ODEs II

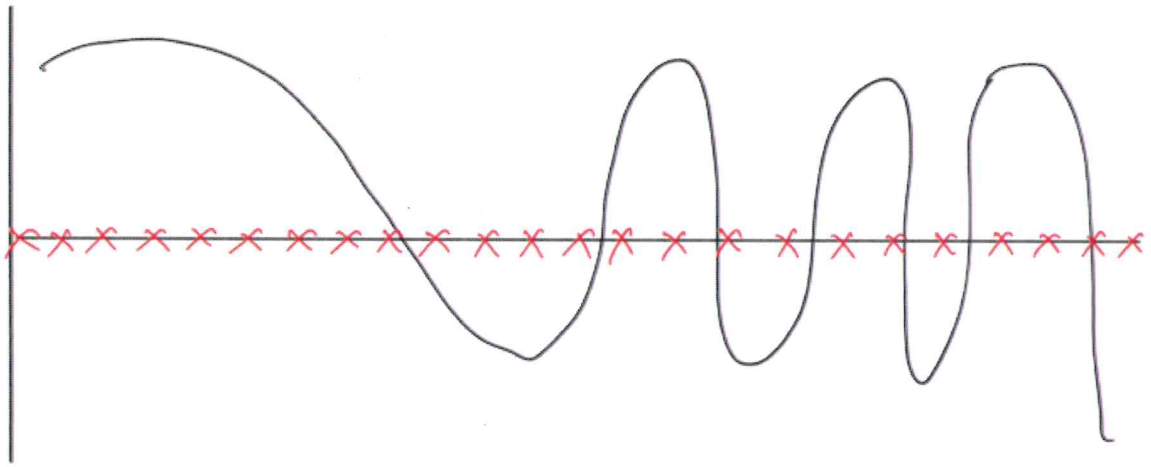
I. Adaptive stepsize control

Problem:

$$\frac{dy}{dx} = f(x, y)$$

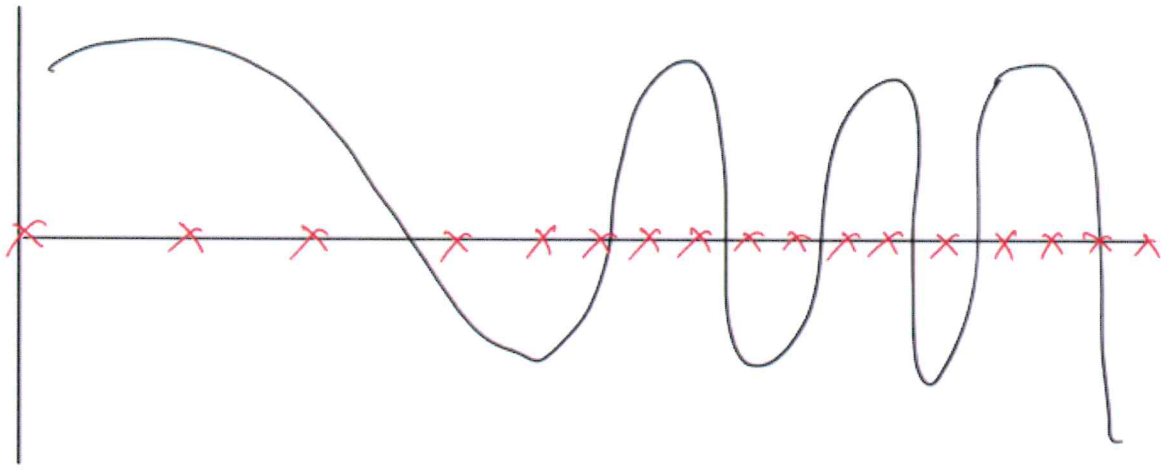
$$y = y_0 \quad \text{at} \quad x = 0$$

Let's say solution looks like



Uniform sampling can be wasteful!
We may want fewer steps in the low-frequency part and more steps in the high-frequency part.

Can we learn and adapt stepsize?



Example: Inspiralling black holes.

As one approaches the merger, everything becomes harder: faster dynamics, stronger fields, stronger radiation, etc.

Use adaptive steps to take giant leaps in early inspiral and small steps near the merger.

Goal: Adapt step size to achieve a given tolerance, tol.

RK4 Recap

Let's consider RK4 : 4 evals per step

$$k_1 = h f(x_n, y_n)$$

$$k_2 = h f\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = h f\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = h f(x_n + h, y_n + k_3)$$

$$y(x+h) = y(x) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)$$

Notation : $y(x)$: Numerical estimate of solution at x.

Also used, $y_n = y(x_n)$; $y_{n+1} = y(x_{n+1})$

$\hat{y}(x)$: Exact solution at x.

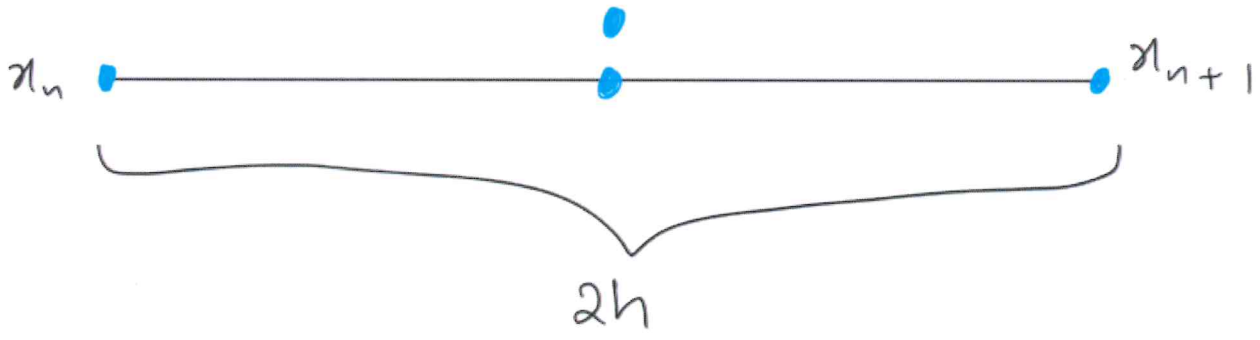
So, for RK4 : $\hat{y}(x) = y(x) + \lambda h^5 + O(h^6)$

To order h^5, λ is a constant.

(λ is related to $\frac{y^{(5)}(x)}{5!}$)

Ia. Step doubling

One big step : 4 evals.

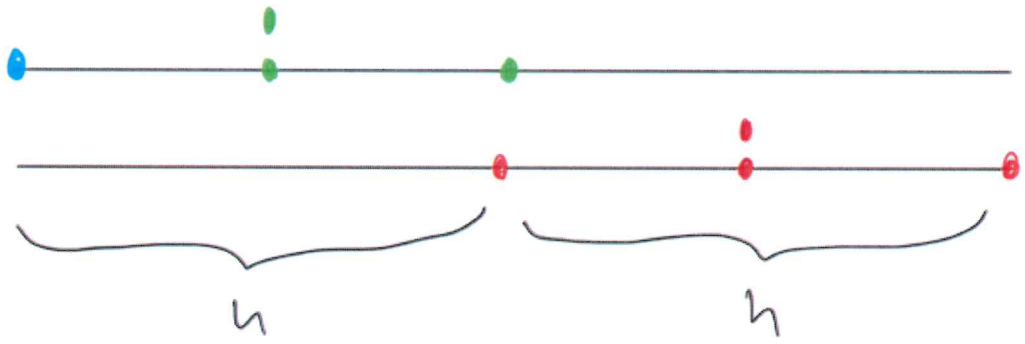


$$\hat{y}(x+2h) = y^{BS}(x+2h) + \lambda (2h)^5 + O(h^6)$$

$\downarrow$

Numerical solution from one Big Step

And two small steps: Only 7 new evals.



$$\hat{y}(x+2h) = y^{ss}(x+2h) + 2\lambda(h^5) + O(h^6)$$

Cost overhead

Total evals: 11

Overhead cost: factor of $\frac{11}{8} = 1.375$

8, because we get the accuracy of the small step.

What do we gain?

Error estimate

$$err = \left| y^{BS}(x+2h) - y^{SS}(x+2h) \right| \quad \text{---(1)}$$

Adjust h, so that err is below the target tolerance tol.

$$err = \left| 2\lambda(h^5) - \lambda(2h)^5 \right|$$

$$err \propto h^5$$

Simple update

Use $err < tol$

$$h_{new} = 0.8\, h \left(\frac{tol}{err} \right)^{1/5}$$

0.8 ↙ extra safety factor

atol and rtol

$$scale = atol + |y| rtol$$

- atol: absolute tolerance
- $|y|$: can be $\max(y(x), y(x+h))$
- rtol: relative tolerance

Use $err < scale$

$$h_{new} = 0.8\, h \left(\frac{scale}{err} \right)^{1/5}$$

General recommendation: atol = rtol = tol

If set of equations $\begin{pmatrix} \dot{y}_1 \\ \dot{y}_2 \end{pmatrix} = \begin{pmatrix} f_1(x, y_1, y_2) \\ f_2(x, y_1, y_2) \end{pmatrix}$ have wildly different scales for $|y_1|$ & $|y_2|$, you could try atol = 0, rtol = tol.

Strategy

Adjust h after each step.

If err is too big, step is rejected, h decreases for the next trial.

Keep in mind, this bounds local error. If you want to bound the global error, you may need to tighten the tolerance, or the safety factor.

1.6. Embedded methods

Can we be smarter?

Embedded RK methods: Pair of RK methods which share most of the function evaluations.

Example 1: RK2 + Forward Euler

$$K_1 = h f(x_n, y_n)$$

$$K_2 = h f\left(x_n + \frac{h}{2}, y_n + \frac{K_1}{2}\right)$$

$$y_{n+1}^{RK2} = y_n + K_2 + O(h^3)$$

$$y_{n+1}^{FE} = y_n + K_1 + O(h^2)$$

No new evaluations!

$$err = \left| y_{n+1}^{RK2} - y_{n+1}^{FE} \right| + O(h^2)$$

$$h_{new} = 0.8\, h \left(\frac{tol}{err} \right)^{1/2}$$

Example 2:

RK4 and RK2 share k_1 and k_2, so with no new evaluations, you get

$$err = \left| y^{RK2} - y^{RK4} \right| + O(h^3)$$

$$h_{new} = 0.8\, h \left(\frac{tol}{err} \right)^{1/3}$$

Smarter examples (Recommended)

1) Fehlberg and later Dormand-Prince

RK5(4): 5^{th} order RK, with embedded 4^{th} order RK.

2) DoPr853: 8^{th} order RK, with embedded 5^{th} and 3^{rd} order RK, combined to get $\sim 8^{th}$ order error estimate.

Ic. Dense output

What if you want uniform time steps? Let's say to take an FFT.

Bad idea: Force adaptive stepper to include the required time steps. This is still wasteful, goes against the spirit of adaptive time steps!

Solution: Dense output, or, interpolate using an interpolation scheme of comparable order to ODE stepper. Embedded interpolation schemes are generally used.

RK5(4) uses 4^{th} order accurate interpolation, with no new evaluations. Sufficient because global error for RK5 is 4^{th} order.
Similarly, DoPr853 uses 7^{th} order accurate interpolation

See Python demo: ODE2_adaptive.ipynb

II. Stiff Equations

Typically, a set of equations where the dependent variables are changing on wildly different time scales.

Example: Common in models for vibrations and chemical reactions.

↓

The name "stiff" comes from large spring constants.

Nuclear reactions with very different half-lifes for different components.

Also show up when computing neutrino emission in NR simulations with neutron stars.

Prototype

$$u' = 998u + 1998v$$

$$v' = -999u - 1999v$$

$$u(0) = 1 \qquad v(0) = 0$$

Solution

$$u = 2e^{-x} + e^{-1000x}$$

$$v = -e^{-x} + e^{-1000x}$$

e^{-x}: steady state

e^{-1000x}: quickly decaying transient

Explicit methods need $h \sim 1/1000$ for stability, even after the transient dies out! See demo: ODE2_stiff.ipynb

II a : Explicit vs Implicit methods

Simplify, take the scalar case

$$y' = -cy \; ; \; y(0) = 0 \; ; \quad c > 0, \qquad \text{---} (1)$$

Solution : $y = e^{-cx}$

II a1 : Forward Euler : An explicit method

$$y_{n+1} = y_n + h\, y_n' + O(h^2)$$

Explicit, because y_{n+1} is given explicitly in terms of the previous step's values, y_n & y_n'.

For (1),

$$y_{n+1} = y_n - chy_n = (1 - ch)y_n$$

If step size $h > 2/c$,

$$|y_n| \longrightarrow \infty, \text{ as } n \longrightarrow \infty.$$

Even though solution goes to 0.

General problem for explicit methods: need small steps for stability, even after the transient dies.

Idea 2: Backward Euler: An implicit method

$$y_{n+1} = y_n + h\, y'_{n+1} + O(h^2)$$

Implicit, because y_{n+1} depends on terms at the current step, y'_{n+1}.

Generally, one needs to do an additional matrix inversion at each time step to get y_{n+1}.

For ①,

$$y_{n+1} = y_n - c\,h\,y_{n+1}$$

$$y_{n+1} = \frac{y_n}{1 + c\,h}$$

$y_{n+1} \to 0$, as $n \to \infty$, independent of h,

"unconditionally stable".

II. More general problems

I.b.1. General linear systems

$$\underline{y}' = -\underline{\underline{C}} \cdot \underline{y}$$

- $\underline{y}'$: vector of size n
- $\underline{\underline{C}}$: $n \times n$ matrix of constants
- $\underline{y}$: vector of size n

Looking for stiffness

Find Eigen values of C : $\lambda_0, \ldots, \lambda_{n-1}$.

if $\dfrac{\max_n \mathrm{Re}(\lambda_n)}{\min_n \mathrm{Re}(\lambda_n)}$ is large,

you may have a stiff problem.

(Not easy to define stiffness)

Explicit methods might need

$$h < \frac{2}{\max_n \mathrm{Re}(\lambda_n)}$$

Implicit methods are recommended in this case. Not all implicit methods are stable, but the generally available ones, like "Radau" in scipy, are stable. See python demo ODE2_stiff.ipynb.

For general problems: Start with explicit methods, look for abnormally small time steps (i.e. use adaptive stepper). Also look for ustable jumps in the solution, as we see in the Python demo.

I L 2 Nonlinear problems

$$\underline{y}' = \underline{f}(\underline{y}, x)$$

Semi-implicit methods: Linearize and apply same methods as above. Needs extra iterations of Newton's method and a good initial guess, on top of matrix inversion at each time step.

See Chapter 17, Sec 5 of Numerical Recipes (3rd Ed).

Higher order methods

Generalizations of RK or Richardson extrapolation methods are generally favored.

Implicit-Explicit methods

Try to get the best of both worlds.

Example: Additive RK. See arxiv:0808.2597

III Boundary value problems

Consider, $\underline{y}' = \underline{f}(x, \underline{y})$; $x \in [x_0, x_f]$

For example, $y'' + y = 0$; $x \in [0, 1]$

(use $v = y$, $v = y'$ to reduce to 1st order)

Initial value problem (IVP)

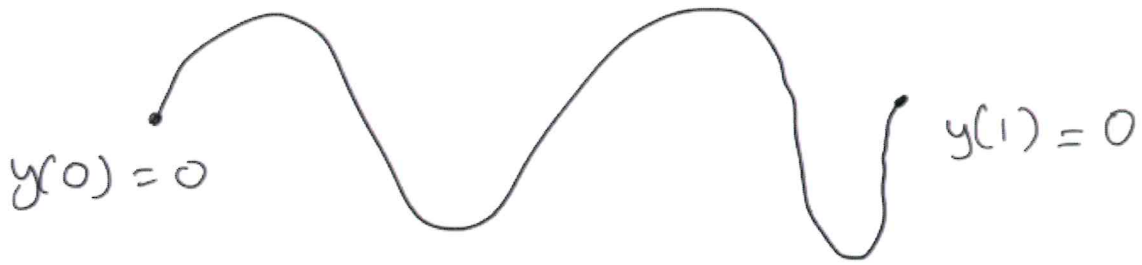
$$y(0) = y'(0) = 0$$

Boundary value problem (BVP)

$$y(0) = y(1) = 0$$

"two-point boundary value problem"

Example 1: String pinned at both ends.



Example 2: TOV equations, for static star

$P(r=0) = P_c$ central pressure

$P(r=R) = 0$

II a Shooting method

Take same prototype,

$$y'' + y = 0 \; ; \; y(0) = y(1) = 0.$$

Use $u = y$, $v = y'$.

$$\left.\begin{aligned} v' + u &= 0 \\ u' &= v \end{aligned}\right| \quad \begin{aligned} v(0) &= 0 \\ u(1) &= 0 \end{aligned} \qquad \text{—— (2)}$$

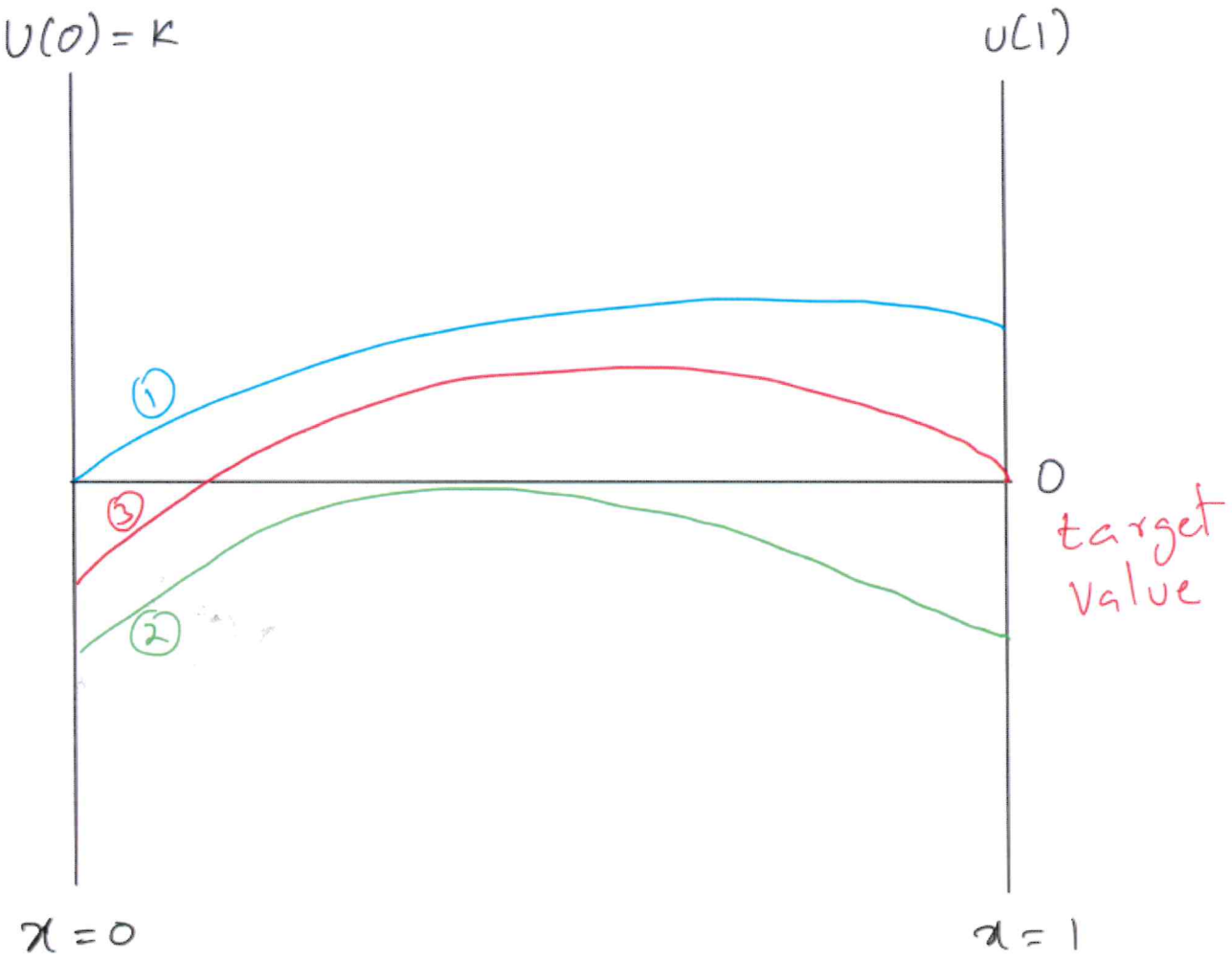
Strategy:

Set $u(0) = k$,

Solve (2) like an IVP,

$$err(k) = | \underbrace{u_k(1)}_{\text{solution at } x=1 \text{ for } u(0)=k.} - 0 |, \qquad u(1)$$

find roots of $err(k)$.



III b: Relaxation

Represent ODE on a finite difference grid, iterate to satisfy ODE & BC at the same time.

IV Recipes

If IVP,

Start with RK method with adaptive stepper, like RK45 or DOPR853 in scipy.

Higher order → big steps, if solution is smooth.

If not smooth, try reducing order.

If stiff problem,

Use implicit method.

Start with RK methods, like Radau in scipy.

If BVP,

Shoot first, relax later.
Relaxation may be better for smooth problems.

Most important: Experiment, and

don't sue me!