Differential Equation Basics

Andrew Witkin and David Baraff
School of Computer Science
Carnegie Mellon University

1. Initial Value Problems

Differential equations describe the relation between an unknown function and its derivatives. To *solve* a differential equation is to find a function that satisfies the relation, typically while satisfying some additional conditions as well. In this course we will be concerned primarily with a particular class of problems, called *initial value problems.* In a canonical initial value problem, the behavior of the system is described by an ordinary differential equation (ODE) of the form

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

where f is a known function (i.e. something we can evaluate given $\mathbf{x}$ and t,) $\mathbf{x}$ is the *state* of the system, and $\dot{\mathbf{x}}$ is $\mathbf{x}$'s time derivative. Typically, $\mathbf{x}$ and $\dot{\mathbf{x}}$ are vectors. As the name suggests, in an initial value problem we are given $\mathbf{x}(t_0) = \mathbf{x}_0$ at some starting time t_0, and wish to follow $\mathbf{x}$ over time thereafter.

The generic initial value problem is easy to visualize. In $2D$, $\mathbf{x}(t)$ sweeps out a curve that describes the motion of a point $\mathbf{p}$ in the plane. At any point $\mathbf{x}$ the function f can be evaluated to provide a 2-vector, so f defines a vector field on the plane (see figure 1.) The vector at $\mathbf{x}$ is the velocity that the moving point $\mathbf{p}$ must have if it ever moves through $\mathbf{x}$ (which it may or may not.) Think of f as *driving* $\mathbf{p}$ from point to point, like an ocean current. Wherever we initially deposit $\mathbf{p}$, the "current" at that point will seize it. Where $\mathbf{p}$ is carried depends on where we initially drop it, but once dropped, all future motion is determined by f. The trajectory swept out by $\mathbf{p}$ through f forms an *integral curve* of the vector field. See figure 2.

We wrote f as a function of both $\mathbf{x}$ and t, but the derivative function may or may not depend directly on time. If it does, then not only the point $\mathbf{p}$ but the the vector field itself moves, so that $\mathbf{p}$'s velocity depends not only on where it is, but on when it arrives there. In that case, the derivative $\dot{\mathbf{x}}$ depends on time in *two ways:* first, the derivative vectors themselves wiggle, and second, the point $\mathbf{p}$, because it moves on a trajectory $\mathbf{x}(t)$, sees different derivative vectors at different times. This dual time dependence shouldn't lead to confusion if you maintain the picture of a particle floating through an undulating vector field.

2. Numerical Solutions

Standard introductory differential equation courses focus on *symbolic* solutions, in which the functional form for the unknown function is to be guessed. For example, the differential equation $\dot{x} = -kx$, where $\dot{x}$ denotes the time derivative of x, is satisfied by $x = e^{-kt}$.

In contrast, we will be concerned exclusively with *numerical* solutions, in which we take discrete *time steps* starting with the initial value $\mathbf{x}(t_0)$. To take a step, we use the derivative function f to calculate an approximate change in $\mathbf{x}$, $\Delta\mathbf{x}$, over a time interval Δt, then increment $\mathbf{x}$ by $\Delta\mathbf{x}$ to obtain the new value. In

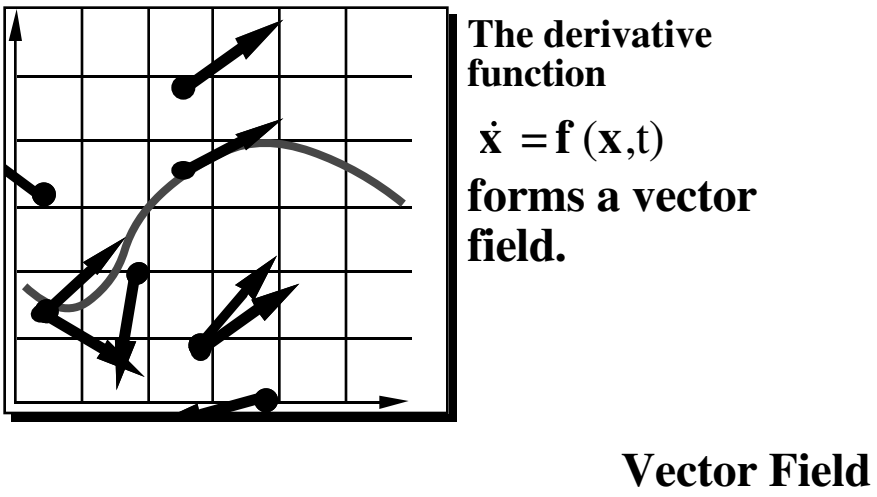


Figure 1: The derivative function $f(\mathbf{x}, t)$. defines a vector field.

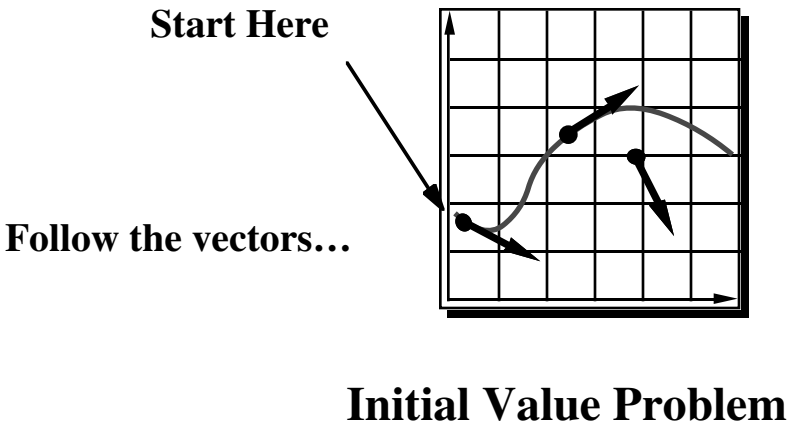
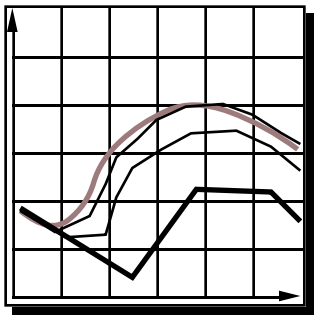


Figure 2: An initial value problem. Starting from a point $\mathbf{x}_0$, move with the velocity specified by the vector field.



- **Simplest numerical solution method**
- **Discrete time steps**
- **Bigger steps, bigger errors.**

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, f(\mathbf{x},t)$$

Euler's Method

Figure 3: Euler's method: instead of the true integral curve, the approximate solution follows a polygonal path, obtained by evaluating the derivative at the beginning of each leg. Here we show how the accuracy of the solution degrades as the size of the time step increases.

calculating a numerical solution, the derivative function f is regarded as a black box: we provide numerical values for $\mathbf{x}$ and t, receiving in return a numerical value for $\dot{\mathbf{x}}$. Numerical methods operate by performing one or more of these *derivative evaluations* at each time step.

2.1. Euler's Method

The simplest numerical method is called Euler's method. Let our initial value for $\mathbf{x}$ be denoted by $\mathbf{x}_0 = \mathbf{x}(t_0)$ and our estimate of $\mathbf{x}$ at a later time $t_0 + h$ by $\mathbf{x}(t_0 + h)$, where h is a *stepsize* parameter. Euler's method simply computes $\mathbf{x}(t_0 + h)$ by taking a step in the derivative direction,

$$\mathbf{x}(t_0 + h) = \mathbf{x}_0 + h\dot{\mathbf{x}}(t_0).$$

You can use the mental picture of a 2D vector field to visualize Euler's method. Instead of the real integral curve, $\mathbf{p}$ follows a polygonal path, each leg of which is determined by evaluating the vector f at the beginning, and scaling by h. See figure 3.

Though simple, Euler's method is not accurate. Consider the case of a $2D$ function f whose integral curves are concentric circles. A point $\mathbf{p}$ governed by f is supposed to orbit forever on whichever circle it started on. Instead, with each Euler step, $\mathbf{p}$ will move on a straight line to a circle of larger radius, so that its path will follow an outward spiral. Shrinking the stepsize will slow the rate of this outward drift, but never eliminate it.

Moreover, Euler's method can be unstable. Consider a $1D$ function $f = -kx$, which should make the point $\mathbf{p}$ decay exponentially to zero. For sufficiently small step sizes we get reasonable behavior, but when

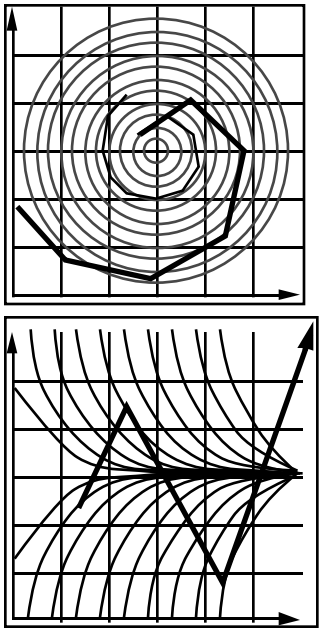


Figure 4: Above: the real integral curves form concentric circles, but Euler's method always spirals outward, because each step on the current circle's tangent leads to a circle of larger radius. Shrinking the stepsize doesn't cure the problem, but only reduces the rate at which the error accumulates. Below: too large a stepsize can make Euler's method diverge.

$h > 1/k$, we have $|\Delta x| > |x|$, so the solution oscillates about zero. Beyond $h = 2/k$, the oscillation diverges, and the system blows up. See figure 4.

Finally, Euler's method isn't even efficient. Most numerical solution methods spend nearly all their time performing derivative evaluations, so the computational cost *per step* is determined by the number of evaluations per step. Though Euler's method only requires one evaluation per step, the real efficiency of a method depends on the size of the steps it lets you take—while preserving accuracy and stability—as well as on the cost per step. More sophisticated methods, even some requiring as many as four or five evaluations per step, can greatly outperform Euler's method because their higher cost per step is more than offset by the larger stepsizes they allow.

To understand how we go about improving on Euler's method, we need to look more closely at the error that the method produces. The key to understanding what's going on is the *Taylor series*: Assuming $\mathbf{x}(t)$ is smooth, we can express its value at the end of the step as an infinite sum involving the the value and derivatives at the beginning:

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\dot{\mathbf{x}}(t_0) + \frac{h^2}{2!}\ddot{\mathbf{x}}(t_0) + \frac{h^3}{3!}\dddot{\mathbf{x}}(t_0) + \ldots + \frac{h^n}{n!}\frac{\partial^n \mathbf{x}}{\partial t^n} + \ldots$$

As you can see, we get the Euler update formula by *truncating* the series, discarding all but the first two terms on the right hand side. This means that Euler's method would be correct only if all derivatives beyond the first were zero, i.e. if $\mathbf{x}(t)$ were linear. The *error term,* the difference between the Euler step and the full, untruncated Taylor series, is dominated by the leading term, $(h^2/2)\ddot{\mathbf{x}}(t_0)$. Consequently, we can describe the error as $O(h^2)$ (read *"Order h squared".*) Suppose that we chop our stepsize in half; that is, we take steps of size $\frac{h}{2}$. Although this produces only about one fourth the error we got with a stepsize of h, we have to take twice as many steps over any given interval. That means that the error we accumulate over an interval t_0 to t_1

depends linearly upon h. Theoretically, using Euler's method we can numerically compute $\mathbf{x}$ over an interval t_0 to t_1 with as little error as we want, by choosing a suitably small h. In practice, a great many timesteps might be required, depending on the error and the function f.

2.2. The Midpoint Method

If we were able to evaluate $\ddot{\mathbf{x}}$ as well as $\dot{\mathbf{x}}$, we could acheive $O(h^3)$ accuracy instead of $O(h^2)$ simply by retaining one additional term in the truncated Taylor series:

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\dot{\mathbf{x}}(t_0) + \frac{h^2}{2}\ddot{\mathbf{x}}(t_0) + O(h^3). \tag{1}$$

Recall that the time derivative $\dot{\mathbf{x}}$ is given by a function $f(\mathbf{x}(t), t)$. For simplicity in what follows, we will assume that the derivative function f does depends on time only indirectly through $\mathbf{x}$, so that $\dot{\mathbf{x}} = f(\mathbf{x}(t))$. The chain rule then gives

$$\ddot{\mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}}\dot{\mathbf{x}} = f' f.$$

To avoid having to evaluate f',which would often be complicated and expensive, we can approximate the second-order term just in terms of f, and substitute the approximation into equation 1, leaving us with $O(h^3)$ error. To do this, we perform another Taylor expansion, this time of the function of f,

$$f(\mathbf{x}_0 + \Delta\mathbf{x}) = f(\mathbf{x}_0) + \Delta\mathbf{x} f'(\mathbf{x}_0) + O(\Delta\mathbf{x}^2). \tag{2}$$

We first introduce $\ddot{\mathbf{x}}$ into this expression by choosing

$$\Delta\mathbf{x} = \frac{h}{2} f(\mathbf{x}_0)$$

so that

$$f(\mathbf{x}_0 + \frac{h}{2} f(\mathbf{x}_0)) = f(\mathbf{x}_0) + \frac{h}{2} f(\mathbf{x}_0) f'(\mathbf{x}_0) + O(h^2) = f(\mathbf{x}_0) + \frac{h}{2}\ddot{\mathbf{x}}(t_0) + O(h^2),$$

where $\mathbf{x}_0 = \mathbf{x}(t_0)$. We can now multiply both sides by h (turning the $O(h^2)$ term into $O(h^3)$) and rearrange, yielding

$$\frac{h^2}{2}\ddot{\mathbf{x}} + O(h^3) = h(f(\mathbf{x}_0 + \frac{h}{2} f(\mathbf{x}_0)) - f(\mathbf{x}_0).$$

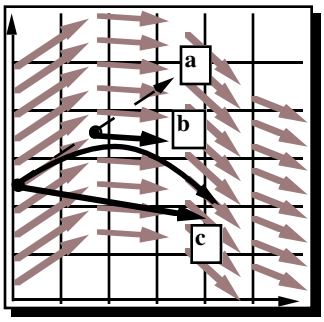
Substituting the right hand side into equation 1 gives the update formula

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h(f(\mathbf{x}_0 + \frac{h}{2} f(\mathbf{x}_0)).$$

This formula first evaluates an Euler step, then performs a second derivative evaluation at the midpoint of the step, using the midpoint evaluation to update $\mathbf{x}$. Hence the name *midpoint method.* The midpoint method is correct to within $O(h^3)$, but requires two evaluations of f. See figure 5 for a pictorial view of the method.

We don't have to stop with an error of $O(h^3)$. By evaluating f a few more times, we can eliminate higher and higher orders of derivatives. The most popular procedure for doing this is a method called Runge-Kutta of order 4 and has an error per step of $O(h^5)$. (The Midpoint method could be called Runge-Kutta of order 2.) We won't derive the fourth order Runge-Kutta method, but the formula for computing $\mathbf{x}(t_0 + h)$ is listed below:

$$\begin{aligned} k_1 &= h f(\mathbf{x}_0, t_0) \\ k_2 &= h f(\mathbf{x}_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}) \end{aligned}$$



a. Compute an Euler step

$$\Delta \mathbf{x} = \Delta t\, \mathbf{f}(\mathbf{x},t)$$

b. Evaluate f at the midpoint

$$\mathbf{f}_{\text{mid}} = \mathbf{f}\left(\frac{x + \Delta x}{2}, \frac{t + \Delta t}{2}\right)$$

c. Take a step using the midpoint value

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t\, \mathbf{f}_{\text{mid}}$$

The Midpoint Method

Figure 5: The midpoint method is a 2nd-order solution method. a) an euler step is computed, b) the derivative is evaluated again at the step's midpoint, and the second evaluation is used to calculate the step. The integral curve—the actual solution—is shown as c.

$$\begin{aligned} k_3 &= hf(\mathbf{x}_0 + \frac{k_2}{2}, t_0 + \frac{h}{2}) \\ k_4 &= hf(\mathbf{x}_0 + k_3, t_0 + h) \\ \mathbf{x}(t_0 + h) &= \mathbf{x}_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4. \end{aligned}$$

3. Adaptive Stepsizes

Whatever the underlying method, a major problem lies in determing a good stepsize. Ideally, we want to choose h as large as possible—but not so large as to give us an unreasonable amount of error, or worse still, to induce instability. If we choose a fixed stepsize, we can only proceed as fast as the "worst" sections of $\mathbf{x}(t)$ will allow. What we would like to do is to vary h as we march forward in time. Whenever we can make h large without incurring too much error, we should do so. When h has to be reduced to avoid excessive error, we want to do that also. This is the idea of adaptive stepsizing: varying h over the course of solving the ODE.

Here we'll be present adaptive stepsizing for Euler's method. The basic idea is as follows. Lets assume we have a given stepsize h, and we want to know how much we can consider changing it.

Suppose we compute two estimates for $\mathbf{x}(t_0 + h)$. We compute an estimate $\mathbf{x}_a$, by taking an Euler step of size h from t_0 to $t_0 + h$. We also compute an estimate $\mathbf{x}_b$ by taking *two* Euler steps of size $h/2$, from t_0 to $t_0 + h$. Both $\mathbf{x}_a$ and $\mathbf{x}_b$ differ from the true value of $\mathbf{x}(t_0 + h)$ by $O(h^2)$. That means that $\mathbf{x}_a$ and $\mathbf{x}_b$ differ from each other by $O(h^2)$. As a result, we can write that a measure of the current error e is

$$e = |\mathbf{x}_a - \mathbf{x}_b|$$

This gives us a convenient estimate to the error in taking an Euler step of size h.

Suppose that we are willing to have an error of as much as 10^{-4} per step, and that the current error is only 10^{-8}. Since the error goes up as h^2, we can increase the stepsize to

$$\left(\frac{10^{-4}}{10^{-8}}\right)^{\frac{1}{2}} h = 100h.$$

Conversely, if we currently had an error of 10^{-3}, and could only tolerate an error of 10^{-4}, we would have to decrease the stepsize to

$$\left(\frac{10^{-4}}{10^{-3}}\right)^{\frac{1}{2}} h \approx .316h.$$

Adaptive stepsizing is a highly recommended technique.

4. Implementation

The ODEs we will want to solve may represent many things—for instance, a collection of masses and springs, some rigid bodies, or a deformable object. We want to implement ODE solvers and the models on which they operate in a way that isolates each from the internal details of the other. This will make it possible to change solvers easily, and also make the solver code reusable. Fortunately, this kind of modularity is not difficult to acheive, since all solvers can be expressed in terms of a small, stereotyped set of operations. Presumably, the system of ODE-governed objects will be embodied in a structure of some kind. The approach is to write type-specific code that operates on this structure to perform the standard operations, then to implement solvers in terms of these generic operations.

From the solver's viewpoint, the system on which it operates is a black-box function $f(\mathbf{x}, t)$. The solver needs to be able to evaluate f, as required, at any values of $\mathbf{x}$ and t, and then to install the updated $\mathbf{x}$ and t when a time step is taken. To support these operations, the object that represents the ODE being solved must be able to handle these requests from the solver:

- Return dim ($\mathbf{x}$). Since $\mathbf{x}$ and $\dot{\mathbf{x}}$ may be vectors, the solver must know their length, to allocate storage, perform vector arithmetic ops, etc.
- Get/set $\mathbf{x}$ and t. The solver must be able to install new values at the end of a step. In addition, a multi-step method must set $\mathbf{x}$ and t to intermediate values in the course of performing derivative evaulations.
- Evaluate f at the current $\mathbf{x}$ and t.

In an object-oriented language, these operations would naturally be implemented as generic functions that are handled in a type-specific way. In a non-object-oriented language generic functions would be faked by installing pointers to type-specific functions in structure slots, or simply by passing the function pointers as arguments to the solver. Later on we will consider in detail how these operations are to be implemented for specific models such as particle-and-spring systems.

References

[1] W.H. Press, B.P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1988.