Week 3: Stability, convergence and higher order ODEs

- Convergence versus stability, higher order ODEs, higher order explicit methods

Dr K Clough, Topics in Scientific computing, Autumn term 2025

Plan for today

1. Revision of last week - Classes, ODEs and integration
2. Convergence - how do you know it has worked?
3. The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)
4. How to make a higher order ODE into a first order one
5. Tutorial this week - classes, multistep methods and second order ODEs

Classes

Classes encapsulate all the attributes of some concept or thing, and all the methods that could be applied to it

```python
# Cat class

class FluffyCat :

    """
    Represents a fluffy cat

    Attribute: colour

    Methods: print the colour of the cat, change colour of cat

    """

    cat_colours = ["black", "ginger", "pink"]

    # constructor function
    def __init__(self, colour = cat_colours[0]):
        self.colour = colour

    def print_colour(self) :
        print(self.colour)

    def change_colour(self, new_colour) :
        assert new_colour in self.cat_colours, 'Need to specify one of the allowed cat colours'
        self.colour = new_colour
```

```python
my_cat = FluffyCat()
my_cat.change_colour(FluffyCat.cat_colours[2])
my_cat.print_colour()

my_cat.change_colour("green") #Returns an error
```

```
pink
```

Ordinary differential equations

What are the features of this ODE?

$$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x^2 + x - 1 = \sin(t)$$

Ordinary differential equations

One independent variable t
so ODE not PDE

One dependent variable
x so dimension 1

$$\frac{d^2x}{dt^2} + \frac{dx}{dt} + x^2 + x - 1 = \sin(t)$$

Second order

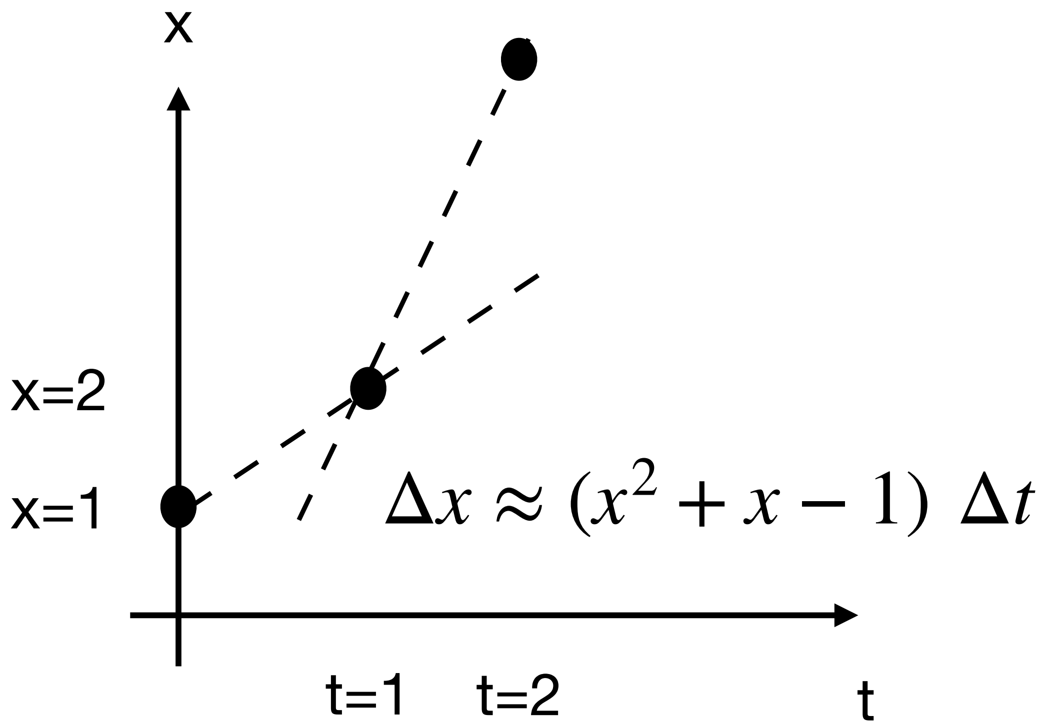
Non linear

Not autonomous

Euler's method

$$\frac{dx}{dt} = x^2 + x - 1$$

$$x(t=0) = 1$$



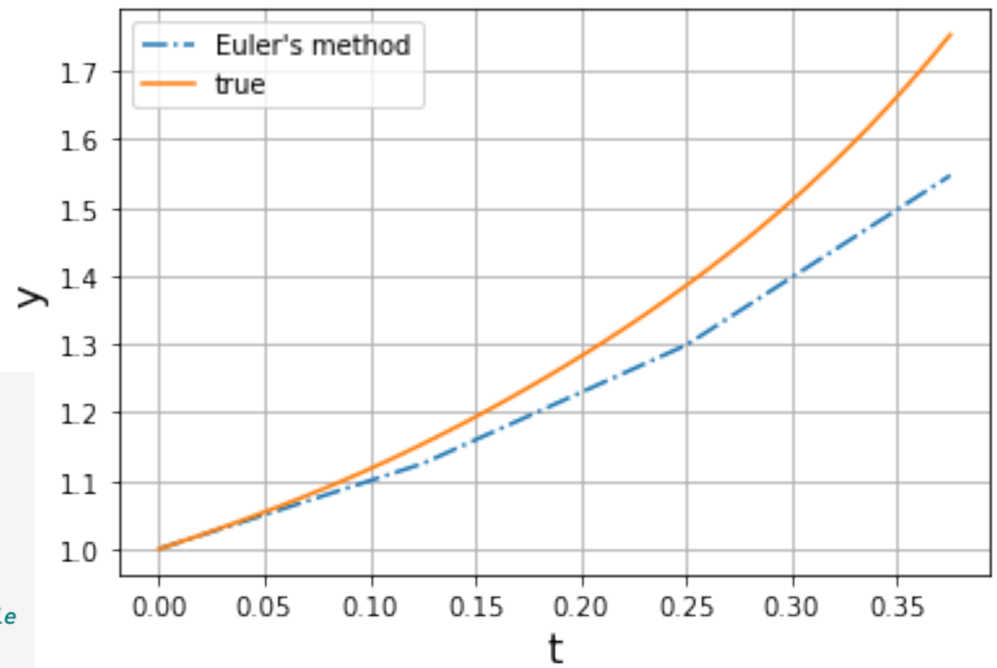
Euler’s method

```python
# Note that the function has to take t as the first argument and y as the second
def calculate_dydt(t, y):
    """Returns the gradient dy/dt for the given function"""
    dydt = y*y + y - 1
    return dydt

max_time = 0.5
N_time_steps = 4
delta_t = max_time / N_time_steps
t_solution = np.linspace(0.0, max_time, N_time_steps+1) # values of independent variable
y0 = np.array([1.0]) # an initial condition, y(0) = y0

# Euler's method
# increase the number of steps to see how the solution changes
y_solution = np.zeros_like(t_solution)
y_solution[0] = y0
for itime, time in enumerate(t_solution) :
    if itime > 0 :
        dydt = calculate_dydt(time, y_solution[itime-1])
        y_solution[itime] = y_solution[itime-1] + dydt * delta_t

plt.plot(t_solution, y_solution, '-.',label="Euler's method")
```



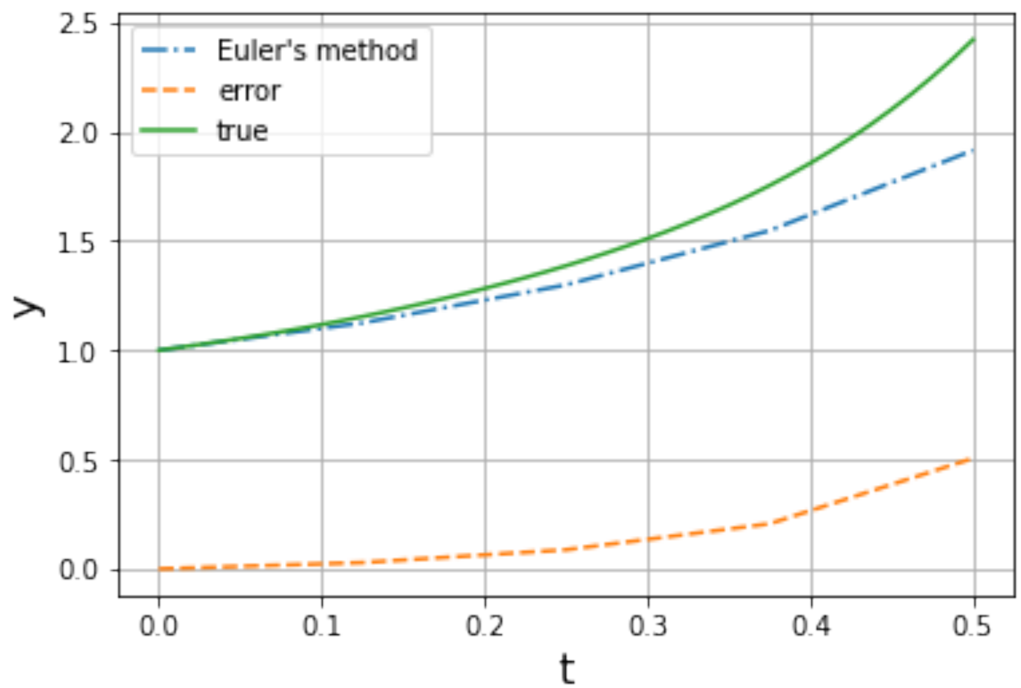
The global error is related to the step size Δt (often also denoted h), so can reduce it, or use a better method to estimate the gradient (more today!)

Plan for today

1. ~~Revision of last week - Classes, ODEs and integration~~
2. Convergence - how do you know it has worked?
3. The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)
4. How to make a higher order ODE into a first order one
5. Tutorial this week - classes, multistep methods and second order ODEs

Convergence

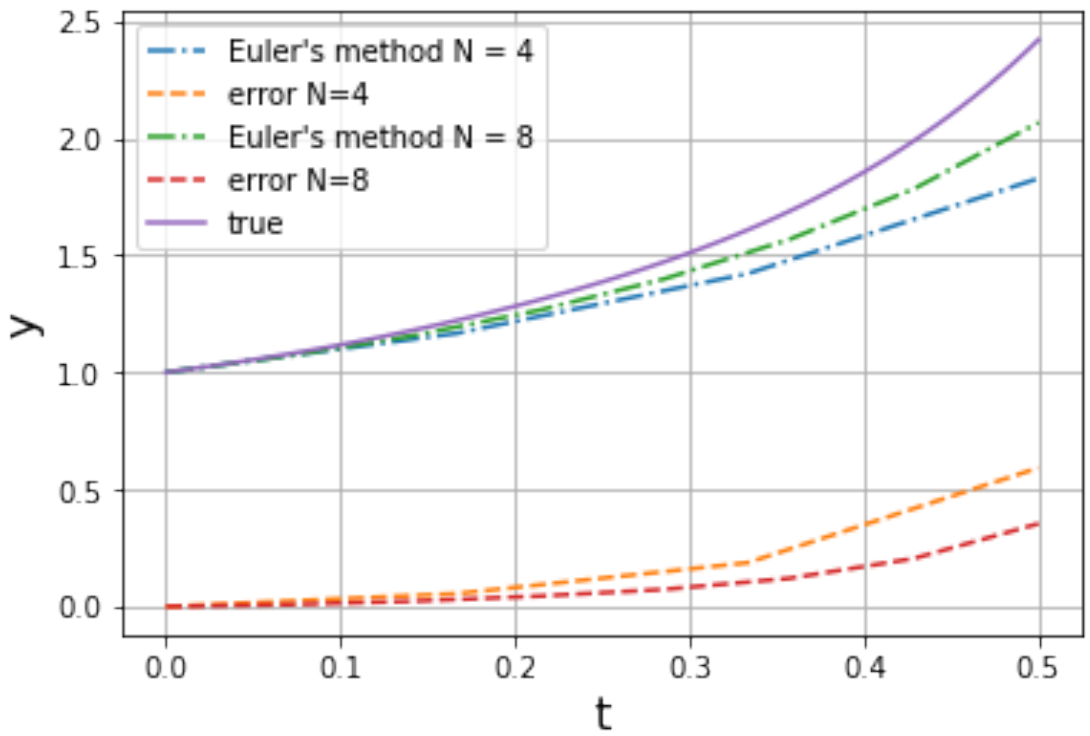
Usually I won't know the solution exactly, so how do I know what I get is right? Should I just trust the solver?



Convergence

Since the method is first order, decreasing the step size by 2 SHOULD decrease the error by 2.

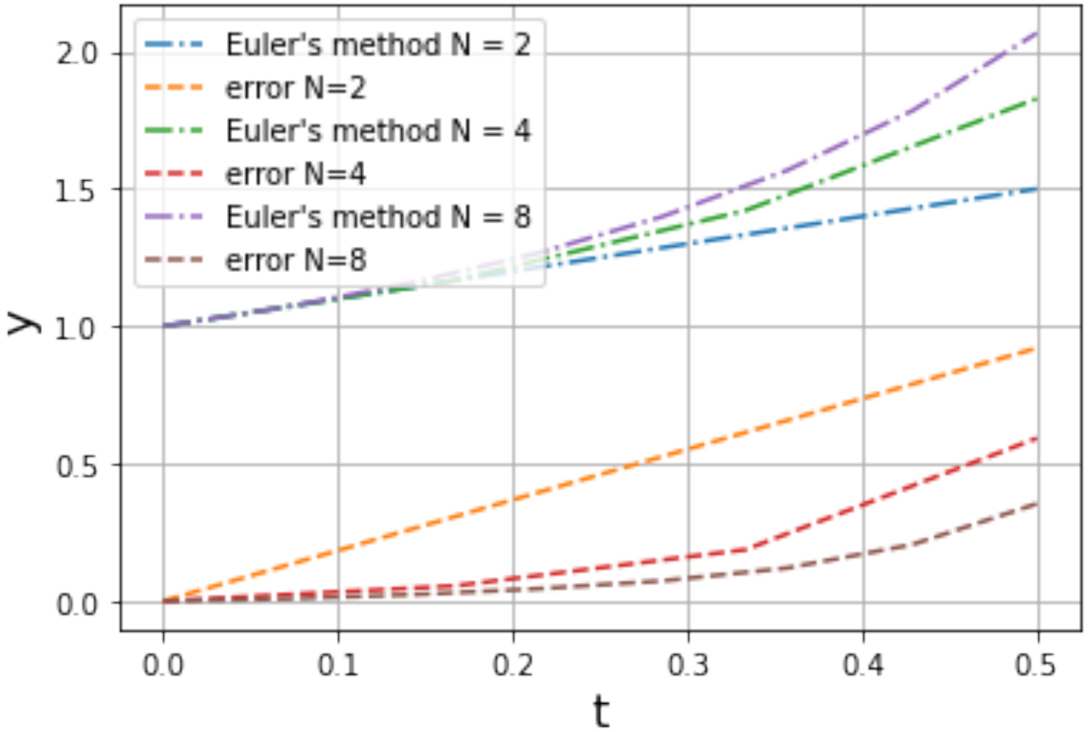
If we can show this, we are **"in the convergence regime"**



Convergence

Where we don't know the solution, we need
3 RESOLUTIONS
to test convergence - if we double the resolution, we know that the differences should scale as

$$\frac{y_{N=8} - y_{N=4}}{y_{N=4} - y_{N=2}} = 1/2$$



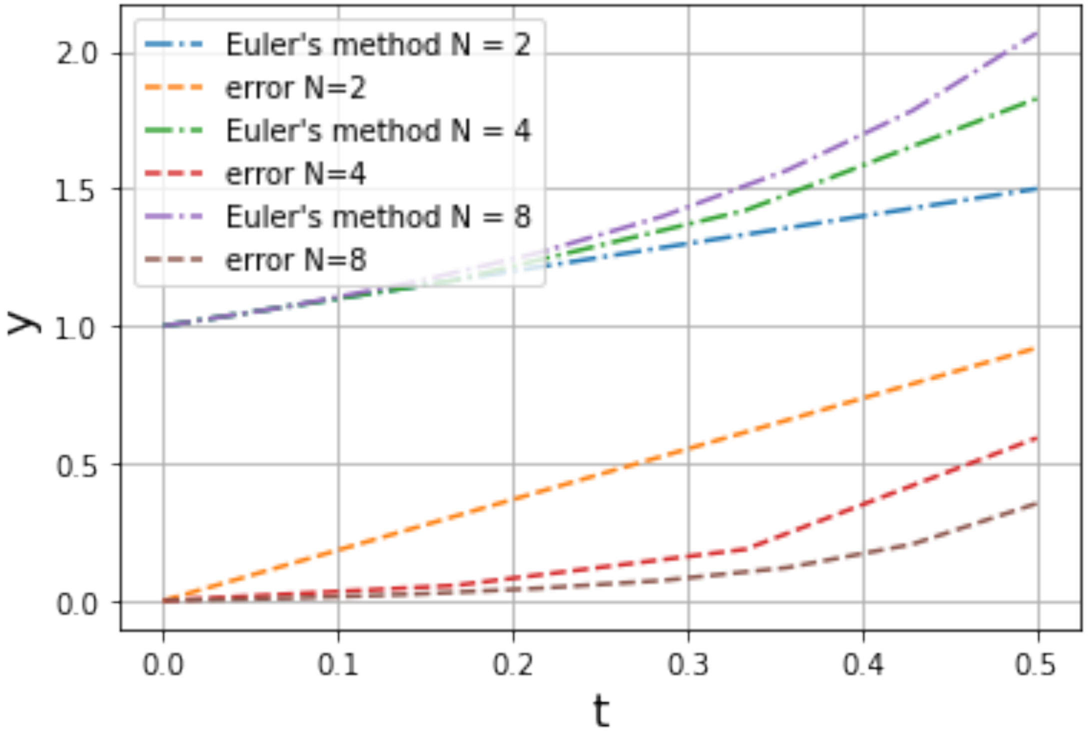
Convergence

Because

$$\frac{(y_{N=8} - y_{true}) - (y_{N=4} - y_{true})}{(y_{N=4} - y_{true}) - (y_{N=2} - y_{true})}$$

$$= \frac{(y_{N=4} - y_{true})/2 - (y_{N=4} - y_{true})}{(y_{N=4} - y_{true}) - 2(y_{N=4} - y_{true})}$$

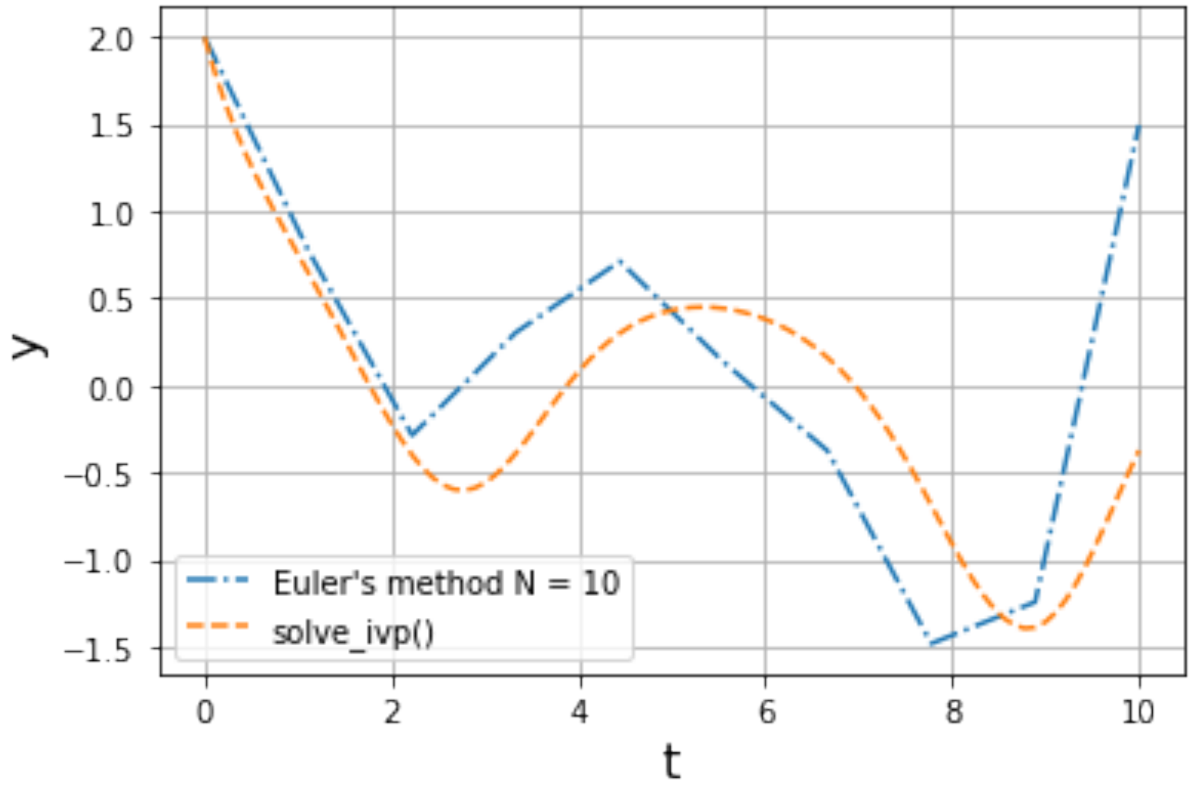
$$= \frac{1}{2}$$



Convergence

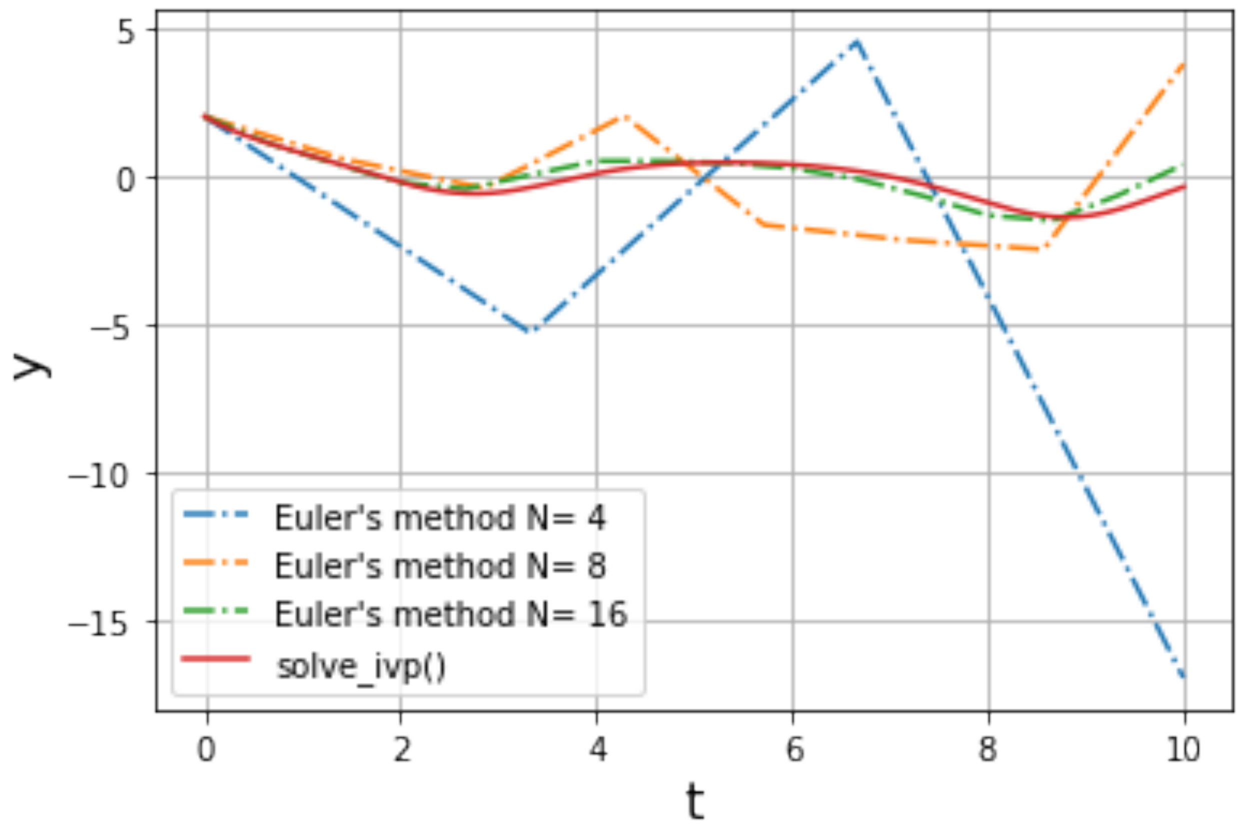
What happens "outside the convergence regime"?

Especially problematic for oscillatory functions, where we need to resolve each wavelength in the solution



Convergence

As a minimum, need to check that increasing resolution does not dramatically change the solution



Plan for today

1. ~~Revision of last week - Classes, ODEs and integration~~
2. ~~Convergence - how do you know it has worked?~~
3. The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)
4. How to make a higher order ODE into a first order one
5. Tutorial this week - classes, multistep methods and second order ODEs

The trouble with Euler's method 1 - convergence

I asserted that the Euler method was 1st order accurate, so error was proportional to the step size h - how did I know this?

First, it comes from the truncated Taylor series expansion of the function

$$y(t_{k+1}) = y(t_k + h) = y(t_k) + h\frac{dy}{dt}\bigg|_{t_k} + O(h^2)$$

Define the error as the value of the function relative to the true value $\bar{y}(t)$

$$\epsilon(t_k) = y(t_k) - \bar{y}(t_k)$$

Can show that $\epsilon(t_{k+1}) = \epsilon(t_k) + O(h^2)$

(*will do on the board, and note provided in QMPlus, but derivation not examinable)*

The trouble with Euler's method 1 - convergence

Can show that $\epsilon(t_{k+1}) = \epsilon(t_k) + O(h^2)$ so **local** truncation error is order h^2

But then the number of steps taken in total is inversely proportional to h

$$N = \frac{t_f - t_i}{h}$$

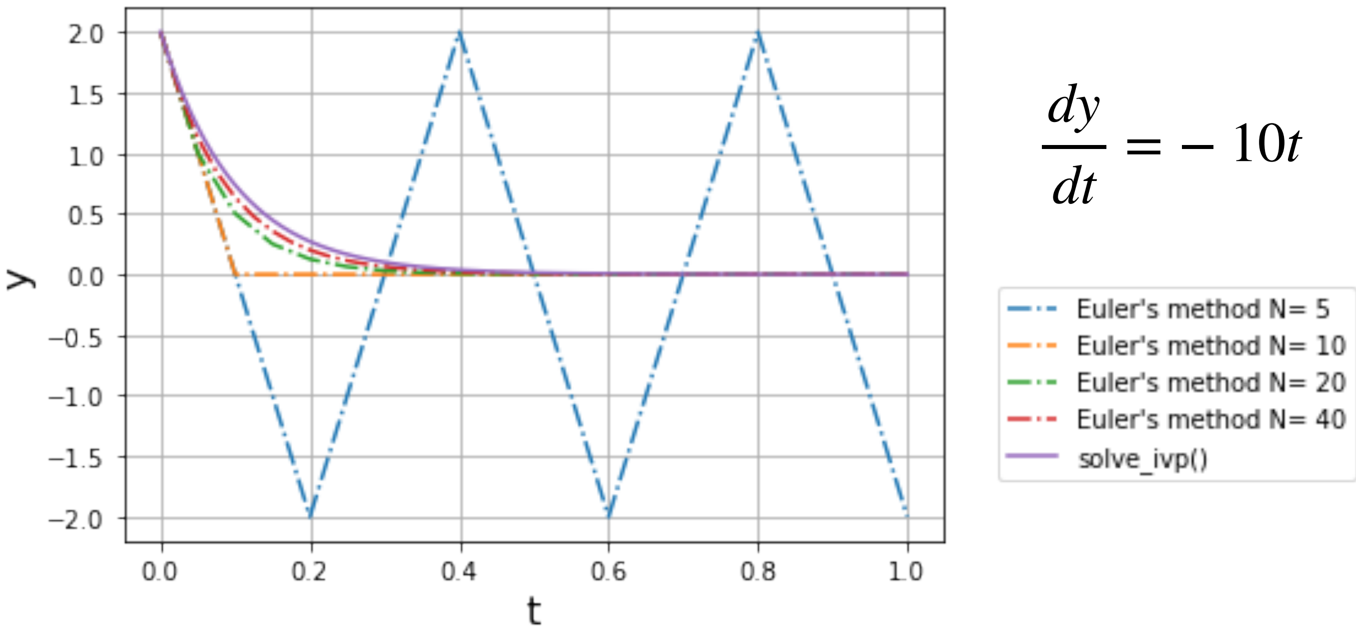
So overall the **global truncation error** is $N \times O(h^2) = O(h)$

We call this a **first order method**.

This means that doubling the number of steps only halves the error, which is not great - we find very **slow convergence** as we increase resolution

The trouble with Euler’s method 2 - stability

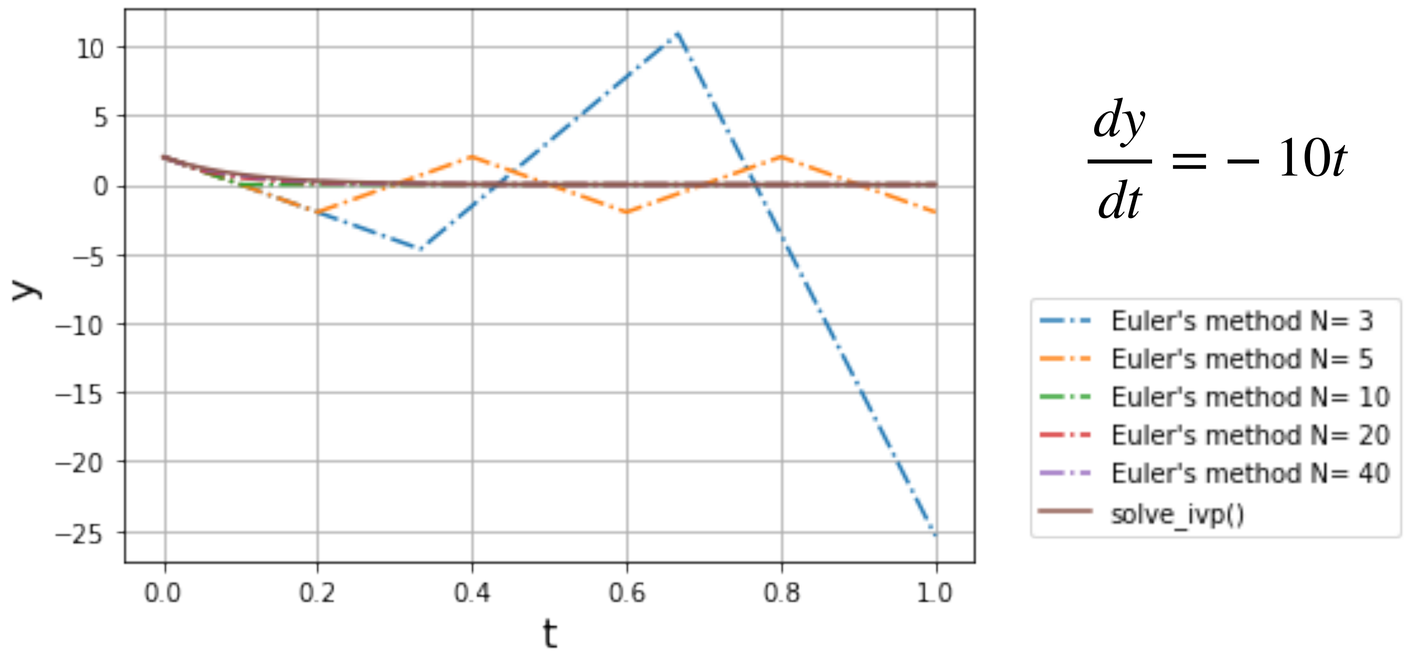
A worse problem is the stability of Euler’s method.



$$\frac{dy}{dt} = -10t$$

The trouble with Euler's method 2 - stability

At low resolutions the error is oscillating and growing exponentially - this is not bad convergence, this is **numerical instability**



$$\frac{dy}{dt} = -10t$$

The trouble with Euler's method 2 - stability

At low resolutions the error is oscillating and growing exponentially - this is not bad convergence, this is **numerical instability:**

A spurious feature in a numerical solution, not present in the exact solution, that grows with time and dominates over the real, physical solution.

We derive it by considering perturbing the solution by a small amount (maybe due to numerical round off errors), so that:

$$y_k = y_k + \delta_k$$

Can show that:

$$\delta_{k+1} = \left(1 + h\frac{\partial f}{\partial y}\right)\delta_k \quad \textit{where} \quad f = \frac{dy}{dt} \qquad \textit{(e.g.} \ \frac{dy}{dt} = -10y\textit{)}$$

(*will do on the board, and note provided in QMPlus, but derivation not examinable)*

The trouble with Euler's method 2 - stability

Can show that:

$$\delta_{k+1} = \left(1 + h\frac{\partial f}{\partial y}\right)\delta_k \qquad \textit{where} \quad f = \frac{dy}{dt} \qquad\qquad \textit{(e.g. } \frac{dy}{dt} = -10y\textit{)}$$

(*will do on the board, and note provided in QMPlus, but derivation not examinable*)

This will grow exponentially when:

$$\left|1 + h\frac{\partial f}{\partial y}\right| > 1 \qquad \Longrightarrow \qquad \frac{\partial f}{\partial y} > 0 \quad \textit{or} \quad \left|\frac{\partial f}{\partial y}\right| > \frac{2}{h}$$

FIX: Using intermediate estimates - the midpoint method

Can achieve stability by using *intermediate estimates* in calculating the full time step.

e.g. the midpoint method is **stable and has second order global error**

$$y_{k+1/2} = y_k + \frac{1}{2}\, h\, f(y_k, t_k) \qquad \textit{where} \qquad f = \frac{dy}{dt}$$

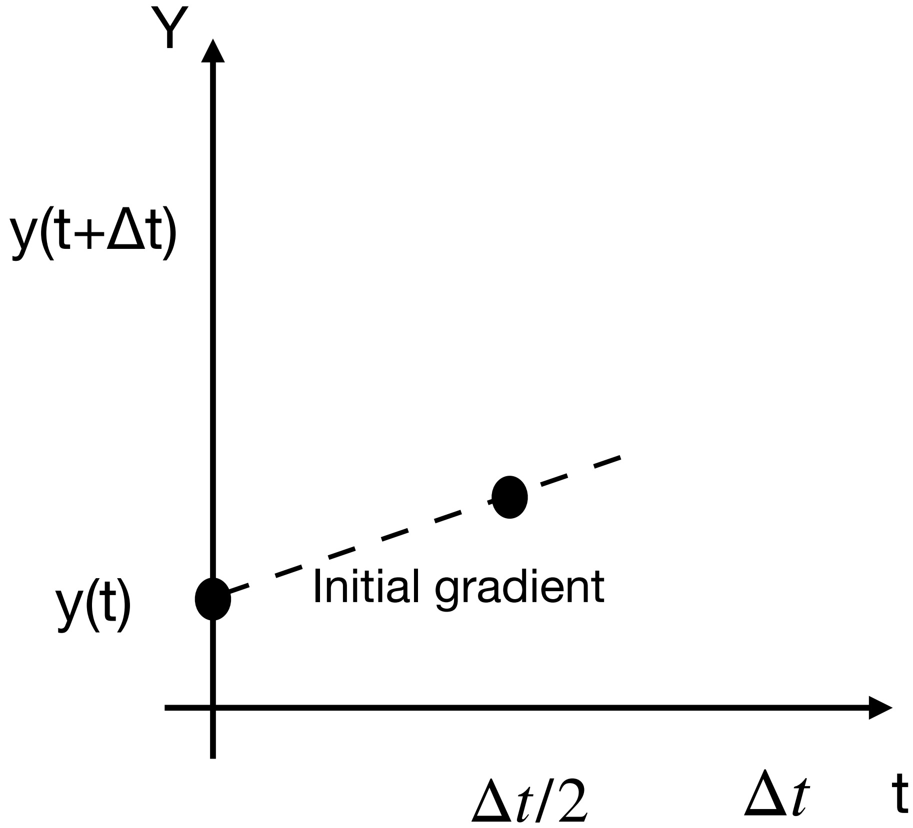
$$y_{k+1} = y_k + h\, f(y_{k+1/2}, t_{k+1/2})$$

Always use this in preference to Euler!

Midpoint method

$$\frac{dy}{dt} = y^2 + y - 1$$

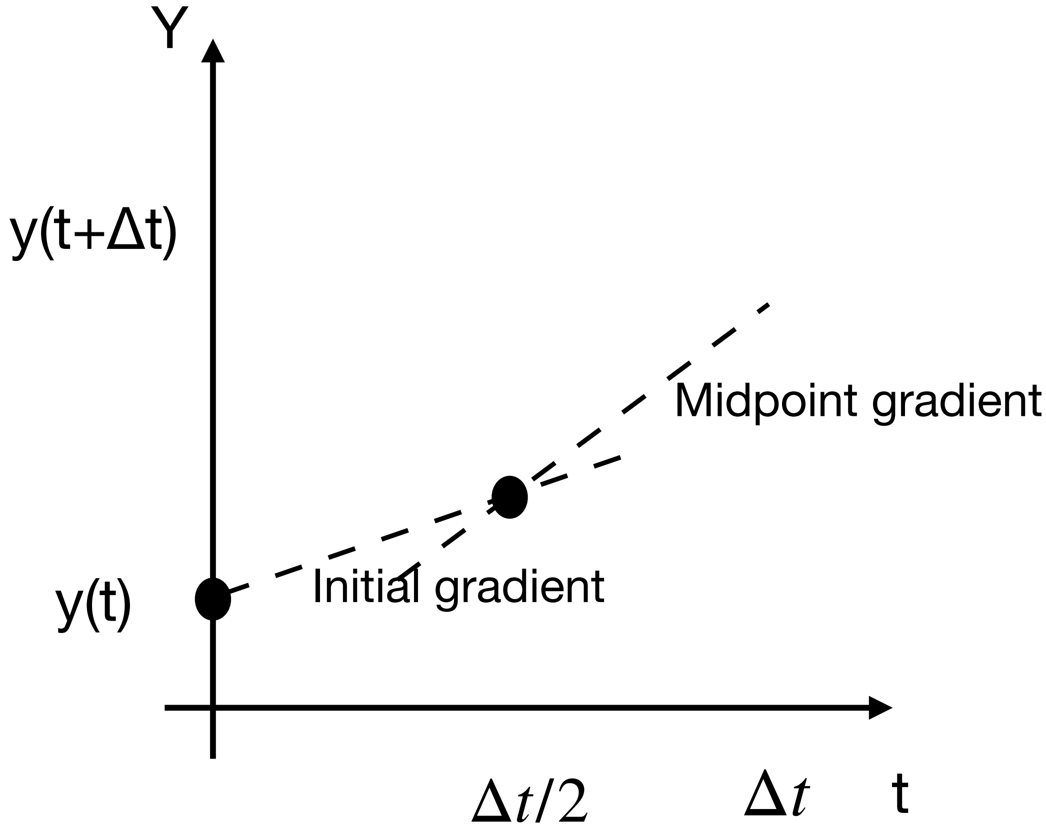
$$y(t = 0) = 1$$



Midpoint method

$$\frac{dy}{dt} = y^2 + y - 1$$

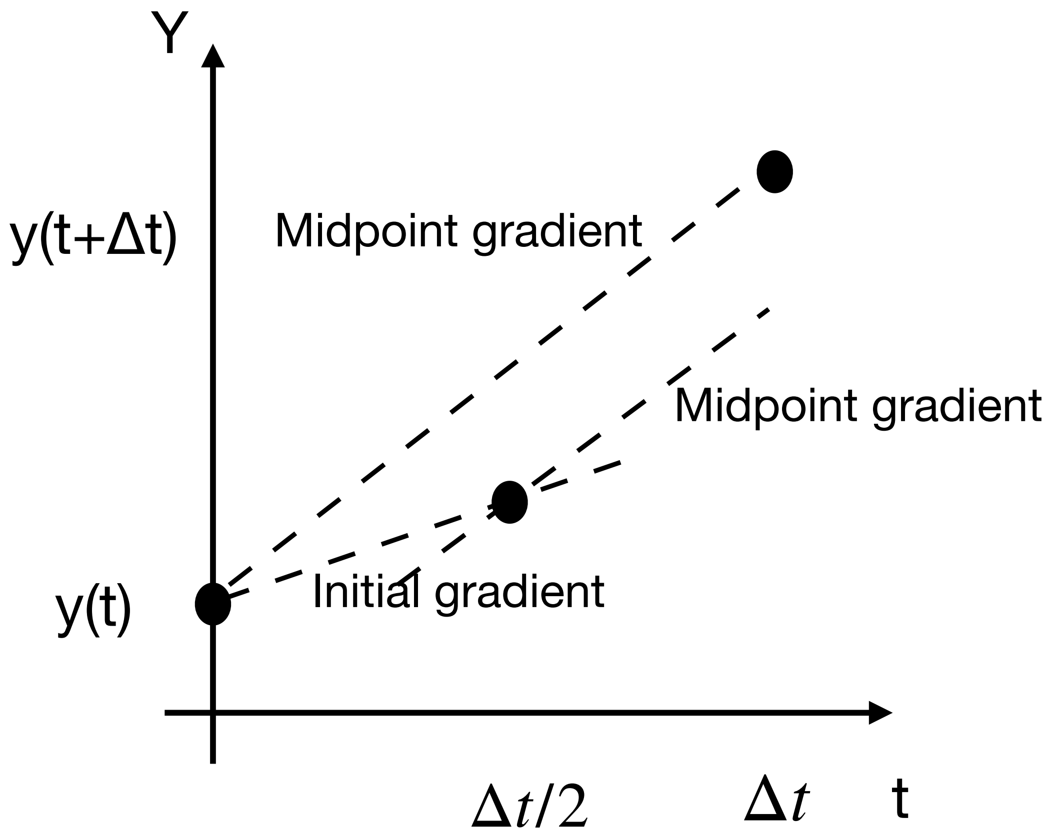
$$y(t = 0) = 1$$



Midpoint method

$$\frac{dy}{dt} = y^2 + y - 1$$

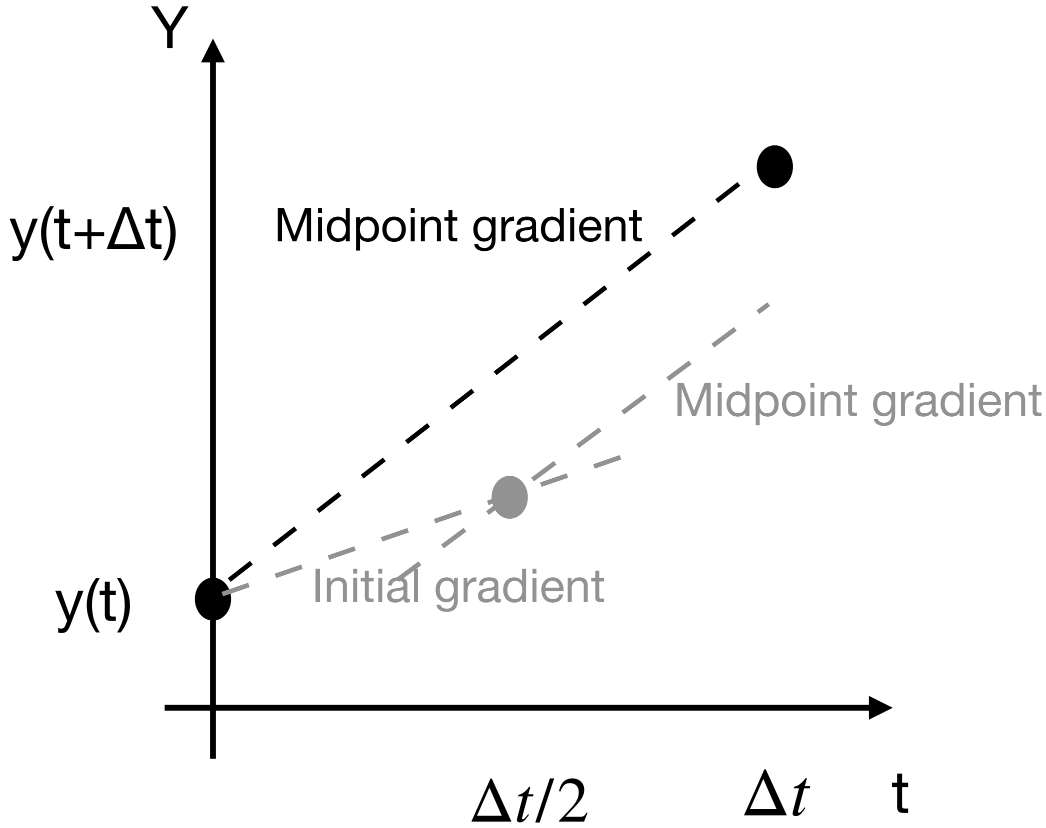
$$y(t = 0) = 1$$



Midpoint method

$$\frac{dy}{dt} = y^2 + y - 1$$

$$y(t = 0) = 1$$



Runge-Kutta methods

Can achieve stability by using *intermediate estimates* in calculating the full time step. How about using even more intermediate points?

The most common method is the 4th order method, often referred to as "RK4"

Explicit Runge–Kutta methods [edit]

The family of explicit Runge–Kutta methods is a generalization of the RK4 method mentioned above. It is given by

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i,$$

where[6]

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

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

Examples [edit]

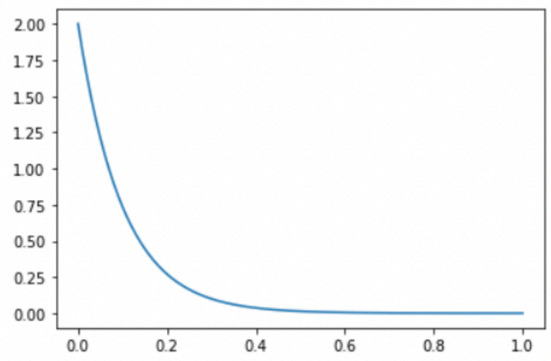
The RK4 method falls in this framework. Its tableau is[13]

$$\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}$$

Scipy's solve_ivp() uses RK45 by default

- This does not mean it is 45th order accurate!!
- The method takes a 4th order RK4 step AND a 5th order RK4 step and uses the difference to estimate the step error. If it is over some threshold ***rtol*** it will reduce the step size it takes.
- For solutions where you need greater accuracy (e.g., many oscillations, or orbits HINT HINT) you may need to reduce rtol.

```
solution = solve_ivp(calculate_dydt, [0,max_time], y0, t_eval=t_solution, rtol=1e-10)
plt.plot(solution.t, solution.y[0], '-', label="solve_ivp()");
```

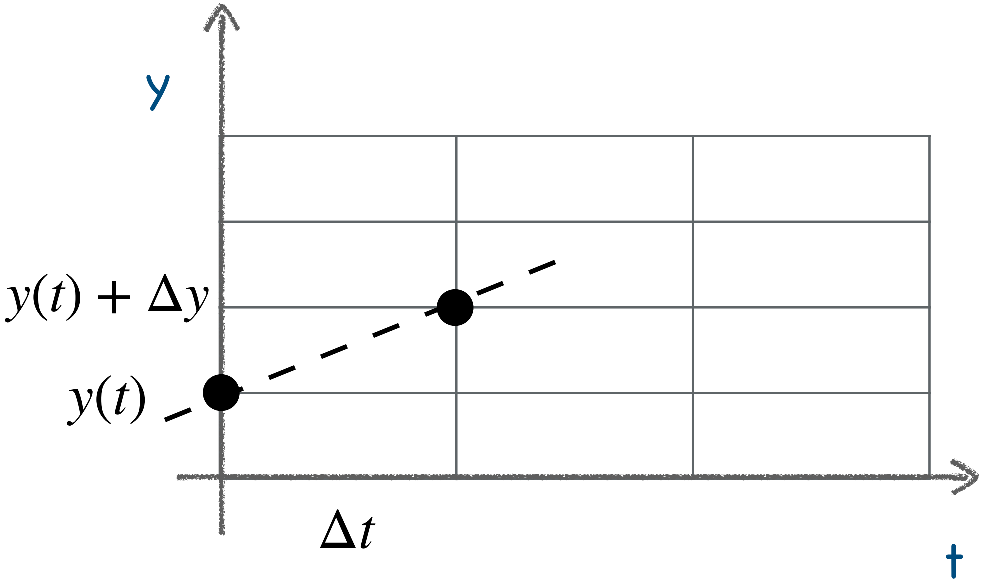


Plan for today

1. ~~Revision of last week - Classes, ODEs and integration~~
2. ~~Convergence - how do you know it has worked?~~
3. ~~The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)~~
4. How to make a higher order ODE into a first order one
5. Tutorial this week - classes, multistep methods and second order ODEs

How do I integrate second order derivatives numerically?

$$\frac{d^2y}{dt^2} - \frac{dy}{dt} + f(y,t) = 0$$

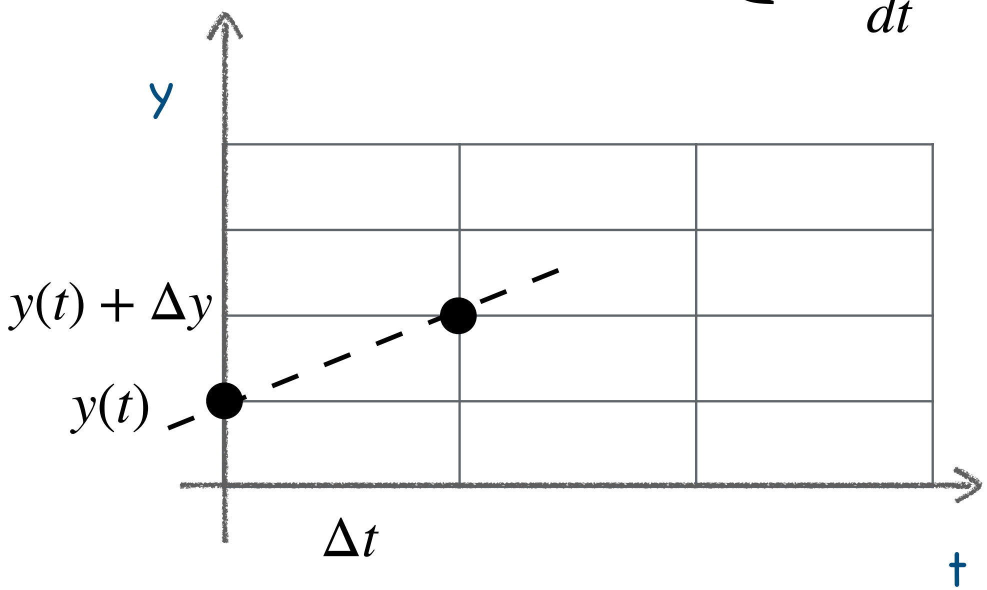


How do I integrate second order derivatives numerically?

$$\frac{d^2y}{dt^2} - \frac{dy}{dt} + f(y,t) = 0$$

$$\begin{cases} \dfrac{dv}{dt} - v + f(y,t) = 0 \\[2ex] \dfrac{dy}{dt} = v \end{cases}$$

1. Decompose the second order equation into two first order ones

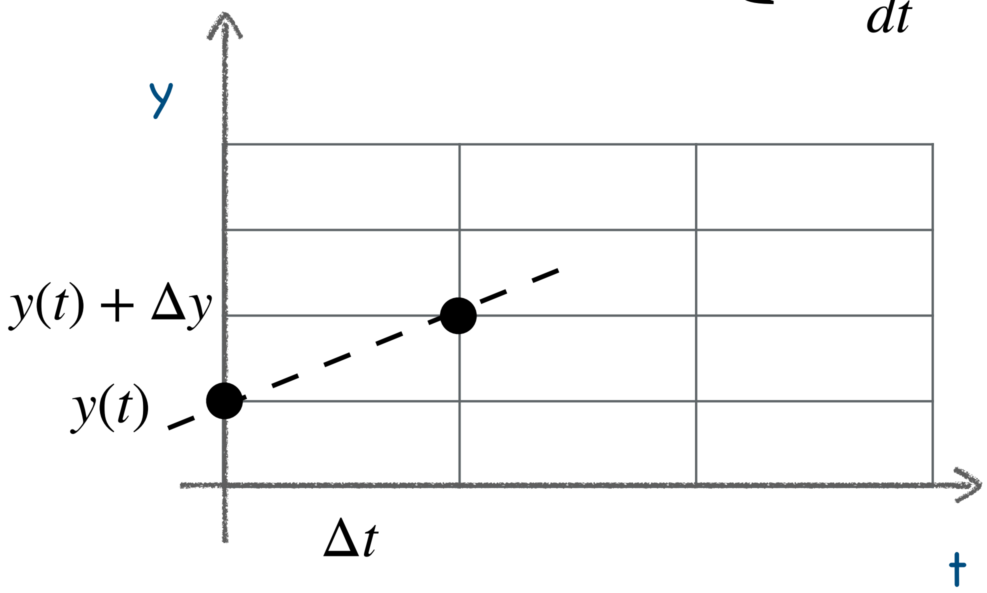


How do I integrate second order derivatives numerically?

$$\frac{d^2y}{dt^2} - \frac{dy}{dt} + f(y,t) = 0$$

$$\begin{cases} \dfrac{dv}{dt} - v + f(y,t) = 0 \\[2ex] \dfrac{dy}{dt} = v \end{cases}$$

1. Decompose the second order equation into two first order ones



$$\Delta v = \Delta t \left(v - f(y,t) \right)$$

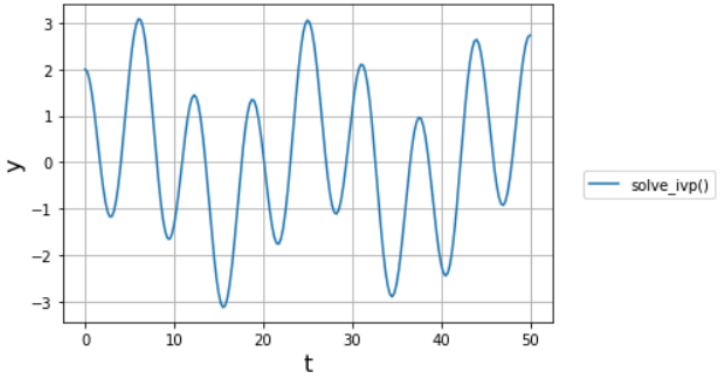
$$\Delta y = v \ \Delta t$$

2. Solve as a dimension 2 first order system

Example: the forced harmonic oscillator

```
# Note that the function has to take t as the first argument and y as the second
def calculate_dydt(t, y):
    """Returns the gradient dy/dt for the forced harmonic oscillator"""
    dydt = np.zeros_like(y)
    dydt[1] = -y[0] + np.sin(0.3*t)
    dydt[0] = y[1]
    return dydt

# Double res
max_time = 50.0
N_time_steps = 200
y0 = np.array([2.0, 0.0])
t_solution = np.linspace(0.0, max_time, N_time_steps+1)
solution = solve_ivp(calculate_dydt, [0,max_time], y0, t_eval=t_solution)
plt.plot(solution.t, solution.y[0], '-', label="solve_ivp()")
plt.grid()
plt.xlabel("t", fontsize=16)
plt.ylabel("y", fontsize=16)
plt.legend(bbox_to_anchor=(1.05, 0.5));
```

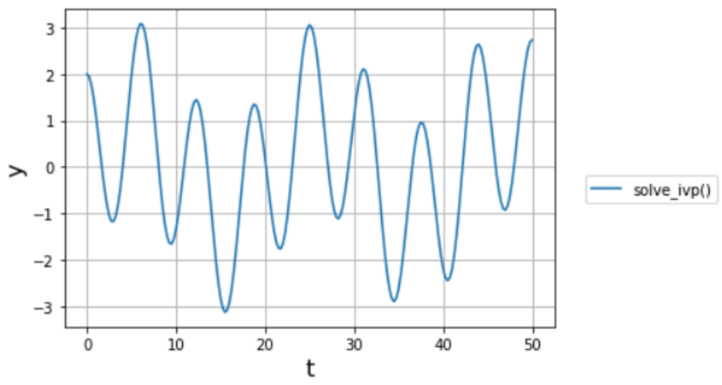


$$\frac{d^2y}{dt^2} + y = \sin(\omega_f t)$$

Example: the forced harmonic oscillator

```
# Note that the function has to take t as the first argument and y as the second
def calculate_dydt(t, y):
    """Returns the gradient dy/dt for the forced harmonic oscillator"""
    dydt = np.zeros_like(y)
    dydt[1] = -y[0] + np.sin(0.3*t)
    dydt[0] = y[1]
    return dydt

# Double res
max_time = 50.0
N_time_steps = 200
y0 = np.array([2.0, 0.0])
t_solution = np.linspace(0.0, max_time, N_time_steps+1)
solution = solve_ivp(calculate_dydt, [0,max_time], y0, t_eval=t_solution)
plt.plot(solution.t, solution.y[0], '-', label="solve_ivp()")
plt.grid()
plt.xlabel("t", fontsize=16)
plt.ylabel("y", fontsize=16)
plt.legend(bbox_to_anchor=(1.05, 0.5));
```



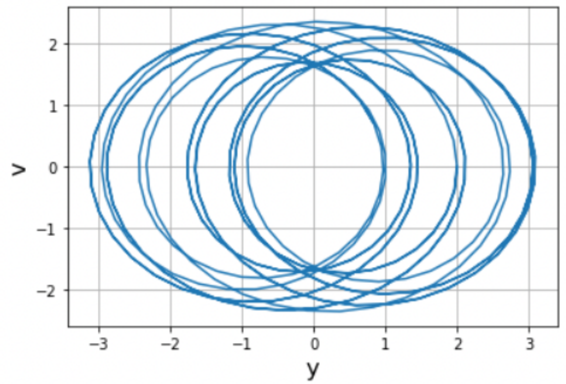
$$\frac{d^2y}{dt^2} + y = \sin(\omega_f t)$$

$$\frac{dv}{dt} = -y + \sin(\omega_f t)$$

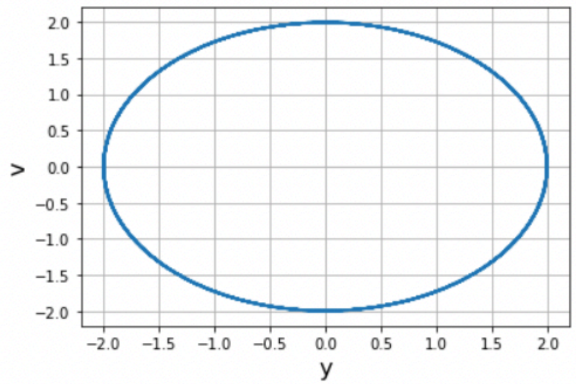
$$\frac{dy}{dt} = v$$

Phase plots for second order systems

Forced harmonic oscillator



Free harmonic oscillator



In a “phase plot” we plot y against v.

This often tells us about the energy in a system, or whether some quantities are conserved.

It also tells us if there is a stable attractor solution - often all initial conditions will drive the system to the same trajectory in phase space.

Plan for today

1. ~~Revision of last week - Classes, ODEs and integration~~
2. ~~Convergence - how do you know it has worked?~~
3. ~~The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)~~
4. ~~How to make a higher order ODE into a first order one~~
5. Tutorial this week - classes, multistep methods and second order ODEs

Tutorial week 4

```python
# ExplicitIntegrator class

class ExplicitIntegrator :

    """
    Contains explicit methods to integrate ODEs

    attributes: the function to calculate the gradient dydt, max_time,
                N_time_steps, method

    methods: calculate_solution, plot_solution

    """
    integration_methods = ["Euler", "MidPoint", "RK4"]

    # constructor function
    def __init__(self, dydt, max_time=0, N_time_steps=0, method = "Euler"):

        self.dydt = dydt # Note that we are passing in a function, this is ok in python
        self.method = method
        assert self.method in self.integration_methods, 'chosen integration method not imp

        # Make these private - restrict getting and setting as below
        self._max_time = max_time
        self._N_time_steps = N_time_steps

        # Derived from the values above
        self._delta_t = self.max_time / self.N_time_steps
        self._t_solution = np.linspace(0.0, max_time, N_time_steps+1)
        self._y_solution = np.zeros_like(self._t_solution)
```

Implement the midpoint method in an ExplicitIntegrator class - more practise with classes

Tutorial week 4

ACTIVITY 3:

Write a class that contains information about the Van der Pol oscillator with a source, and solves the second order ODE related to its motion using scipy's solve_IVP method:

$$\frac{d^2 y}{dt^2} - 2a(1 - y^2)\frac{dy}{dt} + y = f(t)$$

where a is a damping factor. Your class should allow you to pass in the source function f(t) as an argument that can be changed.

HINT: It may help to start with the Ecosystem class in the solutions for last week's tutorial and modify this.

What parts or features of the differential equation tell us if it is:

1. Second or first order
2. Autonomous
3. Linear / non linear
4. Dimension 1 or 2?

Write a VanDerPolOscillator class - 2nd order ODE, need to convert to a first order one