# 19. ODEs III: Taylor methods and adaptivity – controlling the error

# Summary of the previous class

- Trapezoid method: An implicit method

- Modified Euler method
- Runge–Kutta methods
- Stages as Euler steps

# Goals for today

1. **Taylor methods**
   - Taylor series solutions of ODEs
2. **Adaptivity**
   - Controlling the error by varying the step size
   - Embedded Runge–Kutta methods

# Review: Runge–Kutta methods

- We want to solve

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

- To get approximations $\mathbf{x}_n$ of solution $x(t_n)$ at time $t_n$

# Review: Runge–Kutta methods

- We want to solve

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

- To get approximations $\mathbf{x}_n$ of solution $x(t_n)$ at time $t_n$

- Runge–Kutta methods use several **stages**
- Each stage is an evaluation of $f$ at some point
- Depending on previous evaluations

# Runge–Kutta methods reproduce the Taylor series of the true solution

- Runge–Kutta methods calculate the Taylor series of the true solution $x(t)$ around the initial point $x_n$

# Runge–Kutta methods reproduce the Taylor series of the true solution

- Runge–Kutta methods calculate the Taylor series of the true solution $x(t)$ around the initial point $x_n$
- Without actually calculating the Taylor series!

# Runge–Kutta methods reproduce the Taylor series of the true solution

- Runge–Kutta methods calculate the Taylor series of the true solution $x(t)$ around the initial point $x_n$
- Without actually calculating the Taylor series!
- The amazing thing is that a given RK method works for *any* (smooth enough) $f$

# Runge–Kutta methods reproduce the Taylor series of the true solution

- Runge–Kutta methods calculate the Taylor series of the true solution $x(t)$ around the initial point $x_n$
- Without actually calculating the Taylor series!
- The amazing thing is that a given RK method works for *any* (smooth enough) $f$
- However, for high accuracy we need high-order methods

# Runge–Kutta methods reproduce the Taylor series of the true solution

- Runge–Kutta methods calculate the Taylor series of the true solution $x(t)$ around the initial point $x_n$
- Without actually calculating the Taylor series!
- The amazing thing is that a given RK method works for *any* (smooth enough) $f$

- However, for high accuracy we need high-order methods
- E.g. to simulate the solar system for millions of years

# Runge–Kutta methods reproduce the Taylor series of the true solution

- Runge–Kutta methods calculate the Taylor series of the true solution $x(t)$ around the initial point $x_n$
- Without actually calculating the Taylor series!
- The amazing thing is that a given RK method works for *any* (smooth enough) $f$

- However, for high accuracy we need high-order methods
- E.g. to simulate the solar system for millions of years
- RK methods of high order are difficult to find

# Taylor methods

# Taylor series solutions

- **Taylor methods** use the Taylor series of the exact solution
- We have seen that they seem to require calculating high-order derivatives of $f$, such as $f_t$, $f_x$, $f_{xx}$

# Taylor series solutions

- **Taylor methods** use the Taylor series of the exact solution
- We have seen that they seem to require calculating high-order derivatives of $f$, such as $f_t$, $f_x$, $f_{xx}$
- But it is possible to construct the Taylor series solution directly (up to a given order)!

# Taylor series solutions

- **Taylor methods** use the Taylor series of the exact solution
- We have seen that they seem to require calculating high-order derivatives of $f$, such as $f_t$, $f_x$, $f_{xx}$
- But it is possible to construct the Taylor series solution directly (up to a given order)!
- The disadvantage is that we must construct a new Taylor series solution for each ODE

# Taylor method

- Suppose we want to solve $\dot{x}(t) = f(x(t))$, with $x(t = 0) = x_0$

# Taylor method

- Suppose we want to solve $\dot{x}(t) = f(x(t))$, with $x(t = 0) = x_0$
- Suppose $f$ is **analytic**

# Taylor method

- Suppose we want to solve $\dot{x}(t) = f(x(t))$, with $x(t = 0) = x_0$
- Suppose $f$ is **analytic**
- i.e. is equal to its Taylor expansion

$$f(x) = \tilde{f}_0 + \tilde{f}_1 x + \tilde{f}_2 x^2 + \cdots$$

# Taylor method

- Suppose we want to solve $\dot{x}(t) = f(x(t))$, with $x(t = 0) = x_0$
- Suppose $f$ is **analytic**
- i.e. is equal to its Taylor expansion

$$f(x) = \tilde{f}_0 + \tilde{f}_1 x + \tilde{f}_2 x^2 + \cdots$$

- Then (theorem) the solution $x(t)$ exists and is analytic, so has a power series expansion in $t$:

$$x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$$

# Taylor method II

- We have

$$x(0) = x_0$$
$$\dot{x}(0) = x_1$$
$$\ddot{x}(0) = 2x_2$$

$\vdots$

$\cdots$

- How can we calculate the $x_i$?

# Collaboration I

## Taylor method

We are proposing a solution $x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$ with unknown coefficients $x_i$ for the ODE $\dot{x} = f(x)$ with given initial condition $x(t) = x_0$.

1. How could we solve for the coefficients $x_i$?
2. Try to do this for $f(x) = x$

# Taylor method III

- Let's **substitute** the Taylor series for $x(t)$ into the ODE:
  - on the left-hand side we need $\dot{x}(t)$
  - on the right-hand side we need $f(x(t))$

# Taylor method III

- Let's **substitute** the Taylor series for $x(t)$ into the ODE:
  - on the left-hand side we need $\dot{x}(t)$
  - on the right-hand side we need $f(x(t))$
- Both of these give *new Taylor series*:

$$\dot{x}(t) = x_1 + 2x_2 t + 3x_3 t^2 + \cdots$$

- Substituting $x(t)$ into $f(x(t))$ gives a series *in* $t$:

$$f(x(t)) = f_0 + f_1 t + f_2 t^2 + \cdots$$

# Taylor method IV

- Since these two series are equal for all $t$
  *the coefficient of $t^n$ must be equal for each $n$*

# Taylor method IV

- Since these two series are equal for all $t$
  *the coefficient of* $t^n$ *must be equal for each* $n$
- To prove this e.g. differentiate repeatedly

# Taylor method IV

- Since these two series are equal for all $t$
  *the coefficient of $t^n$ must be equal for each $n$*
- To prove this e.g. differentiate repeatedly
- Equate coefficients of each power $t^n$:

$$\begin{aligned} x_1 &= f_0 \\ 2x_2 &= f_1 \\ &\vdots \\ nx_n &= f_{n-1} \end{aligned}$$

# Taylor method IV

- Since these two series are equal for all $t$
  *the coefficient of $t^n$ must be equal for each $n$*
- To prove this e.g. differentiate repeatedly
- Equate coefficients of each power $t^n$:

$$\begin{aligned} x_1 &= f_0 \\ 2x_2 &= f_1 \\ &\vdots \\ nx_n &= f_{n-1} \end{aligned}$$

- We get a **recurrence relation**: $x_n = \frac{f_{n-1}}{n}$

# Taylor method V

- To calculate $f_{n-1}$ we insert the Taylor series for $x(t)$ into $f(x)$

# Taylor method V

- To calculate $f_{n-1}$ we insert the Taylor series for $x(t)$ into $f(x)$
- $f_{n-1}$ is the coefficient of $t^{n-1}$

# Taylor method V

- To calculate $f_{n-1}$ we insert the Taylor series for $x(t)$ into $f(x)$
- $f_{n-1}$ is the coefficient of $t^{n-1}$
- So $f_{n-1}$ can depend only on $x_0$ up to $x_{n-1}$

# Taylor method V

- To calculate $f_{n-1}$ we insert the Taylor series for $x(t)$ into $f(x)$
- $f_{n-1}$ is the coefficient of $t^{n-1}$
- So $f_{n-1}$ can depend only on $x_0$ up to $x_{n-1}$
- Hence from the coefficients $x_0$, …, $x_{n-1}$ we can obtain $x_n$!

# Taylor method V

- To calculate $f_{n-1}$ we insert the Taylor series for $x(t)$ into $f(x)$
- $f_{n-1}$ is the coefficient of $t^{n-1}$
- So $f_{n-1}$ can depend only on $x_0$ up to $x_{n-1}$
- Hence from the coefficients $x_0$, …, $x_{n-1}$ we can obtain $x_n$!
- We generate all coefficients $x_n$ *recursively*, one by one!

# Example

- E.g. Solve $\dot{x} = x^2$ with $x_0 = 1$

# Example

- E.g. Solve $\dot{x} = x^2$ with $x_0 = 1$
- Start with all coefficients unknown *except* $x_0$:

$$x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$$

where red denotes coefficients we don't yet know

# Example

- E.g. Solve $\dot{x} = x^2$ with $x_0 = 1$
- Start with all coefficients unknown *except* $x_0$:

$$x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$$

where red denotes coefficients we don't yet know

- So

$$\begin{aligned} f(x(t)) &= [x(t)]^2 \\ &= (x_0 + x_1 t + \cdots)^2 \\ &= x_0^2 + \mathcal{O}(t) \end{aligned}$$

- Hence $f_0 = x_0^2$

# Example II

- So $x_1 = f_0/1 = x_0^2$

# Example II

- So $x_1 = f_0/1 = x_0^2$
- Now we have $x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$

# Example II

- So $x_1 = f_0/1 = x_0^2$
- Now we have $x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$
- Substitute into $f(x)$ again:

$$f(x(t)) = x_0^2 + t(x_0 x_1 + x_1 x_0) + \mathcal{O}(t^2)$$

# Example II

- So $x_1 = f_0/1 = x_0^2$
- Now we have $x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$
- Substitute into $f(x)$ again:

$$f(x(t)) = x_0^2 + t(x_0 x_1 + x_1 x_0) + \mathcal{O}(t^2)$$

- This gives $f_1$ = (coefficient of $t$) = $2x_0 x_1$

# Example II

- So $x_1 = f_0/1 = x_0^2$
- Now we have $x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$
- Substitute into $f(x)$ again:

$$f(x(t)) = x_0^2 + t(x_0 x_1 + x_1 x_0) + \mathcal{O}(t^2)$$

- This gives $f_1$ = (coefficient of $t$) = $2x_0 x_1$
- Hence $x_2 = f_1/2 = x_0\, x_1$

# Example II

- So $x_1 = f_0/1 = x_0^2$
- Now we have $x(t) = x_0 + x_1 t + x_2 t^2 + \cdots$
- Substitute into $f(x)$ again:

$$f(x(t)) = x_0^2 + t(x_0 x_1 + x_1 x_0) + \mathcal{O}(t^2)$$

- This gives $f_1$ = (coefficient of $t$) = $2x_0 x_1$
- Hence $x_2 = f_1/2 = x_0\, x_1$
- Repeat, including the new coefficient into $x(t)$
- Note: previous $f_i$ are recalculated – this is inefficient

# Alternative viewpoint: Integrals

- An alternative viewpoint is to use the integral formulation of the ODE:

$$x(t) = x_0 + \int_0^t f(x(s))\,ds$$

- Define the $n$th order polynomial approximation $x^{(n)}(t) := x_0 + \cdots + x_n t^n$
- We use **Picard iteration** to calculate $x^{(n)}$ recursively:

$$x^{(n+1)} = x_0 + \int \widehat{f}^{(n)}\left(x^{(n)}\right)$$

- Where $\widehat{f}^{(n)}(x)$ means "$f(x)$ truncated to order $n$"

# Example using integrals

- Let's return to our example:

$$\dot{x} = x^2 \quad \text{with } x^{(0)} := x_0$$

# Example using integrals

- Let's return to our example:

$$\dot{x} = x^2 \quad \text{with } x^{(0)} := x_0$$

$$x^{(1)} = x_0 + \int \left(x^{(0)}\right)^2 = x_0 + \int_0^t x_0^2 \, ds = x_0 + t\, x_0^2$$

# Example using integrals

- Let's return to our example:

$$\dot{x} = x^2 \quad \text{with } x^{(0)} := x_0$$

$$x^{(1)} = x_0 + \int \left(x^{(0)}\right)^2 = x_0 + \int_0^t x_0^2 \, ds = x_0 + t\, x_0^2$$

$$x^{(2)} = x_0 + \int \left(x^{(1)}\right)^2$$

$$= x_0 + \int_0^t (x_0 + s\, x_0^2)^2 \, ds = x_0 + t\, x_0^2 + t^2 x_0^3 + \mathcal{O}(t^3)$$

# Implementation

- How can we automate this in Julia?
- What operations do we need?

# Implementation

- How can we automate this in Julia?
- What operations do we need?

- We are just manipulating polynomials!
- And truncating to a given degree
- So define operations like $*$ on polynomials of degree $n$ that return polynomials of the *same* degree

# Implementation

- How can we automate this in Julia?
- What operations do we need?

- We are just manipulating polynomials!
- And truncating to a given degree
- So define operations like $*$ on polynomials of degree $n$ that return polynomials of the *same* degree
- These manipulations are done with *numeric* coefficients

# Implementation

- How can we automate this in Julia?
- What operations do we need?

- We are just manipulating polynomials!
- And truncating to a given degree
- So define operations like $*$ on polynomials of degree $n$ that return polynomials of the *same* degree
- These manipulations are done with *numeric* coefficients
- In the end, much of “symbolic” calculation manipulating numeric coefficients of polynomials

# Adaptivity: Varying the step size and controlling the error

# Error checking

- Until now we have supposed that each step approximates the function well
- We have always taken the calculated step

# Error checking

- Until now we have supposed that each step approximates the function well
- We have always taken the calculated step
- But we are stepping into the unknown, so we should **verify** if the step is “valid”

# Error checking

- Until now we have supposed that each step approximates the function well
- We have always taken the calculated step
- But we are stepping into the unknown, so we should **verify** if the step is “valid”
- I.e. if the error is “small enough”

# Error checking

- Until now we have supposed that each step approximates the function well
- We have always taken the calculated step
- But we are stepping into the unknown, so we should **verify** if the step is “valid”
- I.e. if the error is “small enough”
- But what can we check against?
- The exact solution is never available

# Collaboration II

## Checking Euler steps

Suppose we are using the Euler method and we take one step.

1. What could we check the step against if we don't have access to the true solution?
2. Can you think of another possibility (for a total of two)?
3. How much computational effort would that require?
4. How can you decide whether the error is small enough?

# Same method, different step sizes

- One solution is to use the *same* numerical method, e.g. Euler

# Same method, different step sizes

- One solution is to use the *same* numerical method, e.g. Euler
- But with *different* step sizes

# Same method, different step sizes

- One solution is to use the *same* numerical method, e.g. Euler
- But with *different* step sizes
- $y_1$ := result after *one* Euler step of length $h$
- $y_2$ := result after *two* consecutive Euler steps, each of length $h/2$

# Same method, different step sizes

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

# Same method, different step sizes

- One solution is to use the *same* numerical method, e.g. Euler
- But with *different* step sizes
- $y_1$ := result after *one* Euler step of length $h$
- $y_2$ := result after *two* consecutive Euler steps, each of length $h/2$
- Local error = $\mathcal{O}(h^2)$ for *both*
- But the constant is different (but related)
- $\Delta y := |y_1 - y_2|$ *measures* the error
- See the problem set

# Alternative: Different methods with different order

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

# Alternative: Different methods with different order

- $y_1 :=$ result after a step with a method of order $p$
- $y_2 :=$ result after a step with a method of order $(p+1)$
- How big will the error be?

# Alternative: Different methods with different order

- $y_1 :=$ result after a step with a method of order $p$
- $y_2 :=$ result after a step with a method of order $(p+1)$
- How big will the error be?
- If the step size is $h$ then $\Delta y := |y_1 - y_2| = Ch^{p+1}$

# Alternative: Different methods with different order

- $y_1 :=$ result after a step with a method of order $p$
- $y_2 :=$ result after a step with a method of order $(p+1)$
- How big will the error be?
- If the 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*
- This measures the approximate error in $y_1$, since $y_2$ is presumably more accurate

# Collaboration III

## Choosing a step size

Suppose we want the error to be $\leq \epsilon$

1. How can we decide whether the step should be accepted?
2. If it is not accepted, what should we do?
3. How should we change things?

# Varying the step size

- Suppose we want the error to be $\leq \epsilon$

# Varying the step size

- Suppose we want the error to be $\leq \epsilon$
- Then we should *choose* the step size $h'$ accordingly

# Varying the step size

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

# Varying the step size

- Suppose we want the error to be $\leq \epsilon$
- Then we should *choose* the 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 $\epsilon$

# Variable step size algorithm

- We can use this to construct an algorithm with **variable step size**:

1. Propose a step and calculate the error $\Delta y$ as above

# Variable step size algorithm

- We can use this to construct an algorithm with **variable step size**:

1. Propose a step and calculate the error $\Delta y$ as above
2. If the error is *too big*, $\Delta y > \epsilon$, then we **reject** the step:

   we remain at the same place, but *decrease* the step size $h$

# Variable step size algorithm

- We can use this to construct an algorithm with **variable step size**:

1. Propose a step and calculate the error $\Delta y$ as above
2. If the error is *too big*, $\Delta y > \epsilon$, then we **reject** the step:

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

   we move with the *current* step size $h$, then *increase* $h$

# Variable step size algorithm II

4. In both cases, the step size gets modified

# Variable step size algorithm II

4. In both cases, the step size gets modified
   - In certain circumstances this can lead to wild increases of $h$
   - So we restrict to at most $h' = 2h$

# Variable step size algorithm II

4. In both cases, the step size gets modified
   - In certain circumstances this can lead to wild increases of $h$
   - So we restrict to at most $h' = 2h$
   - This allows even Euler to “work”
   - But it can lead to taking many tiny steps

# Embedded Runge–Kutta methods

- The above methods require too much computational work

# Embedded Runge–Kutta methods

- The above methods require too much computational work
- E.g. Euler methods need *three* function evaluations for each step for an $\mathcal{O}(h)$ method

# Embedded Runge–Kutta methods

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

# Embedded Runge–Kutta methods

- The above methods require too much computational work
- E.g. Euler methods need *three* function evaluations for each step for an $\mathcal{O}(h)$ method
- This becomes even worse with $p$- and $p+1$-order methods: we need at least $2p+1$ function evaluations
- Amazingly, with Runge–Kutta methods it is possible to find a better solution

# Embedded Runge–Kutta methods II

- Suppose we use an order-$(p+1)$ RK method with $s$ stages

# Embedded Runge–Kutta methods II

- Suppose we use an order-$(p+1)$ RK method with $s$ stages
- Then $x_{n+1} = x_n + h\sum_{i=1}^{s} b_i k_i$, where the $k_i$ are the results of each stage

# Embedded Runge–Kutta methods II

- Suppose we use an order-$(p+1)$ RK method with $s$ stages
- Then $x_{n+1} = x_n + h\sum_{i=1}^{s} b_i k_i$, where the $k_i$ are the results of each stage
- Amazingly, for some RK methods there is a linear combination of the *same* $k_i$'s giving an order-$p$ method

# Embedded Runge–Kutta methods III

- Example: 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

- Example: Bogacki–Shampine BS23:

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

- The last two lines give 3rd- and 2nd-order methods, respectively

# Embedded Runge–Kutta methods III

- Example: Bogacki–Shampine BS23:

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

- The last two lines give 3rd- and 2nd-order methods, respectively
- We need about 3 function evaluations per step for this embedded 2nd / 3rd-order method

# Embedded Runge–Kutta methods IV

- An additional useful property is **FSAL**: “First same as last”

# Embedded Runge–Kutta methods IV

- An additional useful property is **FSAL**: “First same as last”

- If $k_s$ is evaluated at $t_n + h$
- Then ($k_s$ from the previous step) = ($k_1$ for the new step)

# Embedded Runge–Kutta methods IV

- An additional useful property is **FSAL**: “First same as last”

- If $k_s$ is evaluated at $t_n + h$
- Then ($k_s$ from the previous step) = ($k_1$ for the new step)
- So there is no need to re-evaluate

# Embedded Runge–Kutta methods IV

- An additional useful property is **FSAL**: “First same as last”

- If $k_s$ is evaluated at $t_n + h$
- Then ($k_s$ from the previous step) = ($k_1$ for the new step)
- So there is no need to re-evaluate

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

# Summary

- We can generate Taylor methods of arbitrary order
- By recursively calculating the coefficients
- Using polynomial manipulation

- We can calculate the local error by running two different methods for the same step
- We *choose* a variable step size to obtain the desired error
- Embedded Runge–Kutta methods provide an efficient way to do this