16. ODEs III: Error control and variable step size

Last time

- Higher derivatives as systems of 1st-order equations
- Implicit trapezium method $\longrightarrow$ modified Euler method
- Runge–Kutta methods
- Stages as Euler steps

Goals for today

- **Adaptivity**: Vary step size to control error
- Embedded Runge–Kutta methods

Review: Runge–Kutta methods

- Want to solve

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t))$$

- Get approximate solution $\mathbf{x}_n$ at times t_n

Review: Runge–Kutta methods

- Want to solve

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t))$$

- Get approximate solution $\mathbf{x}_n$ at times t_n
- Runge–Kutta methods use several **stages**
- Stage is 1 evaluation of f at some point
- This point depends on previous evaluations

Runge–Kutta II

- Gives nested sequence of evaluations of f
- Reproduces Taylor series (single step) to order h^n
- Local truncation error is $\mathcal{O}(h^{n+1})$

Error checking

- Until now: for Euler and Runge–Kutta we have supposed that each step approximates the function well
- We have calculated the step and always taken it

Error checking

- Until now: for Euler and Runge–Kutta we have supposed that each step approximates the function well
- We have calculated the step and always taken it
- But since we're stepping into the unknown, we should **check** if step is "valid"

Error checking

- Until now: for Euler and Runge–Kutta we have supposed that each step approximates the function well
- We have calculated the step and always taken it
- But since we're stepping into the unknown, we should **check** if step is "valid"
- But what can we check against?
- Exact solution is, of course, never available

Error checking

- Until now: for Euler and Runge–Kutta we have supposed that each step approximates the function well
- We have calculated the step and always taken it
- But since we're stepping into the unknown, we should **check** if step is "valid"
- But what can we check against?
- Exact solution is, of course, never available
- Replace exact solution by a *better* solution

Same method, different step sizes

- One solution: Same method with different step sizes, e.g. Euler

Same method, different step sizes

- One solution: Same method with different step sizes, e.g. Euler
- y_1 := result after *1* Euler step of length h
- y_2 := result after *2* consecutive Euler steps of $h/2$,

Same method, different step sizes

- One solution: Same method with different step sizes, e.g. Euler
- y_1 := result after *1* Euler step of length h
- y_2 := result after *2* consecutive Euler steps of $h/2$,
- Local error = $\mathcal{O}(h^2)$ for *both*
- But constant is different
- $\Delta y := |y_1 - y_2|$ *measures* the error

Same method, different step sizes

- One solution: Same method with different step sizes, e.g. Euler
- y_1 := result after *1* Euler step of length h
- y_2 := result after *2* consecutive Euler steps of $h/2$,
- Local error = $\mathcal{O}(h^2)$ for *both*
- But constant is different
- $\Delta y := |y_1 - y_2|$ *measures* the error
- PS 6

Alternative: Methods with different order

- y_1: result after step with order-p method
- y_2: result after step with order-$(p+1)$ method

Alternative: Methods with different order

- y_1: result after step with order-p method
- y_2: result after step with order-$(p+1)$ method
- If step size is h then $\Delta y := |y_1 - y_2| = Ch^{p+1}$
- Note that C is related to a higher derivative but is *unknown*
- Measures approximate error in y_1, since y_2 is presumably more accurate

Varying the step size

- Suppose we want the error to be a given value ϵ
- Then we should *choose* step size h' accordingly

Varying the step size

- Suppose we want the error to be a given value ϵ
- Then we should *choose* step size h' accordingly
- We need $C(h')^{p+1} \sim \epsilon$

Varying the step size

- Suppose we want the error to be a given value ϵ
- Then we should *choose* step size h' accordingly
- We need $C(h')^{p+1} \sim \epsilon$
- We can get rid of C by dividing!:

$$\left(\frac{h'}{h}\right)^{p+1} = \frac{\epsilon}{\Delta y}$$

- So we should take

$$h' = h\left(\frac{\epsilon}{\Delta y}\right)^{\frac{1}{p+1}}$$

- Alternative: “error per unit time step” should be ϵ

Variable step size algorithm

- Use to construct algorithm with **variable step size**:

1. Propose step and calculate error Δy as above

Variable step size algorithm

- Use to construct algorithm with **variable step size**:

1. Propose step and calculate error Δy as above
2. If error *too big*, $\Delta y > \epsilon$, **reject** step:

   remain at same place but *decrease* step size h

Variable step size algorithm

- Use to construct algorithm with **variable step size**:

1. Propose step and calculate error Δy as above
2. If error *too big*, $\Delta y > \epsilon$, **reject** step:

   remain at same place but *decrease* step size h
3. If error *small enough*, $\Delta y \leq \epsilon$, **accept** step:

   move with *current* step size h, then *increase* h

Variable step size algorithm II

4. In either case, step size modified as above

Variable step size algorithm II

4. In either case, step size modified as above
   - In certain circumstances get wild increases of h
   - So restrict to at most $h' = 2h$

Variable step size algorithm II

4. In either case, step size modified as above
   - In certain circumstances get wild increases of h
   - So restrict to at most $h' = 2h$
   - Allows even Euler to “work”
   - But needs many tiny steps!

Embedded Runge–Kutta methods

- Above methods require too much computational work
- E.g. Euler methods need *3* function evaluations for each step for an $\mathcal{O}(h)$ method
- Even worse with p- and $p+1$-order methods: at least $2p+1$ function evaluations

Embedded Runge–Kutta methods

- Above methods require too much computational work
- E.g. Euler methods need *3* function evaluations for each step for an $\mathcal{O}(h)$ method
- Even worse with p- and $p+1$-order methods: at least $2p+1$ function evaluations
- Amazingly, in the world of Runge–Kutta methods, there is a better solution:
- Suppose we have an order-$(p+1)$ method, given by a certain Butcher tableau (coefficients) defining s stages

Embedded Runge–Kutta methods II

- Recall that $x_{n+1} = x_n + h \sum_{i=1}^{s} b_i k_i$ where the k_i are results of each stage
- Amazingly, for some RK methods the *same* k_i's give order-p method when combined in *different* way
- e.g. Bogacki–Shampine BS23:

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{3}{4} & 0 & \frac{3}{4} & & \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
\hline
 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\
\hline
 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8}
\end{array}
$$

Embedded Runge–Kutta methods III

- Extra useful property:
- **FSAL**: “First same as last”
- k_s is evaluated at $t_n + h$
- So k_s from previous step = k_1 for new step
- No need to re-evaluate

Embedded Runge–Kutta methods III

- Extra useful property:
- **FSAL**: “First same as last”
- k_s is evaluated at $t_n + h$
- So k_s from previous step = k_1 for new step
- No need to re-evaluate

- Modern version: Tsitouras 5/4 method (2011)
- Default in `DifferentialEquations.jl`

Summary

- Calculate local error by two different methods
- *Choose* variable step size to fit desired error
- Embedded Runge–Kutta methods are very efficient