Physically Based Modeling
CS 15-863 Spring 1997
Assignment 1: Differential Equations
Due Thursday Jan 30

The purpose of this assignment is to introduce you to the idea of differential equations as live, squirmy things you can interact with, and to explore some basic properties of numerical solutions. An ulterior motive is to let you make sure you can draw on the screen, read the mouse, etc.

The idea is to implement a 2-D animated particle $\mathbf{x}$ with position $\mathbf{p} = (p_x, p_y)$ whose motion $(\dot{p}_x, \dot{p}_y)$ is governed by one or another differential equation, using Euler's method to solve the equation. You'll animate the particle by updating the display after each timestep.

First, try an "integral curve" mode: each time the mouse is clicked, take the mouse position (M_x, M_y) as an initial value for (p_x, p_y). Then, as you march forward through time, draw a line from the previous position to the new one, so that the particle leaves a trail. Start a new curve each time the mouse is clicked. Don't bother erasing—that way you'll be able to compare bunches of curves. Try the equations $\dot{p}_x = -kp_y$, $\dot{p}_y = kp_x$, where k is a constant, at a variety of step sizes. Also try $\dot{p}_x = -kp_x$, $\dot{p}_y = -kp_y$, also fiddling with the step nzsize. Blends of these two functions (i.e. $\dot{p}_x = -k_1 p_y - k_2 p_x$ and $\dot{p}_y = k_1 p_x - k_2 p_y$ for various values of k_1 and k_2) are also interesting. In vector notation, we'd write this last as

$$\mathbf{x}(t) = \begin{pmatrix} p_x(t) \\ p_y(t) \end{pmatrix}$$

and

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), t) = \begin{pmatrix} -k_1 p_y(t) - k_2 p_x(t) \\ k_1 p_x(t) - k_2 p_y(t) \end{pmatrix}.$$

Next, switch to "movie" mode: draw a spot at the current position, after erasing the old spot. Implement a "mouse spring," as follows: if the mouse position is (M_x, M_y), add a mouse term to the differential equation of the form $\dot{p}_x = k_m(M_x - p_x)$, $\dot{p}_y = k_m(M_y - p_y)$ with $k_m > 0$ which will pull the particle toward the mouse. If possible, use a mouse button to control the spring, enabling it only when the button is down. (You can simplify your code by just making k_m zero whenever the mouse button is *not* pressed. Also, it helps to draw a line between the mouse and the particle when the spring is enabled.) What happens if you make the mouse a repulsor, by making k_m negative? (Can you think of a better function for $\dot{p}_x$ and $\dot{p}_y$ to implement a repulsor?) Try the mouse spring by itself, with the equations from the previous exercise, and with the equation $\dot{p}_x = 0$ and $\dot{p}_y = -kp_y$.

Once you get all that working, you can make the particle more interesting by moving it into a "second-order" dynamics world. So far, you've been computing the velocity of the particle directly, by choosing $\dot{\mathbf{p}} = f(\mathbf{p}, t)$. Now we'd like the particle to behave according to the rule $\ddot{\mathbf{p}}(t) = \mathbf{F}(t)$, where $\mathbf{F}(t) = (F_x(t), F_y(t))$ is the total force acting on our particle at time t. (We're assuming the particle has unit mass.) We'd like to continue to work only with first-order ODE's, so the "state" of the particle has to be enlarged. We now let $\mathbf{x}$ be a vector (p_x, p_y, v_x, v_y) where $(v_x, v_y) = \mathbf{v} = \dot{\mathbf{p}}$

is the particle's velocity. This leads us to the first-order "coupled" equation

$$\frac{d}{dt}\mathbf{x} = \frac{d}{dt}\begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ F_x \\ F_y \end{pmatrix}.$$

Numerically, solving this equation is as easy solving any other ODE. What's different is that the initial conditions now require four parameters to be fully specified, not two: in simulating a Newtonian particle, you must supply the initial position (p_x, p_y) of the particle along with the velocity (v_x, v_y).

To make things simple, you can let the particle have zero velocity initially. Go back and do integral curve mode again. Try the force function $\mathbf{F} = -\mathbf{p} = -(p_x, -p_y)$ again. Try this with *damping*: use $\mathbf{F} = -\mathbf{p} + k_d\mathbf{v}$ and play around with both positive and negative values for k_d.

Now try movie mode. You can implement a mouse spring the same as you did before: let the "mouse force" $\mathbf{F}_M$ be computed by $\mathbf{F}_M = k_m(M_x - p_x, M_y - p_y)$ with $k_m > 0$. In order for your simulation not to get out of control, you'll want to add a damping force $\mathbf{F}_d$ of the form $\mathbf{F}_d = -k_d\mathbf{v}$ where $k_d > 0$, so that the total force $\mathbf{F}$ is $\mathbf{F} = \mathbf{F}_M + \mathbf{F}_D$. For even more fun, combine the functions from integral-curve mode with movie mode e.g. let the particle experience a force $\mathbf{F} = -\mathbf{p}$ in addition to the mouse-spring force.

Another thing that's interesting to do with second-order behavior (but pretty boring in first-order mode) is to simulate some orbital mechanics. Given your single moving particle, let it be attracted to some location $\mathbf{s}$ on the screen with force

$$\mathbf{F} = -k\frac{\mathbf{p}-\mathbf{s}}{\|\mathbf{p}-\mathbf{s}\|}\frac{1}{\|\mathbf{p}-\mathbf{s}\|^2} = \frac{-k(\mathbf{p}-\mathbf{s})}{\|\mathbf{p}-\mathbf{s}\|^3}.$$

See if you can give the particle an initial velocity so that it has a stable orbit. (You might want to draw some indication of where $\mathbf{s}$ is.) To make it interesting, define several source points $\mathbf{s}_1$, $\mathbf{s}_2$, and sum the force exerted on the particles by each source, to find a total net force. You should be able to come up with a configuration of points, values of k, and initial conditions so that the particle's journey traces out an interesting path on the screen. For the ambitious, find a non-symmetric arrangement of 3 or 4 source points that results in a stable orbit (an orbit where the particle doesn't decay into the center of a source, resulting in a numerical explosion, nor is the particle flung off the screen, never to return). Note that stability may be difficult if not impossible using Euler's method for your integeration.

Programming hints: for modularity (later assignments!), try to structure your program as follows. Encode the function $f(\mathbf{x}(t), t)$ as a routine `dxdt` which, in C, might be specified as

```
void dxdt(double t, double x[], double xdot[])
{
        /* Extract data from the state array x[] to get  x(t) */
        ...

        /* Compute  ẋ(t) */
        ...
```

```
        /* Stuff  ẋ(t) into the array xdot[] */
}
```

This way, the method you use to solve the ODE can be completely decoupled from the ODE you're actually trying to solve. Write your Euler method so that it calls the function `dxdt`. If you feel ambitious, modify your program to use the midpoint method instead of Euler: if you've coded up `dxdt` as suggested, the change to the midpoint method shouldn't require more than about a minute or two of programming.