Numerically integrating equations of motion

1 Introduction to numerical ODE integration algorithms

Many models of physical processes involve differential equations: the rate at which some thing varies depends on the current state of the system, and possibly external variables such as time. Here we explore how to numerically solve these equations.

The concrete example which we are considering in this module is dynamics of a pendulum. The angular acceleration of a pendulum bob depends on how far up the pendulum is pulled. This gives the equation of motion

$$\frac{d^2\theta}{d\tau^2} = -\frac{g}{L}\sin(\theta),$$

where I have used τ for time, because we are going to use t for a dimensionless time. In particular if we define $t = \sqrt{L/g}\tau$ the equation of motion becomes

$$\frac{d^2\theta}{dt^2} = -\sin(\theta).$$

This is what is known as going to natural units. It is generally a good idea both for analytic calculations and numerical ones.

To find an approximate solution to this differential equation we discretize time, and use some finite difference approximation for the derivative. Our choice of discretization, and our approximation for the derivative will give different algorithms for solving the equation. For generality, most of the literature on solving differential equations concentrates on systems of first order differential equations. We can make our equation of that form by writing it as

$$\begin{aligned} \frac{d\theta}{dt} &= \omega \\ \frac{d\omega}{dt} &= -\sin(\theta). \end{aligned}$$

For the simplest method one uses a constant *step size* δ. Thus starting from knowing $\theta(t)$ and $\omega(t)$ you use a finite difference approximation to calculate $\theta(t+\delta)$ and $\omega(t+\delta)$, then use these to calculate $\theta(t+2\delta)$... More sophisticated algorithms will use an adaptive step size, where the step is made smaller or larger at different times in order to guarantee a rapid calculation which achieves a predetermined accuracy.

There are several different conceptual ways to come up with approximations of the derivatives. One approach is to use the ”mean value theorem” which says that for a sufficiently well behaved function f, that the slope of the secant line of f on some interval $[t_1, t_2]$ is exactly equal to the slope of f at some intermediate point $\bar{t}$

$$f(t_2) - f(t_1) = (t_2 - t_1) f'(\bar{t}).$$

If we replace $\bar{t}$ with t_1 we get the *forward Euler* approximation

$$\begin{aligned} \theta(t+\delta) &= \theta(t) + \delta\omega(t) \\ \omega(t+\delta) &= \omega(t) - \delta \sin[\theta(t)]. \end{aligned}$$

If we replace $\bar{t}$ with t_2 we get the *backward Euler* approximation

$$\begin{aligned} \theta(t+\delta) &= \theta(t) + \delta\omega(t+\delta) \\ \omega(t+\delta) &= \omega(t) - \delta \sin[\theta(t+\delta)]. \end{aligned}$$

If we replace $\bar{t}$ with $(t_1 + t_2)/2$ we get the *leapfrog* approximation

$$\begin{aligned} \theta(t+\delta) &= \theta(t) + \delta\omega(t+\delta/2) \\ \omega(t+\delta/2) &= \omega(t-\delta/2) - \delta \sin[\theta(t)]. \end{aligned}$$

For an algorithm based on the leapfrog method, we would use a different grid for the angles and the angular velocities. Sometimes this is inconvenient, so this last equation can be made to look more like the others if we shift all of our angular frequencies, writing $\bar{\omega}(t) = \omega(t - \delta/2)$. We then have

$$\begin{aligned} \theta(t+\delta) &= \theta(t) + \delta\bar{\omega}(t+\delta) \\ \bar{\omega}(t+\delta) &= \bar{\omega}(t) - \delta \sin[\theta(t)] \end{aligned}$$

Neglecting the difference between $\bar{\omega}$ and ω gives us the *symplectic Euler* approximation,

$$\begin{aligned} \theta(t+\delta) &= \theta(t) + \delta\omega(t+\delta) \\ \omega(t+\delta) &= \omega(t) - \delta \sin[\theta(t)]. \end{aligned}$$

The forward Euler algorithm is known as *explicit*, meaning that the state variables at a later time are directly calculated from the prior state variables. The backward Euler algorithm is *implicit*, meaning you need to solve an equation (in this case a transcendental equation) to get the later state variables. The symplectic Euler algoritm is *semi-implicit*. In this case since the relationship between $d\theta/dt$ and ω is linear, you can solve the equation analytically, so the sympectic Euler algorithm is no harder to implement than the forward Euler algorithm. Explicit algorithms tend to be less stable than implicit ones. We will discuss this a bit in section 3.

A word of caution: you typically do not want to use one of these simple integration algorithms for any real calculations. There are much better ones.

2 Accuracy – higher order methods

One thing you want your algorithm to do is produce *accurate* results. You want the difference between your approximate solution and the exact solution to be small. Clearly making δ smaller increases the accuracy. On the other hand, smaller δ means that the calculation takes longer, and that more round-off error is accumulated at each step. To evaluate how good an algorithm is for your purposes you want to know how quickly the accuracy increases as you decrease the step size. The standard way to express this is with *big oh* notation. We say $f(\delta) = \mathcal{O}(\delta^n)$ as $\delta \to 0$ if there exists numbers A and ϵ such that for all $\delta < \epsilon$ one finds $f(\delta) \leq A\delta^n$.

One can estimate the accuracy of the algorithms given above by making use of Taylor's theorem, which including the error estimates says $f(x+\delta) = f(x) + \delta f'(x) + \cdots + \delta^n/n! f^(n)(x) + \mathcal{O}(\delta^{n+1})$. To evaluate the forward Euler algorithm we note

$$\begin{aligned}
\theta(t+\delta) &= \theta(t) + \delta\theta'(t) + \mathcal{O}(\delta^2) \\
&= \theta(t) + \delta\omega(t) + \mathcal{O}(\delta^2) \\
\omega(t+\delta) &= \omega(t) + \delta\omega'(t) + \mathcal{O}(\delta^2) \\
&= \omega(t) - \sin[\theta(t)] + \mathcal{O}(\delta^2).
\end{aligned}$$

so the error accumulated in each time step is $\mathcal{O}(\delta^2)$. Since the number of time steps scales as $1/\delta$, the total error in this method is $\mathcal{O}(\delta)$. That is, if you decrease δ the error should decrease linearly.

Similar analysis on the backward Euler and symplectic Euler reveals they are also $\mathcal{O}(\delta)$ methods. On the other hand, the leapfrog method is $\mathcal{O}(\delta^2)$:

$$\begin{aligned}
\theta(t+\delta) &= \theta(t) + \delta\theta'(t) + (\delta^2/2)\theta''(t) + \mathcal{O}(\delta^3) \\
&= \theta(t) + \delta\left[\omega(t) + (\delta/2)\omega'(t)\right] + \mathcal{O}(\delta^3) \\
&= \theta(t) + \delta\omega(t+\delta/2) + \mathcal{O}(\delta^3).
\end{aligned}$$

One general strategy for building up higher order methods is to add extra function evaluations. For illustration, imagine we have a differential equation

$$\frac{d^2 f}{dt^2} = G(f),$$

and we want to step from $f_i = f(t)$ to $f_f = f(t+\delta)$. Well we begin with an Euler step to $\delta/2$, defining

$$f_1 = f_0 + (\delta/2)G(f_0).$$

We then introduce two arbitrary coefficients a and b, and guess

$$f_f^{\text{trial}} = f_0 + \delta\left[aG(f_0) + bG(f_1)\right].$$

One then must choose a and b to make this as accurate as possible. Carrying out the above analysis, we find

$$\begin{aligned} f_f^{\mathrm{trial}} &= f(0) + a\delta f'(0) + b\delta G\left[f(0) + (\delta/2) f'(0)\right] \\ &= f(0) + a\delta f'(0) + b\delta G[f(0)] + \frac{b\delta^2}{2} f'(0) G'[f(0)] + \mathcal{O}(\delta^3) \\ &= f_0 + (a+b)\delta f'(0) + \frac{b\delta^2}{2} f''(0) + \mathcal{O}(\delta^3). \end{aligned}$$

This will be an order δ^3 error in each step if we take $a = 0$ and $b = 1$, yielding an $\mathcal{O}(\delta^2)$ method. This is called the *midpoint* method (and it clearly has a connection with the leapfrog approximation). We can make an even higher order method by taking more than one intermediate time step and following a similar procedure. A typical higher order method is the fourth order explicit Runge-Kutta method (RK4). This is often implemented with an adaptive step size.

3 Stability and fidelity

Sometimes we want to really push our algorithms. For example we might want to integrate to very long times. At fixed step size the errors should grow linearly with time for large time. This could be fixed by decreasing the step size, but at the expense of drastically increasing computation time. Another fix is to use an algorithm for which the behavior remains qualitative correct even when the accuracy is poor.

Here is where we see a difference in the three first order methods that we presented. The forward Euler algorithm is simply unstable. At short times it stays near the exact solution but it slowly drifts off. At long times the energy grows. A crude way to see this is to note that there is effectively a delay between when a force is applied and when it causes and acceleration. One can directly analyze the finite difference equaitons, but it is simpler to think about an analogous differential equation, such as

$$\frac{d^2x}{dt^2} = \frac{1}{m} F[x(t-\epsilon)].$$

Assuming that ϵ is small, this can be approximated by

$$\frac{d^2x}{dt^2} = \frac{1}{m} F[x(t)] - \frac{\epsilon}{m}\frac{dx}{dt}\frac{dF}{dx},$$

and we effectively have a velocity dependent force. If we are near a potential minimum we should have $dF/dx < 0$, which implies that if $\epsilon > 0$ the velocity dependent force tries to increase the velocity. This is clearly unstable. One says that near a potential energy minimum the numerical algorithm has added an *antidamping* term, and the energy tends to grow with time. The same argument says that the backward Euler method (where effectively $\epsilon < 0$) is stable. The backward algorithm has adds a *damping* term, and the energy tends to decrease with time.

This result is clearly generic. Implicit methods tend to be unstable, while explicit methods tend to introduce damping.

As you might guess, the symplectic Euler algorithm falls in between. The energy neither grows nor drops by too much. Below it will be explained *why* it has this behavior. It is important, however, to point out that depending on the problem at hand, the behavior of the symplectic Euler algorithm may not be any better than the others. It is no more accurate than the other Euler algorithms. [For example, the period which it predicts for the pendulum will be no better than the period from the other algorithms – perhaps it might even be worse.] On the other hand there are times where it is useful to use a symplectic integrator. One example is in Molecular Dynamic simulations. If you have ever done any hydrodynamics, you know that many of the properties of fluids depend only on the presence of conservation laws. The symplectic algorithms incorporates such conservation laws.

What makes gives the symplectic Euler algorithm this property is that it preserves the *Poisson bracket*, which mathematically is a *symplectic form.* That is why we call the symplectic Euler algorithm as *symplectic* integrator. Recall that given conjugate variables θ and ω, the Poisson bracket, defined by $\{a, b\}_{\theta\omega} = (\partial a/\partial\theta)(\partial b/\partial\omega) - (\partial a/\partial\omega)(\partial b/\partial\theta)$, has essentially the same properties as the quantum mechanical commutator. The fundamental Poisson bracket is $\{\theta, \omega\}_{\theta\omega} = 1$. If a mapping preserves the fundamental Poisson bracket it will preserve all Poisson brackets. Let θ_0 and ω_0 be time $t = 0$ values of the state variables, and let θ_1 and ω_1 be the variables after one time step of the symplectic Euler algorithm. Straightforward algebra reveals $\{\theta_1, \omega_1\}_{\theta_0\omega_0} = 1$. The same algebra shows that neither the forward nor backward Euler algorithms preserve the Poisson bracket.

A mapping which preserves Poisson brackets is described as *Canonical.* Thinking of it as a map of phase space onto itself, a Canonical transformation preserves areas. Conceptually, a useful way of thinking of a Canonical transformation is that there is always some Hamiltonian which generates equations of motion which will produce any given Canonical transformation. Thus a symplectic algorithm can be thought of as providing the *exact* solution to the motion of a system whose Hamiltonian approximates the one you are interested in. As you make the step size smaller, the two Hamiltonians become closer.

A natural question at this point is how one produces higher order symplectic integrators. One approach is to use the fact that composition of Canonical transformations creates another Canonical transform. Imagine, as in the present case, that the Hamiltonian you are interested in has the form $H = T + V$, with kinetic energy T and potential energy V. We let $H_1 = T$ and $H_2 = V$ be Hamiltonians for fictitious systems. We can approximation the time evolution from time t to $t+\delta$, by first time evolving by time δ using Hamiltonian H_1, and then time δ using Hamiltonian H_2. This is the symplectic Euler method. We can produce higher order methods by interleaving more time evolution steps. For example, evolving by H_1 for time $\delta/2$, then H_2 for time δ, then H_1 for time

$\delta/2$, gives the Verlet algorithm – which is second order.

4 Other ODE integration algorithms

There are a few other ODE integration strategies which are worth mentioning.

- **Multistep methods:** Instead of just using the value of the state variables at time t to calculate the state at time $t + \delta$, multistep methods use the value of the state at several previous times (for example using values at time t and $t - \delta$).
- **Multiderivative methods:** If you can analytically calculate the derivatives of the forces, you can use this information in multiderivative methods.
- **Predictor-Corrector methods:** These first use a prediction step to approximate the value of the state variables at time $t + \delta$. They then use this value to refine the guess. Typically predictor-correlators are multistep methods. Apparently predictor-corrector methods have fallen out of favor lately.
- **Extrapolation methods:** These are conceptually a little different. Essentially they perform the calculation using several different step sizes, then use an extrapolation scheme to estimate what would happen if you took h to zero.

Most modern multipurpose ODE solvers use either some versions of Runge-Kutta or extrapolation methods. The scipy ODE integrator defaults to a predictor-corrector method (Adams), but also supports an implicit multistep method (Gears). The latter is very stable, but not necessarily as efficient. It is best used for "stiff" problems, meaning those with multiple time-scales which can give other integrators trouble.

5 Further Study

A good exercise to bring this all together is to apply the different Euler algorithms to the simple harmonic oscillator. You can actually analytically calculate what the position and velocity of the oscillator will be after N timesteps (hint: you will need to diagonalize a 2-by-2 matrix).