit" dt="0.01"/>
  ```

We have also provided a 'free' feature that you are welcome to use while constructing your creative scene:

- The *vortexforce* node defines a 'vortex force' acting between two particles, which we have implemented and is used in the 'free' feature we discuss in the vortex force section:

  ```
  <vortexforce i="0" j="1" kbs="0.8" kvc="1000.0"/>
  ```

  i and j specify the particles this force acts between. *kbs* controls the strength of the overall 'Biot-Savart' force, while *kvc* controls the amount of viscous drag. You will find that this force works best when one particle's mass is extremely large while the other is very small. The extremely massive particle will generate a vector field through which the lighter particle advects.

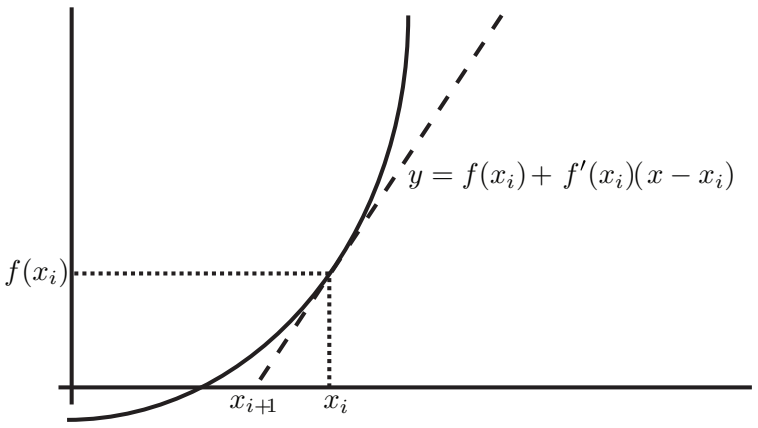

Figure 1: One Step of Newton's Method

3 Introduction to 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.

3.1 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 later 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 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.

3.2 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

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

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 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.

4 Introduction to 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:

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

Let N denote the number of vertices in the system. Here $\mathbf{M}$ is a $2N \times 2N$ mass matrix, in our case a diagonal matrix with masses on the diagonal (entries $0,0$ and $1,1$ are the mass of the first particle, entries $2,2$ and $3,3$ 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 $\mathbf{m} = [m_0, m_0, m_1, m_1, \ldots, m_N, m_N]$. The i^{th} entry 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!

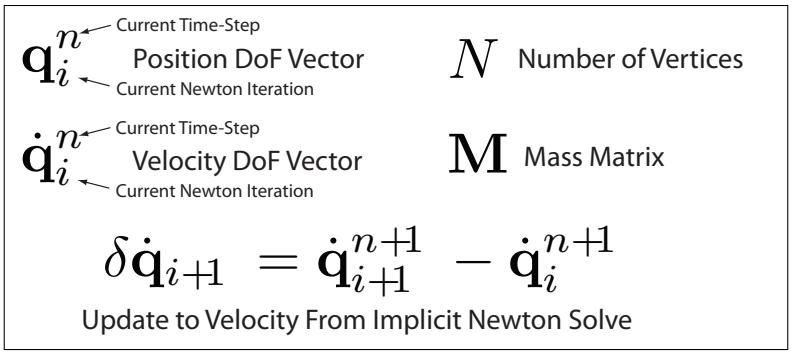

Figure 2: Some Useful Quantities

Observe that we have $4N$ unknowns in our system, where N is the number of vertices. $2N$ of these unknowns are positions, and $2N$ 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 $4N$:

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

Computing the gradient for Newton's method, as discussed in class, we obtain a $4N \times 4N$ matrix:

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

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

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

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 timestep, $\mathbf{q}^n$ 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.

4.1 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 $2N$:

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

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

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

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

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

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

where $\frac{\partial \mathbf{F}}{\partial \mathbf{q}^{n+1}}$ and $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}^{n+1}}$ are evaluated at $(\mathbf{q}^n + h\dot{\mathbf{q}}_i^{n+1}, \dot{\mathbf{q}}_i^{n+1})$. Note that this $2N \times 2N$ system is half the size of our original $4N \times 4N$ 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:

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

5 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. 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

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

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:

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

6 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 4 degrees of freedom, and the corresponding force Jacobian is a 4×4 matrix. You will need to place these components into the 'global' force Jacobian, however.

Consider a system with three vertices, or 6 degrees of freedom. The force Jacobian will be a 6×6 matrix, but we can view it as a 3×3 ($vertices \times vertices$) matrix with 2×2 $(x, y \times x, y)$ blocks. At any given instant, let the force Jacobian be given by

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

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

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

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

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

7 Required Features for Milestone 3

7.1 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 from Milestone II; we have provided these formulae below. In addition, you will have to solve a linear system. We have provided both code to solve a linear system as well as an example with a 'toy' linear system. Copying this example here:

```
// Create a 10 x 10 matrix
MatrixXs A(10,10);
// Fill the matrix with random numbers
A.setRandom();
// Create a vector of length 10
VectorXs b(10);
// Fill the vector with random numbers
b.setRandom();
// Compute the solution to A*x = b
VectorXs x = A.fullPivLu().solve(b);
// Verify that we computed the solution (the residual should be roughly 0)
std::cout << (A*x-b).norm() << std::endl;
```

Please implement linearized implicit Euler in the provided source file *LinearizedImplicitEuler.cpp*. You can find a linear solver example in *ImplicitEuler.cpp*.

7.2 Simple Gravity Force Jacobian

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

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

7.3 Spring Force Jacobian

For two particles in 2D interacting with a spring force, the force Jacobian is a symmetric 4×4 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 2×2 matrix given by

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

k denotes the spring stiffness, $\hat{\mathbf{n}} = (\mathbf{x}_i - \mathbf{x}_j)/|\mathbf{x}_i - \mathbf{x}_j|$ denotes a normalized vector pointing from $\mathbf{x}_j$ to $\mathbf{x}_i$, $l = |\mathbf{x}_i - \mathbf{x}_j|$ denotes the length of the spring, l_0 denotes the rest length of the spring, and $\mathbf{Id}$ 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 $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = 0$.

Augment *SpringForce.cpp* to compute the force Jacobian.

7.4 Spring Damping Force Jacobian

The spring damping force depends on both its particles' positions and its particles' velocities. 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 2×2 matrix given by:

$$\mathbf{K} = -\frac{\beta}{l}(\hat{\mathbf{n}} \cdot (\mathbf{v}_i - \mathbf{v}_j)\mathbf{Id} + \hat{\mathbf{n}}(\mathbf{v}_i - \mathbf{v}_j)^T)(\mathbf{Id} - \hat{\mathbf{n}}\hat{\mathbf{n}}^T)$$

$\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}}$ 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:

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

7.5 Gravitational Force Jacobian

For two particles in 2D interacting with a gravitational force, the force Jacobian is a symmetric 4×4 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 2×2 matrix given by

$$\mathbf{K} = -\frac{G m_1 m_2}{l^3} \left(\mathbf{Id} - 3\hat{\mathbf{n}}\hat{\mathbf{n}}^T\right).$$

$\hat{\mathbf{n}} = (\mathbf{x}_i - \mathbf{x}_j)/|\mathbf{x}_i - \mathbf{x}_j|$ denotes a normalized vector pointing from $\mathbf{x}_j$ to $\mathbf{x}_i$, G denotes the gravitational constant, m_1 and m_2 denote the masses of the first and second particle, and $l = |\mathbf{x}_i - \mathbf{x}_j|$ denotes the distance between the two particles. The gravitational force has no velocity dependence, so $\frac{\partial \mathbf{F}}{\partial \dot{\mathbf{q}}} = 0$.

7.6 Linear Drag Force Jacobian

For each particle, the linear drag force has a 2×2 force Jacobian of:

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

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

8 Fixed Vertices

To fix a vertex, we want an iteration of Newton's method to produce no change in that vertex's position or velocity. One way to accomplish this is to set each of the vertex's degrees of freedom in the right hand side of (2) to 0, to set the row and column corresponding to the vertex's degrees of freedom in the left hand side of (2) to 0, and to set the diagonal entry corresponding to the vertex's degrees of freedom in the left hand side of (2) 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 vertex's velocity, and thus position, is 0.

For example, say we want to fix vertex 5. We would clear entries 10 and 11 in the right hand side of (2). We would also clear rows 10 and 11 and columns 10 and 11 in the left hand side of (2). We would finally set entry $(10, 10)$ and entry $(11, 11)$ of the left hand side of (2) to 1. When we now solve this linear system, the result will be that the x and y coordinate of vertex 5 will remain unchanged.

9 Full Implicit Euler (Extra Credit)

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 timesteps, 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 stepsize. That is, at the end of each iteration of Newton's method, if $|\dot{\mathbf{q}}_{i+1}^{n+1} - \dot{\mathbf{q}}_{i}^{n+1}| < 10.0^{-9}$, we declare that Newton's method has converged.

Implement implicit Euler as detailed in (1) using the source file *ImplicitEuler.cpp*. Note that this feature is NOT required to obtain a 100% on the assignment. The extra credit scenes have been placed in a separate folder, *extracreditassets*. The grading oracle will work with these scenes independently of the required scenes.

10 Vortex Force (Free Feature for Creative Scene)

We have provided a force for you that generates a vortex-like effect. We have also provided an example file, *assets/t1m3/VortexExample/VortexExample.xml*, that demonstrates the use of this force. You are welcome to use this force in one of your creative scenes, but this is not required.

11 Creative Scene

As part of your final submission for this milestone, please include a scene of your design that best shows off your program. Your scene will be judged by a secret of committee of top scientists using the highly refined criteria of:

1. How well the scene shows off this milestone's "magic ingredients" (a la Iron Chef).
2. Aesthetic considerations. The more beautiful, the better.
3. Originality.

Top examples will be posted to the discussion board. Please remember to generate this scene before submitting, as the Codio box will become read-only after you submit.

11.1 Making Movies

Please remember to generate this scene before submitting, as the Codio box will become read-only after you submit. The FOSSSim starter code comes with a PNG outputting utility that can help you create movies from your simulation. To enable it, create a folder named "pngs" in the directory that you are calling your executable from (i.e. your current directory) and run your code with the

```
-g 1
```

flag enabled. This will prompt your program to automatically save a PNG file for each frame of the simulation. This wont work if the "pngs" folder does not exist so you need to create it first before you run the program.

After the simulation finishes, you'll find all the frames in the pngs folder. Then you can make videos out of them with command-line tools such as mencoder and ffmpeg. Both mencoder and ffmpeg are easy-to-use tools available on the Codio boxes. You can execute this command:

```
mencoder mf://pngs/*.png -mf fps=24 -ovc lavc -lavcopts vcodec=msmpeg4v2 -oac copy -o output.avi
```

followed by:

```
ffmpeg -i output.avi -vcodec libx264 -crf 25 output.mp4
```

in order to create an mp4 video you may upload for the Creative portion of the assignment.

Note, ffmpeg can be used to create a video from the png images in one step, although sometimes this will give an error depending on the count and dimensions of input files. Nevertheless, the following command can take pngs and convert them to an mp4 movie in one:

```
ffmpeg -r 24 -f image2 -i ./pngs/frame%05d.png -vcodec libx264 -crf 25 -pix_fmt yuv420p test.mp4
```

Lastly, you may preview your movie by running the following command and looking at the virtual desktop:

```
mplayer <movie_name>
```

Please submit the mp4 file to the Peer Review Assignment portion of this week. An explanation for arguments to both mencoder and ffmpeg can be found online.

FAQ

Theme 1 Milestone 3 FAQ:

Q: How should I interpret the accumulateddUdxdx() and accumulateddUdxdv() functions?
A: Much like we asked you to compute the gradient of energy in previous milestones, we are now interested in the second derivative of the potential energy U, with respect to both x and v variables. Another way to interpret this is we are asking for $-dF/dx = ddU/dxdx$ and $-dF/dv = ddU/dxdv$. The variables passed dx and dv therefore correspond to the differences of positions and velocities calculated against the current one (change from one timestep to another), which may come in useful for computing the Hessian.

Q: Can I use other video capture devices, screen capture, or other tools to generate the creative video?
A: Yes, you may use whatever tool you desire as long as the final output is an AVI file for submission.

Q: When running Newton's method, what sort of criteria should we use against the threshold? Should we check individual components, sum of components, or some other method?
A: Great question, there is no single "correct" criteria for checking convergence; what we are doing is just trying to capture the idea that q_{i+1} does not differ much from q_i. Therefore in practice it is reasonable to

use any norm on the difference vector. For this milestone however, we test with the commonly used norm, the l-2 norm, i.e. $||q_{i+1} - q_i||_2 < 1e - 9$.

Q: Do we need to have all of the addHessV and HessX functions implemented?
A: You need to implement all the HessV and HessX functions, wherever applicable (for example SimpleGravityForce's Jacobian is zero so there is nothing to do). If you don't implement anything the Linearized Implicit Euler is reduced to explicit Euler, which will still look fine but your result won't agree with the Oracle's

Tips:
- Don't forget to pass in dx and/or dv to the Jacobians from the integrator if you plan on using them in the *Force.cpp* function for calculation

- Don't forget to normalize unit vectors

- Be careful when creating xml files. Common errors include assigning colors or edges to elements that don't exist yet, which will result in an error or segfault of some kind.

- The Mass Matrix should be a Mass matrix, even though we store it as a vector.