Week 4: More on ODEs

- Higher order ODEs, higher order explicit methods

Dr K Clough, Topics in Scientific computing, Autumn term 2023

Suggestions for those who still feel unfamiliar with python

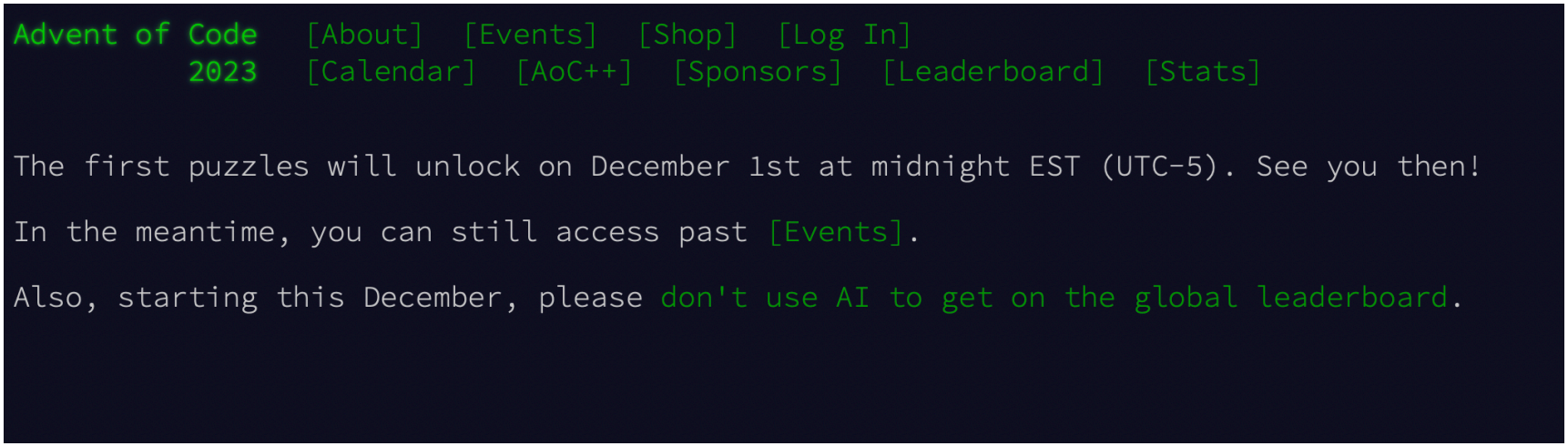
1. Watch the videos that I have recorded of the first two lectures - I have slowed down the introductions here and you can pause and replay them.

2. Check out the (very short and readable) book “Think Python 2” that I recommend on the reading list - it is available open source here: https://greenteapress.com/wp/think-python-2e/ along with example code. (You can skip over Chapters 8-14.)

3. The CodeAcademy course on Python3 can be obtained for free (https://www.codecademy.com/learn/learn-python-3). It says 24 hours but you can probably skim some aspects and do it is less time.

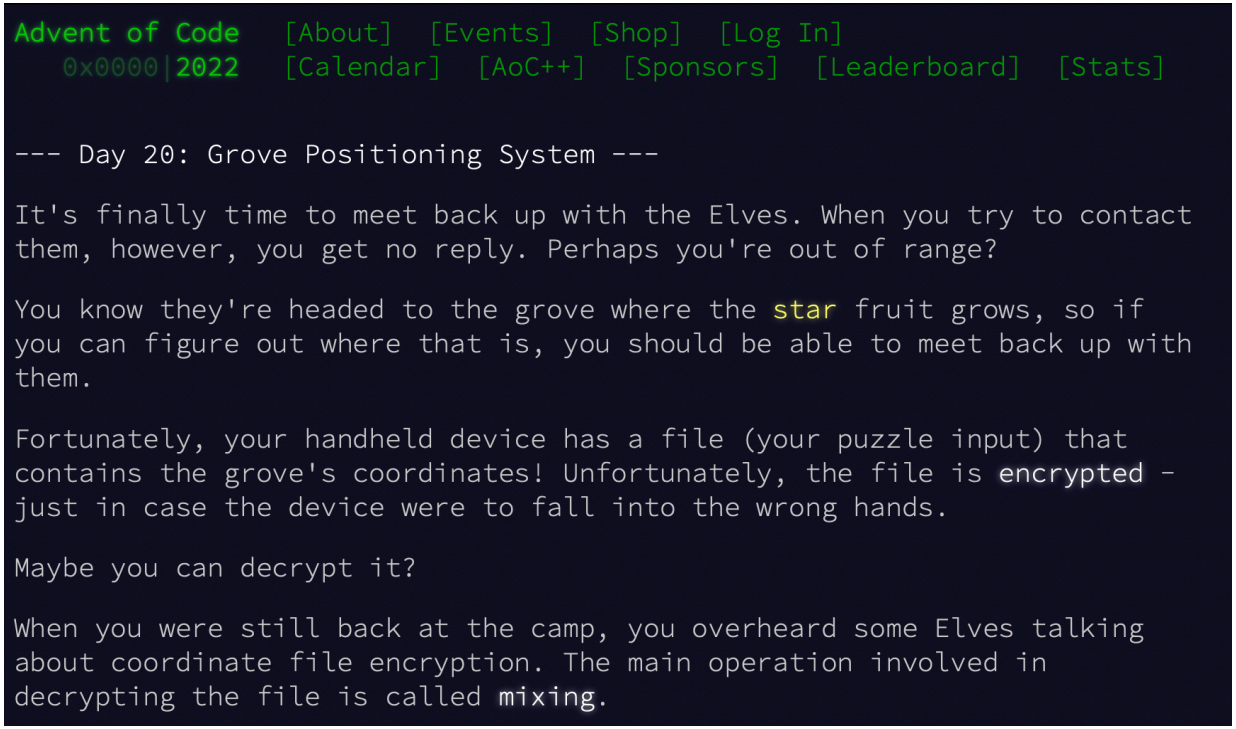
Suggestions for those who want more challenges!

https://adventofcode.com



Suggestions for those who want more challenges!

https://adventofcode.com



Plan for today

1. Revision of last week - Classes, ODEs and integration
2. The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)
3. How to make a higher order ODE into a first order one
4. Coursework - revision of the physical scenario and grading details
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

```
# 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
```

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

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

```
pink
```

Classes

TOP TIP:
A useful strategy for more complicated classes is to keep the init minimal and let the user add attributes later

```
# CatHome class

class CatHome :

    """

    Attributes : some number of cats >=2

    Methods: write the cat colours

    """

    # constructor function
    def __init__(self):
        self.list_of_cats = []
        self.num_cats = 0
        self._is_defined = False
        print("Class requires addition of at least 2 cats to be defined")

    def add_cat(self, new_cat) :

        #Add new cat to the list of cats
        self.list_of_cats.append(new_cat)
        self.num_cats += 1

        if(self.num_cats >= 2) :
            self._is_defined = True
            print("Cat Home definition complete!")
        else :
            print("Need to add another cat to make a cat home")


    def print_cats_colours(self) :
        assert self._is_defined, "Insufficient cats added, Cat Home not defined!"

        for cat in self.list_of_cats :
            cat.print_colour()
```

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

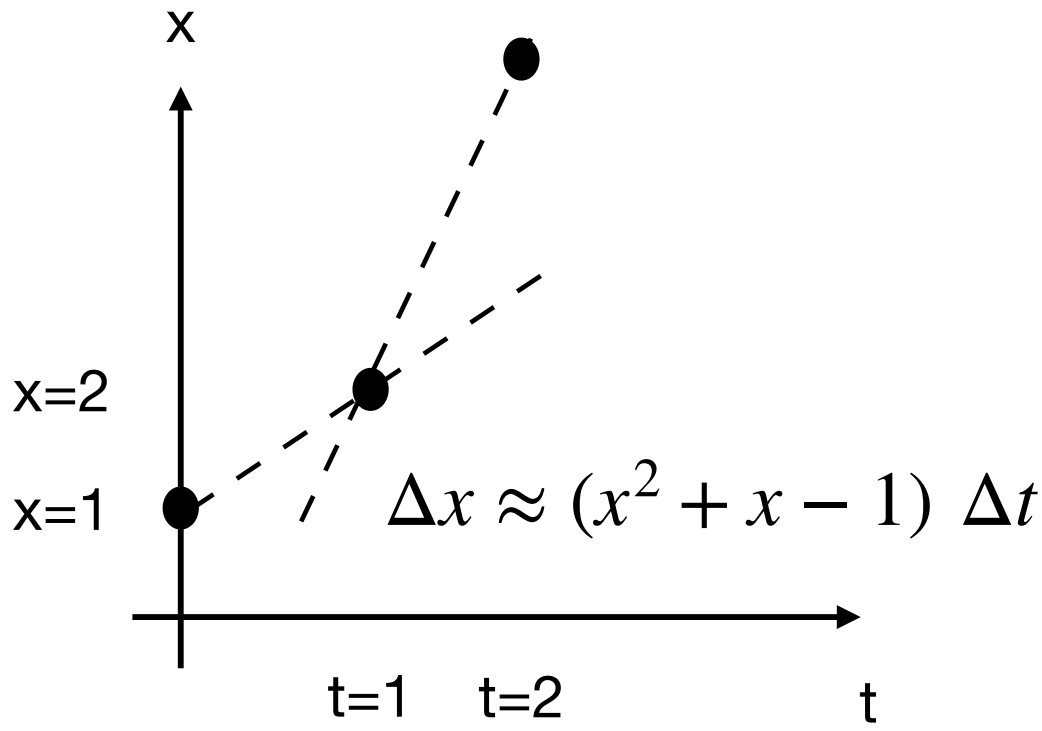
Non linear

Not autonomous

Euler's method

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

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



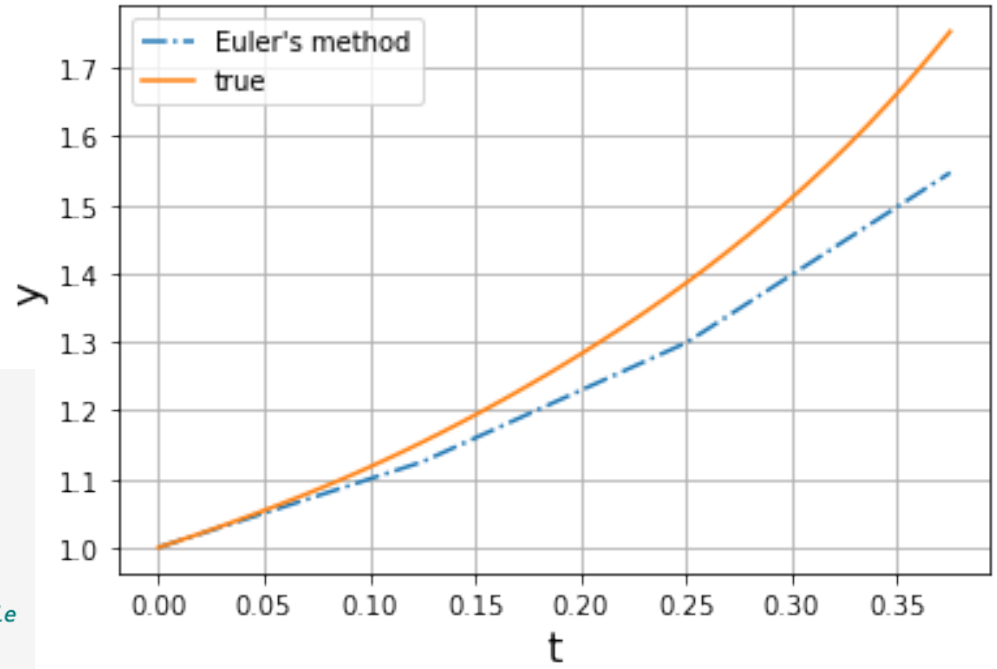
Euler's method

```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")
```



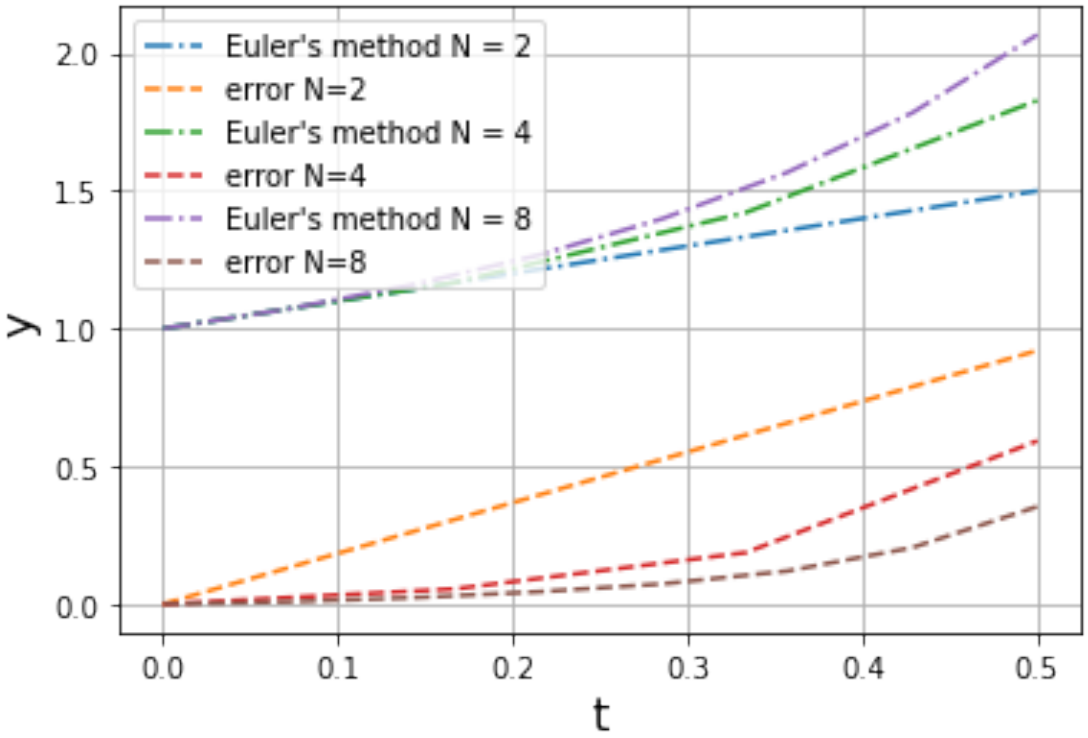
The global error is related to the step size delta_t, so can reduce it, or use a better method to estimate the gradient (more today!)

Convergence of Euler's method (order 1)

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

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

For a k-th order method



Plan for today

1. ~~Revision of last week - Classes, ODEs and integration~~
2. The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)
3. How to make a higher order ODE into a first order one
4. Coursework - revision of the physical scenario and grading details
5. Tutorial this week - classes, multistep methods and second order ODEs

The trouble with Euler's method - 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 (*exercise for tutorial)* that $\epsilon(t_{k+1}) = \epsilon(t_k) + O(h^2)$

The trouble with Euler's method - 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

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

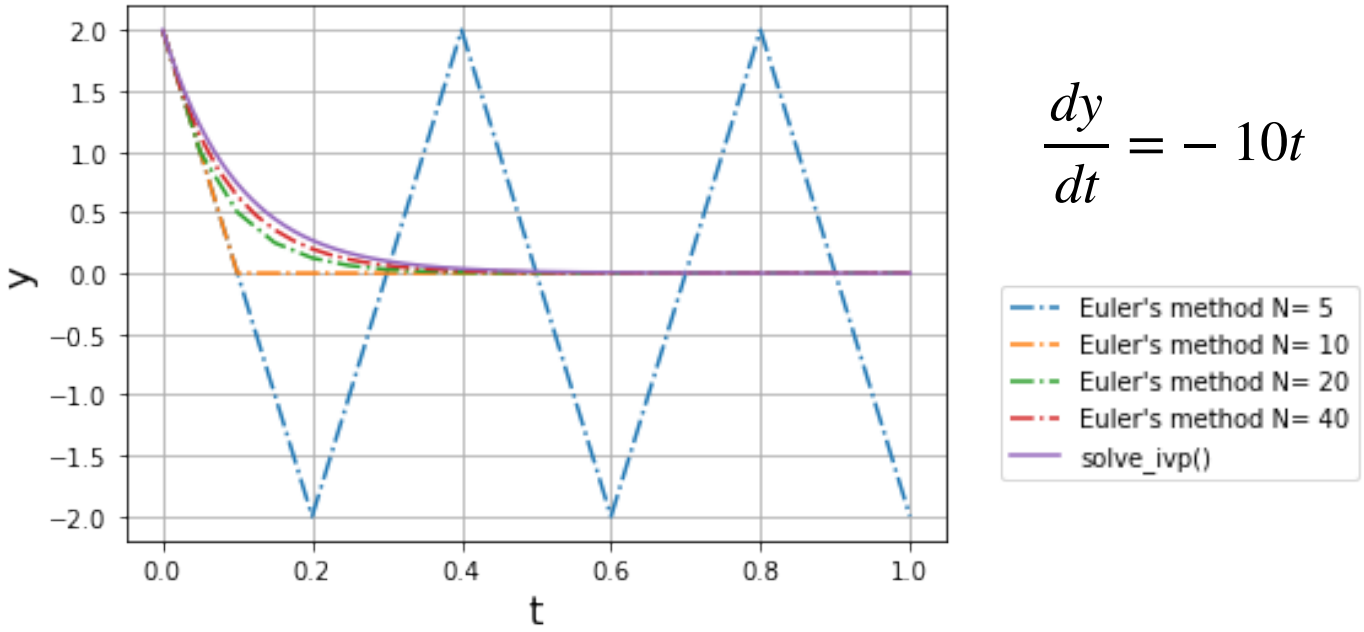
So overall the **global truncation error** is $N \times O(h^2) = O(h)$

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 - stability

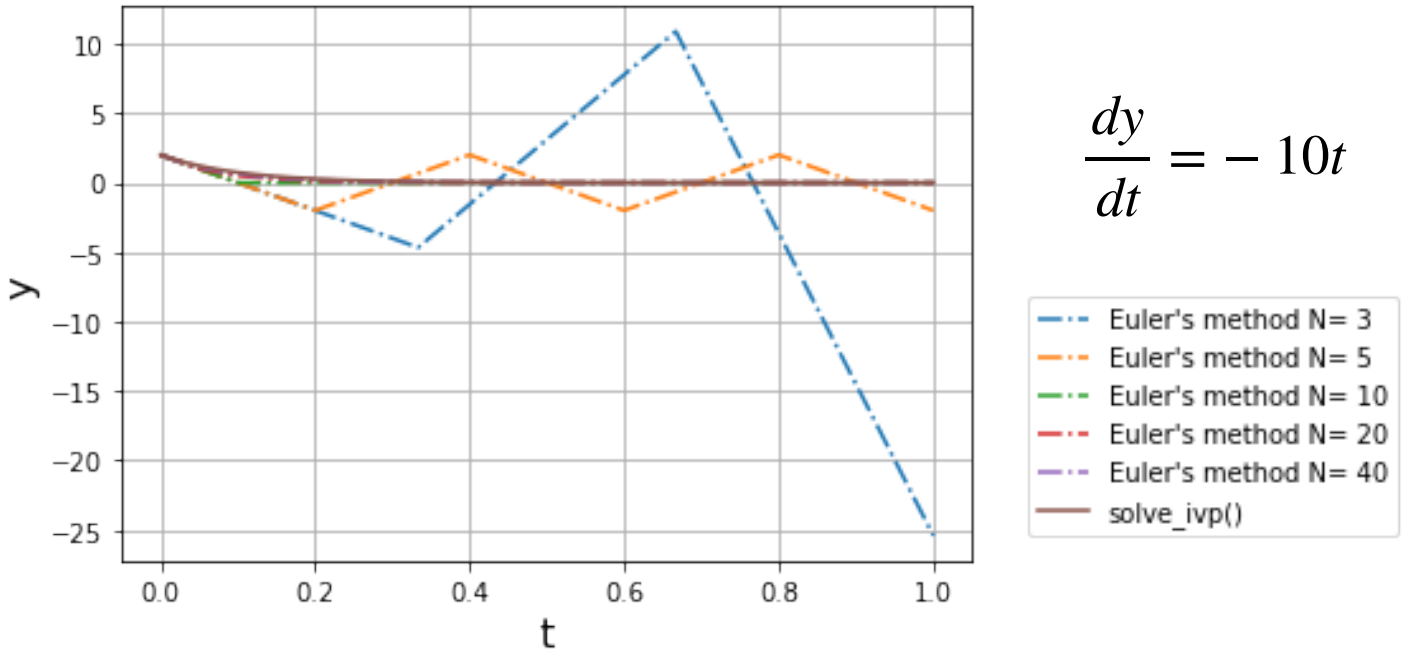
A worse problem is the stability of Euler's method.



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

The trouble with Euler's method - 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 - 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 *(exercise for the tutorial)* that:

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

The trouble with Euler's method - stability

Can show *(exercise for the tutorial)* that:

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

This will grow exponentially when:

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

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**

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

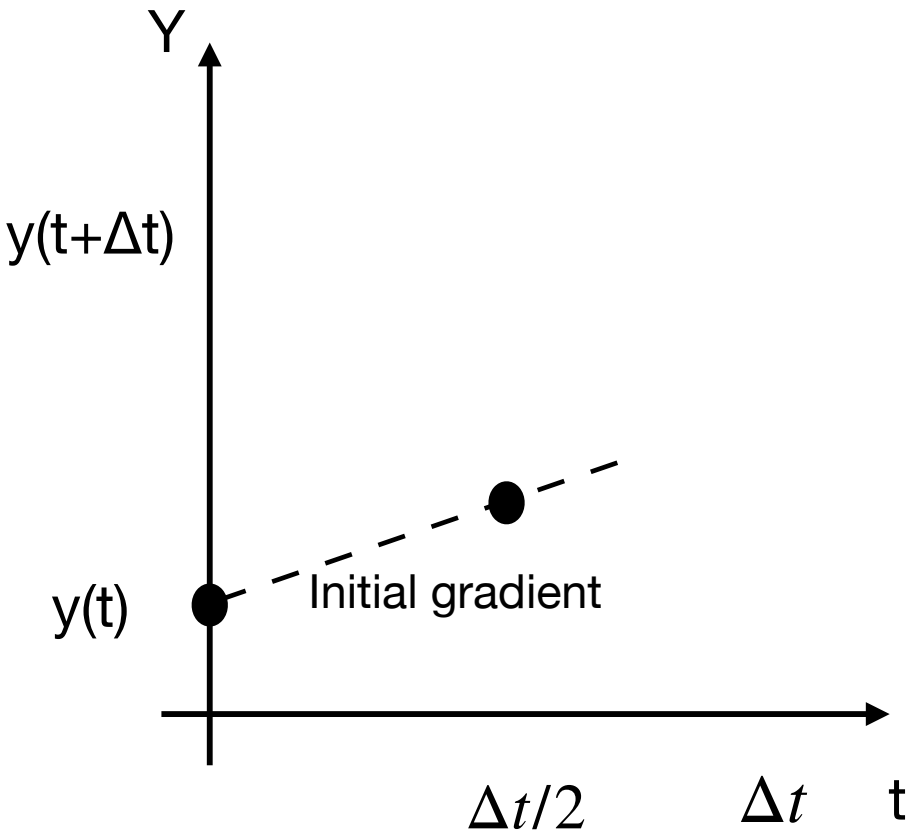
$$y_{k+1} = y_k + \frac{1}{2}\, h\, f(y_{k+1/2}, t_{k+1/2})$$

Always use this in preference to Euler!

Midpoint method

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

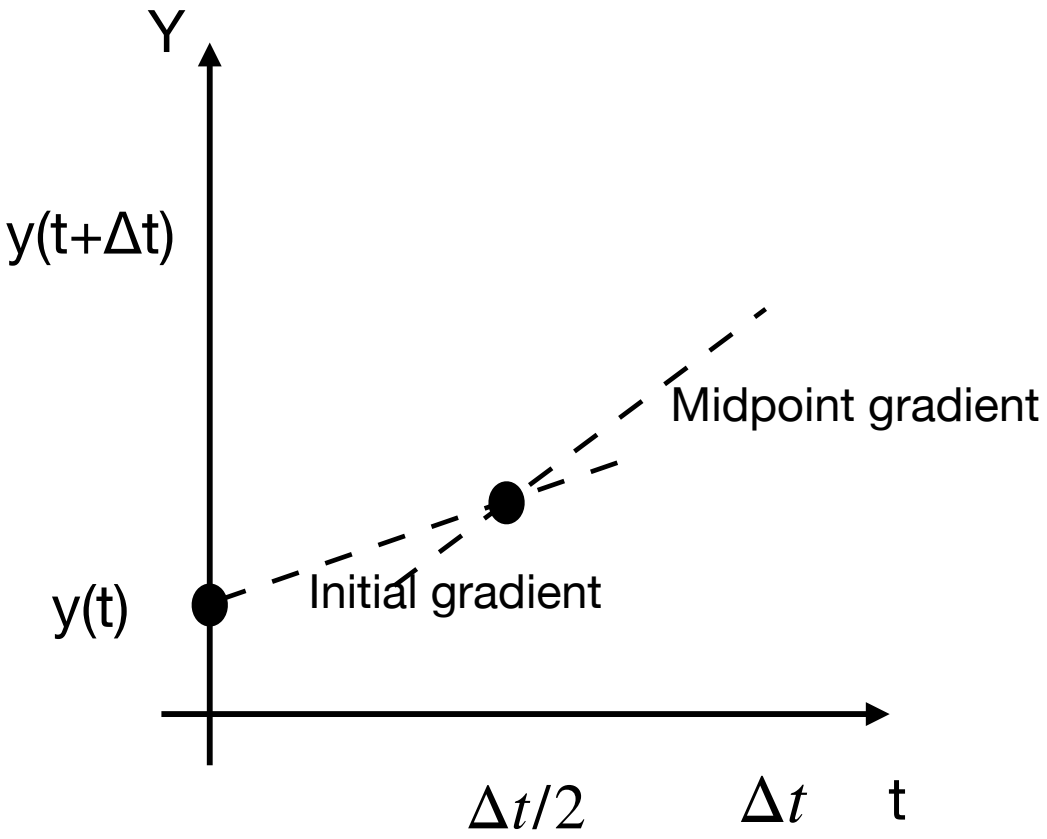
$$y(t = 0) = 1$$



Midpoint method

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

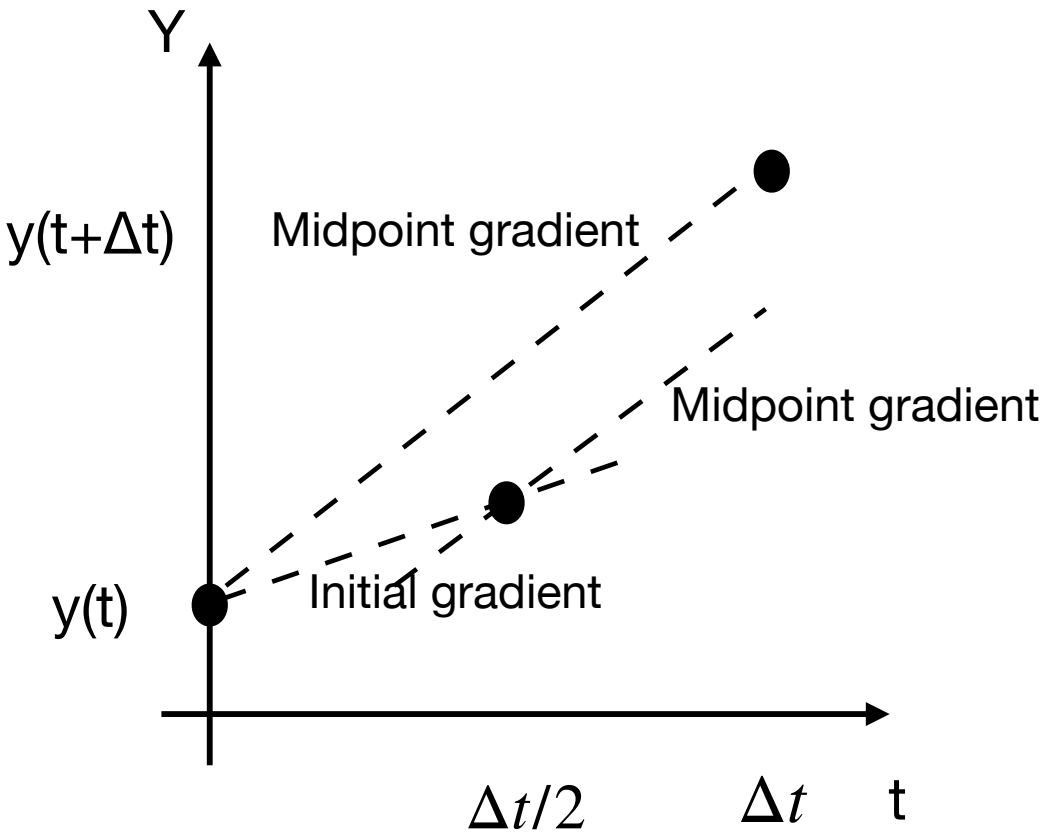
$$y(t = 0) = 1$$



Midpoint method

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

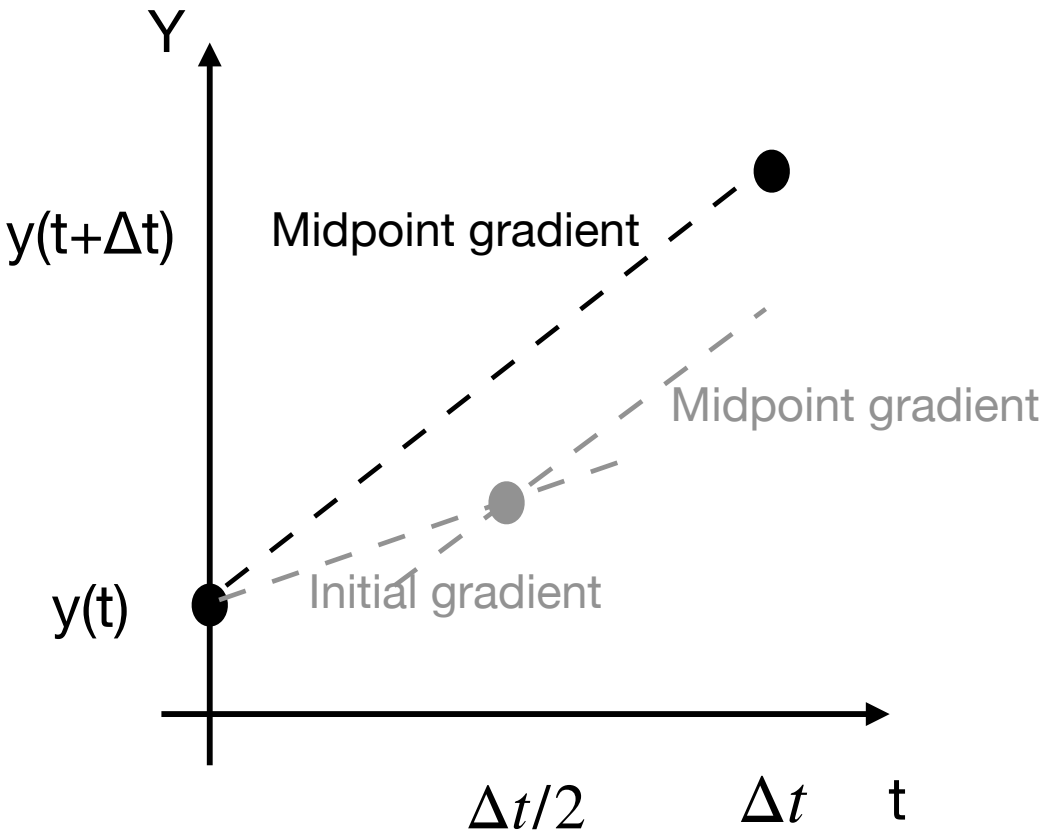
$$y(t = 0) = 1$$



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]

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

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

Examples [edit]

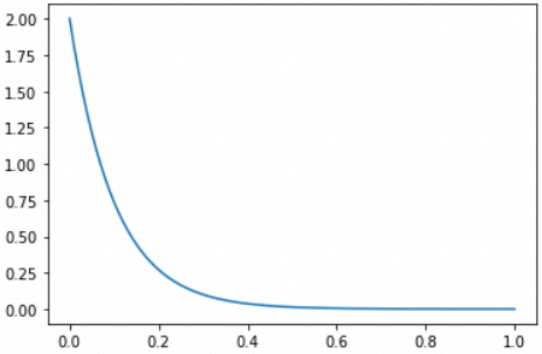
The RK4 method falls in this framework. Its tableau is[13]

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

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()");
```



All method discussed so far have been explicit methods, what is the alternative and why use it?

An explicit result is one where the variable we want, perhaps y_{k+1}, can be written explicitly in terms of quantities we know:

$$y_{k+1} = e^{y_k} + \sin(t_k) + y_k^4 + \dots$$

Implicit methods will instead result in equations like:

$$y_{k+1} + y_{k+1}^4 + 1/y_{k+1} = e^{y_k} + \sin(t_k) + y_k^4 + \dots$$

Where we cannot easily isolate and solve for the quantity we want.

These will be important for ***stiff*** problems (those with several different timescales)

Plan for today

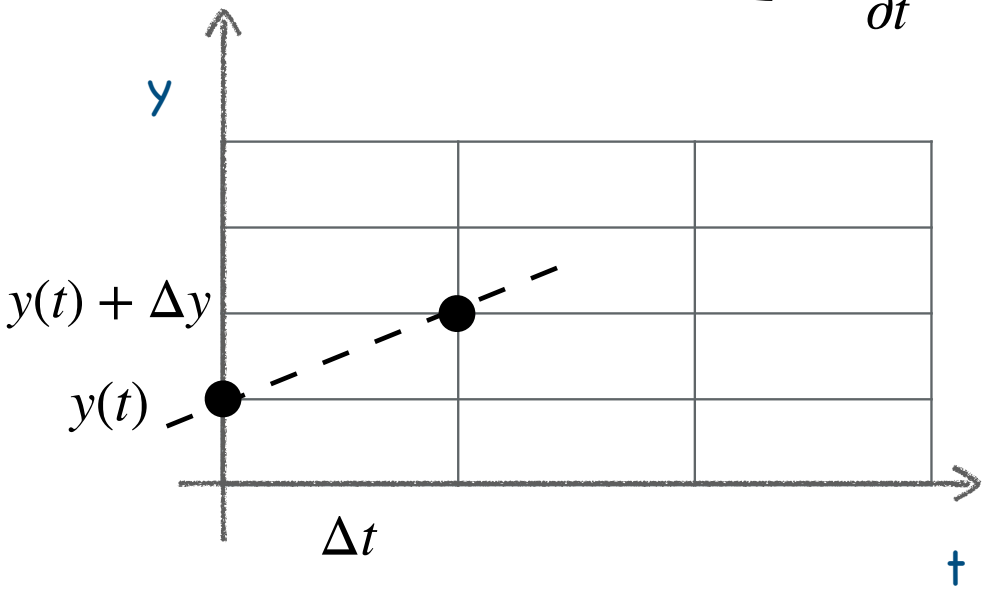
1. ~~Revision of last week - Classes, ODEs and integration~~
2. ~~The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)~~
3. How to make a higher order ODE into a first order one
4. Coursework - revision of the physical scenario and grading details
5. Tutorial this week - classes, multistep methods and second order ODEs

How do I integrate second order derivatives numerically?

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

$$\begin{cases} \dfrac{\partial v}{\partial t} - v + f(y,t) = 0 \\ \dfrac{\partial y}{\partial t} = v \end{cases}$$

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



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

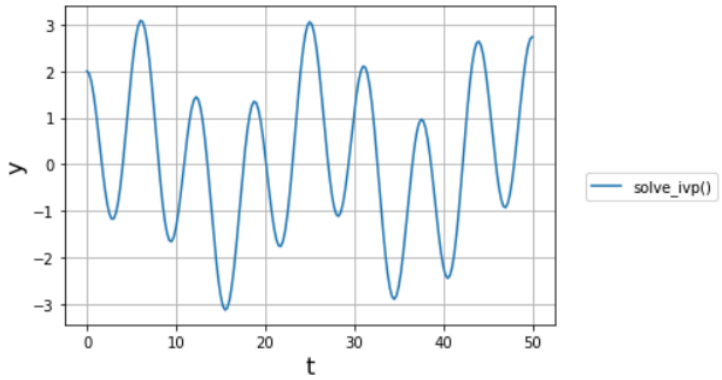
$$\Delta y = v \ \Delta t$$

2. Solve as a dimension 2 first order system

Example: the forced harmonic oscillator

```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 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));
```



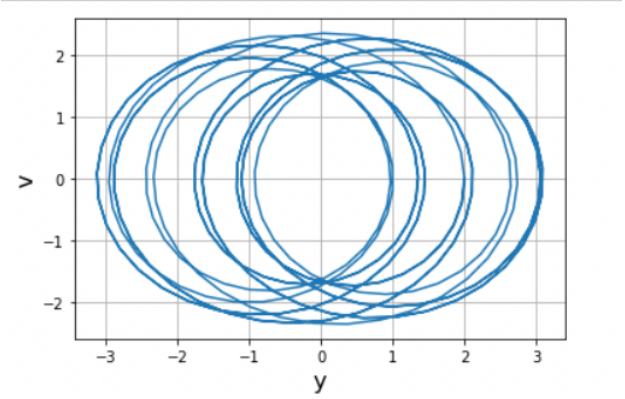
$$\frac{\partial^2 y}{\partial t^2} + y = \sin(\omega_f t)$$

$$\frac{\partial v}{\partial t} = -y + \sin(\omega_f t)$$

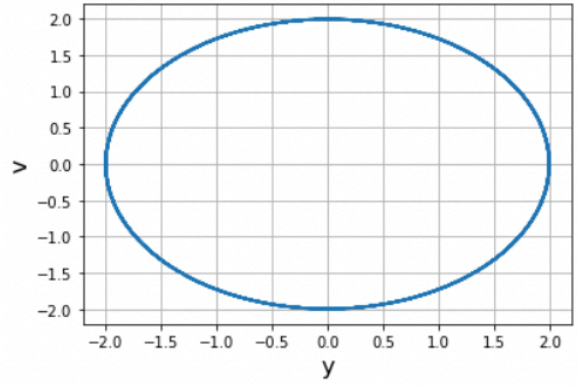
$$\frac{\partial y}{\partial t} = v$$

Example: the forced harmonic oscillator

Forced harmonic oscillator



Free harmonic oscillator



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

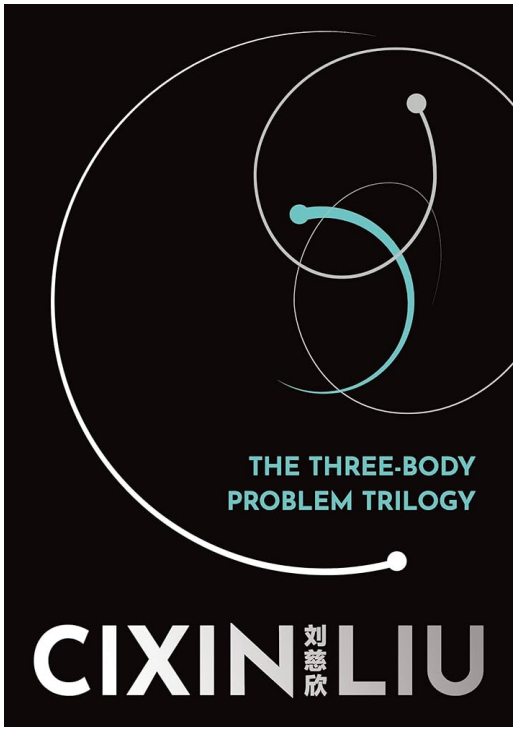
This often tells us about the energy in a system, or whether some quantities are conserved.

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. ~~The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)~~
3. ~~How to make a higher order ODE into a first order one~~
4. Coursework - revision of the physical scenario and grading details
5. Tutorial this week - classes, multistep methods and second order ODEs

Coursework - Trisolaris, the three body problem



Annals of Mathematics, **152** (2000), 881–901

A remarkable periodic solution of the three-body problem in the case of equal masses

By Alain Chenciner and Richard Montgomery

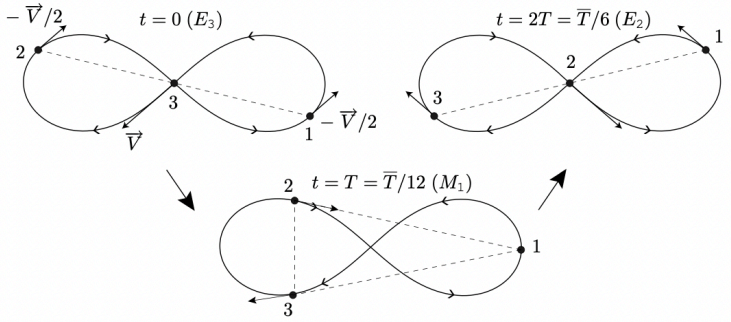


Figure 1 (Initial conditions computed by Carles Simó)

$x_1{=}{-}x_2{=}0.97000436{-}0.24308753i, x_3{=}0$; $\vec{V}{=}\dot{x}_3{=}{-}2\dot{x}_1{=}{-}2\dot{x}_2{=}{-}0.93240737{-}0.86473146i$

$\overline{T}{=}12T{=}6.32591398$, $I(0){=}2$, $m_1{=}m_2{=}m_3{=}1$

Coursework - Trisolaris, the three body problem

Coursework - Trisolaris, the three body problem

Coursework 1 : The three body problem

You should complete your coursework in this notebook and hand it in *via QMPlus* by 5pm on Friday of week 9 (24 November).

The broad goal of the project is to write code to model coplanar stellar systems (i.e., all the motion is in the x and y directions) with up to three stars.

Your code must model the following 3 scenarios

1. A two body system made up of two stars with a mass ratio 1:2 undergoing multiple stable orbits (which may be elliptic). The stars obey Newton's law:

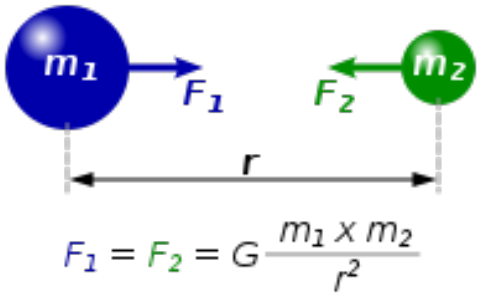
   $$\vec{F}_{12} = -\frac{G m_1 m_2}{|r_{12}|^2}\frac{\vec{r}_{12}}{|\vec{r}_{12}|}$$

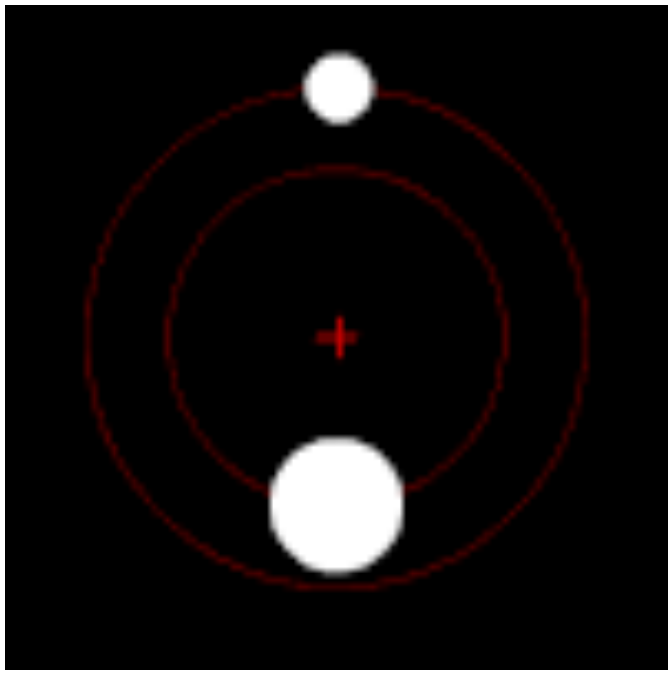
   (You can work in units in which we set $G = 1$ and the masses are order 1 numbers, but you can also choose to work in real units, as you prefer.)
2. A three body system made up of 3 stars of equal mass. Stars are assumed to be point like objects and so they cannot collide with each other (they simply pass through if at the same location). Model the stable solution discovered by Cris Moore and proved by Chenciner and Montgomery, described here and one in which they display chaotic behaviour, with one star being ejected from the system.
3. You now have a system of hypergiants. Hypergiants are the most massive stars, and so they cannot be treated as point like objects. Now if they get within some distance of each other, they should merge (*HINT: maybe we could say that they "add" themselves...*) into a single hypergiant with a combined mass of the two objects. Since momentum is conserved, we will also require that:

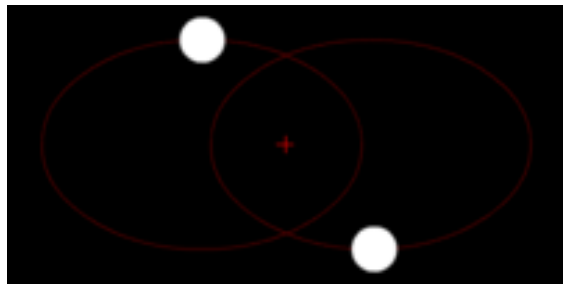
   $$(m_1 + m_2)\vec{v}_{new} = m_1\vec{v}_1 + m_2\vec{v}_2$$

   In this case, set up initial conditions so that you have 3 hypergiants initially, and a few orbits are obtained before a merger of two of the objects. The hypergiants should have 3 similar but different masses, e.g. a ratio of 0.8 : 1.0 : 1.2. You can choose the distance at which they merge, but it should be proportional to the masses of the two objects that are merging.

Coursework - Trisolaris, the three body problem

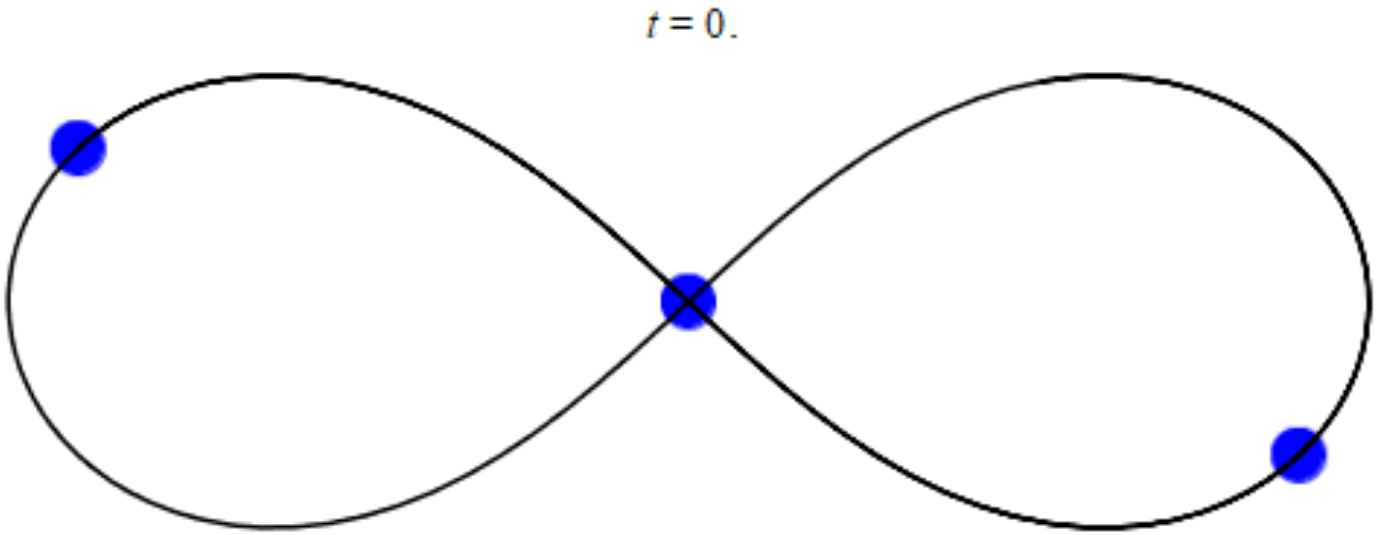






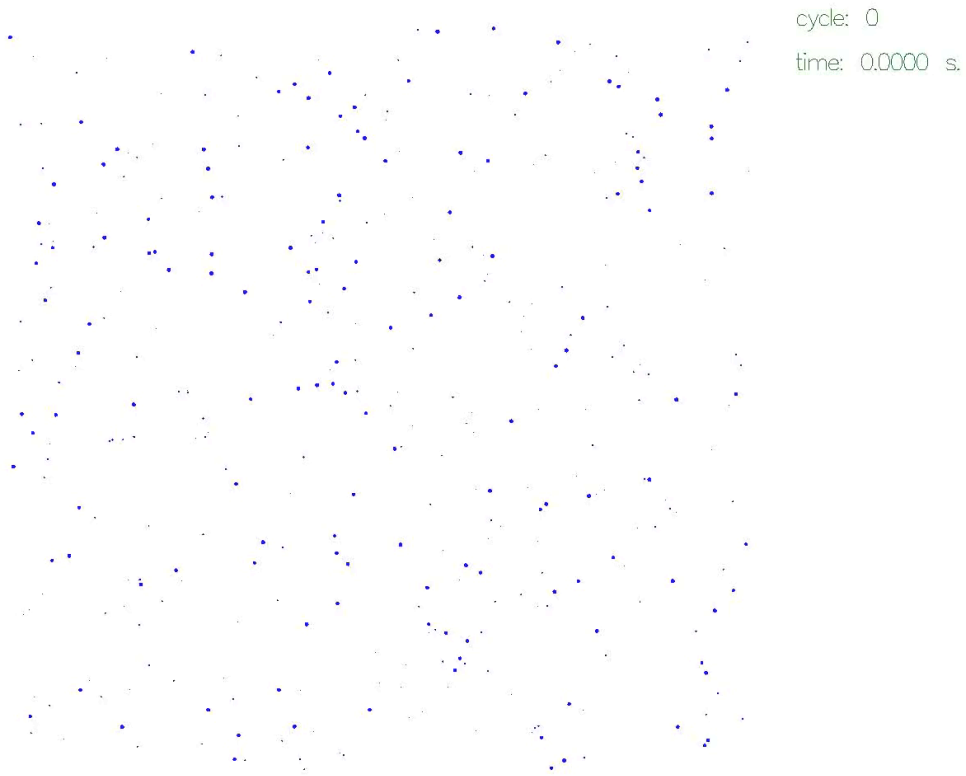
1. Two body problem

Coursework - Trisolaris, the three body problem



2. Three body problem

Coursework - Trisolaris, the three body problem



3. What is they can merge? (You only have to do 3 of these!)

Coursework - Trisolaris, the three body problem

Required components

To obtain full marks your solutions must include the following components:

1. Classes for stars, supergiants and stellar systems with multiple star components. Use of inheritance where possible and appropriate

(HINT: try to think ahead - what features of scenario 1 might you want to reuse in scenarios 2 and 3? You can save time coding by making it sufficiently general from the start.)

2. Plots of the orbital trajectories of the stars over time
3. Phase diagrams for the position and velocities of the component stars.
4. Comparison of two explicit integration techniques - the first should be scipy's solve_ivp() and the second should be the midpoint method (or another Runge Kutta method of specified order). For the latter you should confirm the order of convergence of the solution is as expected.

(HINT: you may want to investigate the rtol parameter for solve_ivp().)

5. Documentation of the code appropriate for new users who have a basic familiarity with python and ODEs (your colleagues on this course, for example!), implemented in markdown around the code blocks.
6. Defensive programming techniques including asserts and tests of key functionality

Marking scheme

- 50% for working code that correctly implements all of the requested physical scenarios
- 20% for use of defensive programming techniques - asserts and tests implemented to prevent user error and check functioning correctly, including a convergence test
- 20% for readability of code, following the agreed naming conventions of the course, appropriate commenting
- 10% for appropriate documentation of the code implemented in markdown format

Plan for today

1. ~~Revision of last week - Classes, ODEs and integration~~
2. ~~The trouble with Euler - stability and how to fix it - methods with intermediate steps (midpoint and explicit Runge Kutta methods)~~
3. ~~How to make a higher order ODE into a first order one~~
4. ~~Coursework - revision of the physical scenario and grading details~~
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 im

        # 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:

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

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