Ordinary Differential Equations – Adaptive Time-Stepping

CS370 Lecture 16 – Feb 8, 2017

Time Step Control

Time stepping methods advance by a discrete time step size h.

So far, we have held h as a prescribed constant for a whole calculation.

Since our error was $O(h^p)$ for some p, depending on the scheme, smaller h $(\rightarrow 0)$ implied smaller error (both local and global).

Time Step Tradeoffs

If small step h yields small error, is smaller always "better" for practical calculations?

No! Smaller h requires more steps to reach a given time t_{final},

$$\#Steps = \frac{t_{final} - t_0}{h} = O(h^{-1}),$$

implying a **higher computational cost**.

But too large h introduces excessive error! (Not to mention stability issues for some schemes…)

Time Step Tradeoffs

$h \to 0$: many steps, computationally expensive, more accurate.

$h \to \text{large}$: fewer steps, so cheaper, but also less accurate.

$\therefore$ should minimize effort/cost by choosing largest h such that error is less than a desired error tolerance:

$$error < tolerance.$$

Can we work out an ideal constant h for a given problem?

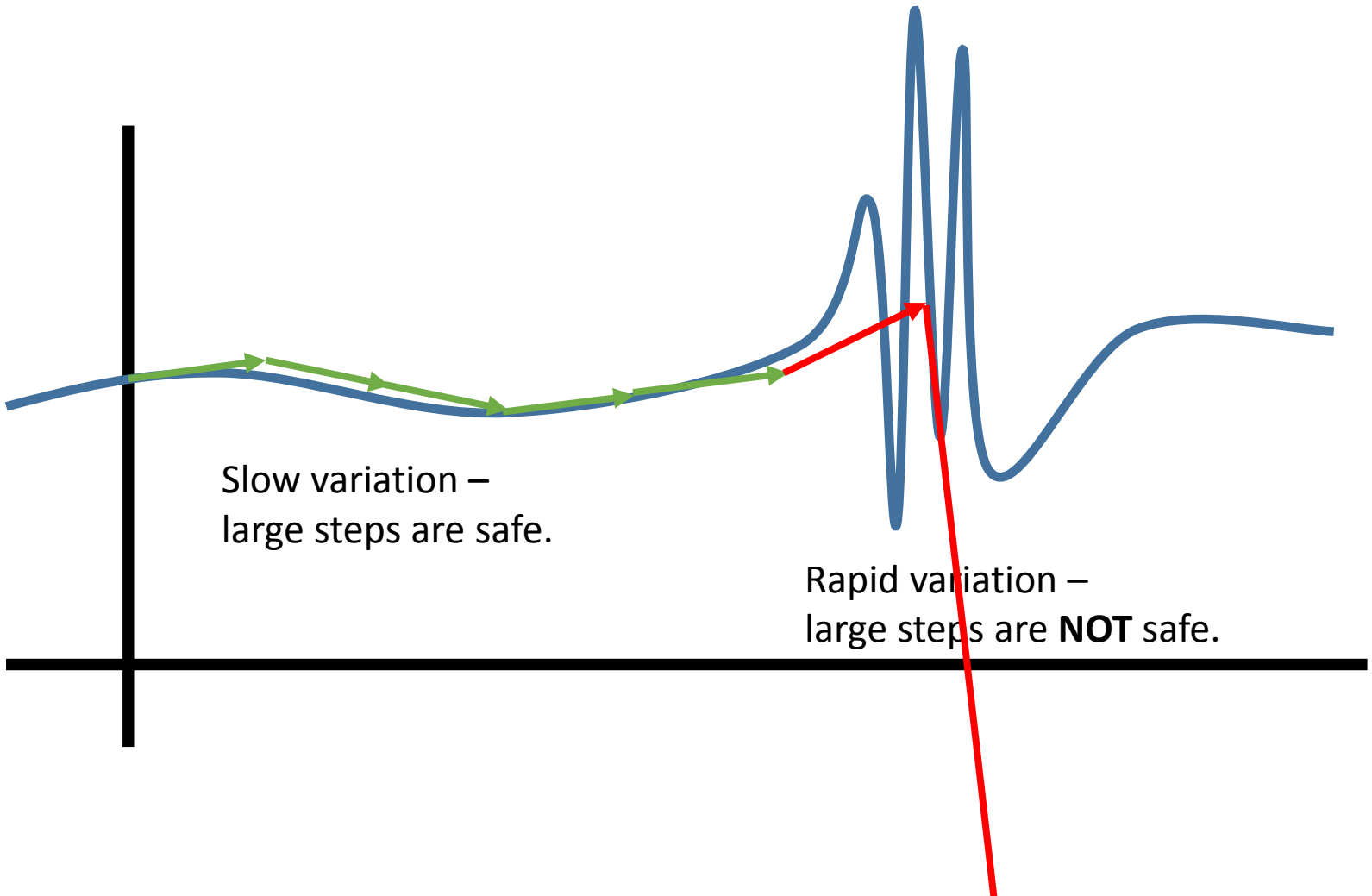
Not necessarily a good idea…

Adaptive Time-Stepping - Motivation

Rate of change of the solution often varies over time!

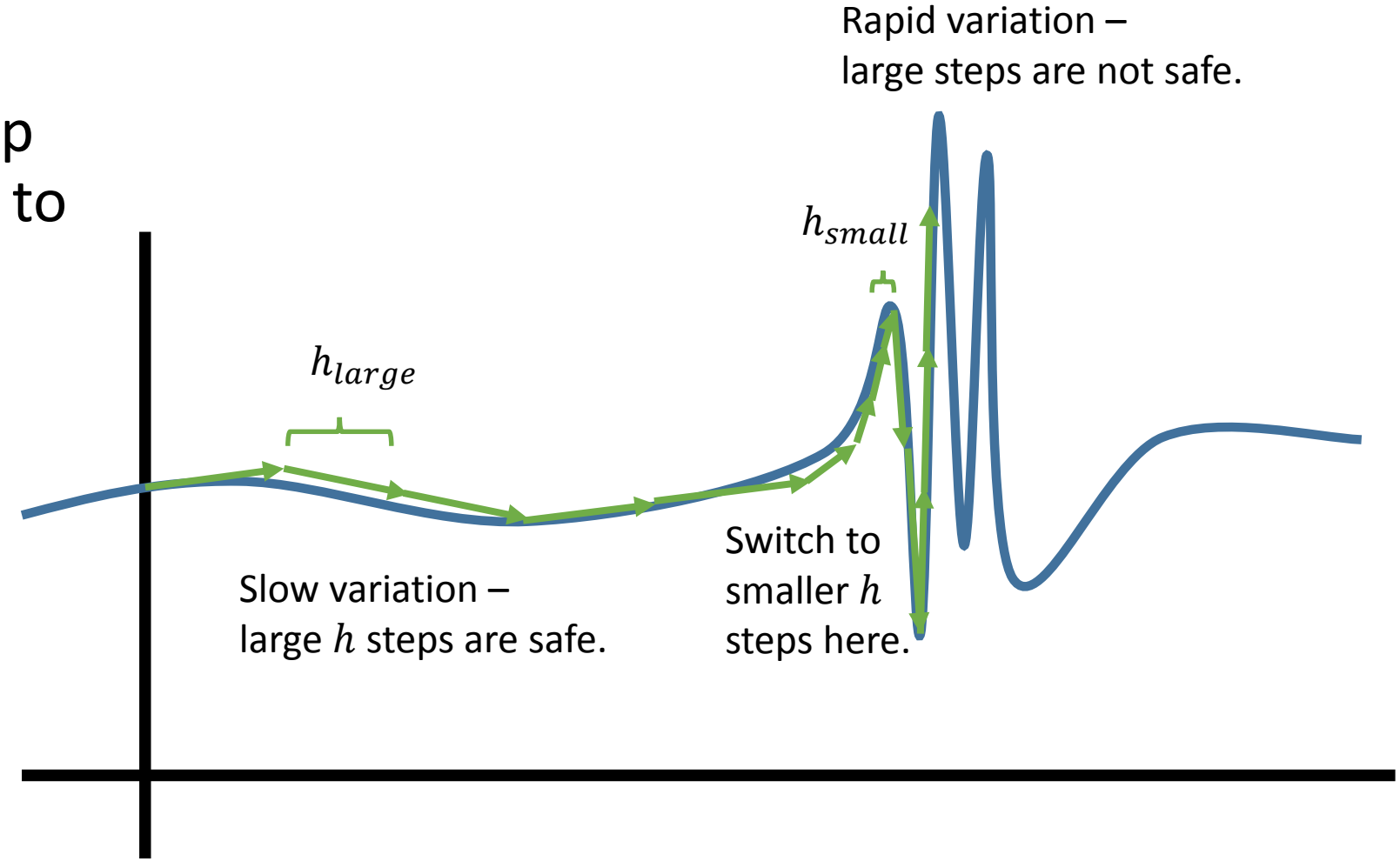
e.g.

- Seasonal variations.
- Sudden strong forces in a physical system.
- Shocks to the stock market.



Adaptive Time-Stepping - Motivation

Idea: adapt the time step during the computation to minimize error while avoiding wasted effort.



Adaptive Time-Stepping – Concept

Another “chicken-and-egg” problem?

If we knew the error for a given h, we could choose a good h to always satisfy $error < tolerance$. But…

…if we already knew the error, we’d know the solution too!

Approach: Use **two** different time-stepping schemes **together**.

Compare their results to *estimate* the error, and adjust h accordingly.

Let’s derive a justification for this!

Adaptive Time-Stepping - Basic Algorithm

1. Compute approximate solutions with two schemes of different orders.
2. Estimate the error by taking their difference.
3. While(error > tolerance)
   - Set h := h/2, and recompute the solutions (1.) & error (2.).
4. Estimate error coefficient, and predict a good *next* stepsize h_{new}.
5. Repeat until end time is reached.

Example: Matlab

Matlab's **ODE45** routine uses "Runge-Kutta-Fehlberg" scheme.
It uses 4th and 5th order Runge-Kutta schemes together:

Matlab's **ODE23** is similar but uses a lower-order pair.

Doesn't this ***double the cost***?
No: evaluation points of f are carefully chosen/weighted to avoid duplicate calls (calling f is often the main cost).

$$
\begin{aligned}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \tfrac{h}{4}, y_n + \tfrac{k_1}{4}) \\
k_3 &= hf(t_n + \tfrac{3h}{8}, y_n + \tfrac{3k_1}{32} + \tfrac{9k_2}{32}) \\
k_4 &= hf(t_n + \tfrac{12h}{13}, y_n + \tfrac{1932k_1}{2197} - \tfrac{7200k_2}{2197} + \tfrac{7296k_3}{2197}) \\
k_5 &= hf(t_n + h, y_n + \tfrac{439k_1}{216} - 8k_2 + \tfrac{3680k_3}{513} - \tfrac{845k_4}{4104}) \\
k_6 &= hf(t_n + \tfrac{h}{2}, y_n - \tfrac{8k_1}{27} + 2k_2 - \tfrac{3544k_3}{2565} + \tfrac{1859k_4}{4104} - \tfrac{11k_5}{40}) \\
y^*_{n+1} &= y_n + \tfrac{25k_1}{216} + \tfrac{1408k_3}{2565} + \tfrac{2197k_4}{4104} - \tfrac{k_5}{5} \text{ with error } O(h^4) \\
y_{n+1} &= y_n + \tfrac{16k_1}{135} + \tfrac{6656k_3}{12825} + \tfrac{28561k_4}{56430} - \tfrac{9k_5}{50} + \tfrac{2k_6}{55} \text{ with error } O(h^5).
\end{aligned}
$$

Time Step Control

Summary:

Adapting the timestep h based lets us avoid excess computational cost, while *approximately* satisfying a given error tolerance.

i.e. we expend **no more effort than we have to** for our answer.