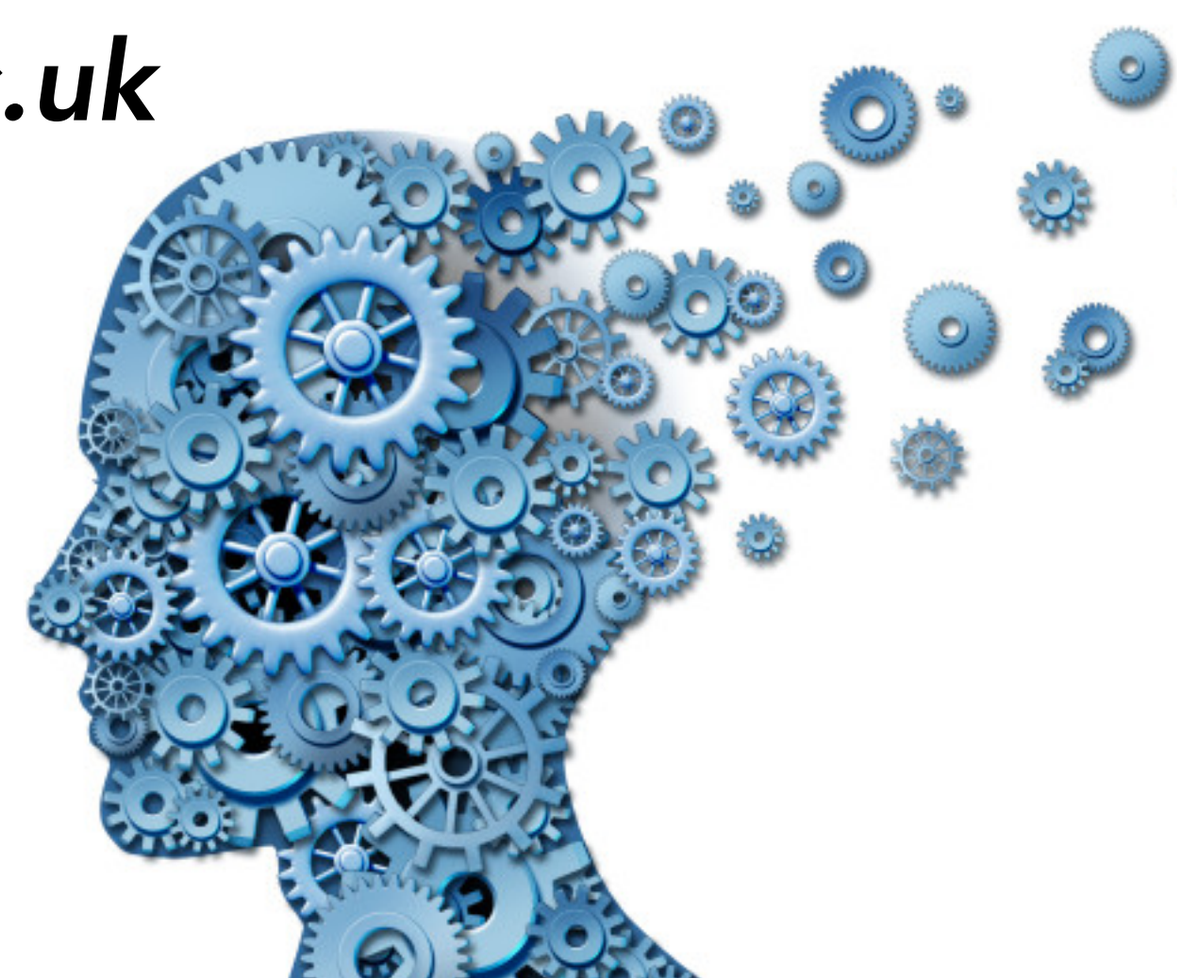
COMS30127/COMSM2127
Computational Neuroscience

Lecture 5: Numerical methods

Dr. Cian O'Donnell

cian.odonnell@bristol.ac.uk



Intended learning outcomes

- [Recap on Taylor series]
- Be able to implement the Euler method of numerical integration.
- Understand the idea behind the Runge-Kutta method.
- Have a basic understanding of how to use automatic ODE solvers.

Taylor series

- The Taylor series is representation of a function as an infinite sum of terms, where the terms are based on the function's derivatives at some point t_0:

$$f(t) = \sum_{n=0}^{\infty} \frac{1}{n!} \frac{d^n f}{dt^n}\bigg|_{t=t_0} t^n$$

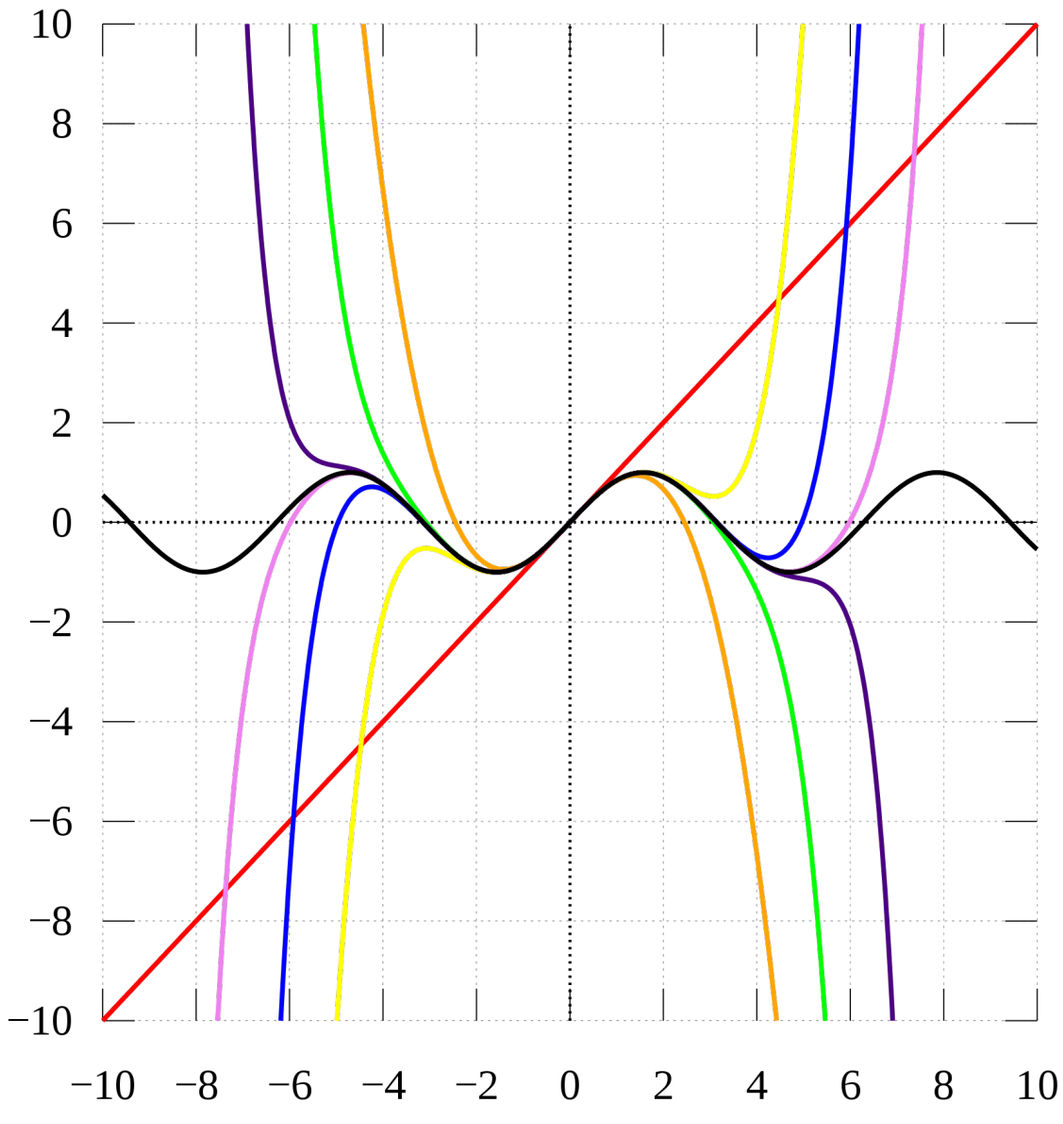
- It is typically used to approximate a function local to some point of interest, by truncating the series after a small number of terms:

$$f(t) \approx f(t_0) + \frac{1}{1!}\frac{df}{dt}(t - t_0) + \frac{1}{2!}\frac{d^2 f}{dt^2}(t - t_0)^2 + \frac{1}{3!}\frac{d^3 f}{dt^3}(t - t_0)^3$$

Taylor series

- For example take $f(t) = \sin t$:

$$\sin t \approx \sin 0 + \cos 0(t) - \frac{1}{2}\sin 0(t^2) - \frac{1}{6}\cos 0(t^3)\ldots$$



https://en.wikipedia.org/wiki/Taylor_series#/media/File:Sintay_SVG.svg

Numerically solving DEs

- In the previous lecture we saw how to solve an easy ODE analytically.
- However for almost any interesting differential equation model of a real-world system, we just can't solve them analytically, perhaps because:
  - the function is nonlinear.
  - we don't know $g(t)$ analytically.
- Do we give up? No! Instead we use computers to approximate the solution function as a series of numbers.
- Most of the algorithms work iteratively: $x(t + \delta t) = f(x(t), t)$
- There are many algorithms for doing this which vary in complexity, accuracy, and computational expense.
- Solving DEs is very much a "no free lunch" problem, there is no globally optimal solver algorithm. Which algorithm performs best varies on a case-by-case basis.

Euler's method

Leonhard Euler
1707-1783

- The Euler method is the simplest way to numerically solve a differential equation.
- It is based on fitting a straight line tangent to the function at the current point in time t, and extrapolating forward to the function value at the next time step $t + \delta t$.
- Take the Taylor series for some function $f(t)$ taken about the current timepoint $t = n\delta t$, with a view to estimating the function at the next timepoint:

$$f(n\delta t + \delta t) = f(n\delta t) + \left.\frac{df}{dt}\right|_{t=n\delta t} \delta t + \frac{1}{2}\left.\frac{d^2 f}{dt^2}\right|_{t=n\delta t} (\delta t)^2 + \ldots$$

$$\approx f(n\delta t) + \left.\frac{df}{dt}\right|_{t=n\delta t} \delta t$$

y = c + m x

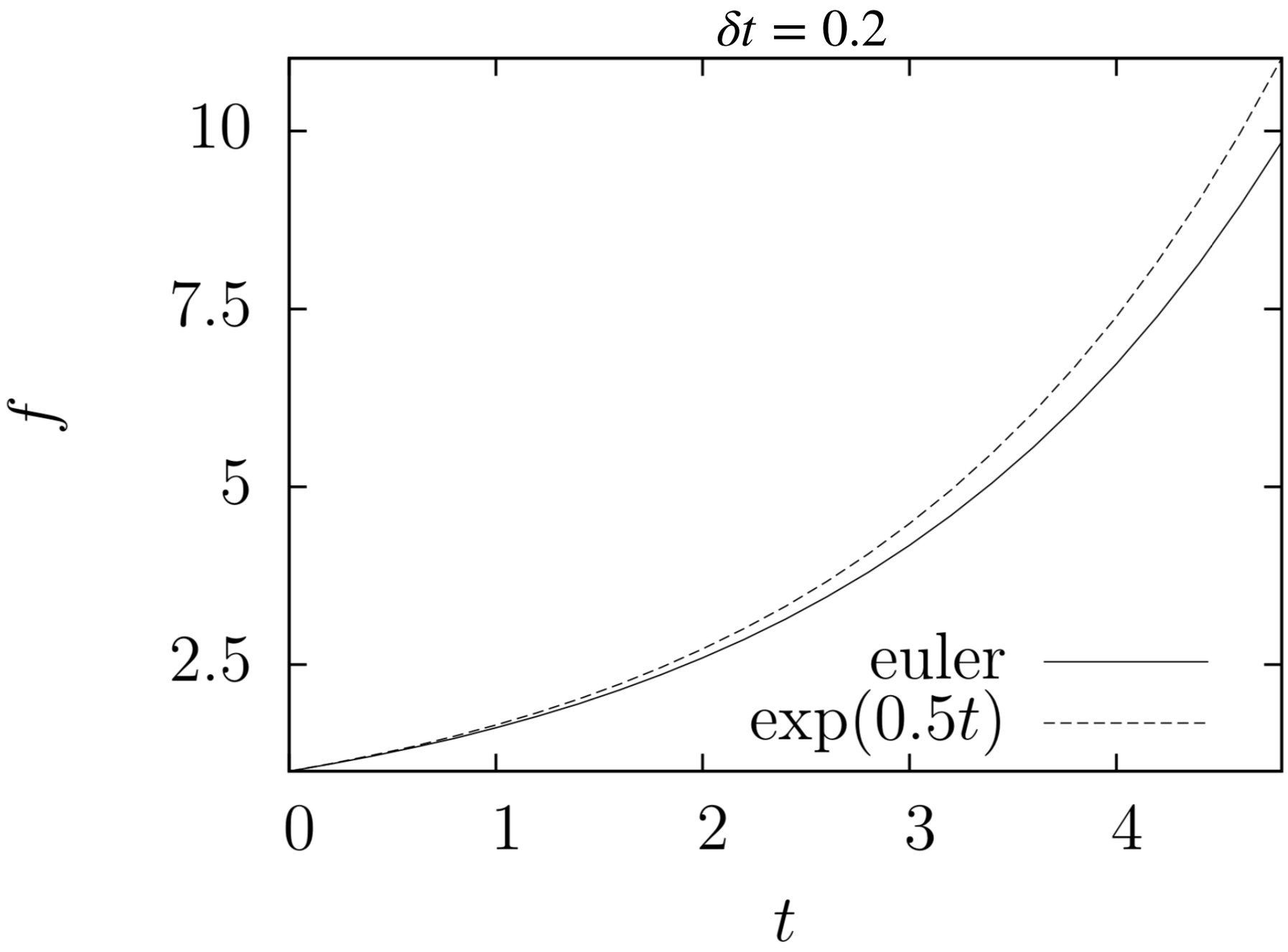
Euler's method

- This approximation works well if δt is small, because then δt^2, δt^3, etc are even smaller, and the Taylor series terms associated with them become negligible.

- However it also means that the error of the approximation scales $\sim \mathcal{O}(\delta t^2)$.

- We can say $f(t + \delta t) = (\text{Euler approximation}) + \mathcal{O}(\delta t^2)$

Euler's method



- From Conor's notes: https://github.com/coms30127/2019_20/notes/numerical

Runge-Kutta method

- Runge-Kutta is conceptually similar to Euler, except it accounts for more terms of the Taylor series expansion.
- The most common version of Runge-Kutta is "fourth order": $f(t+\delta t) = (\text{R-K approximation}) + \mathcal{O}(\delta t^5)$.
- The derivation is quite involved, here is the method assuming we are given $\frac{df}{dt} = G(f,t)$:
  - First calculate the following values:

$$k_1 = G(f_n)$$

$$k_2 = G(f_n + \frac{1}{2}\delta t k_1)$$

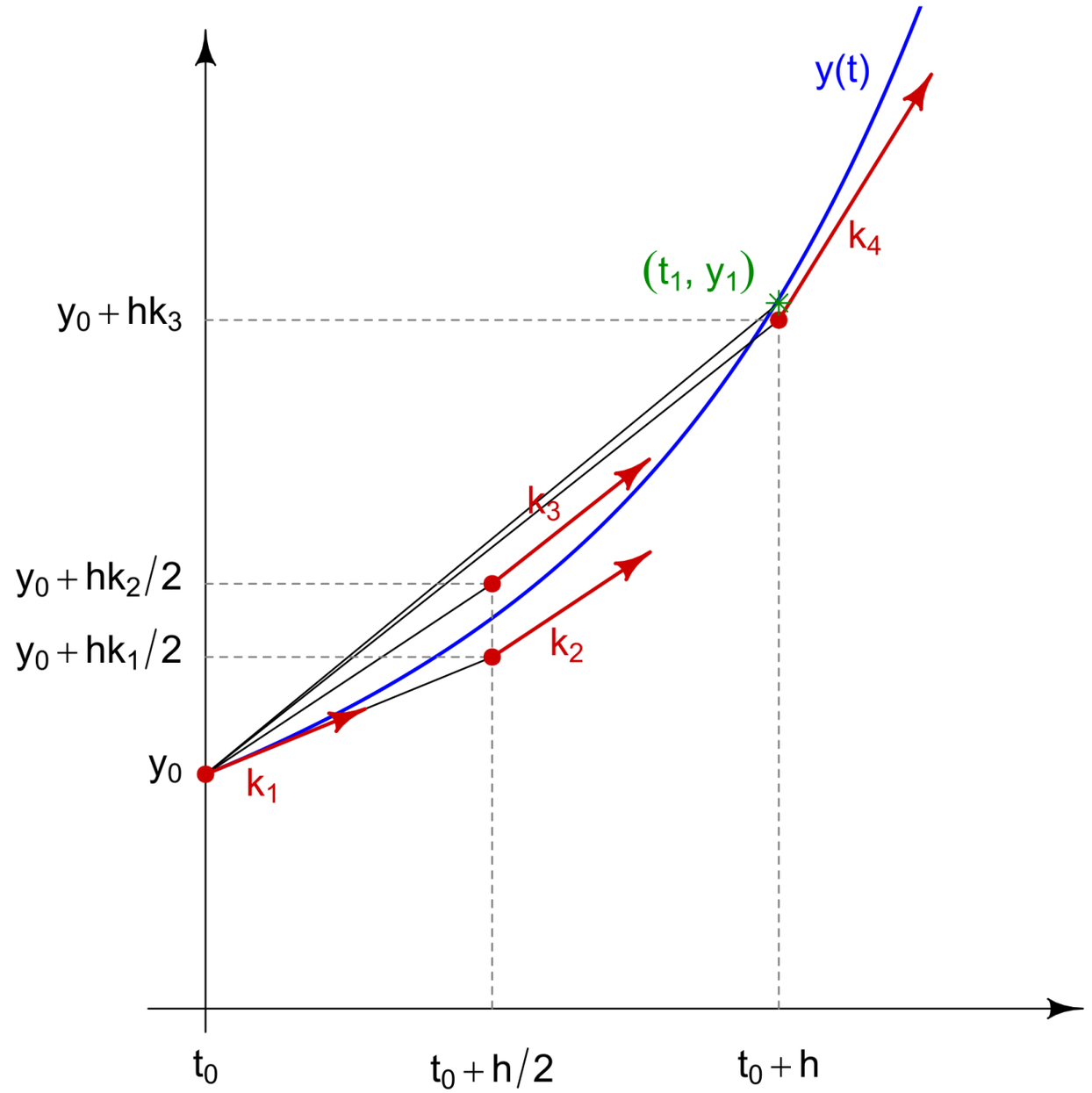
$$k_3 = G(f_n + \frac{1}{2}\delta t k_2)$$

$$k_4 = G(f_n + \delta t k_3)$$

  - Then the final estimate for the function at the next time point is:

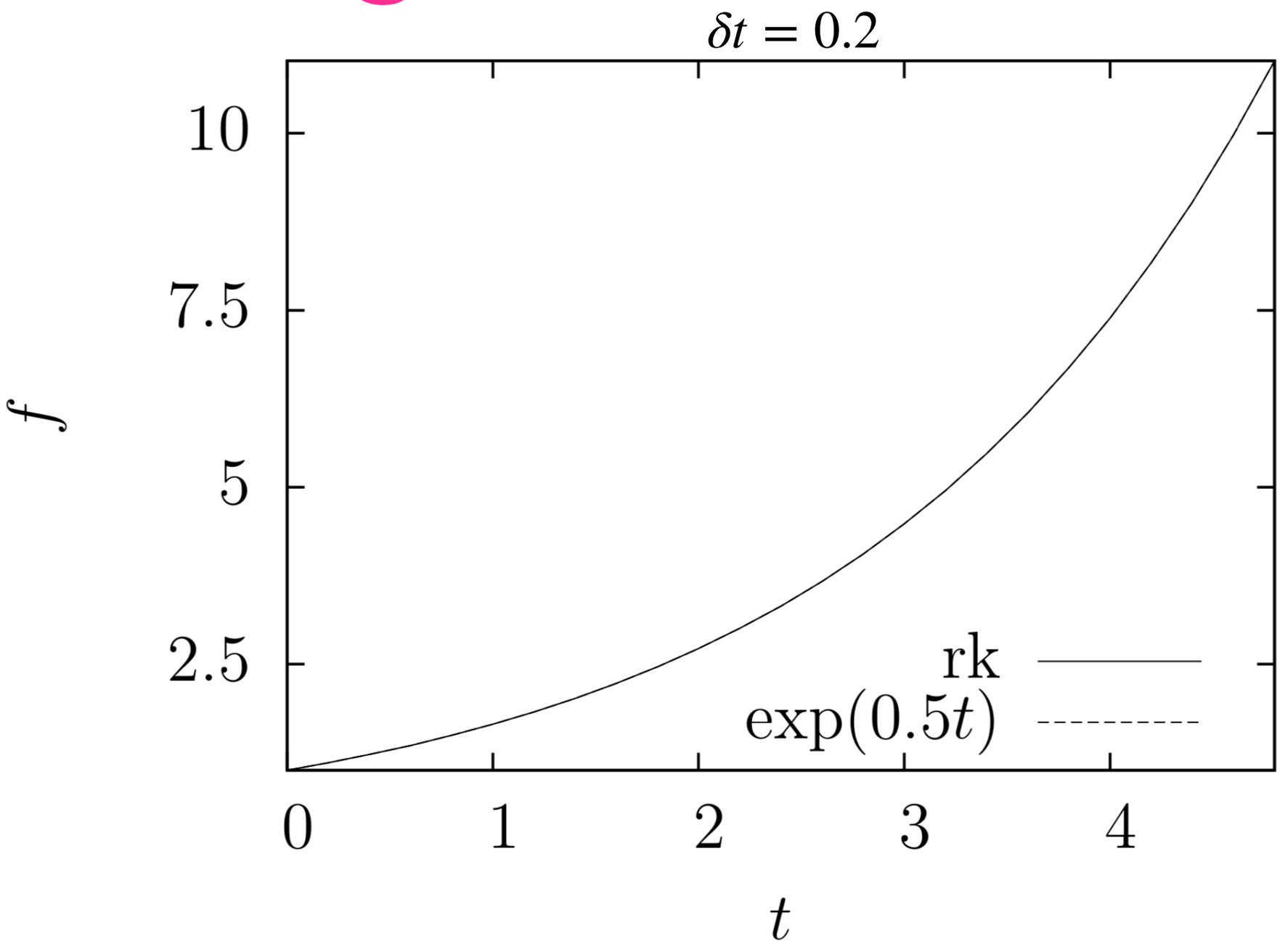
$$f_{n+1} = f_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\delta t$$

Runge-Kutta method



- There is a pretty good diagram with slightly different notation at: https://en.wikipedia.org/wiki/Runge–Kutta_methods#The_Runge–Kutta_method

Runge-Kutta method



- From Conor's notes: https://github.com/coms30127/2019_20/notes/numerical

Euler vs RK in practise

- Both Euler and Runge-Kutta are widely used in practise.
- Because of its simplicity, Euler is especially common for quick-and-dirty models when people want to manually code something. You will use it in your coursework.
- Coding Runge-Kutta is more involved (usually though we just use a DE solver package that someone else has written in a black-box way).
- As you might guess Runge-Kutta is also computationally more expensive than Euler, per timestep.
- However RK's gain in accuracy over Euler usually more than offsets the increased computation time. This means that with Runge-Kutta we can get away with bigger timesteps, making it more efficient than Euler overall.

How do we set the timestep?

- The only user-defined parameter in Euler and RK is the timestep size. How do we choose it?

- We could use the Taylor series to estimate the magnitude of the error, decide what error we are comfortable living with, and set the timestep accordingly.

- However the size of the error likely varies at different points in the solution.

- In practise we can just test a range of timestep sizes and see at what value the numerical solution starts to stabilise.

- This choice can be informed by examining the timescales in the DEs. Usually we would want our timestep to be $< 10\times$ smaller than the fastest time constant in the differential equation.

ODE solving in practise

- Euler and RK are two (families of) ODE solver methods from a large set.
- They are both examples of "fixed timestep" solvers: the user chooses the timestep and leaves it the same for the duration of the computation.
- However many ODE problems are "stiff", meaning that they need a **painfully** small timestep to avoid numerical instability.
- In systems where the solution move between periods of fast changes and periods of slow changes, we can do better with "adaptive timestep" methods.
- Adaptive methods look at the magnitude of the derivatives online while solving, and take larger timesteps while the derivatives are small. The size of the timestep is computed to match some user-specified "tolerance" for the error.

ODE solving in practise

- Most of the time we hand the job of solving to pre-written packages.
- Common languages used in scientific computing often have ODE solver packages with many method options:

Python	MATLAB	Julia
vode	ode45	Euler
zvode	ode23	Midpoint
lsoda	ode113	SSPRK22
dopri5	ode15s	SSPRK33
dop853	ode23s	SSPRK104
	ode23t	RK4
	ode23tb	BS3
	ode15i	DP5
		Tsit5
		BS5
		Vern6
		Vern7
		TanYam7
		DP8
		TsitPap8
		Vern8
		Vern9
		Feagin10
		Feagin12
		Feagin14
		ImplicitEuler
		Trapezoid
		Rosenbrock23
		Rosenbrock32

Summary

- Recap on Taylor series.
- Euler method: basically estimating the function at next timestep by extrapolating a tangent about the current point.
- Runge-Kutta method: like Euler but with a weighted average for the derivative to fit the extrapolation, by using higher order terms in the Taylor series.
- Some tips on how to solve ODEs in practise.
- Suggest also reading Conor's elegant notes: https://github.com/coms30127/2019_20/notes/numerical