Ph 22.1 – Return of the ODEs: higher-order methods

-v20130102-

Introduction

This week we are going to build on the experience that you gathered in the Ph20, and program more advanced (and accurate!) solvers for ordinary differential equations. The final result of this assignment, a general-purpose *Runge–Kutta* ODE integrator, will be very close to the numerical routines that you might one day use in your own research.

Rewriting higher-order ODEs systems as first order systems

You will often encounter ordinary differential equations containing nth order derivatives. One useful trick is that a single nth order equation can always be reduced to a *system* of equations that contain only first derivatives! This is done by defining new variables. For example, the second-order ODE

$$\frac{d^2x(t)}{dt^2} + q(t)\frac{dx(t)}{dt} = r(t) \tag{1}$$

can be rewritten as two first-order equations by defining a new variable $v(t)$:

$$\frac{dx(t)}{dt} = v(t), \quad \frac{dv(t)}{dt} = r(t) - q(t)v(t). \tag{2}$$

The most obvious choices for the new variables are the first and higher derivatives of the original variables, but occasionally you may want to use other functions of them to reduce error.

Because you can always reduce an ODE (or a system of ODEs) to first order equations, you can therefore write *any* set of ODEs in the following form:

$$\frac{d\xi_i(t)}{dt} = f_i(\xi_1, \xi_2, \ldots, \xi_N, t), \quad i = 1, \ldots, N. \tag{3}$$

Don't be confused by the notation: here we have written N coupled *first-order* differential equations, labeled by the index i. The unknown functions are called $\xi_1(t)$, $\xi_2(t)$, …, $\xi_N(t)$, and these denote *all* the variables (positions *and* velocities).

The solution to a set of ODEs depends not only on the differential equations, but also on the *boundary conditions*. Here we will consider the most straighforward *initial-value problems*: the boundary conditions are the specified values of all the ξ_i at the initial time $t = t_0$.

Improving on the Euler method.

In Ph20, we implemented a few variants of the *Euler method* for the numerical integration of ODEs. The Euler method is *first order*, which means that the *local* error introduced by each integration step [taking us from $\xi(t)$ to $\xi(t+h)$] scales like h^2: that is, for the equation $d\xi/dt = f[\xi(t)]$ (and using forward Euler),

$$\xi_{\text{fe}}(t+h) = \xi(t) + hf[\xi(t)] = \xi(t+h) + O(h^2). \tag{4}$$

Here we use $\xi(t+h)$ for the *exact* value of ξ at time $t+h$ and we use $\xi_{\text{fe}}(t+h)$ for the *approximate* value obtained using our forward Euler step. Thus, if we reduce the stepsize to half its original

value, we should get an error four times smaller. By contrast, the *global error* across a *fixed* time interval Δt scales only as h, because when we reduce the stepsize h we must also increase the total number of steps by the same amount to get to the same Δt. This global error scaling is why we call the Euler methods "first-order".

The reason that Equation (4) works is because of Taylor's theorem: you will notice that Equation (4) is just the beginning (appropriately, up to the *first order*) of the Taylor expansion

$$\begin{aligned} \xi(t+h) &= \xi(t) + h\frac{d\xi(t)}{dt} + \frac{h^2}{2!}\frac{d^2\xi(t)}{dt^2} + \ldots + \frac{h^p}{p!}\frac{d^{(p)}\xi(t)}{dt^p} + O(h^{p+1}) \quad (5) \\ &= \xi(t) + hf[\xi(t)] + \frac{h^2}{2!}\frac{df[\xi(t)]}{dt} + \ldots + \frac{h^p}{p!}\frac{d^{(p-1)}f[\xi(t)]}{dt^{p-1}} + O(h^{p+1}). \quad (6) \end{aligned}$$

So you might ask what happens if, instead of truncating the Taylor expansion at first order, we kept more terms in the expansion? Can we then generalize the Euler method to write pth-order numerical schemes where the local error scales as $O(h^{p+1})$? Sure! We begin with a very simple second-order scheme, the *midpoint method*. Instead of taking a full Euler step from t to $t+h$, we first take a half step,

$$\tilde{\xi} \equiv \xi(t) + \frac{h}{2}f[\xi(t)], \quad (7)$$

defining an approximate value $\tilde{\xi}$ for ξ at time $t + h/2$. We now use the new value $\tilde{\xi}$ to compute the value of the derivative f at the midpoint (hence the name of this method), which we use to take a full Euler step,

$$\xi_{\rm mp}(t+h) = \xi(t) + hf[\tilde{\xi}]. \quad (8)$$

To convince ourselves that this method is indeed second order [that is, that $\xi_{\rm mp}(t+h) = \xi(t+h) + O(h^3)$], we insert Eq. (7) in Eq. (8),

$$\begin{aligned} \xi_{\rm mp}(t+h) &= \xi(t) + hf\left[\xi(t) + \frac{h}{2}f[\xi(t)]\right] = \xi(t) + h\left(f[\xi(t)] + \frac{h}{2}f[\xi(t)]\frac{df[\xi(t)]}{dx} + O(h^2)\right) \quad (9) \\ &= \xi(t) + hf[\xi(t)] + \frac{h^2}{2!}\frac{df[\xi(t)]}{dx}f[\xi(t)] + O(h^3) \quad (10) \\ &= \xi(t) + hf[\xi(t)] + \frac{h^2}{2!}\frac{df[\xi(t)]}{dt} + O(h^3), \quad (11) \end{aligned}$$

which, as you can see, reproduces the terms of Eq. (6) up to order $O(h^3)$.

Generalizations of this scheme, with more intermediate steps, make up the family of *Runge–Kutta* methods. Schemes with q intermediate steps are known as $(q+1)$-*stage* Runge–Kutta. The one-stage Runge–Kutta method is essentially the Euler method, while the two-stage Runge–Kutta methods are the midpoint method and its variants obtained by moving the intermediate point around. As for Euler, both explicit and implicit schemes are possible. It is not the case, in general, that a q-stage method will be p'th-order accurate: this ceases to be true for $q \geq 4$. So we shall hit the sweet spot and program an implementation of a fourth-order explicit Runge–Kutta method, with local error scaling as h^5.

A fourth-order Runge–Kutta method

Consider a system of first-order differential equations of the form given in Eq. (3) or, with slightly different notation,

$$\frac{d\boldsymbol{\xi}(t)}{dt} = \mathbf{f}(\boldsymbol{\xi}, t), \quad (12)$$

where bold variables represent N-dimensional vectors (so the vector $\boldsymbol{\xi}$ is the same as the vector $(\xi_1, \xi_2, \ldots, \xi_N)$ similar to that which we defined in Eq. (3)—we just use boldface vectors so that we don't need to write all those messy subscripts). For our implementation of the fourth-order Runge–Kutta scheme, we evaluate the Euler step $h \times \mathbf{f}$ four successive times: once at the beginning of the integration interval, twice at the midpoint of the interval (using two different estimates of $\boldsymbol{\xi}$), and once at the end of the interval:

$$\mathbf{k}_1 = h \times \mathbf{f}\left(\boldsymbol{\xi}(t), t\right) \tag{13}$$

$$\mathbf{k}_2 = h \times \mathbf{f}\left(\boldsymbol{\xi}(t) + \mathbf{k}_1/2, t + h/2\right) \tag{14}$$

$$\mathbf{k}_3 = h \times \mathbf{f}\left(\boldsymbol{\xi}(t) + \mathbf{k}_2/2, t + h/2\right) \tag{15}$$

$$\mathbf{k}_4 = h \times \mathbf{f}\left(\boldsymbol{\xi}(t) + \mathbf{k}_3, t + h\right). \tag{16}$$

We then combine these four evaluations to get the final composite step

$$\boldsymbol{\xi}_{\mathrm{RK4}}(t+h) = \boldsymbol{\xi}(t) + \frac{1}{6}\left\{\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right\}, \tag{17}$$

which has local error $O(h^5)$. This method is a standard way of integrating ODEs, but as it is written here it is still missing one crucial ingredient (*adaptive stepsize control*) that makes it really efficient. But that's the object of next week's assignment.

Runge–Kutta methods are not the final word on ODE integration. You should be aware that at least two other important classes of algorithms (*Bulirsch–Stoer* and *multistep*) have been studied extensively, and are available in common numerical libraries. If you wish, you can start reading about them in *Numerical Recipes*.

Assignment

1. Review the third assignment of Ph20, which dealt with simple techniques for solving ODEs.

2. Go back to the harmonic-oscillator system studied in Ph20.3 and compare (using simultaneous plots) the evolution of the global error for the explicit Euler and midpoint methods. Modify the code written for the Ph20.3 to implement the midpoint method; use the same h for both methods, setting it large enough that the error can grow appreciably within the time of integration.

3. Compare the scaling of the global error for these two methods when you move from a stepsize of h, to $h/2$, to $h/4$, to $h/8$, to $h/16$. This is known as a 'convergence plot'.

4. Write a general subroutine in Python implementing the fourth-order Runge–Kutta method described above. In particular, consider the set (12) of N first-order differential equations, and write function that takes the values $\boldsymbol{\xi}_{\mathrm{old}}$ of all your variables at time t, and returns their values $\boldsymbol{\xi}_{\mathrm{new}}$ at the time $t + h$, as computed using Eqs. (13)–(17). The function call should be similar to

   $$\boldsymbol{\xi}_{\mathrm{new}} = \texttt{rungekutta}(\boldsymbol{\xi}_{\mathrm{old}}, t, h, \texttt{func}),$$

   where $\boldsymbol{\xi}_{\mathrm{old}}$ and $\boldsymbol{\xi}_{\mathrm{new}}$ are vector-like objects (e.g. numpy arrays), t and h are respectively the initial time and the timestep for a single Runge–Kutta step (the values in Eqs. (13)–(17)), and $\mathbf{f} = \texttt{func}(\boldsymbol{\xi}, t)$ is a function that returns the derivatives $\mathbf{f}(\boldsymbol{\xi}, t)$.

The function `rungekutta` represents the *algorithm* routine in the nested program structure discussed in Ch. 16.0 of *Numerical Recipes*. You will also write a *stepper* function that calls `rungekutta` repeatedly, and some *driver* code that sets up your problem and calls the driver with the right parameters. Remember it is important for your sanity to keep the driver, stepper, and RK4 routines as separate subroutines—spaghetti code invites confusion and is a breeding ground for bugs.

5. Use your Runge–Kutta routine to solve the simplest problem of celestial mechanics: a small mass m (such as the Earth) moving in a central gravitational potential M/r (such as the Sun's). Setting for simplicity all the masses and Newton's gravitational constant G to 1, and using Cartesian coordinates x and y in the orbital plane, the equations of motion can be written as

$$x'(t) = v_x(t), \qquad v_x'(t) = -\frac{x(t)}{r(t)^3}, \tag{18}$$

$$y'(t) = v_y(t), \qquad v_y'(t) = -\frac{y(t)}{r(t)^3}, \tag{19}$$

where $r(t) = \sqrt{x^2(t) + y^2(t)}$. Choose initial conditions that represent a circular orbit (*Hint*: set the initial radius R and velocity v so that the centripetal acceleration v^2/R equals the gravitational force $1/R^2$). Plot the evolution of x against y to see if your Runge–Kutta integrator returns the expected orbit.

Symplectic Methods (Optional)

The midpoint and fourth-order Runge-Kutta methods we have seen so far belong to the same family of methods as the explicit Euler method we implemented in Ph 20. In Ph 20, we also ran into the *symplectic* Euler method, which was only a slight modification to the Euler method:

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h\mathbf{v}(t) \tag{20}$$

$$\mathbf{v}(t+h) = \mathbf{v}(t) + h\,\mathbf{a}(\mathbf{x}(t+h)), \tag{21}$$

where the acceleration $\mathbf{a}(\mathbf{x}) = -\mathbf{x}$ was a function solely of the position of the simple harmonic oscillator. This scheme had the useful property that it was fully time-reversible, and it conserved the area bounded by the trajectory of the simple harmonic oscillator in phase-space – we called it *symplectic*. It turned out that this allowed the method to conserve energy, on average: while there were regular fluctuations in the total energy, they always self-corrected and the total energy error remained within certain bounds. Runge-Kutta methods, on the other hand, do not respect the conserved quantities that we physicists so cherish: we can beat the error down with short timesteps and high-order schemes, but there is never a guarantee that quantities such as energy will not have arbitrarily-large long-term deviations. We would like to have methods that combine both symplecticity and higher-order accuracy. Such methods do exist.

The Leapfrog

The Leapfrog method is the workhorse of large N-body simulations. Prized for its robustness and simplicity, it is a second-order accurate symplectic method that is used in many massively-parallel cosmological simulation codes such as GADGET and GIZMO. A timestep in these codes has two

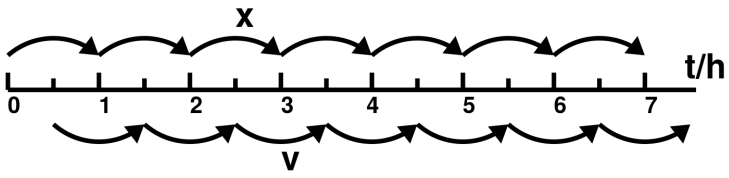

Figure 1: Schematic of the timestep structure of the Leapfrog scheme. Credit: Peter Young.

parts: a "drift" step and a "kick" step. The "drift" step updates the position based on the past velocity:

$$\mathbf{x}\left(t+h\right)=\mathbf{x}\left(t\right)+h\,\mathbf{v}\left(t+h/2\right). \tag{22}$$

Note that this is identical to Eq. 20 except that we are using the value of $\mathbf{v}$ at the *half timestep value*, $t+h/2$. The Leapfrog is so named because it is traditionally presented in a way where we *stagger* the positions and velocities in time, as two people playing leapfrog advance at staggered positions in space (See Fig. 1). After the drift, the kick updates the velocity to the next timestep-centered value:

$$\mathbf{v}\left(t+3h/2\right)=\mathbf{v}\left(t\right)+h\,\mathbf{a}\left(\mathbf{x}\left(t+h\right)\right). \tag{23}$$

It may be the case that we don't want to deal with this confusing business of time-centered velocity values. Luckily, there is a completely equivalent expression for the timestep that avoids half-timestep values:

$$\mathbf{x}\left(t+h\right)=\mathbf{x}\left(t\right)+h\,\mathbf{v}\left(t\right)+\frac{1}{2}h^{2}\,\mathbf{a}\left(\mathbf{x}\left(t\right)\right), \tag{24}$$

$$\mathbf{v}\left(t+h\right)=\mathbf{v}\left(t\right)+h\,\frac{\mathbf{a}\left(\mathbf{x}\left(t+h\right)\right)+\mathbf{a}\left(\mathbf{x}\left(t\right)\right)}{2}. \tag{25}$$

The Assignment (Optional)

1. Convince yourself that the timestep described by Eqs. 22-23 can be rewritten as Eqs. 24-25.

2. In the problem of orbital motion we solved in the first part of this assignment, there are four conserved quantities: the specific orbital energy $\mathcal{E}=\frac{1}{2}v^{2}-\frac{1}{r}$, and the components of the specific angular momentum $\mathbf{h}=\mathbf{x}\times\mathbf{v}$. Plot the error of these constants of motion versus time for a 4th-order Runge Kutta run that spans many orbits. What happens?

3. Write a stepper function that implements the condensed Leapfrog iteration in Eqs. 24-25. Repeat your orbit integration with this new stepper, and again look at the error in the constants of motion, plotting them on top of the Runge Kutta result. How does this new behaviour differ?

Beyond Leapfrog

As with Runge-Kutta methods, arbitrarily high-order symplectic methods can be constructed. In very large N-body simulations, approximations are used to reduce the amount of work required to sum the contributions of all particles to the force on a single particle (see Assignment 22.4); typical force errors are on the order of a few per cent. For this reason, there is little accuracy to be gained from going to higher-order schemes. However, in problems where the force can be computed

exactly, such as planetary systems or sufficiently-small star clusters, it is advantageous to use symplectic integrators of higher order. A good pedagogical resource on higher-order symplectic methods is The Art of Computational Science.