Simple-ODEs

December 9, 2024

Supporting textbook chapters for week 6: Chapters 8.1, 8.2, 8.5.1 to 8.5.3

- Euler method
- Runge-Kutta methods
- Leapfrog Methods
  - Bulirsch-Stoer

Consider a first-order ODE that is a function of only one variable, with an initial condition(s). $\frac{\mathrm{d}x}{\mathrm{d}t} = f(x,t) \quad \text{with} \quad x(t=0) = x_0$. These equations can be impossible to solve analytically, but easy to solve on a computer.

Later in the course (not this week), we'll do:

- nD: $\quad \dfrac{\mathrm{d}x_i}{\mathrm{d}t} = f_i(x_1, \ldots, x_n, t) \quad \text{with} \quad x_i(t=0) = x_{i0}$.
- higher order, e.g.:
$$\frac{\mathrm{d}^3 x}{\mathrm{d}t^3} = f(x,t) \quad \Leftrightarrow \quad \frac{\mathrm{d}x}{\mathrm{d}t} = v, \ \frac{\mathrm{d}v}{\mathrm{d}t} = a, \ \frac{\mathrm{d}a}{\mathrm{d}t} = f.$$
- coupled sets of DEs

1 SciPy Built-in Solvers

- SciPy has built-in ODE solvers called `odeint` (older) and `solve_ivp` (preferred) located in the `scipy.integrate` module. https://docs.scipy.org/doc/scipy/tutorial/integrate.html
- `odeint` is very powerful, but lots of hidden automatic black-box workings make it difficult to track what is actually going on (e.g. to get an error estimate).
- `solve_ivp` allows more user control and tunable knobs.
- If you don't need an error estimate for your specific application, then just use `odeint` or `solve_ivp` and hope the computation time is acceptable. However, if it does matter, then you can write your own ODE solver

2 Euler Method

- Start at $t = t_0, \ x = x_0$
- Discretize into time-steps t_i with constant spacing h
- At each time-step, find x, using x at the previus time-step and f: \$ x_{i} = x_{i-1} + hf(x_i) \$
- The Euler method has error $O(h^2)$ at each step

- integrating across the whole interval: global error is $O(h)$ (see eqn 8.8, p. 331 in textbook):

$$\text{Taylor expansion} \Rightarrow x(t+h) = x(t) + h\frac{dx}{dt} + \overbrace{\frac{h^2}{2}\frac{d^2x}{dt^2}}^{\epsilon} + O(h^3)$$

$$\sum \epsilon = \sum_{k=0}^{N-1} \frac{h^2}{2}\left.\frac{d^2x}{dt^2}\right|_{x_k,t_k} = \frac{h}{2}\sum_{k=0}^{N-1} h\left.\frac{df}{dt}\right|_{x_k,t_k} \approx \frac{h}{2}\int_a^b \frac{df}{dt}dt = \frac{h}{2}\left[f_b - f_a\right]$$

- For some applications, this is good enough. But for others, we need to do better!

3 Runge-Kutta (RK) Methods

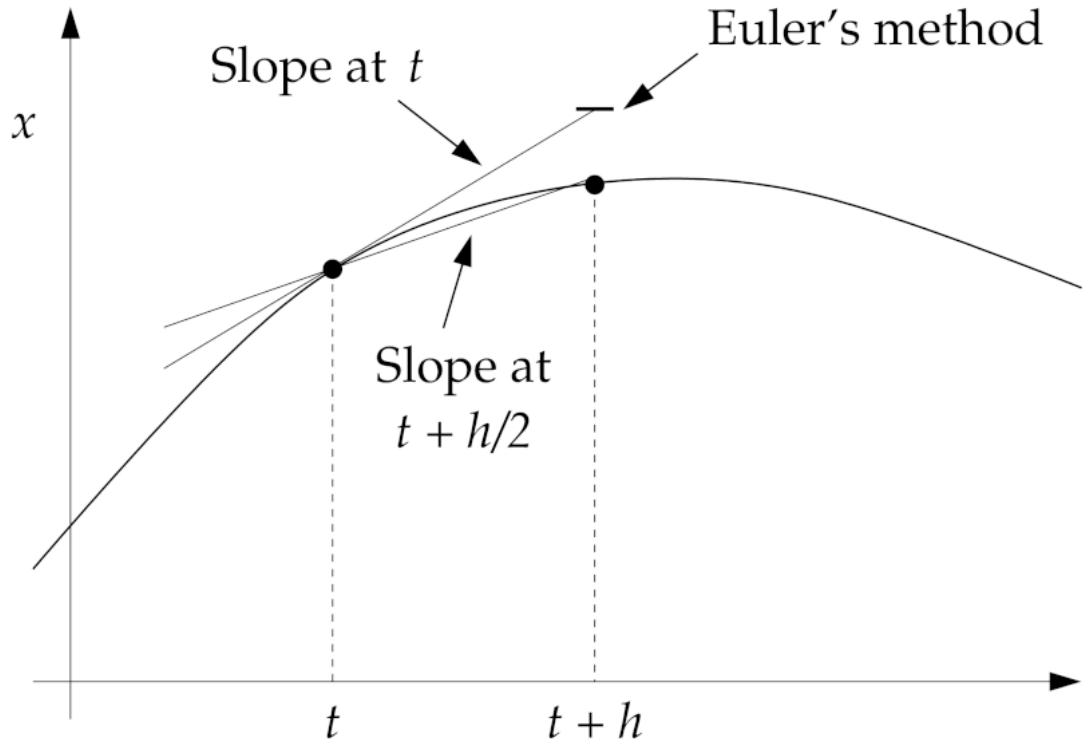
3.1 2nd-order (RK2)

- Use the middle point $t + h/2$ and evaluate with Euler's method,

$$x\left(t + \frac{h}{2}\right) \approx x(t) + \frac{h}{2}f[x(t), t]$$

- Slope at $t + \frac{h}{2} \approx f\left[x(t) + \frac{h}{2}f\left[x(t), t\right], t + \frac{h}{2}\right]$

$$\Rightarrow \boxed{x(t+h) = x(t) + hf\left[x(t) + \frac{h}{2}f[x(t), t], t + \frac{h}{2}\right]}$$



RK2 usually coded by defining intermediate quantities:

- $k_1 = hf(x, t)$ as preliminary step before $x(t + h/2)$,

- $k_2 = hf\left(x + \frac{k_1}{2}, t + \frac{h}{2}\right)$,
- $x(t+h) = x(t) + k_2$.

Note: $O(h^3)$ error at each step, usually $O(h^2)$ global error.

Coding Euler:

[ ]:
```
for t in tpoints:
    x += h*f(x, t)
```

Coding RK2:

[ ]:
```
for t in tpoints:
    k1 = h*f(x, t)
    k2 = h*f(x + 0.5*k1, t+0.5*h)
    x += k2
```

3.2 4th-order (RK4)

- Various Taylor expansions at various points in the interval ⇒ higher-order RK's.
- RK4 is reasonable to code yourself. Higher-order methods ⇒ use canned routines.
- End result, after tedious algebra:
  1. $k_1 = hf(x, t)$,
  2. $k_2 = hf\left(x + \frac{k_1}{2}, t + \frac{h}{2}\right)$,
  3. $k_3 = hf\left(x + \frac{k_2}{2}, t + \frac{h}{2}\right)$,
  4. $k_4 = hf\left(x + k_3, t + h\right)$,
  5. $x(t+h) = x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$.

Coding RK4:

[ ]:
```
for t in tpoints:
    k1 = h*f(x, t)
    k2 = h*f(x+0.5*k1, t+0.5*h)
    k3 = h*f(x+0.5*k2, t+0.5*h)
    k4 = h*f(x+k3, t+h)
    x += (k1 + 2*k2 + 2*k3 + k4)/6
```

3.2.1 Error Estimation

- Very accurate method: error is $\epsilon = ch^5$ at each time step h, c constant (order h^4 globally).
- Error after 2 time steps: $\approx 2ch^5$.
- Error after 1 time step of $2h$: $\approx c(2h)^5 = 32ch^5 \gg 2ch^5$
- The difference is $(32-2)ch^5 = 30ch^5 = 30\epsilon$.
- To estimate error: run ODE solver twice with h (to get x_1), once with $2h$ (to get x_2), divide difference by 30.

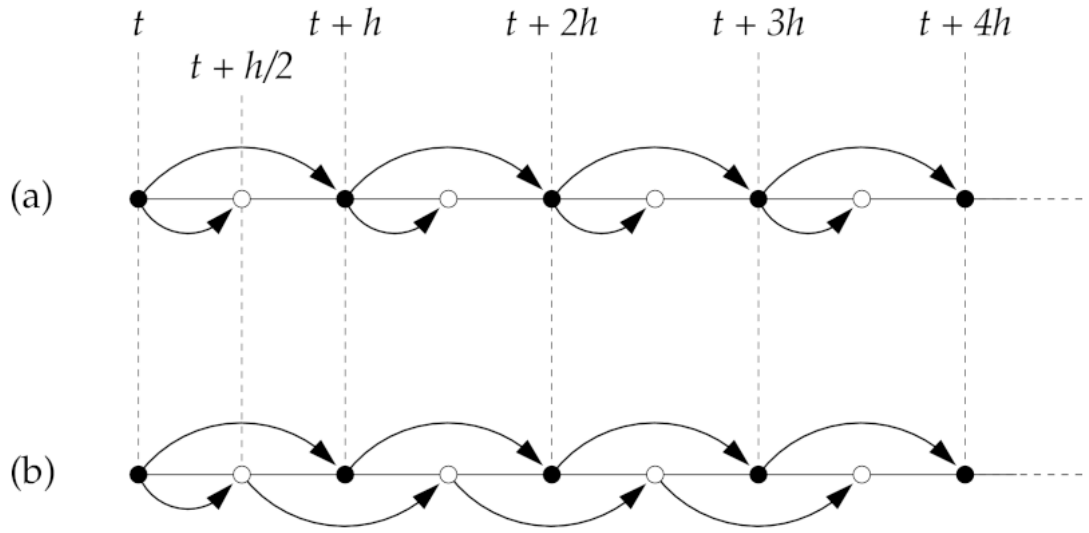
$$\epsilon = ch^5 = \frac{1}{30}(x_1 - x_2).$$

3.2.2 Adaptive time stepping

- Suppose target error is δ *per unit time* (physical time t, not computational step).
- If
$$\rho = \frac{h\delta}{\epsilon} = \frac{30h\delta}{|x_1 - x_2|} = \frac{30\delta}{ch^4} > 1,$$
then h is too small (as in, could be bigger, saving computational resources while still reaching target accuracy) and can be adjusted to $h' = h\rho^{1/4}$ to get $\rho' = 1$.
  - Still achieves target error, which is $h'\delta$ for step of size h'.
  - Saves calculation time.
- If $\rho < 1$, the time step is too large and needs to be adjusted down by the same factor.
  - We also need to repeat our calculation to get the desired accuracy.
  - This will guarantee meeting error target.
- We test if we need to adjust by performing the calculation twice (we retrieve x_1 and x_2), testing if we met the target, and adjusting h.
- Overall, despite extra work (up to 3 RK4 steps per time step), program often faster because resolution focused where it's needed.

4 Leapfrog methods

- RK2: Use mid-point location to jump to $t + h$
$$x(t+h) = x(t) + hf\left[x + \frac{h}{2}f(x,t), t + \frac{h}{2}\right]$$
- Leapfrog: use each point as a mid-point.
$$x(t+h) = x(t) + hf\left[x + \frac{h}{2}f(x,t), t + \frac{h}{2}\right],$$
$$x\left(t + \frac{3}{2}h\right) = x\left(t + \frac{h}{2}\right) + hf[x(t+h), t+h].$$
  - Requires Euler for the first half-step, from t to $t + h/2$



- It is **time-reversible!** Just take the 'reverse path' to 'retrace your steps' backwards
  - But the time-step has to be constant

4.1 Error Estimation

- Time-reversible ⇒ error ϵ is an **odd** function of h:

  $$\epsilon(-h) = -\epsilon(h)$$

  ⇒ Taylor expansion is made of **odd** powers of h,

  $$\epsilon(h) = c_3 h^3 + c_5 h^5 + ...$$

  ⇒ cumulative error is **even** in h.
- Simplest implementation, based on RK2, has $O(h^2)$ global error
- Each improvement we apply would gain two orders of accuracy, if we could first eliminate all even powers in ϵ due to first 1/2 step.

4.2 Modified Midpoint Method

Do the Leapfrog method, then do **both** the whole integer **and** the forward Euler 1/2-step.

$$\begin{aligned} x_{n-1/2} &= x_{n-3/2} + hf(x_{n-1}, t + H - h), \\ x_n &= x_{n-1} + hf(x_{n-1/2}, t + H - h/2) \approx x(t + H) \\ x'_n &= x_{n-1/2} + hf(x_n, t + H) \approx x(t + H) \end{aligned}$$

Then do the following adjustment:

$$x(t + H)_{final} = \frac{x_n + x'_n}{2}$$

… and you have canceled the even powers (MMP method).

This is not a trivial result (cf. Gragg 1965 for proof; https://doi.org/10.1137/0702030 PDF posted on Quercus if you're curious).

4.3 Bulirsch-Stoer method

MMP method rarely used by itself, but is the basis for the powerful Bulirsch-Stoer (don't abbreviate this…) method: * Take 1 single MMP step of size $h_1 = H$ to get estimate

$$x(t + H) = R_{1,1}.$$

* R stands for "Richardson extrapolation", remember this from Romberg integration? * Now take 2 MMP steps of size $h_2 = H/2$ to get second estimate of

$$x(t + H) = R_{2,1}.$$

- Since we know the MMP has 2nd order and even total error, we can write both of these estimates as

  $$\begin{aligned} x(t + H) &= R_{1,1} + c_1 h_1^2 + O(h_1^4) \quad \text{and} \\ x(t + H) &= R_{2,1} + c_1 h_2^2 + O(h_2^4). \end{aligned}$$

- Using the relationship between the step sizes: $h_1 = 2h_2$, we can equate these expressions to get

$$R_{1,1} + 4c_1h_2^2 + O(h_2^4) = R_{2,1} + c_1h_2^2 + O(h_2^4)$$

$$\Rightarrow \; c_1h_2^2 = \frac{1}{3}(R_{2,1} - R_{1,1}) + O(h_2^4).$$

- If we plug this back in to the expression for $x(t+H)$ above we get a new estimate called $R_{2,2}$:

$$x(t+H) \approx R_{2,2} + \boxed{O(h_2^4)}$$

$$x(t+H) = \underbrace{R_{2,1} + \frac{1}{3}(R_{2,1} - R_{1,1})}_{R_{2,2}} + \boxed{O(h_2^4)}.$$

* 2 different grid spacings (H and $H/2$) $\rightarrow$ expression for the leading error term $\rightarrow$ replace it with our estimates for these grid spacings, i.e., $R_{1,1}$ and $R_{2,1}$. * We have reduced the error in our estimate by 2 orders! *(which was possible because the errors were even)*

You can keep going! The power in this method is that you keep cancelling 2 powers in the error for every new grid spacing you consider. * Continue to refine grid to find new estimates and error estimates, and stop when error is acceptable

- Error:

$$x(t+H) = R_{n,m+1} + O(h_n^{2m+2})$$

- Recursion relation:

$$R_{n,m+1} = R_{n,m} + \frac{R_{n,m} - R_{n-1,m}}{[n/(n-1)]^{2m} - 1} \quad \text{and} \quad h_n = \left(\frac{n-1}{n}\right) h_{n-1}.$$

Can make a similar extrapolation table as for Romberg Integration, so you only need to use MMP (for calculating $R_{m,1}$) and the recursion relation:

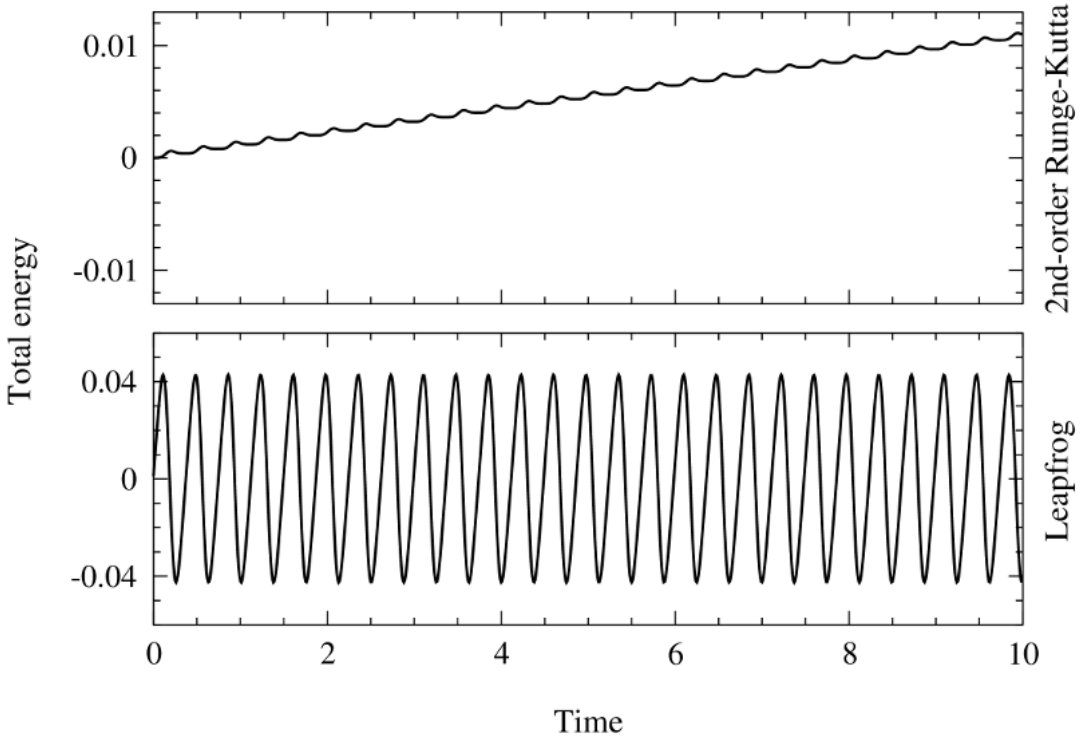
$$\begin{array}{lllll}
n=1: R_{1,1} & & & & \\
 & \searrow & & & \\
n=2: R_{2,1} & \rightarrow R_{2,2} & & & \\
 & \searrow & \searrow & & \\
n=3: R_{3,1} & \rightarrow R_{3,2} & \rightarrow R_{3,3} & & \\
 & \searrow & \searrow & \searrow & \\
n=4: \underbrace{R_{4,1}}_{MMP} & \rightarrow R_{4,2} & \rightarrow R_{4,3} & \rightarrow R_{4,4} &
\end{array}$$

4.4 More on time-reversal

Many physics problems involve systems that are symmetric or invariant in time (related to conservation of energy). So when modeling these systems, often want a solution that can be run 'backwards in time'

Time shift/reversal doesn't work with RK

- e.g. for RK2, everything "resets" at $t + h$, so the info at the mid-point is lost and the reverse path is not a "retracing of the steps".
- Graphically, reverse is not just taking the same points and drawing the arrows backwards. The positions of the points change, depending if you start at the beginning and work forwards or start at the end and work backwards!
- A manifestation of this: solution 'drifts' with time, see example below (from model of a non-linear pendulum system)



5 Pros and Cons of each method

RK2: * ⊕ Easily extended to RK4 * ⊕ Possible to use adaptive time step * ⊖ not time-reversible * ⊖ not great accuracy

RK4: * ⊕ good accuracy * ⊕ Possible to use adaptive time step * ⊖ not time-reversible

Leapfrog:

- ⊕ time-reversible
- ⊕ can be extended to higher-order methods
- ⊖ not great accuracy
- ⊖ time step has to be constant (not adaptive)

Bulirsch-Stoer:

- ⊕ time-reversible
- ⊕ accuracy can be improved until desired level if reached
- ⊖ time step has to be constant (not adaptive)

[ ]: