Numerical Solutions to ODEs
Part II

CMPT 419/983

18/09/2019

Stiff Equations

- ODEs with components that have very fast rates of change
  - Usually requires very small step sizes for stability

- Example: $\dot{x}_1 = ax_1$ with forward Euler
  - Stability requires $|1 + ha| \leq 1$
  - For $a = -100$, we have $|1 - 100h| \leq 1 \Leftrightarrow h \leq 0.02$

- Small step size is required even if there are other slower changing components like $\dot{x}_2 = x_1 - x_2$
  - Implicit methods (eg. backward Euler) are useful here

$$\dot{x}_1 = -100x_1$$
$$\dot{x}_2 = x_1 - x_2$$

$$\dot{x} = \begin{bmatrix} -100 & 0 \\ 1 & -1 \end{bmatrix} x$$

Stiff Equations

Eigenvalues of A are $\sigma(A) = \{-1, -100\}$

- Example:

$$\dot{x}_1 = -100x_1$$
$$\dot{x}_2 = x_1 - x_2$$

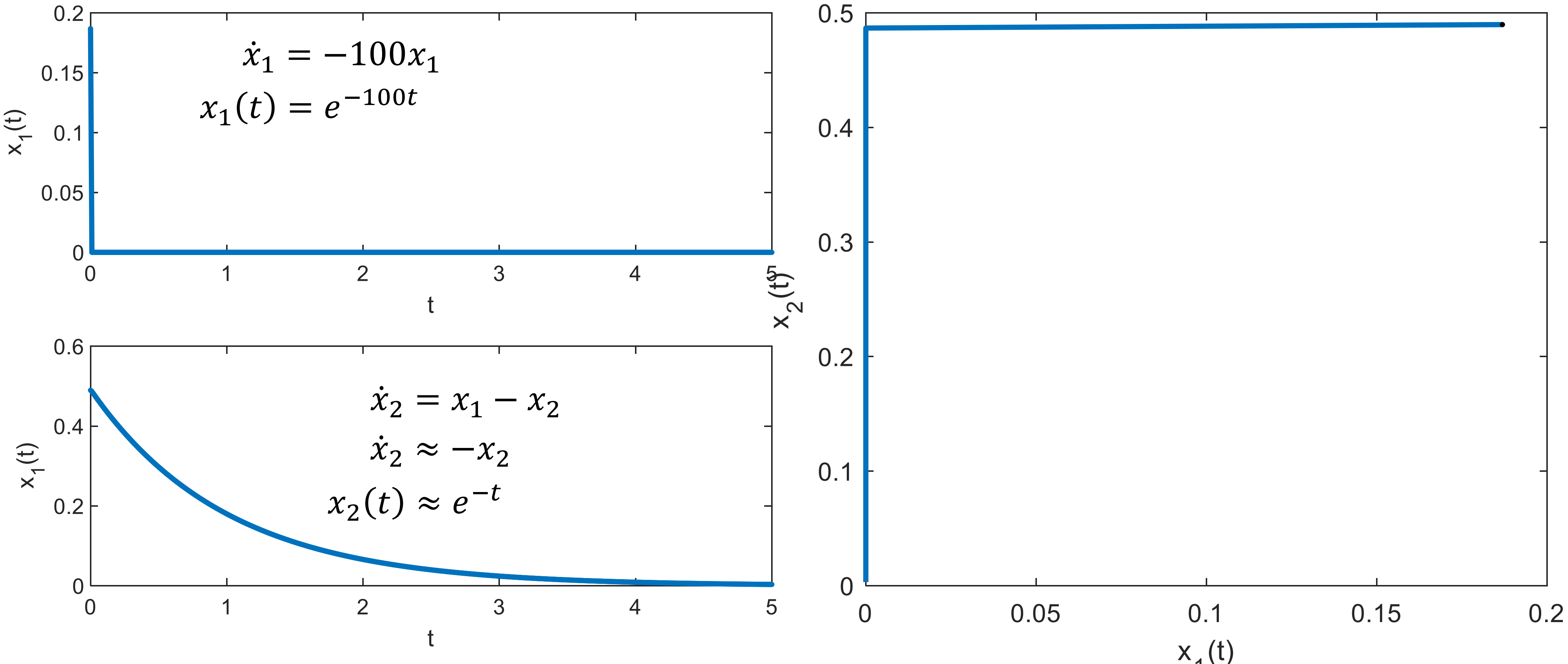
$$\dot{x} = Ax = \begin{bmatrix} -100 & 0 \\ 1 & -1 \end{bmatrix} x$$

  - Forward Euler:

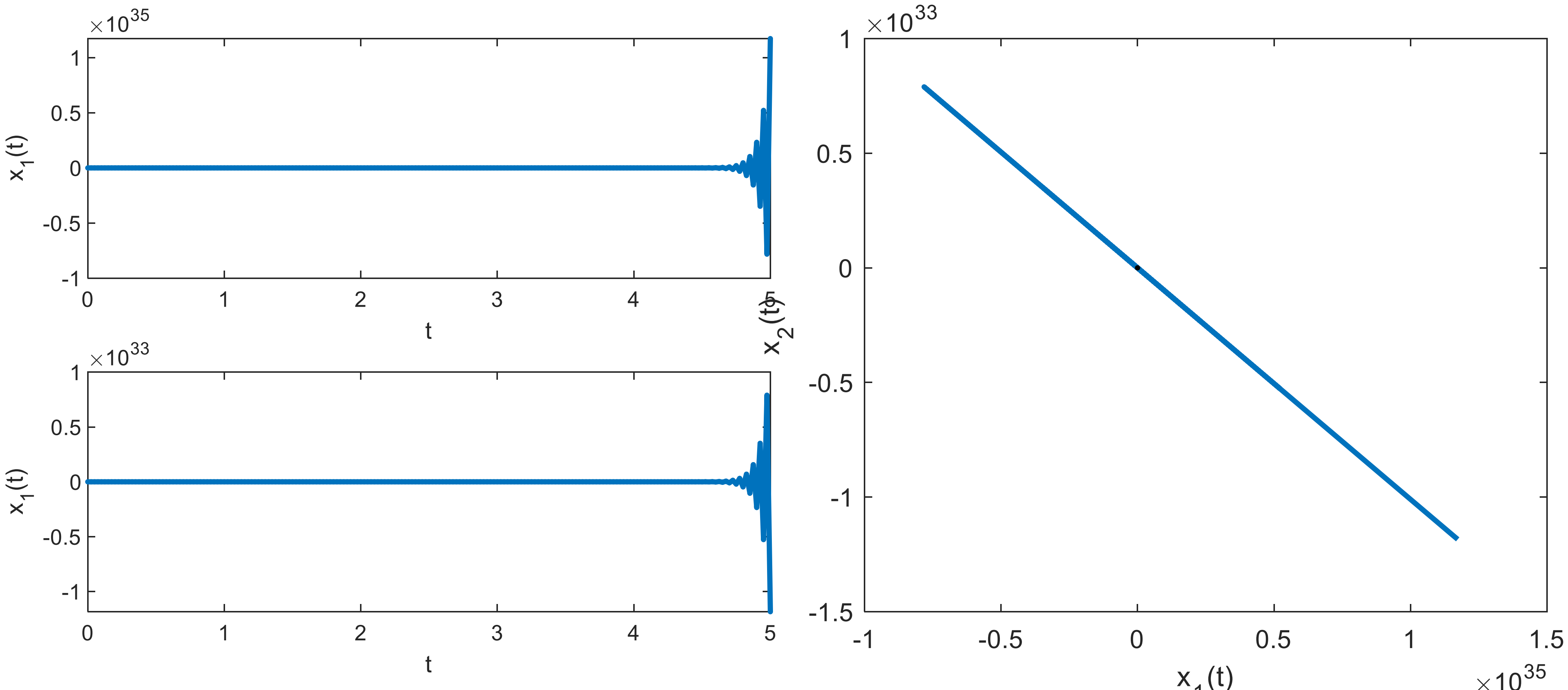
$$\begin{aligned} y^{k+1} &= y^k + hf\left(y^k\right) \\ &= y^k + hAy^k \\ &= (I + hA)y^k \end{aligned}$$

    - Eigenvalues of hA: $-h, -100h$
    - Eigenvalues of $I + hA$ are $\{1 + h\sigma(A)\}$: $1 - h$ and $-100h$
    - So, we need $|1 - h| < 1$ and $|1 - 100h| < 100 \Rightarrow h < 0.02$

Forward Euler, $h = 0.01$

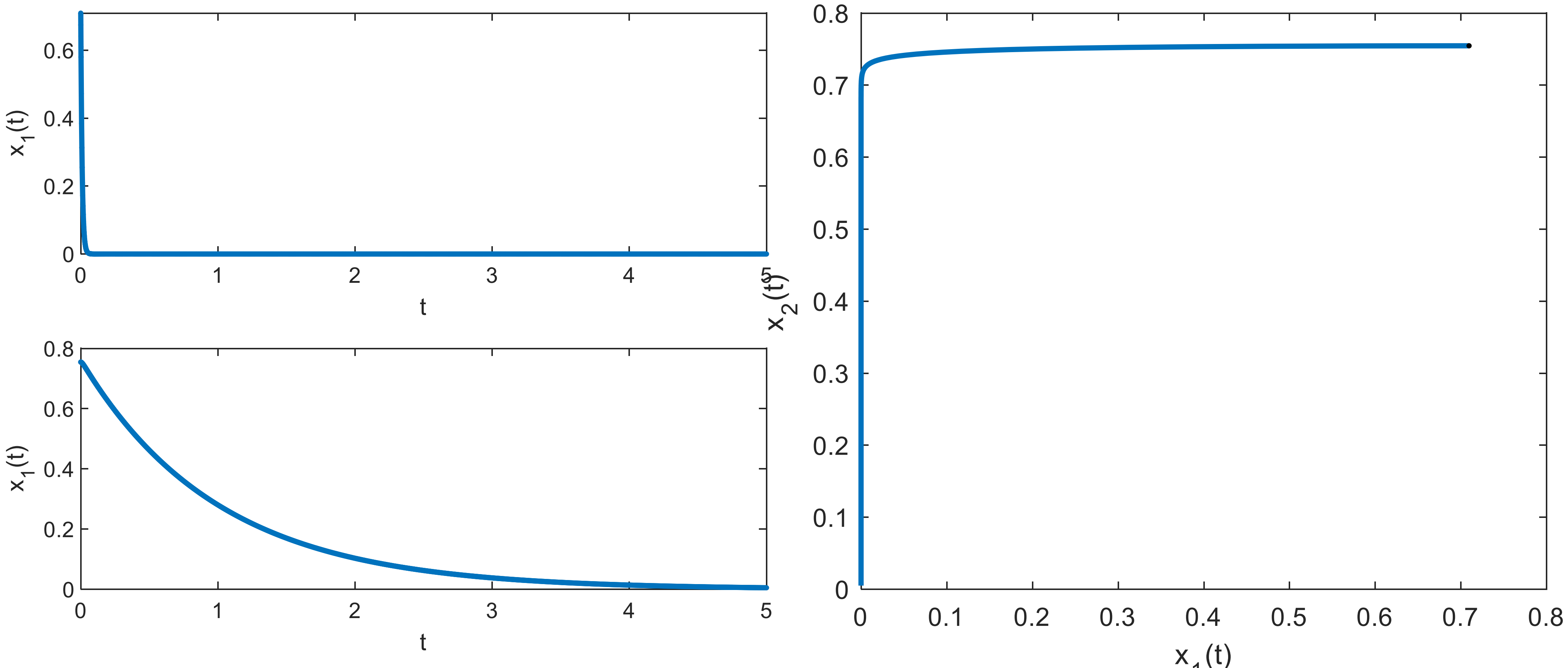


Forward Euler, $h = 0.025$



Matlab’s ode45 Solver (Explicit Method)

- Automatically chosen variable time steps: $h \approx 0.002$ to $h \approx 0.008$

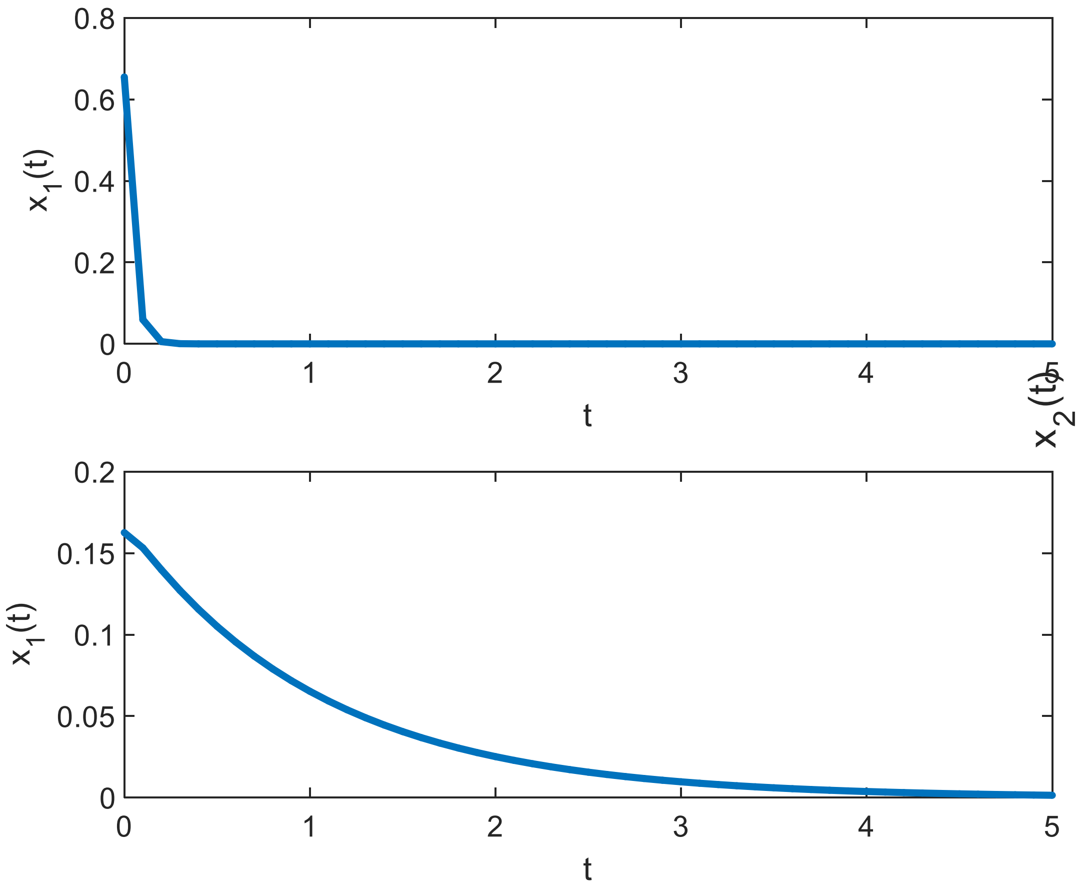


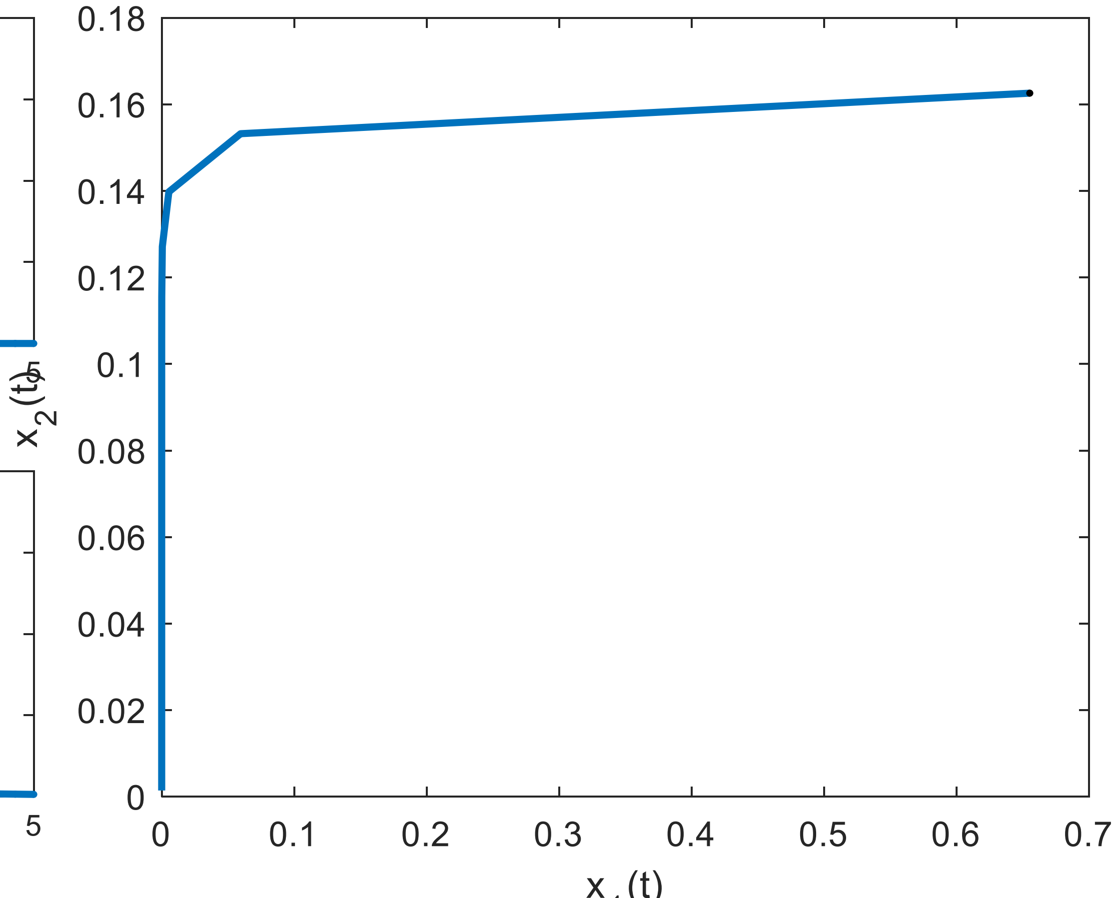
Backward Euler, $h = 0.01$

- Our system: $\dot{x} = Ax$
- Backward Euler:
  - $y^{k+1} = y^k + hf\left(y^{k+1}\right)$
  - $y^{k+1} = y^k + hAy^{k+1}$
  - $(I - hA)y^{k+1} = y^k$
  - $y^{k+1} = (I - hA)^{-1}y^k$
    - Eigenvalues of $(I - hA)^{-1}$ are $\left(1 - h\sigma(A)\right)^{-1}$
    - No restrictions on h if eigenvalues of A have negative real part

Backward Euler, $h = 0.1$

- Not super accurate, but stable
- Relatively slow for the same h due to inverse: $y^{k+1} = (I - hA)^{-1} y^k$





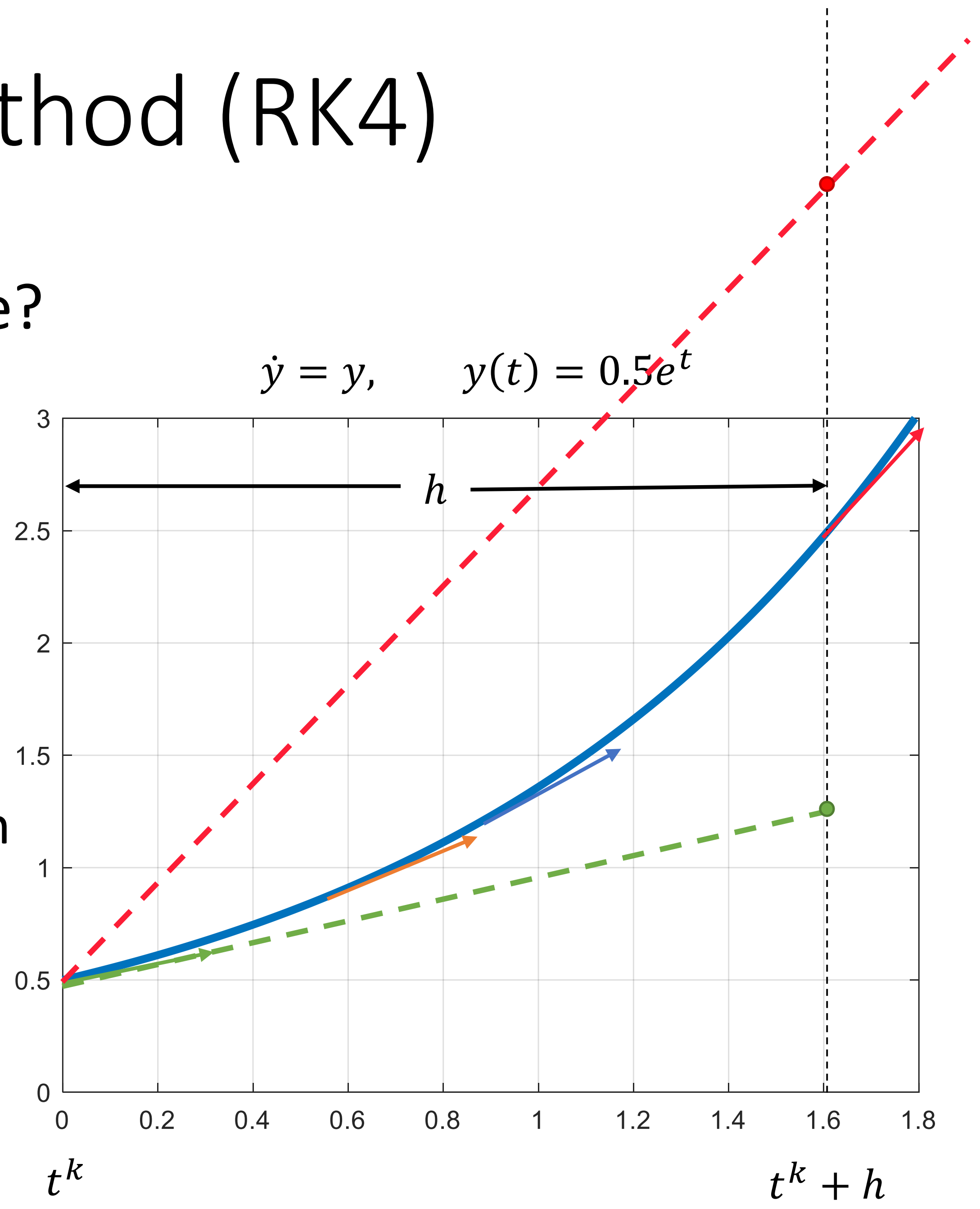
Numerical Solutions of ODEs

- In general, $\dot{x} = f(x, u)$ does not have a closed-form solution
  - Instead, we usually compute numerical approximations to simulate system behaviour
  - Done through discretization: $t^k = kh,\ u^k \coloneqq u(t^k)$
    - h represents size of time step
  - Goal: compute $y^k \approx x(t^k)$

- Key considerations
  - Consistency: Does the approximation satisfy the ODE as $h \to 0$?
  - Accuracy: How fast does the solution converge?
  - Stability: Do approximation error remain bounded over time?
  - Convergence: Does the solution converge the true solution as $h \to 0$?

Classical Runge-Kutta Method (RK4)

- Main consideration: what slope to use?
  - Forward Euler: slope at beginning
    $$y^{k+1} = y^k + hf\left(y^k, u^k\right)$$
  - Backward Euler: slope at the end
    $$y^{k+1} = y^k + hf\left(y^{k+1}, u^k\right)$$
  - In general, we can use anything between t^k and t^{k+1}
  - Classical Runge-Kutta: weighted average

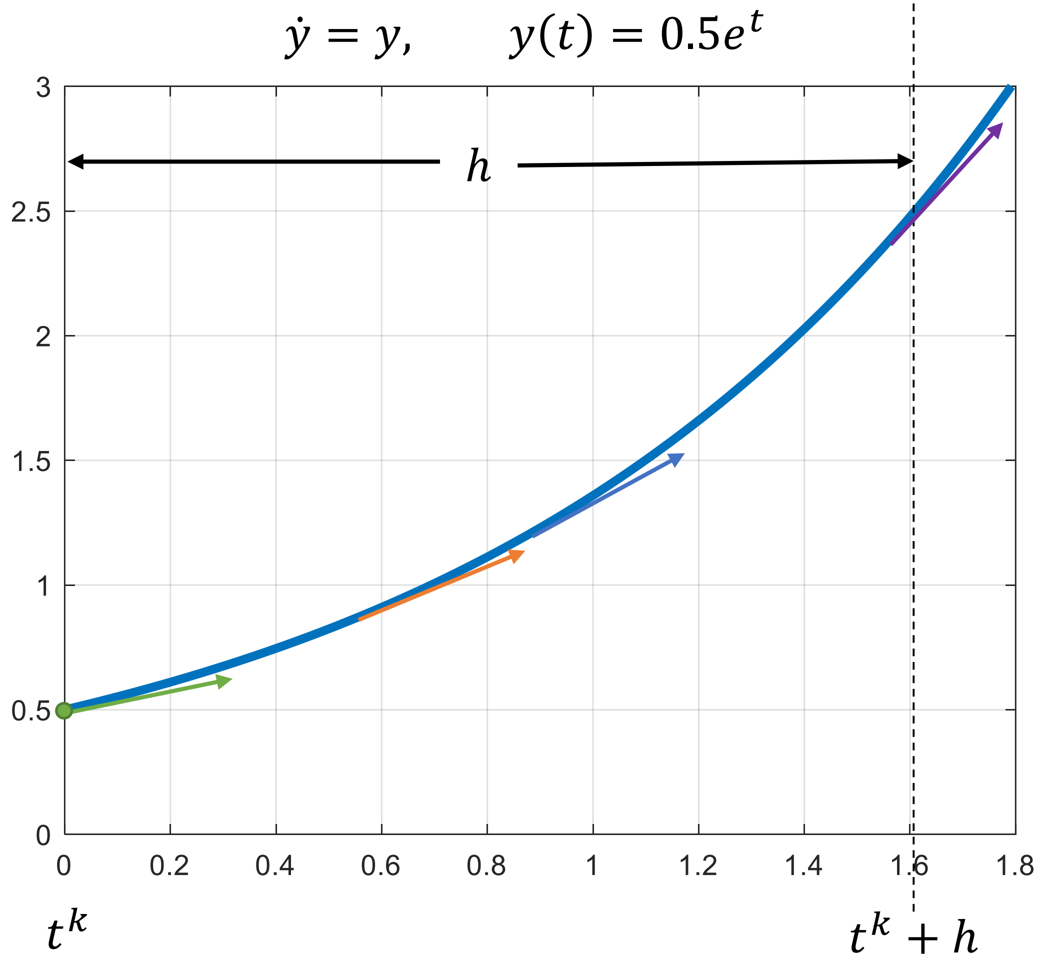


Classical Runge-Kutta Method (RK4)

- Main consideration: what slope to use?
  - Weighted average

- $y^{k+1} = y^k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
  - $k_1 = hf\left(t^k, y^k\right)$
  - $k_2 = hf\left(t^k + \frac{h}{2}, y^k + \frac{k_1}{2}\right)$
  - $k_3 = hf\left(t^k + \frac{h}{2}, y^k + \frac{k_2}{2}\right)$
  - $k_4 = hf\left(t^k + h, y^k + k_3\right)$

- Properties
  - Equivalent to Simpson's rule
  - 4$^{\text{th}}$ order accuracy



Classical Runge-Kutta Method (RK4)

- One of the most widely used methods
  - $y^{k+1} = y^k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
    - $k_1 = hf(t^k, y^k)$
    - $k_2 = hf\left(t^k + \frac{h}{2}, y^k + \frac{k_1}{2}\right)$
    - $k_3 = hf\left(t^k + \frac{h}{2}, y^k + \frac{k_2}{2}\right)$
    - $k_4 = hf(t^k + h, y^k + k_3)$
- Intuitively: estimate y^{k+1} using weighted average of slopes
- Mathematically: can show
  - Consistency: $\frac{\|e^k\|}{h} \rightarrow$ as $h \rightarrow 0$
  - Stability for small enough h
  - Consistency + stability $\Leftrightarrow$ convergence (4th order)

Numerical Solutions: Discussion

- Stiff equations
- Approximation errors
  - Typically cannot be used to prove system properties
- Simulations cannot capture all system behaviours
- Libraries:
  - Matlab: ode__ → ode45, ode113, etc. (ode__s for stiff equations)
  - Python: scipy.integrate.odeint
  - C++: odeint