COMS W4167: Mass-Spring Systems

Theme I, Milestone II

Introduction

In Milestone II of Theme I, you will implement a new integrator, Symplectic Euler, as well as spring, gravitational, and damping forces.

Prerequisites

This milestone requires familiarity with the *gradient* for a multivariate function. To understand and complete this assignment, you should be comfortable with calculating (by hand) the gradient of a function in coordinates.

Test yourself: can you show that $\nabla_{(x,y,z)} \frac{1}{2}(x^2+y^2)+z = (x,y,1)$. Does $\nabla_{(x,y)} \frac{1}{2}(x+y)^2 \overset{?}{=} (x+y, x+y)$?

You can review the geometric notion of a gradient, and the calculation of gradients, from any multivariable Calculus text, such as:

Gilbert Strang. *Calculus.* Wellesley-Cambridge Press. 1991.
§13.4 Directional Derivatives and Gradients, pp. 490–496.

Presently, a link is available on the web:
`https://ocw.mit.edu/resources/res-18-001-calculus-online-textbook-spring-2005/textbook/MITRES_18_001_strang131134.pdf`

1 Policies

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 the discussion board to converse with other students, the TAs, and the instructor.

HOWEVER, you may NOT share source code or hardcopies 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. Source code should be yours and yours only. Do not cheat. For more details, please refer to the academic honesty on EdX.

All other policies

All the policies that applied to our previous homework assignment (including but not limited to collaboration, grading, lateness, submission) apply to this assignment without modification. Please review these policies as described in Theme I Milestone I and on EdX before beginning this assignment.

2 New XML Features

In addition to the xml tags from Milestone I, Milestone II adds the new features:

1. The *integrator* node now accepts the type "symplectic":

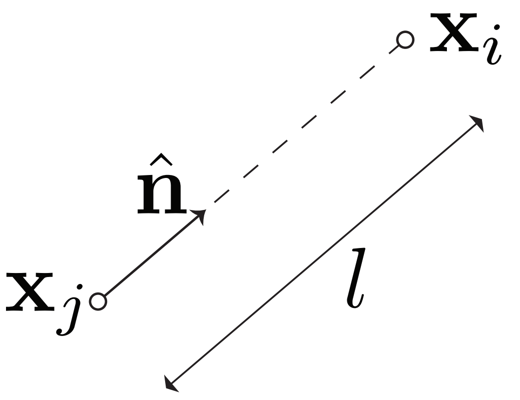

Figure 1: Geometry in Spring and Gravitational Force

```
<integrator type="symplectic-euler" dt="0.01"/>
```

2. The *springforce* node adds a spring to the system:

   ```
   <springforce edge="0" k="2.0" l0="1.5" b="0.1" />
   ```

   The edge property sets the edge this spring is associated with, the k attribute sets the stiffness of the spring, and the l0 attribute sets the rest length of the spring. The optional property b introduces an internal damping force to the spring. If b is not specified, it defaults to 0.

3. The *gravitationalforce* node adds an attractive force acting between two particles:

   ```
   <gravitationalforce i="0" j="1" G="0.000118419"/>
   ```

   The properties i and j set the particles the force acts on, and the G attribute linearly scales the magnitude of the gravitational force.

4. The *dragdamping* node adds a force to the system that resists the motion of all particles:

   ```
   <dragdamping b="3.0"/>
   ```

   The property b is a constant that linearly scales the magnitude of the damping force.

3 Required Features for Milestone II

3.1 Spring Force

The potential energy of a spring, or 'harmonic oscillator,' is given by:

$$\boxed{U(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{2}k\left(l(\mathbf{x}_i, \mathbf{x}_j) - l_0\right)^2}$$

Let $l(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2}$ denote the distance between the two particles, let l_0 denote the spring's rest length, and let k denote the spring's stiffness. Note that l_0 is typically a constant, although one could 'script' this parameter's value to achieve an artistic effect (for example, one could increase l_0 to achieve an inflation-like effect). See Figure 1. Computing the gradient of the potential with respect to one of the particles, we find by the chain rule that

$$\nabla_{x_k} U = k\,(l - l_0)\,\nabla_{x_k} l.$$

Computing $\nabla_{x_k} l$, we find that

$$\nabla_{x_k} l = \frac{1}{2}((\mathbf{x}_i - \mathbf{x}_j)^2)^{-\frac{1}{2}} 2(\mathbf{x}_i - \mathbf{x}_j)\nabla_{x_k}(\mathbf{x}_i - \mathbf{x}_j) = \nabla_{x_k}(\mathbf{x}_i - \mathbf{x}_j)\frac{\mathbf{x}_i - \mathbf{x}_j}{\sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2}} = \nabla_{x_k}(\mathbf{x}_i - \mathbf{x}_j)\hat{\mathbf{n}}$$

where $\hat{\mathbf{n}} = (\mathbf{x}_i - \mathbf{x}_j)/\sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2}$ is a unit-length vector pointing from vertex x_j to vertex x_i. For vertex i, $\nabla_{x_i}(\mathbf{x}_i - \mathbf{x}_j) = 1$, and for vertex j, $\nabla_{x_j}(\mathbf{x}_i - \mathbf{x}_j) = -1$.

This tells us that the gradient of the distance between two particles is parallel to a vector between the two particles (e.g. $\hat{\mathbf{n}}$). Let us take a step back and see what we can learn from this result. Consider the interpretation of the gradient as the direction of maximum change. If we move a particle in either direction perpendicular to $\hat{\mathbf{n}}$, the length between the particles will increase. This implies that the particle is at a local minimum along the direction perpendicular to $\hat{\mathbf{n}}$, and thus that the directional derivative in this direction is 0. Therefore, the gradient, or direction of maximum change, must point along $\hat{\mathbf{n}}$, and our result makes intuitive sense.

We conclude that the gradients of the potential with respect to each vertex are given by:

$$\boxed{\nabla_{x_i} U = k\,(l - l_0)\,\hat{\mathbf{n}}}$$

$$\boxed{\nabla_{x_j} U = -k\,(l - l_0)\,\hat{\mathbf{n}}}$$

Observe that the force (minus the gradient, don't forget that pesky minus sign!) is directed along the vector between the particles, as we would expect for a spring. Further, observe that the gradients, and thus forces, sum to 0. That is, our solution obeys Newton's third law.

Edit the provided source file *SpringForce.cpp* to compute this potential energy and its gradient. Please make sure you add the force's contribution to the proper location in the 'global' force vector. For example, if the spring acts on particles 2 and 5, you will add contributions to the 4^{th}, 5^{th}, 10^{th}, and 11^{th} entries of the 'global' force vector.

3.2 Gravitational Force

The potential energy of two particles interacting via gravity is given by:

$$\boxed{U(\mathbf{x}_i, \mathbf{x}_j) = -\frac{Gm_i m_j}{l}}$$

As before, let $l(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(\mathbf{x}_i - \mathbf{x}_j)^2}$ denote the distance between the two particles. Computing the gradient of the potential with respect to one of the particles, we find by the chain rule that

$$\nabla_{x_k} U = \frac{Gm_i m_j}{l^2}\nabla_{x_k} l$$

where $\nabla_{x_k} l$ is again given by $\nabla_{x_i} l = \hat{\mathbf{n}}$ and $\nabla_{x_j} l = -\hat{\mathbf{n}}$. Thus, the gradients of the potential with respect to each vertex are given by:

$$\boxed{\nabla_{x_i} U = \frac{Gm_i m_j}{l^2}\hat{\mathbf{n}}}$$

$$\boxed{\nabla_{x_j} U = -\frac{Gm_i m_j}{l^2}\hat{\mathbf{n}}}$$

As a 'sanity check' we immediately observe that this force is central and obeys Newton's third law (Newton's third law states the familiar 'for every action there is an equal and opposite reaction').

Edit the provided source file *GravitationalForce.cpp* to compute this potential energy and its gradient.

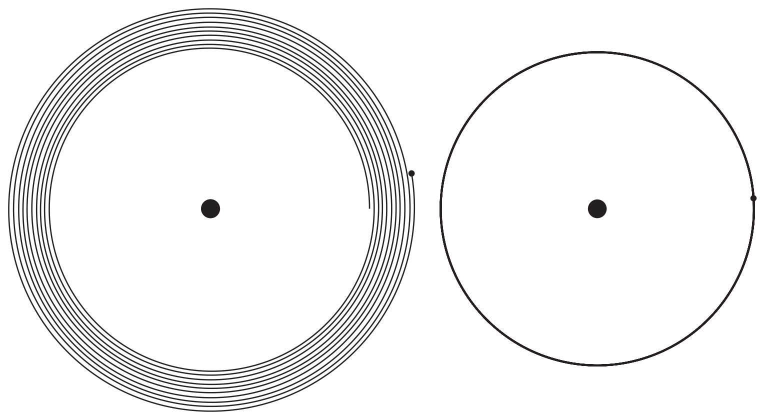

Figure 2: Comparison of Orbits for Explicit Euler (left) and Symplectic Euler (right)

3.3 Linear Damping Force

All of the forces we have introduced thus far are conservative; that is, we can assign a scalar valued potential to each point in space, and the difference in potential between any two points is independent of the path taken between the points. There are useful forces that are not conservative, however, such as friction and drag. Note that a damping force, by its very (dissipative) nature, is not conservative, i.e., it does not act to preserve total energy. You will implement one such force that linearly resists a particle's motion. That is, for each particle in your system, the force's magnitude is given by

$$\mathbf{F}_i = -\beta \mathbf{v}_i$$

where β is a scalar damping constant.

Edit the provided source file *DragDampingForce.cpp* to compute this force. Watch out for sign errors!

3.4 Spring Damping Force

The linear damping force models the motion of an object in a 'thick' fluid, and will eventually damp ALL motion in the scene. In contrast, we can introduce a force that models internal dissipation within a spring. Unlike the linear damping force, this internal force only damps motion that compresses or extends the spring. Given two particles i and j interacting with a spring force, the spring damping force is given by

$$\mathbf{F}_i = -\beta \hat{\mathbf{n}} \cdot (\mathbf{v}_i - \mathbf{v}_j)\hat{\mathbf{n}}$$

$$\mathbf{F}_j = \beta \hat{\mathbf{n}} \cdot (\mathbf{v}_i - \mathbf{v}_j)\hat{\mathbf{n}}$$

where β is a constant to scale the magnitude of the damping force, and $\hat{\mathbf{n}}$ is defined as in the *SpringForce* section. Note that while the spring damping force is not conservative (and it leads to a loss of energy), it does obey Newton's third law, and therefore it does conserve the total momentum of both particles.

Augment the spring force in the source file *SpringForce.cpp* to include this damping force.

3.5 Symplectic Euler

The time integrator we implemented in the first milestone is called *Explicit Euler.* It is called explicit because of the specific order it evaluates the position and velocity updates: the values for the next time step are all computed by directly evaluating expressions using the values of the last time step. From a *finite differencing* point of view, Explicit Euler can be interpreted as the following

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

Now take a closer look at the first equation. The equation uses $\dot{\mathbf{q}}^n$, the instantaneous change rate of $\mathbf{q}$ at the last time step n, to represent the average change rate of $\mathbf{q}$ between the current time step n and the next time step $n+1$. Of course this is only *approximately* true, and we'll look at the error introduced by this approximation (i.e. the *discretization error*) later in the course. Because the instantaneous point in time picked to represent the average is the left endpoint (i.e. time step n) of the interval, this discretization scheme is called *forward differencing.* The second equation is thus the same forward differencing applied to $\dot{\mathbf{q}}$. It is important to realize that this is not the only reasonable way to do the approximation; for example $\dot{\mathbf{q}}^{n+1}$ is obviously as good an approximation of the average change rate between time step n and $n+1$ as is $\dot{\mathbf{q}}^{n+1}$. Using the right endpoint (i.e. time step $n+1$) is naturally called *backward differencing.* This yields the following integrator:

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

The problem with this formulation is that the unknowns (the values for time step $n+1$) are present on both sides of the formula and it becomes an equation that needs to be solved, rather than a direct evaluation. We'll follow down this path next week – it leads to an important class of integrators (called *Implicit* integrators) that possess desirable properties at the cost of more computation. On the other hand, what else can we do without requiring solving equations? One possible scheme is to use forward differencing for $\dot{\mathbf{q}}$, and backward differencing for $\mathbf{q}$:

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

Next-time-step quantity $\dot{\mathbf{q}}^{n+1}$ is still present on the right hand side of the first equation, but this term actually becomes known if we evaluate the second equation first. Therefore, no equation needs to be solved. This time integration scheme is called *Symplectic Euler.*

Symplectic Euler integrator is sometimes also called semi-implicit Euler or forward-backward Euler. We adopt the name "Symplectic Euler" as a standard. The 'update rule' is:

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

Notice that the velocity update depends only on the position and velocity at the previous timestep, while the position update depends on the velocity at the current timestep. Contrast this with explicit Euler, where both updates depend on the previous step's position and velocity.

To qualitatively verify the behavior of symplectic Euler, compare to the behavior of explicit Euler with the test examples *assets/GravityTests/test00explicit.xml* and *assets/GravityTests/test00symplectic.xml.* Explicit Euler should produce an unstable orbit and 'spiral outward,' while Symplectic Euler should produce a stable orbit. For your personal edification, plot and compare the total energy with explicit Euler and symplectic Euler. See Figure 2 for examples of these orbits.

As an additional test, run the scene *assets/GravityTests/test00explicit.xml.* For the default timestep and spring stiffness, you should simply see an oscillating spring. Experiment with different spring stiffnesses, masses, and time-steps. For large values of $\frac{k}{m}$ and for large time-steps, you should observe ‘explosions’ with explicit Euler. Conduct these same tests with symplectic Euler.

Implement symplectic Euler using the provided source file *SymplecticEuler.cpp*.

4 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.

4.1 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 won’t 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 2 FAQ:

Q: Where can I find an implementation of the previous theme so I can see what I did wrong?
A: Every theme builds on itself, this means that the starter code for this assignment includes a correct implementation to the previous assignment. To discourage cheating, we will not release the solutions, but you are free to post questions on the discussion boards if you want help with a previous assignment.

Q: When implementing the spring force, how do I make sure the force acts on the correct edges and how do I index into the correct particles to apply the force?
A: The code is abstracted out to handle most of this and make it very generic to use. Each 'Force' reference in the system state contains information about who it applies to. In the case of the spring force, make use of the endpoint member variables to get the indices of the particles involved. Each spring interaction will be its own force reference, containing the specific information about that particle-particle (edge) interaction and coefficients.

Q: After the first assignment I read up on the Eigen class and found lots of useful functions and shortcuts in their API, can I use them?
A: Yes, anything from Eigen library is allowed. We don't grade your submissions based on your coding style, only on its execution result.

Q: What does '.segment<>()' mean on a vector?
A: This is an Eigen function for accessing the members of a vector, the notation vec.segment<n>(i) means you are accessing a 'n' size consecutive segment of Vector 'vec', starting at the index 'i'.