MA3227 Numerical Analysis II

Lecture 17: Adaptive Time-Stepping

Simon Etter

2019/2020

Adaptive Time-Stepping

Introduction

Solving an ODE $\dot{y} = f(y)$ using the algorithms presented in Lecture 16 requires us to specify

- the model parameters f, y_0 and T, and
- the numerical parameters `step_function` and `n_steps`.

See `plot_solution()` for a concrete example.

Having to specify `step_function` and `n_steps` is quite a nuisance in practice. We hardly ever care what these parameters are as long as both accuracy and runtime are acceptable.

Unfortunately, we usually cannot predict a-priori which combinations of `step_function` and `n_steps` achieve this goal, so we have to try different values until we find such a combination.

We will discuss how to choose `step_function` later. For the moment, let us fix `step_function = euler_step` so we can focus on how to choose `n_steps`.

Adaptive Time-Stepping

How to choose `n_steps`

The simplest way of choosing `n_steps` is to simply set this number to a very large value, e.g. `n_steps = 100_000`. This works for toy problems, but may be unpractical for large-scale problems.

The next simplest way of choosing `n_steps` is as follows:

- Start with a small `n_steps`.
- Keep doubling `n_steps` until the solution stops changing.

For a single set of model parameters, this can easily be done by hand, and writing code which does this automatically is not difficult; see `integrate_adaptively()` and `plot_adaptive_solution()`.

Moreover, if this procedure stops with $\texttt{n\_steps} = 2^L$, then we have made

$$\sum_{k=0}^{L} 2^k = \frac{2^{L+1} - 1}{2 - 1} = 2\,\texttt{n\_steps} - 1$$

calls to `step_function`, i.e. `integrate_adaptively()` is at most twice as expensive as `integrate()` with the optimal `n_steps`.

Adaptive Time-Stepping

How to choose `n_steps` (continued)

Finally, we can get rid of the factor two by using that the error $e(n)$ in Runge-Kutta methods satisfies $e(n) = \mathcal{O}\left(n^{-p}\right)$ with a known p.

For large enough n, we therefore expect that there is a constant C such that $e(n) \approx Cn^{-p}$. This C can be estimated by estimating the error for some moderate number of steps n_{trial} and setting

$$e(n_{\text{trial}}) = C\, n_{\text{trial}}^{-p} \qquad \Longleftrightarrow \qquad C = e(n_{\text{trial}})\, n_{\text{trial}}^{p}.$$

Once we know C, we can estimate the number of steps n_{ideal} required to satisfy $e(n_{\text{ideal}}) = \varepsilon$ using

$$e(n_{\text{ideal}}) = C\, n_{\text{ideal}}^{-p} = \varepsilon \quad \Longleftrightarrow \quad n_{\text{ideal}} = \left(\frac{C}{\varepsilon}\right)^{1/p} = n_{\text{trial}} \left(\frac{e(n_{\text{trial}})}{\varepsilon}\right)^{1/p}.$$

An easy way to estimate n_{trial} is to use a higher-order Runge-Kutta method to estimate the error in a lower-order method (e.g. use midpoint to estimate the error in Euler). See `integrate_adaptively_2()`.

Adaptive Time-Stepping

Discussion

The fundamental issue underlying the above discussion is that we generally cannot guarantee upper bounds on both the runtime and the accuracy.

Fixing `n_steps` or `tol` represent different ways to resolve this situation:

- Fixing `n_steps` guarantees the runtime, but then we have to put up with whatever accuracy we get.
- Fixing `tol` guarantees the accuracy, but then we have to put up with whatever runtime we get.
  More accurately, we *hope* that fixing `tol` guarantees the accuracy. It is possible that comparing solutions with different `n_steps` as in `integrate_adaptively()` underestimates the error. This is a fundamental issue which cannot be overcome in general, so usually we just ignore it.

The previous slide may seem to suggest that the two adaptive approaches sketched above are close to optimal in the category of "fixed accuracy, whatever runtime" algorithms.

This is not the case, and the remainder of this lecture will explain how we can do better.

Adaptive Time-Stepping

Runge-Kutta convergence theory, revisited

While preparing this lecture, I noticed that the discussion regarding the convergence of Runge-Kutta methods can be both simplified and slightly adapted to simplify the discussion in this lecture. Have a look at slides 10ff in Lecture 16 to see what changed.

The main new result is the bound

$$\left\|\tilde{y}_n - y_n\right\| \leq \sum_{k=1}^{n} \exp\bigl(L\,(t_n - t_k)\bigr)\,\|\tilde{\Phi}_k(\tilde{y}_{k-1}) - \Phi_k(\tilde{y}_{k-1})\|,$$

which shows that total error is upper bounded by the sum of the local errors multiplied by the appropriate stability factors.

In the following, I will assume that either $T \lesssim \frac{1}{L}$ or the stability of the ODE is better than the exponential worst-case shown above. This is a reasonable assumption, since if violated we cannot hope to obtain solutions with reasonable accuracy.

Consequently, I will assume in the following that the total error is proportional to the local error with a proportionality constant $\lesssim 10$.

Adaptive Time-Stepping

Runge-Kutta convergence theory, revisited (continued)

For Euler's method, we have seen that

$$\tilde{y}(t) = y(0) + f\big(y(0)\big)\, t,$$
$$y(t) = y(0) + \dot{y}(0)\, t + \ddot{y}(0)\, \tfrac{t^2}{2} + \mathcal{O}(t^3)$$

and hence the local error is given by

$$\tilde{\Phi}_k(\tilde{y}_{k-1}) - \Phi_k(\tilde{y}_{k-1}) = \dddot{\tilde{y}}(t_{k-1})\, \tfrac{(t_k - t_{k-1})^2}{2} + \mathcal{O}\big((t_k - t_{k-1})^3\big)$$

where we set $\ddot{\tilde{y}}(t_{k-1}) = f'(\tilde{y}_{k-1})\, f(\tilde{y}_{k-1})$.

Ignoring the higher-order terms, we see that the local error is small if either $\ddot{\tilde{y}}(t_k)$ or $(t_k - t_{k-1})^2$ is small. This result is easily generalised to higher-order Runge-Kutta methods and leads to the following rule-of-thumb:

Runge-Kutta methods are accurate even for large step sizes if $y(t)$ is approximately a straight line. Runge-Kutta methods are inaccurate for large step sizes if $y(t)$ is "curvy".

Adaptive Time-Stepping

Runge-Kutta convergence theory, revisited (continued)

In our experiments so far, we have kept the time-step $t_k - t_{k-1}$ constant, but $\ddot{y}(t) = \lambda^2 \exp(-\lambda t)$ vanishes for $t \to \infty$. This means that the local errors will decrease exponentially for large t, see `plot_error()`.

We conclude that choosing small time steps for large times t is a waste of compute time since according to the bound on slide 6, the total error will be dominated by the large errors in the first few steps.

Rather, we should start with a small step size and then gradually increase the step size, or more generally we should choose $t_k - t_{k-1}$ small if $y(t)$ is curvy and we should choose $t_k - t_{k-1}$ large if $y(t)$ is approximately straight or even constant.

Adaptive Time-Stepping

Adaptive time-stepping

Recall: in the beginning of this lecture we discussed the "double until converged" and "extrapolate convergence" schemes for choosing `n_step`.

Both of these schemes also work for choosing the local step size. Usually, the "double until converged" scheme is used to choose the initial step size, and then the "extrapolate convergence" scheme is used to adapt the step size, using the previous step size as the trial step size.

For the "extrapolate convergence" scheme, the $e(\Delta t_{\text{trial}})$ error is usually estimated using two Runge-Kutta methods of different orders. This can be done very cheaply if these methods are nested, i.e. if the lower-order method only requires evaluations $f(\tilde{y})$ which are computed anyway in the higher-order method.

Example

Recall the formulae for the Euler and midpoint methods:

$$\text{Euler:} \quad \tilde{y}(t) = y_0 + f(y_0)\, t$$

$$\text{Midpoint:} \quad \tilde{y}(t) = y_0 + f\left(y_0 + f(y_0)\, \tfrac{t}{2}\right) t$$

These methods are nested since Euler requires only $f(y_0)$ which is computed anyway in the midpoint method.

Adaptive Time-Stepping

Discussion

Adaptive time-stepping is probably closer to engineering than it is to mathematics: it involves many heuristic arguments and rule-of-thumb approximations, and despite our best efforts it does not guarantee that the resulting algorithm is optimally efficient or yields results of a guaranteed accuracy. Instead, the aim is to have an algorithm which simply "works" for most problems.

This lecture outlined the main components of an adaptive time-stepping algorithm. Writing code which incorporates these components is not difficult, but tedious and not very insightful. Instead, I demonstrate the effect of adaptive time-stepping using the `DifferentialEquations.jl` Julia package. See `plot_de_solution()`.

We observe that adaptive time-stepping indeed manages to determine a $\tilde{y}(t)$ which is indistinguishable from the exact solution up to plotting resolution, and it does so without any input regarding the step size.

Adaptive Time-Stepping

Discussion (continued)

Moreover, looking at the output of `plot_stepsize()` we observe that adaptive time-stepping leads to time steps which start off small and then gradually increase as expected.

However, we also observe that the time steps stop increasing at some point. Explaining how this phenomenon comes about and how it can be overcome will be the topic of the next lecture.

To conclude, I demonstrate in `adaptive_example()` how adaptive time-stepping can detect a sharp transition in the solution and use very small time-steps around this transition while using large time steps far away.

Adaptive Time-Stepping

Summary

- Extrapolation of convergence parameter:

$$e(n) = C\, n^{-p} \quad \longrightarrow \quad n_{\text{ideal}} = n_{\text{trial}} \left(\frac{e(n_{\text{trial}})}{\varepsilon} \right)^{1/p}$$

- Runge-Kutta methods are accurate even for large step sizes if $y(t)$ is approximately a straight line. Runge-Kutta methods are inaccurate for large step sizes if $y(t)$ is "curvy".
- The performance of Runge-Kutta methods can be improved by adapting the step size depending on the curvature of the solution.