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

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 \( n \)th order derivatives. One useful trick is that a single \( n \)th 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

\[
\frac{d^2 x(t)}{dt^2} + q(t) \frac{dx(t)}{dt} = r(t)
\]

(1)
can be rewritten as two first-order equations by defining a new variable \( v(t) \):

\[
\frac{dx(t)}{dt} = v(t), \quad \frac{dv(t)}{dt} = r(t) - q(t)v(t).
\]

(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:

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

(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 \( \xi_1(t), \xi_2(t), \ldots, \xi_N(t) \), 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 straightforward initial-value problems: the boundary conditions are the specified values of all the \( \xi_i \) 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),

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

(4)

Here we use \( \xi(t + h) \) for the exact value of \( \xi \) at time \( t + h \) and we use \( \xi_{fe}(t + h) \) 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 $\Delta t$ 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 $\Delta t$. 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

$$\xi(t + h) = \xi(t) + h \frac{d\xi(t)}{dt} + \frac{h^2}{2!} \frac{d^2\xi(t)}{dt^2} + \ldots + \frac{h^p}{p!} \frac{d^p\xi(t)}{dt^p} + O(h^{p+1}) \quad (5)$$

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

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 $p$th-order 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 $\xi$ 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,

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

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

$$\xi_{mp}(t + h) = \xi(t) + hf \left[ \xi(t) + \frac{h}{2} f[\xi(t)] \right] = \xi(t) + h f[\xi(t)] + \frac{h^2}{2!} f[\xi(t)] \frac{df[\xi(t)]}{dx} + O(h^3) \quad (9)$$

$$= \xi(t) + hf[\xi(t)] + \frac{h^2}{2!} f[\xi(t)] \frac{df[\xi(t)]}{dx} + O(h^3) \quad (10)$$

$$= \xi(t) + hf[\xi(t)] + \frac{h^2}{2!} f[\xi(t)] \frac{df[\xi(t)]}{dx} + O(h^3), \quad (11)$$

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,

$$\frac{d\xi(t)}{dt} = f(\xi, t), \quad (12)$$
where bold variables represent $N$-dimensional vectors (so the vector $\mathbf{\xi}$ 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 $\mathbf{\xi}$), and once at the end of the interval:

$$k_1 = h \times \mathbf{f}(\mathbf{\xi}(t), t) \quad (13)$$
$$k_2 = h \times \mathbf{f}(\mathbf{\xi}(t) + k_1/2, t + h/2) \quad (14)$$
$$k_3 = h \times \mathbf{f}(\mathbf{\xi}(t) + k_2/2, t + h/2) \quad (15)$$
$$k_4 = h \times \mathbf{f}(\mathbf{\xi}(t) + k_3, t + h). \quad (16)$$

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

$$\mathbf{\xi}_{RK4}(t + h) = \mathbf{\xi}(t) + \frac{1}{6} \{k_1 + 2k_2 + 2k_3 + k_4\}, \quad (17)$$

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 of $N$ first-order differential equations, and write function that takes the values $\mathbf{\xi}_{old}$ of all your variables at time $t$, and returns their values $\mathbf{\xi}_{new}$ at the time $t + h$, as computed using Eqs. (13)–(17). The function call should be similar to

$$\mathbf{\xi}_{new} = \text{rungekutta}(\mathbf{\xi}_{old}, t, h, \text{func}),$$

where $\mathbf{\xi}_{old}$ and $\mathbf{\xi}_{new}$ are vector objects (see the Python hints below), $t$ and $h$ are respectively the initial time and the timestep for a single Runge–Kutta step (the values in Eqs. (13)–(17)), and $\mathbf{f} = \text{func}(\mathbf{\xi}, t)$ is a function that returns the derivatives $\mathbf{f}(\mathbf{\xi}, t)$.
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

\[
\begin{align*}
  x'(t) &= v_x(t), & v'_x(t) &= -\frac{x(t)}{r(t)^3}, \\
  y'(t) &= v_y(t), & v'_y(t) &= -\frac{y(t)}{r(t)^3},
\end{align*}
\]

(18)

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.

**Python hints**

We have already covered passing Python functions as arguments in the fourth week of classes. The other ingredient that you need for this week’s assignment is a home-grown `Vector` class, which you will use to hold the ODE variables \( \xi \) and to perform arithmetic on them.

We start by noticing that the built-in `list` object is almost what we need: it can be initialized as in

```python
>>> x = [1.0, 2.0, 3.0, 4.0],
```

and its items can be accessed as `x[0]`, `x[1]`, and so on. Lists can also be `printed` easily. Unfortunately, `x + [5.0, 6.0, 7.0, 8.0]` and `2 * x` do not do what we expect for a vector, because they perform list concatenation (try them out!).

However, it is not hard to write functions that sum two lists element-by-element, or multiply them by a scalar. For instance,

```python
def vectoradd(x,y):
    s = []
    for i in range(0,len(a)):
        s.append(x[i] + y[i])
    return s
```

or rather more succinctly and elegantly (Python code should always be beautiful),

```python
def vectoradd(x,y):
    return map(lambda a,b : a+b,x,y)
```
The trick is now to write our `Vector` class as a derived class of `list` that overloads (or defines) the arithmetic operators that we require. You have learned about Python classes in this week’s reading.

```python
class Vector(list):
    """A list-based vector class""

def __add__(self, other):
    """Element-by-element addition, or addition of constant""

    try:
        return Vector(map(lambda x,y: x+y, self, other))
    except TypeError:
        return Vector(map(lambda x: x+other, self))

[...]
```

With this definition, you can create `Vector` objects as in

```python
>>> x = Vector([1.0, 2.0, 3.0])
>>> y = Vector([4.0, 5.0, 6.0])
```

and get the correct addition behavior

```python
>>> x + y
[5.0, 7.0, 9.0]
```

In the partial definition of `Vector` given above, the `try/except` construct is needed because `__add__` could be called for either the addition of two `Vectors`, or for the addition of a `Vector` and a single real number, in which case the `TypeError` exception is generated when `map` tries to apply the indexing operator `[]` to the number. The enclosing `Vector()` is needed because `map` would otherwise return a simple list.

Other operators that you might want to implement (filling in the `...` above) are `__neg__` (unary negation), `__sub__` (subtraction), `__mul__` (multiplication), `__div__` (division), `__radd__`, `__rsub__`, `__rmul__`, `__rdiv__` (addition, subtraction, multiplication, and division from the right), and perhaps others.

Have fun building up your `Vector` class with all the facilities that you think can be useful. Save it in a separate file (say, `Vector.py`), and import it with `from Vector import *`. We shall reuse it extensively in the coming weeks.