MAS212 Scientific Computing and Simulation

Dr. Sam Dolan

School of Mathematics and Statistics,
University of Sheffield

Autumn 2017

http://sam-dolan.staff.shef.ac.uk/mas212/

G18 Hicks Building
s.dolan@sheffield.ac.uk

Today's lecture

- Methods for solving ODEs
  1. Euler's method (review)
  2. The midpoint method (review)
  3. Runge-Kutta methods
- Linear multi-step methods:
  1. A two-step method
  2. Adams-Bashforth methods
- Implicit methods:
  1. The Backward Euler method
  2. Adams-Moulton methods

Recap: Euler's method

$$\boxed{\frac{dx}{dt} = f(x,t) \qquad x(t_0) = x_0}$$

Euler's method

$$\boxed{x_{k+1} = x_k + h\,f\left(x_k, t_k\right)}$$

Algorithm:

- Divide the domain $[t_0, t_f]$ into n equally-spaced intervals:

$$h \equiv \frac{t_f - t_0}{n} \qquad \text{and} \qquad t_k = t_0 + kh$$

- Start with the initial condition x_0 at t_0
- Apply $x_{k+1} = x_k + h\,f\left(x_k, t_k\right)$ **once** to get y_1 from y_0
- Now iterate (repeat) ...

Runge-Kutta methods

Runge-Kutta methods

$$x_{j+1} = x_j + \sum_{i=1}^{s} b_i k_i$$

where s is the order of the method, and

$$\begin{aligned}
k_1 &= h f(x_j, t_j) \\
k_2 &= h f(x_j + a_{21} k_1, t_j + c_2 h) \\
k_3 &= h f(x_j + a_{31} k_1 + a_{32} k_2, t_j + c_3 h) \\
&\dots \\
k_s &= h f(x_j + a_{s1} k_1 + a_{s2} k_2 + \dots + a_{s,s-1} k_{s-1}, t_j + c_s h)
\end{aligned}$$

- A method is specified by the coefficients b_i, c_i and a_{ij}.
- Euler's method and the midpoint method are members of the **Runge-Kutta family of methods** that use intermediate estimates in a systematic way.

Runge-Kutta methods

- A method is specified by the coefficients b_i, c_i and a_{ij}.
- Butcher tableau are used to give these coefficients:

$$\begin{array}{c|ccccc} 0 & & & & & \\ c_2 & a_{21} & & & & \\ c_3 & a_{31} & a_{32} & & & \\ \dots & \dots & \dots & \dots & & \\ c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\ \hline & b_1 & b_2 & \dots & b_{s-1} & b_s \end{array}$$

- **Example:** Midpoint method

$$\begin{aligned} k_1 &= hf(x_j, t_j) \\ k_2 &= hf(x_j + \tfrac{1}{2}k_1, t_j + \tfrac{1}{2}h) \\ y_{j+1} &= x_j + k_2. \end{aligned}$$

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \end{array}$$

Runge-Kutta methods

- The most well-used version is the **classical Runge-Kutta method** or **RK4 method** :

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

- This is a fourth-order accurate method.
- `odeint` uses this method.

Example:

(a) Numerically solve the ODE

$$\frac{dx}{dt} = e^{-t/10}\sin(t)\sin(x), \qquad x(0) = 1,$$

in the range $t \in [0, 20]$ using the **3rd order Runge-Kutta method** with Butcher tableau

0			
1/2	1/2		
1	−1	2	
	1/6	4/6	1/6

(b) Apply the ratio test to show that your method is 3rd-order accurate.

Solution: part (a)

- Import modules:

```
import numpy as np
import matplotlib.pyplot as plt
```

- A function to calculate the derivative function from x and t:

```
def dx_dt(x, t):
    return np.sin(t)*np.sin(x)*np.exp(-0.1*t)
```

- A function to implement one step of the RK3 method:

```
def rk3_step(x, t, h):
        """See Butcher tableau for RK3 method"""
    k1 = h*dx_dt(x, t)
    k2 = h*dx_dt(x+k1/2.0, t+h/2.0)
    k3 = h*dx_dt(x-k1+2*k2, t+h)
    x_next = x + (k1 + 4*k2 + k3)/6.0
    return x_next
```

Solution: part (a)

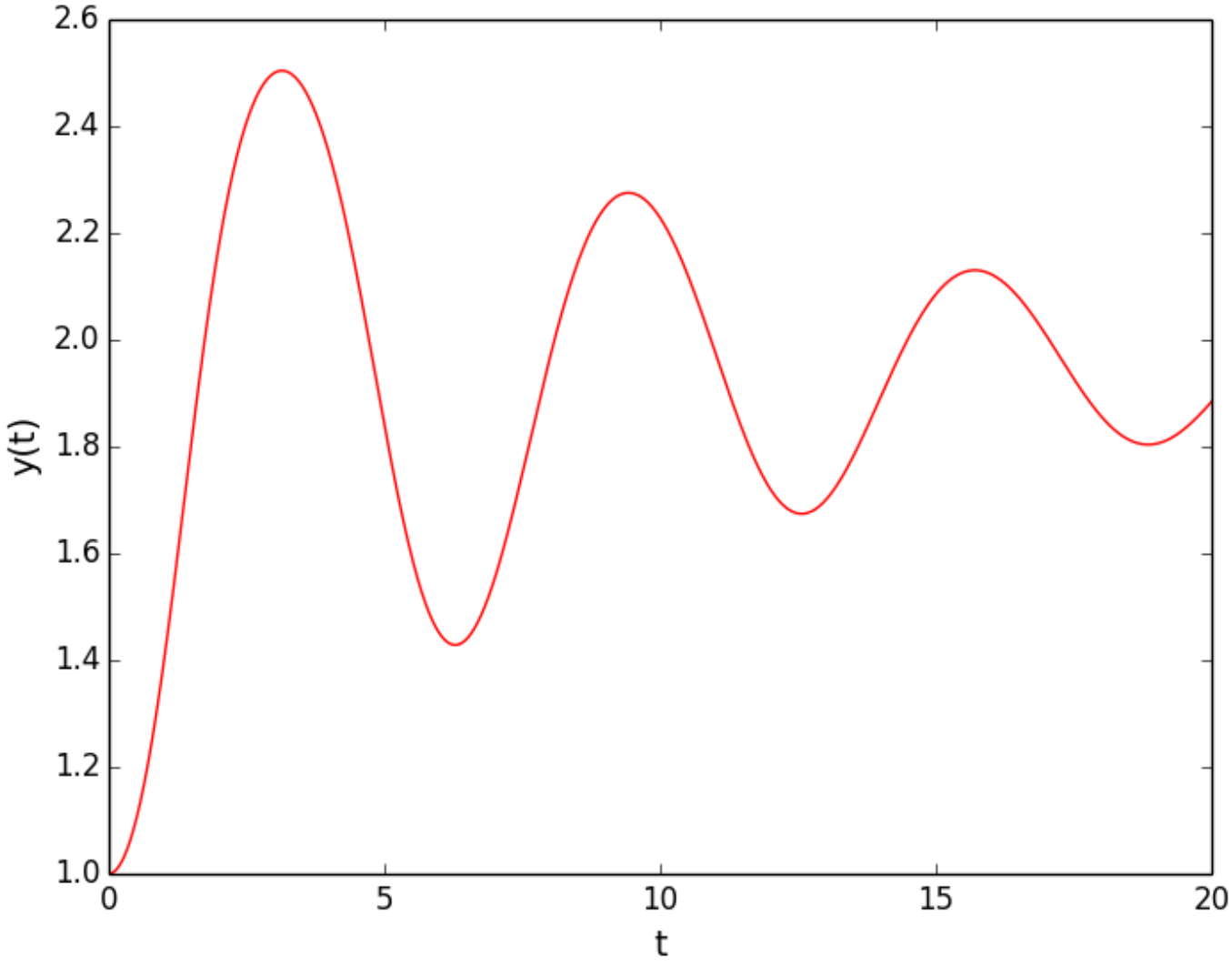
- A function to solve over the range $0 \leq t \leq 20$:

```
def rk3_solve(x0, h, tmin, tmax):
    n = int((tmax - tmin)/h)
    ts = np.zeros(n+1); xs = np.zeros(n+1)
    xs[0] = x0; ts[0] = tmin
    for k in range(n):
        ts[k+1] = ts[k] + h
        xs[k+1] = rk3_step(xs[k], ts[k], h)
    return ts, xs
```

- Now test with step size $h = 0.01$:

```
x0 = 1.0; h = 0.01; tmin = 0.0; tmax = 20.0;
ts, xs = rk3_solve(x0, h, tmin, tmax)
plt.plot(ts, xs, 'r-')
plt.xlim(tmin, tmax)
plt.xlabel("t"); plt.ylabel("x(t)")
plt.show()
```

Solution: part (a)



Solution: part (b)

- **Q.** How do we know the implementation of the method is truly **third-order accurate**? (i.e. that the **global truncation error** scales with h^3).
- **A.** Apply the **ratio test**: see lab class 5, Q1(c).
- Let $x_{(h)}(t)$ be a numerical solution with time step h.
- Now form the ratio

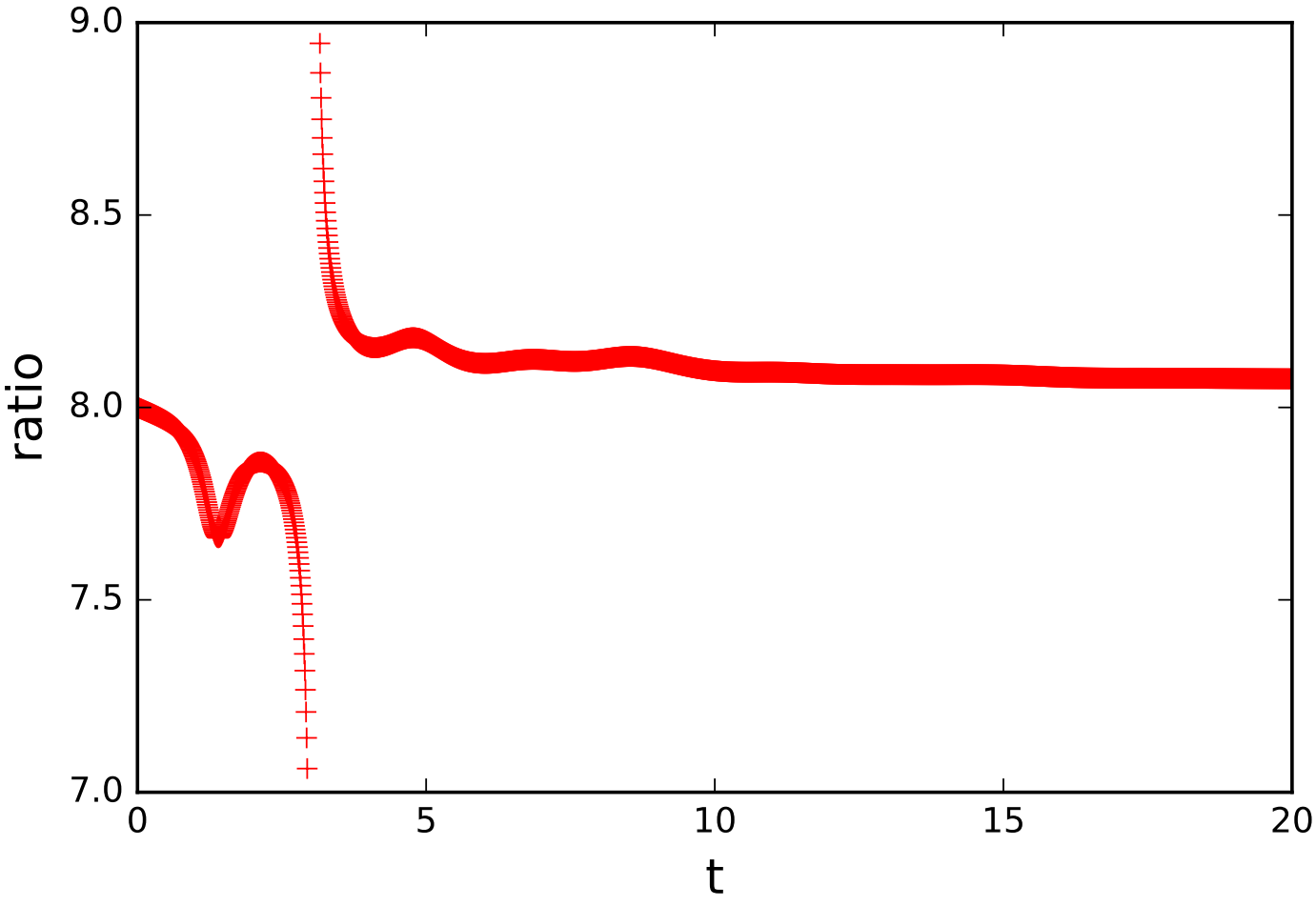
$$r_n = \frac{x_{(4h)}(t_n) - x_{(2h)}(t_n)}{x_{(2h)}(t_n) - x_{(h)}(t_n)}$$

- As $h \to 0$, we expect $r \to 2^s$, where s is the order of the method. (*Exercise:* show this).
- As the RK3 method is third order accurate we expect $r \to 8$.

Solution: part (b)

```
x0 = 1.0; h = 0.01; tmin = 0.0; tmax = 20.0;
ts0, xs0 = rk3_solve(x0, h, tmin, tmax)
ts1, xs1 = rk3_solve(x0, h/2.0, tmin, tmax)
ts2, xs2 = rk3_solve(x0, h/4.0, tmin, tmax)
rs = (xs0 - xs1[::2]) / (xs1[::2] - xs2[::4])
plt.plot(ts0, rs, 'ro')
plt.xlim(tmin, tmax); plt.ylim(0, 12)
plt.xlabel('t', fontsize=14); plt.ylabel('ratio', fontsize=14)
plt.show()
```

Solution: part (b)



Multi-step methods

- Runge-Kutta methods take one step forward by combining several intermediate steps:
- The intermediate steps are **thrown away** . . . seems wasteful & inefficient
- Multi-step methods **reuse** information from earlier times
- **Linear** multi-step methods get x_{n+1} from a linear combination of $\{x_n, x_{n-1}, \ldots\}$.

Linear multi-step methods

$$\boxed{\frac{dx}{dt} = f(x,t), \qquad x(t_0) = x_0}$$

- **Example: The two-step method:**

$$x_{n+1} = x_n + h\left(\frac{3}{2}f(x_n,t_n) - \frac{1}{2}f(x_{n-1},t_{n-1})\right)$$

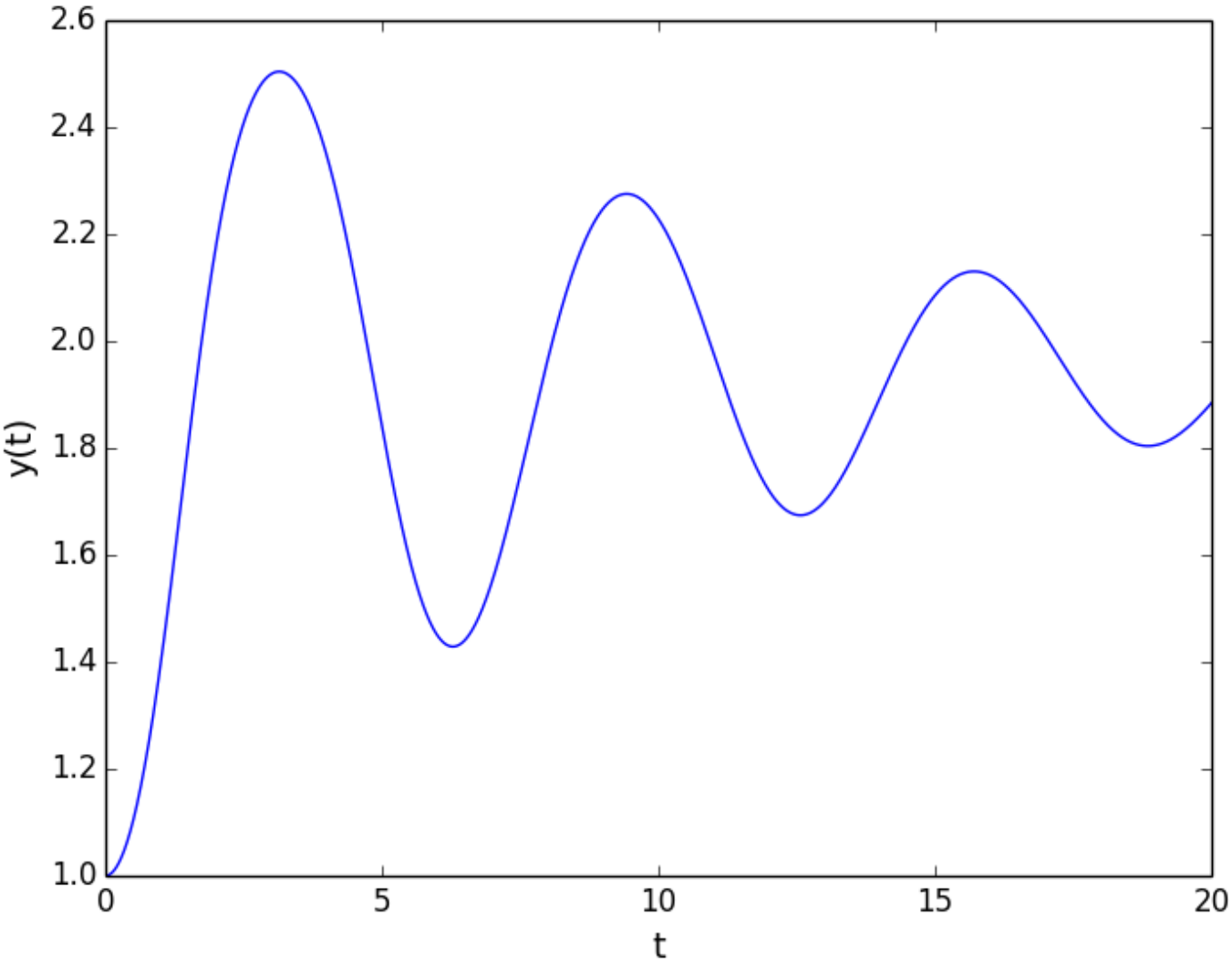
- The method uses both x_n and x_{n-1}.
- It is second order accurate (*Exercise:* show this).
- **Q**: How do we take the very first step?
- **A**: Use an alternative method of same order.

Implementation of the two-step method

$$x_{n+1} = x_n + h\left(\frac{3}{2}f(x_n, t_n) - \frac{1}{2}f(x_{n-1}, t_{n-1})\right)$$

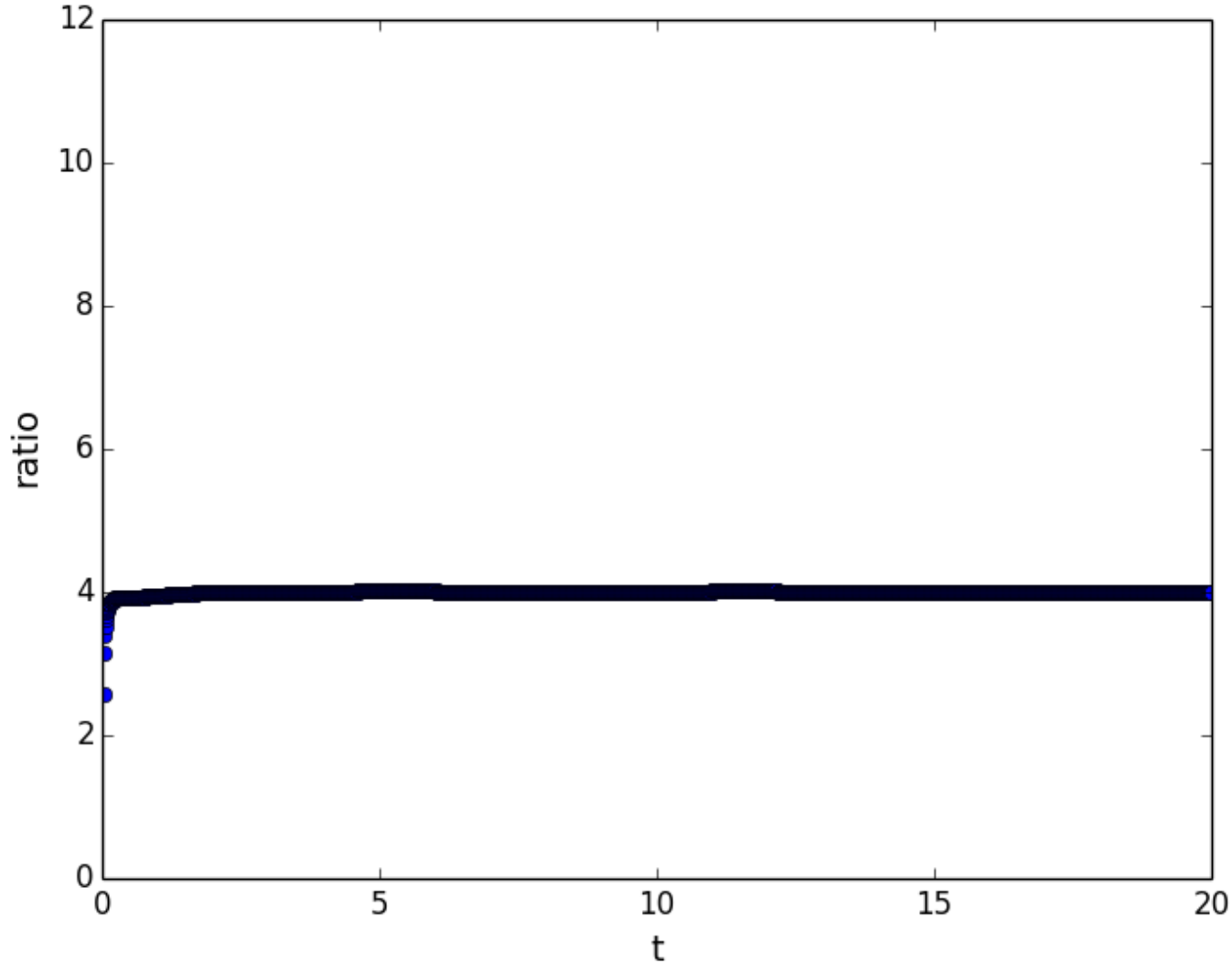
```
def twostep_solve(x0, h, tmin, tmax):
    n = int((tmax - tmin)/h) + 1
    ts = np.zeros(n)
    ts[0] = tmin
    xs = np.zeros(n)
    xs[0] = x0  # Use midpoint method for first step.
    xhalf = xs[0] + 0.5*h*dx_dt(xs[0], ts[0])
    xs[1] = xs[0] + h*dx_dt(xhalf, ts[0]+h/2.0)
    # Now apply the two-step method.
    for k in range(1,n-1):
        ts[k+1] = ts[k] + h
        xs[k+1] = xs[k] + (h/2.0)*(3*dx_dt(xs[k], ts[k])
                                   - dx_dt(xs[k-1], ts[k-1]))
    return ts, xs
```

Plot



(as before)

Ratio test



(demonstrating the two-step method is 2nd order accurate)

Adams-Bashforth methods

- The two-step method is an example of a linear multi-step method
- **Explicit** linear multi-step methods go by the name of Adams-Bashforth methods:

$$
\begin{aligned}
y_{n+1} &= y_n + h\, f_n && \text{(Euler method)} \\
y_{n+1} &= y_n + h\left(\frac{3}{2} f_n - \frac{1}{2} f_{n-1}\right) && \text{(2-step method)} \\
y_{n+1} &= y_n + h\left(\frac{23}{12} f_n - \frac{4}{3} f_{n-1} + \frac{5}{12} f_{n-2}\right) \\
y_{n+1} &= y_n + h\left(\frac{55}{24} f_n - \frac{59}{24} f_{n-1} + \frac{37}{24} f_{n-2} - \frac{3}{8} f_{n-3}\right) \\
y_{n+1} &= \dots
\end{aligned}
$$

where

$$f_k = f(y_k, t_k)$$

Implicit Methods

Stiff sets of equations

- **Stiffness** occurs when there are two or more **very different scales** in t over which the dependent variables **x** are changing.
- Consider this innocent-looking 2D linear system:

$$\begin{aligned} \dot{x} &= 998x + 1998y, & x(0) &= 1, \\ \dot{y} &= -999x - 1999y, & y(0) &= 0. \end{aligned}$$

Stiff sets of equations

- **Stiffness** occurs when there are two or more **very different scales** in t over which the dependent variables **x** are changing.
- Consider this innocent-looking 2D linear system:

$$\begin{aligned} \dot{x} &= 998x + 1998y, & x(0) &= 1, \\ \dot{y} &= -999x - 1999y, & y(0) &= 0. \end{aligned}$$

- It has a simple exact solution,

$$x(t) = 2e^{-t} - e^{-1000t}, \quad y(t) = -e^{-t} + e^{-1000t}.$$

- But . . . **all** our explicit methods become **unstable** when applied to this system, if the step size is $h \gtrsim 1/1000$ (**!**) [see Lab Class 6].

Implicit methods

- Methods are only useful if they are **stable**, i.e. not subject to spurious divergences
- **Explicit** methods lead to instabilities for stiff equations. Even worse stability issues when applied to **partial** differential equations
- **Implicit** methods have different (often better) stability properties, but are harder to implement
- **Example:** The **backwards Euler method**

$$x_{n+1} = x_n + h\, f(x_{n+1}, t_{n+1})$$

- Implicit, because we cannot write x_{n+1} as an explicit function of x_n (unless f is linear).

Stability: forward Euler method

- Consider a set of linear equations with constant coefficients,

$$\dot{\mathbf{x}} = -\mathbf{C} \cdot \mathbf{x},$$

  where $\mathbf{C}$ is a positive-definite matrix (all eig.values $\lambda_i > 0$).

- **Explicit differencing** (**forward** method) gives

$$\mathbf{x}_{n+1} = (\mathbf{I} - h\mathbf{C}) \cdot \mathbf{x}_n$$

- $\mathbf{C}$ positive-definite $\Rightarrow$ $\mathbf{C} = \mathbf{A}^{-1}\mathbf{\Lambda}\mathbf{A}$ with $\mathbf{\Lambda}$ diagonal.

$$\begin{aligned} \mathbf{x}_n &= (\mathbf{I} - h\mathbf{C})^n \cdot \mathbf{x}_0 \\ &= \mathbf{A}^{-1}(\mathbf{I} - h\mathbf{\Lambda})^n \mathbf{A} \cdot \mathbf{x}_0. \end{aligned}$$

- As $n \to \infty$, the solution **converges** iff $1 - h\lambda_{\max} > -1$,

$$\Rightarrow \quad h < \frac{2}{\lambda_{\max}}.$$

Stability: backward Euler method

- **Implicit differencing** (**backward** method) gives

$$\begin{aligned} (\mathbf{I} + h\mathbf{C})\,\mathbf{x}_{n+1} &= \mathbf{x}_n \\ \Rightarrow \quad \mathbf{x}_{n+1} &= (\mathbf{I} + h\mathbf{C})^{-1} \cdot \mathbf{x}_n \end{aligned}$$

- It follows that

$$\Rightarrow \quad \mathbf{x}_n = \mathbf{A}^{-1}\,(1 + h\mathbf{\Lambda})^{-n}\,\mathbf{A} \cdot \mathbf{x}_0.$$

- This converges for **any** $h > 0$, since

$$(1 + h\lambda_i)^{-1} < 1$$

  (**C** is positive-definite $\Leftrightarrow$ $\lambda_i > 0$).

- Backward differencing is **robust** for **stiff** problems

The backwards Euler method

$$x_{n+1} = x_n + h\, f(x_{n+1}, t_{n+1})$$

- **Q**. How do we actually find x_{n+1}?
- **A1.** If ODE is linear, then find the matrix inverse.
- **A2.** If ODE is non-linear, apply an iterative method such as fixed-point iteration or Newton-Raphson method at each step.
- **Fixed-point iteration:**

$$x_{k+1}^{[0]} = x_k, \qquad x_{k+1}^{[i+1]} = x_k + h\, f(x_{k+1}^{[i]}, t_{k+1})$$

- Iterate until some convergence criterion is met:

$$\left| x_{k+1}^{[i+1]} - x_{k+1}^{[i]} \right| \leq \epsilon$$

The backwards Euler method

```
def backEuler(h=0.01, xmin=0.0, xmax=1.0):
    n = round((xmax - xmin)/h)
    print(n)
    ts = xmin + h*np.arange(0,n+1)
    xs = np.zeros(n+1); ys = np.zeros(n+1)
    xs[0] = 1; ys[0] = 0;
    a = 1; b = 1000;
    Amat = np.matrix([[1-998*h, -1998*h], [999*h, 1+1999*h]])
    Ainv = np.linalg.inv(Amat)
    for k in range(n):
        x, y = xs[k], ys[k]
        xs[k+1] = Ainv[0,0]*x + Ainv[0,1]*y
        ys[k+1] = Ainv[1,0]*x + Ainv[1,1]*y
    return ts, xs, ys
```