CS 322 Project 3: Springies

out:	Thursday 29 March 2007
code due:	**Wednesday 25 April 2007**
writeup due:	**Friday 27 April 2007**

1 Background

Many problems in the real world can be represented as systems of differential equations. A differential equation is one where both variables and the derivatives of those variables can appear. Differential equations can be classed into two broad categories: Ordinary Differential Equations (ODEs) and Partial Differential Equations (PDEs). In an ODE, the only derivatives are with respect to a single independent variable (usually, but not always, time), while in a PDE there can be multiple independent variables and derivatives with respect to those independent variables. In addition, differential equations have an order, which specifies the highest derivative present in the equation; however, there is a mechanism for transforming any second or greater order ODE into an equivalent first order ODE. In this assignment, we will be dealing exclusively with first order ODEs.

First order ODEs are typically written as

$$\dot{\mathbf{u}} = \mathbf{G}(\mathbf{u}, t)$$

where we use the dot notation to signify derivatives with respect to the independent variable (in this case time, t). We can convert higher order ODEs to first order ODEs through the following process: suppose we have a second order ODE of the form $\ddot{\mathbf{u}} = \mathbf{G}(\mathbf{u}, \dot{\mathbf{u}}, t)$. We introduce a new variable $\mathbf{v} = \dot{\mathbf{u}}$. Note that $\dot{\mathbf{v}} = \ddot{\mathbf{u}}$. Thus, we can rewrite our second order ODE as

$$\begin{bmatrix} \dot{\mathbf{u}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{G}(\mathbf{u}, \mathbf{v}, t) \end{bmatrix}$$

Note that this only has first derivatives, and so it is now a first order ODE.

1.1 Particle Systems

One of the most well known ODEs is Newton's Second Law of Motion. This is a second order ODE that relates the force on a particle to the acceleration of the particle (the second derivative of position with respect to time). More formally, on a single particle in one dimension this can be expressed as

$$f(x, \dot{x}, t) = m\ddot{x}$$

where x is the position of the particle, t is the independent variable corresponding to time, f is a function that returns the force on the particle given a position, velocity and current time, m is the mass of the particle, and $\dot{x}$ and $\ddot{x}$ are the first and second derivatives of x with respect to time. This can be expanded to describe the motion of a collection of n particles in 2D motion via the system of equations

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = \mathbf{M}\ddot{\mathbf{x}}$$

where $\mathbf{x}$ is now a $2n$ vector of the positions of all particles, and the masses of the particles are collected in a diagonal square matrix $\mathbf{M}$ that is (not unsuprisingly) called the *mass matrix*. For 2D motion such as in this assignment, $\mathbf{M}$ is a $2n \times 2n$ diagonal matrix where $M_{2i,2i} = M_{2i+1,2i+1} = m_i$, i.e. the mass of particle i.

1.2 Forces

The force function above typically has many components, all summed together. Some common forces are listed below

1.2.1 Gravity

Gravity is the simplest force — it is a constant force in the same direction as some vector (usually down). The magnitude of the force due to gravity is proportional to the mass of a particle, and the constant of proportionality is called g. On earth, $g = 9.8m/s^2$.

1.2.2 Damping

Damping is a velocity-based force that always opposes the current direction of movement. It has a damping parameter k_d which controls how strong the force is. Damping can be either a general viscous damping on velocity (which crudely models movement through fluid) or damping on the actions of another force (in particular spring forces, which will be covered in the next section). For a particle i, viscous damping is usually expressed as $\mathbf{f}_i(\dot{\mathbf{x}}) = -k_d\dot{\mathbf{x}}_i$.

1.2.3 Linear Springs

Linear springs are a commonly used force that attempts to keep pairs of particles (or a particle and a fixed point) a specified distance away from each other. They have a stiffness parameter k_s that controls how strong the force is, a damping parameter k_d, and a rest length r. The force can be separated into two components – the positional component and the damping component. The positional force component for a spring between two particles i and j on particle i is $\mathbf{f}_i = k_s(\|\mathbf{x}_j - \mathbf{x}_i\| - r)\frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|}$, and the force on particle j is $\mathbf{f}_j = -\mathbf{f}_i$. Breaking this equation down, we see that it is operating in the direction of $\mathbf{x}_j - \mathbf{x}_i$ (that is, the vector pointing from $\mathbf{x}_i$ to $\mathbf{x}_j$) that has been normalized to be a unit vector by dividing by the length. The magnitude of the force is proportional to the difference between the current distance between the two points and the desired distance, scaled by the stiffness parameter.

Damping of springs is ideally done only in the direction of the spring. This is accomplished by taking the difference in velocities and taking the dot product with the direction of the spring, applying the result along the direction of the spring. So, if $\mathbf{d} = \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|}$, then the damping force $\mathbf{f}_i = k_d((\dot{\mathbf{x}}_j - \dot{\mathbf{x}}_i) \cdot \mathbf{d})\mathbf{d}$ (and the negation of that for $\mathbf{f}_j$).

1.2.4 Fixed Position Springs

There are a few variations on springs which can be important. One are springs that act between a particle and a fixed point. In this case the spring force is still computed as above, substituting the position of the fixed point in for the position of one of the particles, but there is only a force applied to the particle; no force can (or should) be applied to the fixed point.

Another variation involves spring forces that only act on a single particle in a given direction. For instance, if a particle is inside a wall, a spring force can be used to pull that particle to the surface. This spring force is a force between the particle and the closest point on the wall. Note that this closest point will move over time, and so it is different from a fixed point spring, but it will still result in a force being applied only to the particle. As an example, if a particle is inside a vertical wall, a spring force acting in the horizontal direction towards the outside of the wall can bring the particle to the surface. Even if the particle has motion parallel to the wall, the force will always act horizontally to push it to the surface.

1.3 Integrators

Because there are usually no closed form solutions for most interesting ODEs (including the general particle mechanics ODE), these problems are usually solved using numerical approximation. Solving ODE initial value problems (problems for which the initial conditions are specified) is a well-researched and generally well-understood area and has resulted in a rich variety of *integrators* available. An *integrator* is an algorithm for advancing an ODE from a given state to a new state via advancement of the independent variable. For ODEs where the independent variable is time, this corresponds to stepping the system forward in time by some amount, called the timestep h. Integrators have different properties such as the order (which is a measure of how accurate the answer is in terms of the size of h) and stability (which is a measure of what timesteps are safe to take for a given system without causing the solution to diverge).

Some common explicit integrators are Forward Euler (Moler p. 191), Midpoint (Moler p. 192), 4th Order Runge-Kutta (used by the MATLAB command `ode45`), and Backwards Euler.

1.4 Constraints

In some cases we may not want points to move at all over time, regardless of what forces are being applied to them. In the framework, these points are indicated by having negative mass. For explicit solvers like the ones you will be implementing, it is usually sufficient to set the derivatives of position and velocity of the constrained points to be zero.

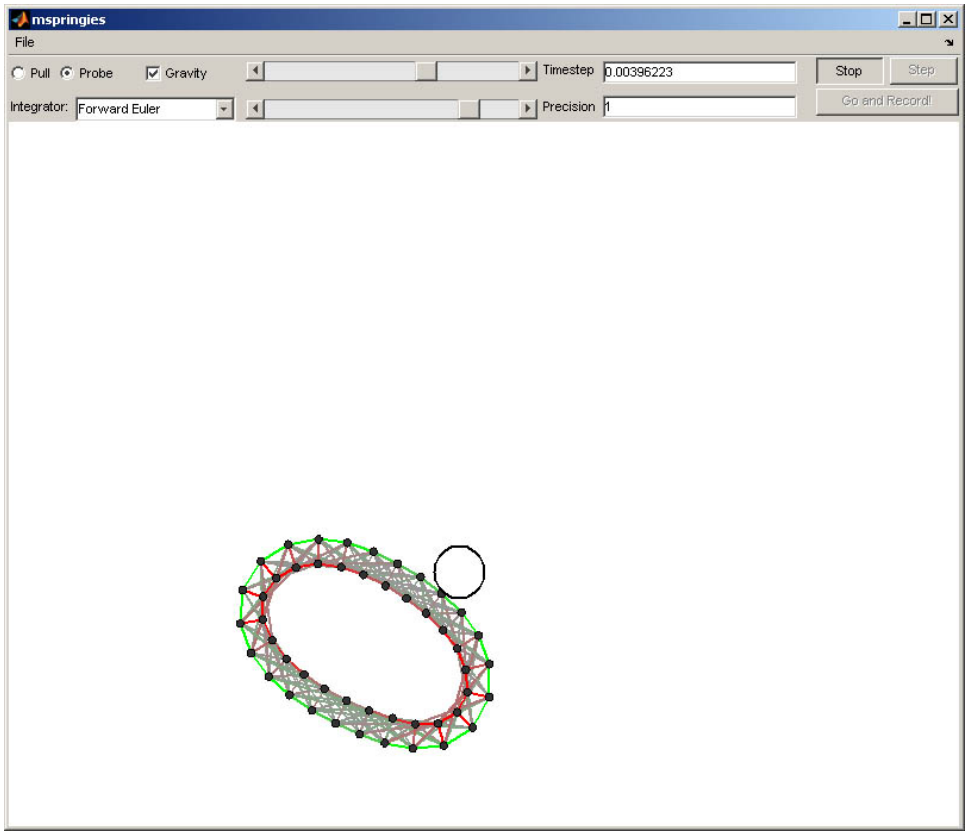

Figure 1: A screenshot of the program in action

2 Assignment

In this assignment you will build a simple mass-spring simulator that allows the user to load scene descriptions from a file, select different integrators, and interact with the simulation using both "probe" (a circular wall centered at the current mouse point) and "pull" (a spring between a given particle and the current mouse point) tools. It also models gravity, viscous damping, spring, and wall collision forces with the two horizontal and two vertical walls.

The program can be started up with the command `mspringies()`. The GUI allows for different forces to be turned on and off, different integrators to be selected, and various scene definition files (specified in the xspringies format) to be loaded. It also allows for the results of the simulation to be saved to disk and loaded in later for data analysis.

You are responsible for writing a function `stepScene` that takes the current state $\mathbf{u}$, the current scene structure `scene`, the desired timestep h, the maximum time t_s, the error precision, and the integrator to use. It uses the specified integrator to advance the ODE from the current state to its value at t_s, taking steps of at most h in size for fixed-step integrators and keeping the error under the precision for adaptive solvers.

As part of writing this assignment, you should also write another function that takes the state and current scene description and evaluates the ODE function $\mathbf{G}(\mathbf{u})$ for the first order ODE corresponding to the equations of motion for the system. You will call this function from your integrators in order to advance the system.

In your writeup you are also responsible for providing answers to the questions asked in the Questions section below.

3 Guidance

3.1 Getting Started

There are many ways to order the work involved in this assignment. One reasonable way of starting would be to use the following order:

1. Write the function to evaluate the ODE for the equations of motion, at first with just gravity forces
2. Write a forward Euler integrator in `stepScene`, which allows you to test your ODE function.
3. Add viscous damping forces to your ODE function
4. Add wall forces to your ODE function
5. Add spring forces to your ODE function
6. Add probe / pull forces to your ODE function
7. Write a midpoint integrator, `ode45` integrator, and `ode15s` integrator
8. Write an adaptive forward Euler integrator

3.2 The Scene

The scene that is passed to `stepScene` is specified in a MATLAB structure variable. A struct is an object that has fields, where each field has a name and a value stored in that field (which can itself be a struct, and so on). Struct fields are accessed using the familiar dot notation; for instance, given a struct `scene` with a field `Gravity` that is itself a struct with fields `Enabled` and `Direction`, values can be accessed as `scene.Gravity.Direction`, etc. The scene passed in has many fields describing the properties of the scene, as specified below:

Field name	Type	Description
`Gravity`	struct	Structure containing fields corresponding to the force of gravity. `Enabled` is set to 1 if gravity is on, 0 if gravity is off. `Direction` is a 1×2 vector containing the direction and magnitude of the acceleration due to gravity. By default this is $[0, -9.8]$, but it can be changed in the scene file
`Viscosity`	number	The coefficient of viscous damping in the scene
`TopWall`	0 or 1	A boolean flag specifying whether the top wall is on (1) or off (0)
`LeftWall`	0 or 1	A boolean flag specifying whether the left wall is on (1) or off (0)
`RightWall`	0 or 1	A boolean flag specifying whether the right wall is on (1) or off (0)
`BottomWall`	0 or 1	A boolean flag specifying whether the bottom wall is on (1) or off (0)
`Mass`	$n \times 1$ vector	A vector containing the mass of particle i
`Springs`	$n \times 5$ matrix	A matrix containing the spring connections. Each row is one spring. The first column contains the index of the first particle, the second column contains the index of the second particle, the third column contains the stiffness (k_s), the fourth contains the damping coefficient (k_d), and the fifth contains the rest length of the spring.
`WallStiffness`	number	Stiffness of the spring used to handle collisions with the walls and probe
`WallDamping`	number	Damping coefficient of the spring used to handle collisions with the walls and probe
`Probe`	struct	Structure containing fields corresponding to the probe. `Enabled` is set to 1 if the probe is on and 0 otherwise. `Position` is a 1×2 vector containing the current position of the probe, and `Radius` is the radius of the probe.
`PullForce`	struct	Structure containing fields corresponding to the pull force. `Enabled` is set to 1 if the pull force is on and 0 otherwise. `Position` is a 1×2 vector containing the current position of the mouse. `Particle` is the index of the particle that is being pulled. `RestLength` is the rest length of the spring, and `Stiffness` and `Damping` are the stiffness and damping coefficients of the spring, respectively

3.3 The State

The current state of the system as it is passed to `stepScene` is as a $n \times 4$ matrix, where each row corresponds to a particle, with the first two entries being the (x, y) coordinates of its position and the second two entries being the (x, y) coordinates of its velocity vector.

3.4 Evaluating the ODE function

The ODE function you write should take the current state and the scene description as input and produce the value of the ODE function $\mathbf{G}(\mathbf{u}, t)$ for the first order ODE corresponding to the equations of motion of the scene. Because our forces are independent of the current time t, it is not necessary to have t be an argument to the function. The function should return the appropriate derivatives for each component in the state.

`ode45` and `ode15s`, however, expect that the state and the value of the ODE function are column vectors. Because of this, you will have to write your ODE function to accept and return state vectors, rather than state matrices.

3.4.1 Wall "Collisions" and Probes

When a particle passes through a wall, we will model it as a linear spring acting in the direction perpendicular to the wall and with a magnitude to push the particle back outside. Damping should be applied as well to the particle, in the same manner as in the linear springs section.

The left and right walls are at $x = 0$ and $x = 750$, while the bottom and top walls are at $y = 0$ and $y = 550$, respectively.

The probe is an interactive tool that is represented as a circular wall at the current mouse point. Collisions with the probe should be treated in the same fashion as collisions with a wall – a linear spring force should be applied in the direction from the particle to the closest point on the surface.

3.4.2 Pull Force

The pull force is an interactive force that is represented as a spring of a set length between a specified particle and the mouse point. Again, it should be modeled as another linear spring in your code, again involving only one point.

3.5 `stepScene`

`stepScene` is responsible for advancing the system forward in time by some amount `timeToStepTo`, using the specified integrator. The integrators are as follows:

1. (`mode = 1`) Fixed-step size Forward Euler (see Moler p. 191)
2. (`mode = 2`) Fixed-step size Midpoint (see Moler p. 192)
3. (`mode = 3`) Adaptive step size Forward Euler
4. (`mode = 4`) MATLAB's adaptive ODE solver `ode45`
5. (`mode = 5`) MATLAB's adaptive ODE solver `ode15s`

For the fixed stepsize integrators, the integrator should advance the system forward according to the integrator by stepsizes of at most `timestep`. Note that this may require a smaller step at the end to step exactly to `timeToStepTo`.

3.6 Adaptive Forward Euler

For adaptive size Forward Euler, you should start with a current stepsize `currTS` of `timestep`. Until you have reached the max time `timeToStepTo`, you should repeat the following loop

```
currTime = 0;
successes = 0;
while (currTime < timeToStepTo)
  y_1 = forward_euler_step(state) of stepsize currTS;
  y_2 = midpoint_step(state) of stepsize currTS;
  err = y_2 - y_1;
  if (max(err) > precision)
    currTS = currTS / 2;
    successes = 0;
  else
    state = y_1;
    successes = successes+1;
    currTime = currTime + currTs;
    if (successes > 3)
      currTs = currTs*2;
    end
    if (currTime + currTs > timeToStepTo)
      currTs = timeToStepTo - currTime;
    end
end
```

What this loop does is compute both the Forward Euler and Midpoint steps and then checks the error between them. If they differ by more than the specified precision, it halves the timestep and tries again. Otherwise, it uses the result of the Forward Euler computation and advances forward. If it is able to advance forward three times without any error, it will try to double the timestep. Note that we use the result of the Forward Euler computation instead of the presumably more accurate Midpoint evaluation. There is typically some debate about whether to use the lower or higher order approximation, with advocates of the former pointing out that the error analysis is typically only valid for the lower order answer. In any case, you should feel free to experiment with using the results of either one in your computations.

3.7 `ode45` and `ode15s`

Both `ode45` and `ode15s` take as an argument a *function handle* to the ODE function to evaluate. This function should be of the form $\mathbf{G}(t, \mathbf{u})$ (note the order of arguments, and the presence of the time variable t). A function handle in MATLAB is a variable that contains a function, and can be called just like any other function. In order to allow your ODE function to be evaluated by one of these two methods, it should be written to take and return the state and derivative of the state as vectors, and not matrices. If your ODE function is called `evalODEFunction`, for instance, in `stepScene` you can use

```
opts = odeset(...)
```

```
ode45(@(t, x) evalODEFunction(x, scene), [0 timeToStepTo], myState, opts)
```

The `@(t, x)` syntax creates an *anonymous function* and passes its handle to `ode45`. This anonymous function takes the passed in state `x` and passes it to our ODE function, along with the scene variable from `stepScene`. We are allowed to do this because our anonymous function can reference variables that are in the scope of `stepScene`. Note also that we disregard the `t` parameter as it is not needed.

You should use `odeset` to set the precision for the adaptive algorithms accordingly. One idea is to use the precision field as the Relative Error, and set the Absolute Error to be relatively low (say 0.01 to 0.001).

4 Questions

1. Run `pend.xsp` with both Forward Euler and Midpoint. What range of timesteps result in expected behavior for each integrator? Describe what happens when things go bad. What happens as the timesteps get close to the boundary between good behavior and bad behavior? Give a rough bound on the stability region of each integrator on this problem.

2. What error does Forward Euler and Midpoint produce? Run each integrator on `single_pendulum.xsp` with a fixed timestep for several choices of timestep and record the results, then compare the error from `ode45` with a small error precision. What can you conclude from your plots about the order of accuracy of both Forward Euler and Midpoint?

   Note that the recorder generates a MAT file that contains two fields, `PointData`, which is a $n \times 4 \times k$ 3D array where `PointData(:, :, i)` is the state matrix at the i-th step, and `Time`, which is a $k \times 1$ vector that contains the current time at the i-th step. Also note that the recorder only records every other frame for space considerations.

5 Submission and FAQ

Submission will be through CMS. A FAQ page will be kept on the course website detailing any new questions and their answers brought to the attention of the course staff. Your submission should include all MATLAB code you used. Your code should be well-commented and vectorized as much as possible.

Your writeup should discuss the answers to the questions raised in the assignment, as well as general observations of stability and speed of the various integrators applied to various scenes.

6 Starred Problems

1. Implement Backwards Euler as another integrator. See the attached documentation for more details.