CSCI 8980 Assignment 1:
Mass-Spring Systems and Time Integration

Due date: Monday, October 3, 2016

In this assignment, you will implement three different time integration schemes and use them to simulate a damped harmonic oscillator and a mass-spring system. The `#define` variables at the top of the `main.cpp` file in the starter code will let you change between these combinations.

A Notes section has been added to the end of this handout, which includes compilation instructions and other implementation hints. Read at least the compilation instructions in Sec. 4.1 before trying to compile the starter code.

1 Explicit integration

The starter code defines an abstract `PhysicalSystem` class that serves as a generic interface for time integrators to operate on. Also included is an example subclass of it, `DampedHarmonicOscillator`, that implements a one-dimensional oscillator with viscous damping,

\[ mx'' = -kx - cx'. \]  

(1)

First, implement forward Euler integration in the class `ForwardEuler`. In this case, you will not need to call `getInertia`, `getForces`, or `getJacobians`; apart from `getState` and `setState` you will only need `getAccelerations`.

Next, implement the second-order Runge-Kutta method in `RungeKutta2` using the same approach.

Apply your implementation to a `DampedHarmonicOscillator` with parameters \( m = 1 \), \( k = 100 \), \( c = 2 \) and initial conditions \( x(0) = 1 \), \( x'(0) = 0 \). Use a time step \( \Delta t = 1/100 \) to integrate from \( t = 0 \) to \( t = 1 \), and report the value of \( x(1) \).

For reference, the analytical solution is

\[ x(1) = \frac{33\cos(3\sqrt{11}) + 11\sin(3\sqrt{11})}{33e} \]  

(2)

which is approximately \(-0.3369\). The forward Euler approximation should be about \(-0.536\) while the RK2 approximation should be \(-0.333\).

What happens if you take larger time steps of \( \Delta t = 1/25 \)?
2 Mass-spring systems

The starter code also includes some code that reads a 2D triangle mesh and produces a Mesh2D object. If we treat the vertices of the mesh as particles and the edges as springs, we can deform any such mesh as though it were elastic.

A skeleton implementation of ParticleSystem that inherits from PhysicalSystem is provided. Complete it by defining Particle, Force, SpringForce, and AnchorForce classes and using them to implement ParticleSystem::getState etc. (You can omit ParticleSystem::getJacobians until the next part.) Finally, implement a function to initialize a ParticleSystem from a Mesh2D, and another to copy the updated positions of the particles back to the Mesh2D’s vertices.

In the main program, create a mass-spring system using the same parameters as the last part \((m = 1, k_s = 100, k_d = 2)\). Inside the while loop, perform one time step of \(\Delta t = 1/60\) each iteration using RK2. At this point, you should have a mass-spring system whose vertices you can drag around with the mouse. However, if you agitate the system too much or if you increase the stiffness \(k_s\), the system is liable to explode.

(If nothing happens when you try to drag the mesh vertices around, did you remember to update them with the new particle positions?)

3 Implicit integration

Finally, implement backward Euler time integration in the BackwardEuler class. You can use the provided wrapper for Eigen’s conjugate gradient solver.

First test it on the damped harmonic oscillator. With the same parameters and \(\Delta t = 1/100\), you should get \(x(1) \approx -0.219\). Observe that the magnitude of the result is smaller than the true value because the backward Euler method exhibits artificial damping.

Next, apply backward Euler to your mass-spring system. For this, you will need to implement the Jacobians of the spring forces. Once you have that working, your simulator should be rock-solid even with arbitrarily large \(k_s\) (although it will be more and more damped).

4 Notes

4.1 Compilation instructions

The starter code depends on Eigen for linear algebra and GLFW for window management, and uses SCons (a modern replacement for Make) for compilation. If you’re using Linux, I recommend using your package manager to install them. On Ubuntu, I believe they’re libeigen3-dev, libglfw3-dev, and scons. (On the CSE Labs machines, Eigen and SCons are already installed, but you will need to download and compile GLFW from source.)
To compile the starter code, run `scons` in the code directory. This reads the build recipe in the `SConstruct` file and runs the compiler and linker. If you compiled GLFW from source, you will need to edit `SConstruct` first to tell SCons where to find it; see the instructions in the file.

4.2 Mouse interaction

The starter code includes some code for mouse-driven forces. Whenever the user drags a particle, we will apply a spring force between it and the position of the mouse pointer. Finding the clicked-on particle and updating the other end of the spring whenever the mouse moves is taken care of by the `updateMouseForce` function, but you will need to connect it to the rest of your simulation. Once you have a mass-spring system constructed, do

```cpp
AnchorForce mouseForce(nullptr, Vector2d(0,0), 1000, 50);
massSpringSystem.forces.push_back(&mouseForce);
```

to add the mouse force to it, and then call

```cpp
updateMouseForce(ms, mouseForce, window.mousePos(), window.mouseDown());
```

on every iteration of the `while` loop. (The `AnchorForce` class is written so that if its particle pointer is null, it doesn’t do anything. Whenever the user clicks on a particle, we update the pointer to the particle so the spring starts acting on it.)

4.3 Block vectors and matrices

Eigen has some convenient functions for working with vectors and matrices in block form like

\[
x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}, \quad J = \begin{bmatrix} J_{11} & J_{12} & \cdots \\ J_{21} & J_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}. \tag{3}
\]

For a vector \(x\), the method \(x\text{.segment}(i, n)\) returns the segment of length \(n\) starting at index \(i\), i.e. containing the entries \(x[i], \ldots, x[i+n-1]\). You have both read and write access to the segment, for example you can do

\(x\text{.segment}(i, n) = y\)

where \(y\) is a vector of length \(n\). You can also do \(+=, *=, \text{and everything else.}\)

Similarly, for a matrix \(M\), you can get/set an \(m\text{-by-}n\) block via \(M\text{.block}(i,j, m,n)\). This will be useful for assembling the Jacobians.