Numerical integration for graphics simulation

Steve Marschner
CS 5643 Spring 2023
Cornell University

For simulation we need to solve ODEs. That is, there is an unknown function y that we have to find from an equation that tells us something about its derivative. One standard form is

$$y'(t) = f(t, y(t))$$

In most interesting cases we can't derive a closed-form solution for y so we instead ask for an algorithm that can call the function f many times and compute sufficiently accurate approximations to $y(t_k)$ for many t_k. Such algorithms are called ODE solvers, numerical integrators, or when the context is clear just "solvers" or "integrators."

Pretty much all integrators used for simulation are structured in terms of time steps: we assume we know $y_k = y(t_k)$ and we solve some equation to compute $y_{k+1} \approx y(t_{k+1})$ where $t_{k+1} = t_k + h$ and h is known as the step size. The process to get from t_k to t_{k+1} is called a "time step" (though sometimes h is called the time step).

An important question with ODE solvers is how accurate the answers are. At each step the approximation we compute has some error, and that error will accumulate as we repeat this process to travel far forward in time. For the integrator to be useful at all, we expect that the accumulated error at some fixed time T to decrease as we take more steps, and for this to happen the error in an individual step has to decrease faster than h does. Because the methods we use are all based on polynomial approximations, our error bounds are normally polynomials, so this means the largest useful error in one time step is $O(h^2)$.

For instance if we integrate our system from time 0 to time T in N equal-sized steps, then $h = T/N$ and if the error in each step is $O(h^2)$ then the error after N steps is $O(h)$. In this case we would call the integrator a "first-order method" which means two things. An individual step is "first-order accurate" meaning its approximation accounts for all the terms up through first order (so that its error is second order). And the final result at a fixed time has first-order error.

Some basic integrators

Most integrators are derived by writing down a Taylor expansion of y and using it to write some equation involving y_k and some values of f that can be solved to find y_{k+1}. If we expand $y(t)$ around t_k we get

$$\begin{aligned} y(t) &= y_k + y'(t_k)(t - t_k) + \dots \\ y(t_{k+1}) &\approx y_k + hf(t_k, y_k) \end{aligned}$$

and if we disregard the error we have the simple timestep formula:

$$y_{k+1} = y_k + hf(t_k, y_k).$$

This is known as Euler's method, or "forward Euler," or "explicit Euler."

On the other hand if we expand $y(t)$ around t_{k+1} we get

$$\begin{aligned} y(t) &= y(t_{k+1}) + y'(t_{k+1})(t - t_{k+1}) \\ y(t_k) &\approx y_{k+1} - hf(t_{k+1}, y_{k+1}) \end{aligned}$$

and the timestep equation is

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}).$$

Note that this is an equation that has to be solved numerically, since y_{k+1} appears as an argument to f. This is known as the backward, or implicit, Euler's method.

Another way to think of these is in terms of finite difference approximations to the derivative. The familiar forward difference formula for estimating a $y'(t_k)$ is

$$y'(t) \approx \frac{y(t+h) - y(t)}{h}$$

Applying this formula for $t = t_k$, multiplying both sides by h, and solving for $y(t_k + h)$ results in the forward Euler step. If instead we use the backward difference formula

$$y'(t) \approx \frac{y(t) - y(t-h)}{h}$$

and apply it for $t = t_{k+1}$ then we get the (implicit) timestep equation for the backward Euler method. This explains the names "forward" and "backward."

In each of these cases the discarded error term was $O(h^2)$.

If we expand around $t = (t_k + t_{k+1})/2$, we can get a more accurate method. To neaten the notation I'm going to use subscripts for "old," "midpoint," and "new" so that $t_o = t_k$, $t_m = t_k + h/2$, and $t_n = t_{k+1} = t_k + h$. Also $y_o = y_k = y(t_o)$, $y_m \approx y(t_m)$, and $y_n = y_{k+1} \approx y(t_n)$. Now let's write the Taylor series for y around t_m and keep one more term:

$$y(t) = y(t_m) + y'(t_m)(t - t_m) + \frac{1}{2}y''(t_m)(t - t_m)^2 + O[(t - t_m)^3]$$

Now if we compute the step $y_n - y_o$ the quadratic term cancels and we have:

$$\begin{aligned} y(t_o) &= y(t_m) + \frac{h}{2}y'(t_m) + \frac{h^2}{8}y''(t_m) + O(h^3) \\ y(t_n) &= y(t_m) - \frac{h}{2}y'(t_m) + \frac{h^2}{8}y''(t_m) + O(h^3) \qquad [1] \\ y(t_n) - y(t_o) &= hy'(t_m) + O(h^3) \end{aligned}$$

To use this formula we need an estimate for $y'(t_m) = f(t_m, y(t_m))$, and to avoid losing our nice cubic error, we need its error to be $O(h^2)$. We can get a suitable estimate using Euler's method.

$$y(t_m) = y_o + \frac{h}{2}f(t_o, y_o) + O(h^2)$$

so we can use the estimate

$$y_m = y_o + \frac{h}{2}f(t_o, y_o) = y(t_m) + O(h^2)$$

to compute an estimate of $y'(t_m)$:

$$\begin{aligned} f(t_m, y(t_m)) &= f(t_m, y_m) + f'(t_m, y_m)(y(t_m) - y_m) + O[(y(t_m) - y_m)^2] \\ &= f(t_m, y_m) + f'(t_m, y_m)O(h^2) + O(h)^4 \\ &= f(t_m, y_m) + O(h^2) \end{aligned}$$

(Here by f' I mean the derivative of f with respect to y.) Plugging this into our formula above for $y(t_m) - y(t_o)$ we get

$$\begin{aligned} y(t_n) - y(t_o) &= h[f(t_m, y_m) + O(h^2)] + O(h^3) \\ &= hf(t_m, y_m) + O(h^3) \end{aligned}$$

This leads to an integration method known as the Midpoint method, which uses two evaluations of f but gives more accuracy:

$$\begin{aligned} y_m &= y_k + hf(t_k, y_k)/2 \\ y_{k+1} &= y_k + hf(t_k + h/2, y_m) \end{aligned}$$

I feel the argument with the Taylor series is the clearest, but there is a quicker explanation you might also like. The central difference formula

$$y'(t) \approx \frac{f(t + h/2) - f(t - h/2)}{h}$$

is a more accurate way to estimate a derivative; its error is second-order. The midpoint method is using a first-order-accurate estimate of $y'(t_m)$ with this formula to get a second-order-accurate estimate of $y(t_n)$.

This kind of approach can be continued to cancel out the contributions of higher order derivatives to the error, leading to integrators with higher orders of accuracy. The best known such integrators are the Runge-Kutta family, which includes methods that use at least p evaluations of f to compute a timestep with order of accuracy p. The forward Euler and midpoint methods are both Runge-Kutta methods, and a fourth-order Runge-Kutta integrator (often abbreviated RK4) is often recommended for problems requiring high accuracy.

However, one rarely sees higher than second-order integrators being used for animation, and there are a few reasons for this. First, stability and conservation of energy are often of more importance than accuracy, so low order methods that have these properties may be preferred. Also, animation problems often produce systems that are not differentiable to high enough order to satisfy the conditions required for methods like RK4 to deliver the accuracy they are designed for.

Generalizing the problem

ODEs always have a single independent variable (time, for animation), but they can have many dependent variables. This does not actually change much in the derivation of the methods, but it changes our mental picture a bit! The unknown function is vector valued so we write

$$\mathbf{f}(\mathbf{x}(t), \dot{\mathbf{x}}(t)) = 0$$

and when it's easy to solve for the derivative, we can put it in the form

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

I'm switching to dots as a notation for time derivatives, as is conventional in mechanics, and to $\mathbf{x}$ as the name for the unknown function of time. You'll notice I left out t as an argument of $\mathbf{f}$. If we want $\mathbf{f}$ to depend on t we can just add another component to $\mathbf{y}$ that has constant derivative 1, and $\mathbf{f}$ can depend on that. So even though time might be handled specially in practice we don't need it cluttering up our notation.

Now that the unknown is vector valued, you can think of the solution as a path through a multidimensional space, often known as a "state space" since a single value of $\mathbf{x}$ represents the complete state of the system (i.e. nothing else is remembered from one step to the next). An ODE with a vector-valued unknown function can also be thought of as a system of single-variable ODEs.

The other thing we need to generalize is to allow for higher order derivatives. For animation we normally only need up to second derivatives so we can write

$$\mathbf{f}(\mathbf{x}(t), \dot{\mathbf{x}}(t), \ddot{\mathbf{x}}(t)) = 0$$

or in the common case that the equations come already solved for the second derivative,

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

In principle this also doesn't change things much, because of a standard trick: by introducing extra variables we can express the same problem as a first-order ODE. The idea is to expand the state to include both the unknown and its first derivative:

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

(I'm using the letter **v** to suggest velocity, which makes sense if **x** is a position.) Without actually changing the function **f**, we have managed to write our second-order ODE as a first-order ODE (meaning that it has only first derivatives in it).

One way of seeing why we needed to include the velocity in the system state is that once you have a second-order system the position is no longer enough to predict the future. For the simplest possible example consider the second-order equation for a free particle:

$$\ddot{\mathbf{x}} = 0$$

The particle will keep moving at its current velocity forever; clearly predicting where it will be in the future requires knowing what velocity it has now.

Integrators for second-order systems

After reading the preceding section you might think we are done, that the second-order nature of our ODEs can be safely encapsulated some very useful integrators have been developed that are specialized for second-order systems.

Two integrators that are well known in mechanics are the Verlet methods and the Leapfrog method. Both are second-order integrators specialized for conservative systems where forces depend only on position:

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

The "velocity Verlet" method uses ideas similar to the midpoint method to construct second-order approximations of the new position and velocity. and uses an estimate of $\mathbf{v}(t_m)$ to update **x**. Start with the second-order midpoint step from [1]:

$$\mathbf{x}(t_n) = \mathbf{x}(t_o) + h\,\mathbf{v}(t_m) + O(h^3)$$

and use an Euler step to estimate $\mathbf{v}(t_m)$:

$$\mathbf{v}(t_m) = \mathbf{v}(t_o) + \frac{h}{2}\mathbf{a}(t_o) + O(h^2)$$

Substituting gets us the step equation for the position

$$\begin{aligned} \mathbf{x}(t_n) &= \mathbf{x}(t_o) + h(\mathbf{v}(t_o) + \frac{h}{2}\mathbf{a}(t_o) + O(h^2)) + O(h^3) \\ &= \mathbf{x}(t_o) + h\,\mathbf{v}(t_o) + \frac{h^2}{2}\mathbf{a}(t_o) + O(h^3) \\ \mathbf{x}_n &= \mathbf{x}_o + h\,\mathbf{v}_o + \frac{h^2}{2}\mathbf{a}_o \end{aligned}$$

With this second-order approximation of $\mathbf{x}(t_n)$ we can compute an approximation to $\mathbf{a}(t_n)$:

$$\mathbf{a}(t_n) = \mathbf{f}(\mathbf{x}(t_n)) = \mathbf{f}(\mathbf{x}_n) + O(\mathbf{x}(t_n) - \mathbf{x}_n) = \mathbf{f}(x_n) + O(h^3)$$

and we name it $\mathbf{a}_n = \mathbf{f}(\mathbf{x}_n)$. Now having approximations to both $\mathbf{a}(t_o)$ and $\mathbf{a}(t_n)$ we can average them to get a first-order approximation to $\mathbf{a}(t_m)$:

$$\mathbf{a}(t_m) = \frac{\mathbf{a}_o + \mathbf{a}_n}{2} + O(h^2)$$

To prove this order of approximation:

$$\begin{aligned} \mathbf{a}(t) &= \mathbf{a}(t_m) + \dot{\mathbf{a}}(t_m)(t - t_m) + O[(t - t_m)^2] \\ \mathbf{a}(t_o) + \mathbf{a}(t_n) &= 2\mathbf{a}(t_m) - \frac{h}{2}\dot{\mathbf{a}}(t_m) + \frac{h}{2}\dot{\mathbf{a}}(t_m) + O(h^2) \end{aligned}$$

And as in the midpoint method, this first-order approximation to the derivative of $\mathbf{v}$ enables a second-order approximation of $\mathbf{v}(t_n)$:

$$\mathbf{v}(t_n) = \mathbf{v}(t_o) + h\,\frac{\mathbf{a}_o + \mathbf{a}_n}{2} + O(h^3)$$

and the time-step formulas are

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + h\,\mathbf{v}_k + \frac{h^2}{2}\mathbf{a}_k \\ \mathbf{a}_{k+1} &= \mathbf{f}(\mathbf{x}_{k+1}) \\ \mathbf{v}_{k+1} &= \mathbf{v}_k + h\,\frac{\mathbf{a}_k + \mathbf{a}_{k+1}}{2} \end{aligned}$$

Note that the previous step's value of $\mathbf{a}$ is remembered, so only one call to $\mathbf{f}$ is required for each time step. The values of both $\mathbf{x}$ and $\mathbf{v}$ are second-order accurate—remember it took two calls to $\mathbf{f}$ per iteration for the midpoint method to achieve this accuracy.

A similar but arguably more elegant method is the leapfrog method. The idea here is to sample position and velocity at interleaved times, so that if $\mathbf{x}$ is sampled at times t, $t + h$, $t + 2h$, ..., then $\mathbf{v}$ is sampled at times $t + 0.5$, $t + 1.5$, $t + 2.5$, ... Then for every step we already have the velocity at the midpoint of the time interval for updating $\mathbf{x}$, and we can easily compute the acceleration at the midpoint of the time interval for updating $\mathbf{v}$. This leads to very simple time-step formulas:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k + h\,\mathbf{v}_{k+0.5} \\ \mathbf{v}_{k+1.5} &= \mathbf{v}_{k+0.5} + h\,\mathbf{f}(\mathbf{x}_{k+1}) \end{aligned}$$

These equations look as simple as for Euler's method, and use the same number of $\mathbf{f}$ evaluations, but because of the half-step offset (which is only possible because $\mathbf{f}$ depends only on $\mathbf{x}$) it achieves second-order accuracy.

So Verlet and leapfrog integration give you second-order accuracy essentially for free, but this property depends on $\mathbf{f}$ being independent of $\mathbf{v}$, which is often not the case in animation applications (where we want damping, friction, and atmospheric drag which all introduce velocity dependent forces). We can

hack in an approximation for $\mathbf{v}$ at a nearby point in time, but this will take us back to first-order accuracy.

Another feature of these two integrators that explains their popularity in mechanics is that they exactly preserve some invariants that ought to be preserved. Some familiar invariants that are conserved in closed mechanical systems are total energy, linear momentum, and angular momentum. When the equations $\mathbf{f}$ properly model a closed system, the exact solutions to the equation $\ddot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \dot{\mathbf{x}})$ will conserve these invariants, and since our solutions approximate exact solutions as $h \to 0$, all the methods we've discussed *approximately* conserve the invariants. But for any particular value of h there will be some error, and after the error accumulates for a while the solution can be qualitatively implausible, showing a pendulum that swings higher and higher or a spinning object that slows for no reason. In practice, these errors can mean having to use very small step sizes to get qualitatively reasonable behavior. This is a particularly important concern for animation, where we often don't need high accuracy and don't want to have to pay for it in order to get long-term stability.

In particular, these methods both are *symplectic* integrators, meaning that they preserve state-space volume in a certain sense. This is not the same as preserving energy, but it does mean that when integrating oscillatory systems (with no damping or other dissipation) they are able to produce consistent orbits that don't grow or shrink. In the context of animation this tends to mean they can produce lively motion (small wiggles don't damp out too fast) without blowing up (small wiggles growing uncontrollably), often at step sizes much larger than you would need with one of the earlier methods that don't have this property. But these methods can't be applied directly when the acceleration needs to depend on both position and velocity. (If you run across code that claims to be using Verlet but their models include dissipation, they are doing it wrong and not seeing the second-order accuracy.)

This brings us to the last of the simple mechanics-oriented integrators we'll look at, and one which is a good first method to try for most physics-based animations. You can think of it as the leapfrog method without the interleaving, or as a merger of the explicit and implicit Euler methods. Its time-step formulas are:

$$\mathbf{v}_{k+1} = \mathbf{v}_k + h\,\mathbf{f}(\mathbf{x}_k, \mathbf{v}_k)$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k + h\,\mathbf{v}_{k+1}$$

Since this can be described as applying explicit Euler to velocity and implicit Euler to position, one name for it is "the semi-implicit Euler method." This is how we usually write it in graphics but it's a special case of something a little more general, where the system takes the form

$$\begin{bmatrix}\dot{\mathbf{x}}\\ \dot{\mathbf{v}}\end{bmatrix} = \begin{bmatrix}\mathbf{f}(\mathbf{x}, \mathbf{v})\\ \mathbf{g}(\mathbf{x}, \mathbf{v})\end{bmatrix}$$