CS667 **Lecture 12: Numerical Ordinary Differential Equations** 3 March 2005

Jeff Budsberg Lecturer: Steve Marschner

1 Introduction

As we get into more specialized volume light transport computations, and also into simulations of physical systems, we end up with differential equations. Unfortunately, we cannot solve these as simply as we were able to in the derivation of the volume rendering equation. This gets us off the familiar turf of integration by point sampling and monte carlo methods, and leads us to a whole new toolbox of methods for solving differential equations.

A soild introduction to these topics is about $1\frac{1}{2}$ courses worth of material, so obviously this lecture will only be a brief overview of differential equations and the culture that goes with the methods for solving them. Generally, the more interesting problems need numerical solutions (not analytical), and for actual problems there is more learning to do about particular methods that we are unable to cover in such a brief overview.

ODE: ordinary differential equation
An ODE is an equality involving a function and its derivatives, where the unknown function is a function of *one variable*. An ODE of order n is an equation of the form

$$f(x, y(x), y'(x), y''(x), \cdots, y^n(x)) = 0$$

where you have a single independent variable x, a vector-valued function, f, and its derivatives. Problems involving ordinary differential equations can always be reduced to the study of sets of *first-order* differential equations. For example, the equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

can be written as two first-order equations:

$$\frac{dy}{dx} = z(x)$$
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

where z is a new variable. The usual new choice for the new variables is to let them just be derivatives of each other (and the original variable). This exemplifies the procedure for an arbitrary ODE. Thus, the generic problem for an ordinary differential equation is reduced to a set of n coupled *first-order* differential equations for the functions $y_i, i = 1, 2, ..., N$, having the form $\frac{dy_i(x)}{dx} = f_i(x, y_1, \cdots, y_N), i = 1, \cdots, N$. This problem is described here:

Canonical problem:

$$y'(x) = f(x, y(x)), \text{ where } y \text{ is a vector and } x \text{ is a scalar}$$

or

$$\left.\begin{aligned} y_1'(x) &= f_1(x, y_1(x), y_2(x), \cdots, y_n(x)) \\ &\vdots \\ y_n'(x) &= f_n(x, y_1(x), y_2(x), \cdots, y_n(x)) \end{aligned}\right\} \text{a system of coupled ODEs}$$

PDE: partial differential equation
PDEs are similar to ODEs (as they are also an equality involving a function and its derivatives), although the unknown function is a function of *more than one variable.* A PDE is an equation of the form

$$f(x, u, \nabla u, \nabla\nabla u, \cdots) = 0.$$

Note this just indicates dependence on all the partial derivatvies of u; $\nabla\nabla u$ is the partial derivative with respect to one variable and then another (We use this notation due to lack of consistent notation of a derivative with respect to multiple variables).

So really, *an ODE is just a PDE only one independent variable*, but the methods to evaluate them tend to look different (for the most part). Hence, we tend to think of them as two different classes of problems, since the methods for one do not necessarily work for the other. We will focus more on PDEs in a future lecture.

2 ODEs

The first thing you need to know about ODEs is that we can always make them into first-order ODEs (which we saw before). Here is our example from above in a different notation:

$$y''(x) + g(x)y'(x) - r(x) \Longrightarrow \begin{array}{l} y'(x) = z(x) \\ z'(x) = r(x) - g(x)z(x) \end{array}$$

For each order of derivative you eliminate, you introduce one more unknown function.

The second thing to remember with ODEs is that the problem specification is incomplete without knowledge of the nature of the problem's *boundary conditions.* Boundary conditions can be as simple as discrete numerical values or as complex as nonlinear equations, but are not maintained outside of thier specified constraints. Typically, the numerical methods used in solving ODEs are directed by the nature of the boundary conditions and you usually need to constrain the solution with one boundary condition per unknown. The two catergories of boundary conditions are illustrated below.

3 Initial Value Problem

If all of the boundary conditions are specified at the starting time, it's an *initial value problem.* An example of such a problem arises when trying to describe a ball's vertical movement over time as it is tossed in the air. This simple system with constant acceleration is depicted below. Here, one must specify the ball's starting position and velocity to have a solution for $y_1' = -g$ and $y_2' = y_1$.

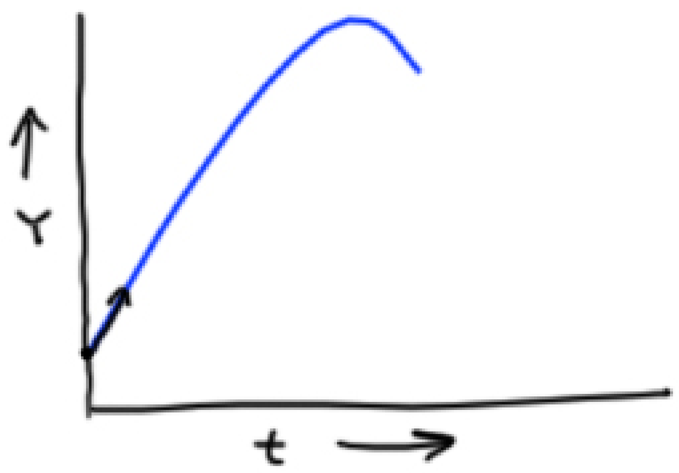

Figure 1: Vertically thrown ball example

Another typical example of an intial value problem is the mass on a spring, where you again need the initial position and velocity. One can then put the system in motion to find a solution for $y_1' = -ky_2$ and $y_2' = y_1$, which oscilaltes around the zero point.

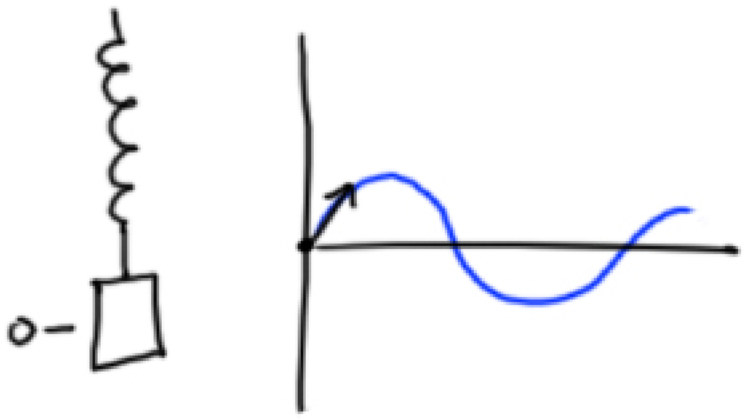

Figure 2: Harmonic oscilator example

4 Boundary Value Problem

A different kind of problem arises if the boundary conditions are specified at more than one x. For instance, say we want to throw a ball in a waste basket—this involves getting the ball to be at a particular point at a given time. Another way of phrasing this is that you know where you are throwing from and the final position, but don't care about the velocity. Explicitly, in this case the boundary conditions are $y(0) = 0$ and $y(1) = k$.

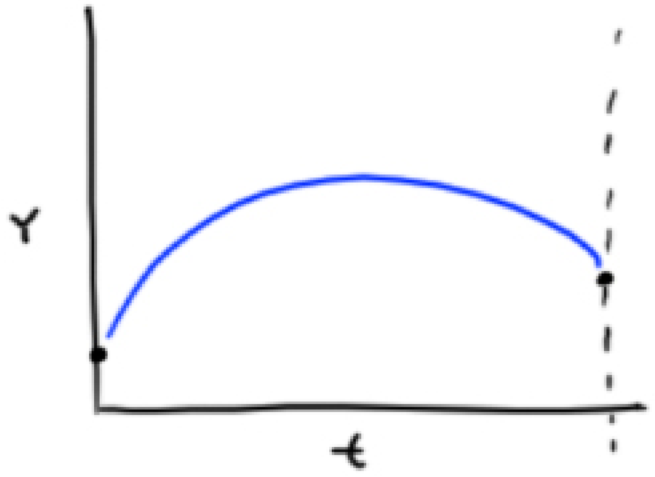

Figure 3: Tossing a ball in a trashcan example

Even though the equations are exactly the same, the solution methods will look different because of the different boundary conditions. A similar situation happens in the following case of a hanging chain in a static state. The boundary conditions are the two heights of the anchor points of the chain.

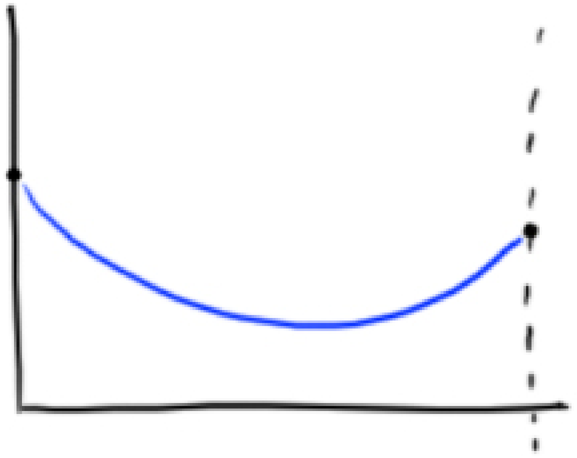

Figure 4: Static hanging chain example

5 Solution Methods

The simplest possible method for ODEs is to assume the slope is constant from the current known point x_1 to the next point at $x_2 = x_1 + h$ (where h is the step size). Then, $y_{n+1} = y_n + hf(x_n, y_n)$ This is a very old method known as *Euler's method (Forward Euler)*. It is super-cheap, since it depends only on the first function point, but is not very good. It does get used, for example, for particle systems in games where you want to spend very little time, but really might not care much about accuracy.

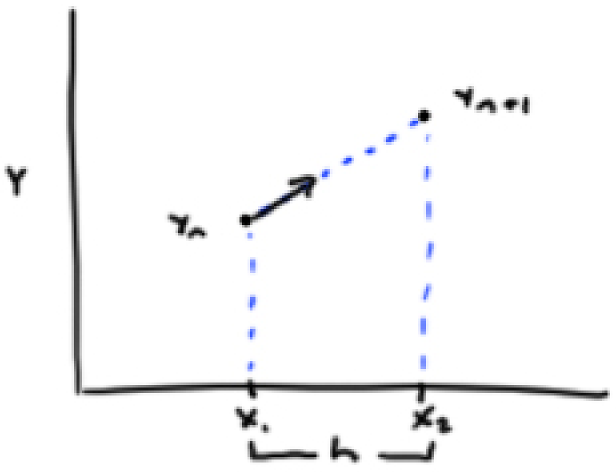

Figure 5: Forward Euler method

So, what is the difference betwen y_{n+1} and the real y_{n+1}? Let's look at the accuracy of this method. The standard approach to do this is to expand in Taylor series and compare.

$$y(x_n + \Delta) = y(x_n) + \Delta y'(x_n) + \frac{\Delta}{2^2} y''(x_n) + O(\Delta^3)$$
$$y_{n+1} - y(x_n + h) = \frac{h}{2^2} y''(x_n) + O(\Delta^3) = O(h^2)$$

The error is of order h^2 and this makes it a *first order accurate* method (the method cancels out errors up to order 1). We can think of this error intuitively, that the method assumes the function is straight, which will systematically wander towards the outside of the curve. In the figure below you notice that the approximations in black wander to the outside of the real solution in blue.

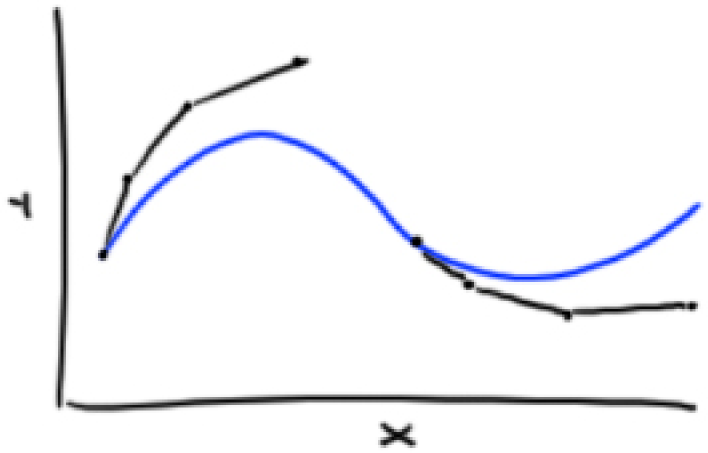

Figure 6: Forward Euler error

Hence, Forward Euler can easily be unstable and we often need to use a very small time step to get a decent answer after many iterations. Thus, this implementation is not recommended for any practical use. However, to get a higher order of accuracy, let's expect the direction to change. Since the derivative at x_1 is not representative of the average; instead try the derivative at the halfway point. This is called the *Midpoint method (second order Runge-Kutta)*:

$$k_1 = hf(x_n, y_n)$$
$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2})$$
$$y_n + 1 = y_n + k_2 + O(k^3)$$

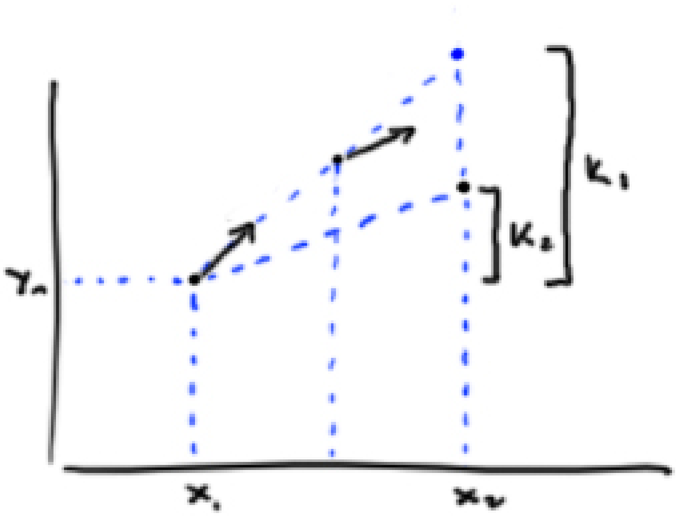

Figure 7: Midpoint method

It turns out that doing things this way introduces a symmetry that cancels the h^2 term, making this a *second order accurate* method.

briefly, $y_n + \frac{k_1}{2}$ is a first-order approximation of $y(x_n, \frac{h}{2})$
$f(x_n + h/2, y_n + \frac{k_1}{2})$ is a first-order approximation of $y'(x_n + \frac{h}{2})$
k_2 is a second-order approximation of $y(x_1 + h) - y(x_1)$

One can keep doing this, cancelling more terms, leading to a family of Runge-Kutta methods that work on this same principle: make k extra function evaluations to increase the order by k. Runge-Kutta is a simple and robust model and is a good general candidate (for well-behaved problems), but is usually not the fastest method.

There are *many other methods*; all of which have their own costs and benifits.

Note: *An adaptive stepsize is also very important*, but beyond the scope of this lecture. You want to estimate that error term (at some additional cost) and use some heuristics to adjust the step size so that the error is low enough but not unnessesarily low.

6 Stability

Consider the linear example with constant coefficients:

$$\begin{aligned} y' &= -cy \\ f(x,y) &= -cy \end{aligned}$$

This should lead to nice exponential decay. If we assume that $f(x, y)$ varies slowly, this is a reasonable model for local behavior, even for nonlinear equations or non-constant coefficients. Let's solve using Euler integration, which leads us to:

$$\begin{aligned} y_n + 1 &= y_n + h(-cy_n) \\ &= (1 - ch)y_n \end{aligned}$$

What are the behaviors we get out of this? It's easy to see, since $y_n = (1 - ch)^n y_0$.
case 1: if $0 < 1 - ch < 1 \rightarrow$ decay; where the higher the timestep, the higher the error
case 2: if $-1 < 1 - ch < 0 \rightarrow$ oscilates and decays
case 3: if $1 - ch < -1 \rightarrow$ oscilates and diverges

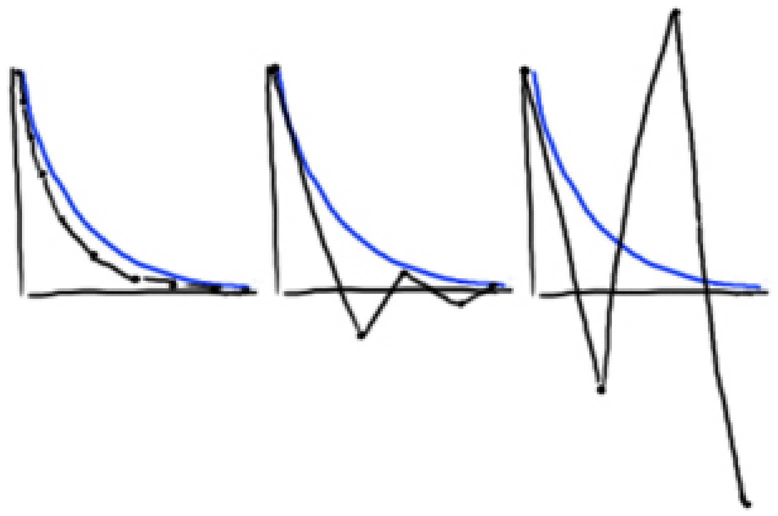

Figure 8: Forward Euler stability; cases 1-3 correspond from left to right

This method is unstable overall, but we classify it as *conditionally stable*, where the condition is $h < \frac{2}{c}$. In real solution, c is the time constant of exponential decay, and h needs to be small enough to resolve on that scale. This means we cannot use large stepsizes on this problem, even though we know exactly how it behaves. An alternate way of doing this update is called *Backward Euler*, which is almost identical to its explicit relative.

Rather than	$y_{n+1} = y_n + hf(x_n, y_n)$,
we use	$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$.

We can see that this will steer us towards the inside of curves (very loosely speaking), and this tends not to blow up (like Forward Euler).

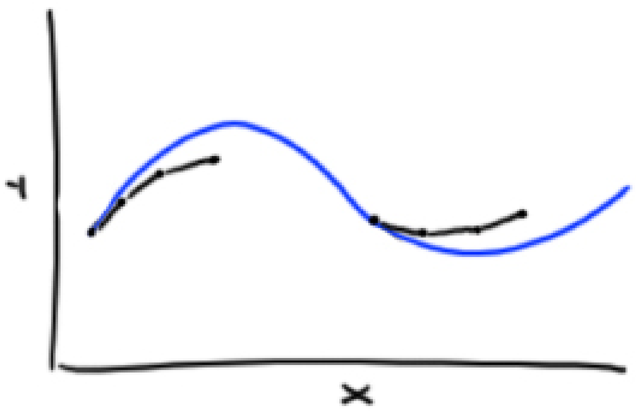

Figure 9: Backward Euler

Stability analysis for our simple problem expands out to:

$$
\begin{aligned}
y_{n+1} &= y_n + h(-cy_{n+1}) \\
y_{n+1}(1+hc) &= y_n \\
y_{n+1} &= \underbrace{\frac{1}{1+hc}}_{constant} y_n
\end{aligned}
$$

Note that since h and c are both positive, the constant term is between 0 and 1 and the solution decays . This makes this method stable for all h. This means that although the estimate may be grossly inaccurate, it will never diverge. Thus, it is *unconditionally stable*. This is characteristic of all implicit methods like this.

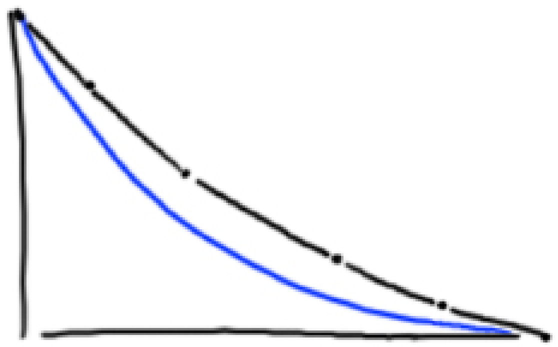

Figure 10: Backward Euler's Stabilty

One situation where stability is an important issue is in cloth simulation, where a typical problem can occur when the cloth solver error grows exponentially and explodes!

Key points for ODEs:

- intial value vs boundary value problems
- order of accuracy (first, second,...) of method
- stability (yes/no/conditional?) of method