CS 4: Lecture 7
Wednesday, February 8, 2006

PHYSICAL SIMULATIONS

In lab, we've seen how to compute trajectories in which a projectile undergoes a constant acceleration, using the equation x = x0 + v0 t + a t^2 / 2. But what if the acceleration is not constant? The equation doesn't work any more.

If we know the velocity or acceleration as a function of time, there are two ways we might be able to determine the position of an object at a given time.
- An _analytical_solution_: find an explicit expression for the position as a function of time, like x(t) = x0 + v0 t + a t^2 / 2. Unfortunately, this is not always possible.
- A _numerical_solution_: find an _approximate_ solution based on a computer simulation. Unfortunately, this doesn't give 100% accuracy, but we can often get very close. This is what we'll look at today.

Creating a computer simulation takes several steps.
- Physics: understand how the quantities are related.
- Discretization: the position of an object is a continuous function of time, but computers can only do a finite number of computations, so we're going to look at a discrete (finite) set of "points in time." We need to figure out an _algorithm_ to do this.
- Simulation: write and run code that calculates the discretized values. Understand the output.

Let's look at these steps.

Physics

Suppose you drive a car whose velocity at time t is v(t) = sin t miles/minute, where t is measured in minutes, and the sine is taken over degrees (not radians). You start at time t = 0 with a velocity of v(0) = 0, and stop when your velocity peaks at 1 mile/minute after 90 minutes. How far have you driven?

Let x(t) express how far you've driven at time t. Remember that velocity is the derivative of distance with respect to time, so

$$\frac{dx(t)}{dt} = v(t) = \sin t^{o} . \qquad \text{<-- A reminder that t is in degrees, not radians.}$$

This is called a _differential_equation_. To solve it, you also need to know the _initial_condition_ x(0) = 0. (If you started driving at x(0) = 10, you'd obviously end up in a different place than if you start driving at x(0) = 0.)

The analytical way to solve this differential equation is to integrate, giving

$$x(t) = \int_0^t \sin \frac{\pi}{180} y^{o} \, dy. \qquad \text{<-- Here we convert y to radians so you can integrate the sine function.}$$

However, there are many differential equations that arise in practice that nobody knows how to integrate or solve. For example, if you want to account for the effects of wind resistance when you simulate slingshotting a ball along a trajectory, there is no known analytical solution. If you want to simulate three planets orbiting each other under gravitational forces, there is no known analytical solution. (There's a solution for two planets--they revolve each other in elliptical orbits--but not for three.) So let's look at how to approximate the integral numerically.

Discretization

Here's the idea. Suppose we know the position x(t) at time t, and we want to know the position x(t + T) at time t + T, where T is a small positive number. Suppose that the velocity is a constant, v, during this period of time. Then

$$x(t + T) = x(t) + v T.$$

Unfortunately, in our problem, v(t) is not constant! But if we choose T to be really small, then v(t) is _close_ to constant during that period of time. So although we won't get x(t + T) exactly right, we'll be close. And the smaller we make T, the less v(t) will vary during that time, so the closer we'll be. If we could make T as small as the infinitesimals used in calculus, we'd get exactly the right answer. Computers can't do infinitesimals, but they can make T pretty small if you're willing to wait long enough.

Here's a mathematical justification for this approximation. If we express x(t + T) as a Taylor series, expanded around t, we have

$$x(t + T) = x(t) + \frac{dx(t)}{dt} T + \frac{1}{2} \frac{d^2 x(t)}{dt^2} T^2 + \frac{1}{6} \frac{d^3 x(t)}{dt^3} T^3 + \ldots$$

If T is small enough, then T^2 is really small, and T^3 is absolutely tiny. The smaller you make T, the more the first two terms dominate all the rest. So if T is small enough, we can ignore the terms involving T^2, T^3, etc., giving the approximation

```
  x(t + T) ˜ x(t) + v(t) T.
```

Our strategy is to approximate a sequence of values of x(t), each at a different time t.

```
  x(0)  = 0.
  x(T)  ˜ x(0)  + v(0)  T.         <- The "˜" means "is approximately equal"
  x(2T) ˜ x(T)  + v(T)  T.            and is actually written as two squiggles,
  x(3T) ˜ x(2T) + v(2T) T.            one over the other, when I'm not stuck
  x(4T) ˜ x(3T) + v(3T) T.            with ASCII.
```
... and so on, until we get to the end of the 90-minute trip. To compute the distance x(90), repeat 90/T times.

This is called a _discretization_ of the original integral we wanted to solve, because we compute a finite number of discrete _timesteps_ of duration T, instead of an infinite number of infinitesimal timesteps. This particular discretization is called _Euler's_method_, or alternatively _forward_differencing_. We'll see a more accurate discretization soon.

This strategy is also called _numerical_integration_, because we are approximating the integral numerically. We are approximating the area under the sine curve by adding the areas of a bunch of tall, thin rectangles (one for each timestep).

```
         ____
       __||||
     __||||||
    _||||||||           Each rectangle has width T and height v(t), where t is
   _|||||||||           the time at the left edge of the rectangle.
  _||||||||||
```

Of course, the flat tops of the rectangles don't match the curve of the sine, so our approximation of x(t) is not perfectly accurate. But we can improve the accuracy by making T smaller. However, we have to do 90/T timesteps, so better accuracy means a longer computation time.

Simulation

Now we can write code to compute the length of our drive.

```
  public static void main(String[] args) {
    double duration = 90.0;                           // Duration of simulation
    long timesteps = 10;                              // Number of timesteps
    double steptime = duration / (double) timesteps;  // Time for one timestep

    double distance = 0.0;                            // Distance traveled
    long step = 0;                                    // Current step number
    while (step < timesteps) {
      double time = steptime * (double) step;
      /* We must convert ‘time’ from degrees to radians to use Math.sin */
      double velocity = Math.sin(Math.PI / 180.0 * time);
      distance = distance + velocity * steptime;
      step++;
    }
    System.out.println("You drove " + distance + " miles.");
  }
```

Each iteration of the loop performs one timestep of the simulation.

Let’s look at the output of this program...for a bunch of different choice of the variable ‘timesteps’.

# of timesteps	Miles driven (output)
1	0.
2	31.
5	47.8
10	52.6
100	56.84
1,000	57.251
10,000	57.2912
100,000	57.29533

As the number of timesteps increases, the solutions _converge_--you can see they’re getting closer to the same answer. In this case, we can evaluate the integral exactly, so we know the exact answer is about 57.295779. Not only is the simulation converging--it’s converging to the right answer. (Unfortunately, there are many problems in scientific computing where it’s quite hard to devise a numerical method that converges to the right answer.)

A Better Discretization

Instead of using Euler’s method, let’s use a method called the _trapezoid_rule_ to integrate v(t). Instead of approximating the integral with a bunch of rectangles, we’ll use a bunch of tall thin trapezoids whose tops are tilted. Both endpoints of each trapezoid top lie on the sine curve.

```
          _-/
       _-/|||
     _/||||||
    /||||||||
   /|||||||||
  /||||||||||
```

Mathematically, this means that during the timestep that takes us from t to t + T, we don’t assume that the velocity is v(t) over the whole interval. Instead, we assume the velocity is the _average_ of v(t) and v(t + T).

$$x(t + T) \approx x(t) + \frac{v(t) + v(t + T)}{2} T.$$

We can implement this method by changing just one line of code.

```
      double velocity = 0.5 * Math.sin(Math.PI / 180.0 * time) +
            0.5 * Math.sin(Math.PI / 180.0 * (time + steptime));
```

Let’s look at the output using our improved program.

# of timesteps	Euler’s method	Trapezoid rule
1	0.	45.
2	31.	54.3
5	47.8	56.82
10	52.6	57.18
100	56.84	57.2946
1,000	57.251	57.295768
10,000	57.2912	57.29577939
100,000	57.29533	57.2957795119

The exact answer is about 57.2957795131. Observe that the trapezoid rule converges to the correct solution a _lot_ faster than Euler’s method! In 1,000 timesteps, the trapezoid rule is already more accurate than Euler’s method after 100,000 timesteps.

Postscript: Why the Trapezoid Rule Works (not examinable)

Why is the trapezoid rule so much more accurate? Expand x’(t + T) around t.

$$\frac{dx(t + T)}{dt} = \frac{dx(t)}{dt} + \frac{d^2 x(t)}{dt^2} T + \frac{1}{2} \frac{d^3 x(t)}{dt^3} T^2 + \ldots$$

Take the Taylor series for x(t + T), add T/2 times the left-hand side of this equation, and subtract T/2 times the right-hand side.

$$x(t + T) = x(t) + \frac{1}{2} \frac{dx(t)}{dt} T + \frac{1}{2} \frac{dx(t + T)}{dt} T - \frac{1}{12} \frac{d^3 x(t)}{dt^3} T^3 + \ldots$$

Ta-daa! The T^2 terms magically cancelled each other out. The trapezoid method uses the first three terms, so its error is proportional to T^3. If T is small, then T^3 is tiny--smaller than the T^2 error of Euler’s method.