Floating point and ODEs

Number representations in computers

```
0.1  + 0.2 == 0.3
```

```
False
```

- What's going on?

Integers

- Something simpler

```
1 + 1 == 2
```

```
True
```

- Integers can be represented in binary

```
3 == 0b11 # Ooctal `0o` or hexadecimal `0h`
```

```
True
```

- Binary string representation using `bin` function

```
bin(-2)
```

```
'-0b10'
```

- Python allows for arbitrarily large integers
- No possibility of overflow or rounding error

```
2**100
```

```
1267650600228229401496703205376
```

- Only limitation is memory!

- Numpy integers are a different story

```
import numpy as np
np.int64(2**100)
```

```
OverflowError: Python int too large to convert to C long
```

- Since NumPy is using C the types have to play nicely
- Range of integers that represented with 32 bit `numpy.int32`s is $\approx \pm 2^{31} \approx \pm 2.1 \times 10^9$ (one bit for sign)
- 64 bit `numpy.int64`s lie in range $\approx \pm 2^{63} \approx \pm 9.2 \times 10^{18}$
- Apart from the risk of overflow when working NumPy's integers there are no other gotchas to worry about

Floating point numbers

- $0.1 + 0.2 \neq 0.3$ in Python is that **specifying a real number exactly would involve an infinite number of bits**
- Any finite representation necessarily approximate
- Representation for reals is called floating point arithmetic
- Essentially scientific notation

$$\text{significand} \times \text{exponent}$$

- Named *floating point* because number of digits after decimal point not fixed

---

- Requires choice of base, and Python's floating point numbers use binary
- *Numbers with finite binary representations behave nicely*

```
0.125 + 0.25 == 0.375
```

```
True
```

- For decimal numbers to be represented exactly we'd have to use base ten. Can be achieved with `decimal` module:

```
from decimal import *
Decimal('0.1') + Decimal('0.2')
```

```
Decimal('0.3')
```

- But: there's nothing to single out decimal representation in physics (as opposed to, say, finance)

- A specification for floating point numbers must give

  1. Base (or *radix*) b
  2. Precision p, the number of digits in the significand c. Thus $0 \leq c \leq b^p - 1$.
  3. A range of exponents q specifed by emin and emax with $\text{emin} \leq q + p - 1 \leq \text{emax}$.

- With one bit s for overall sign, a number then has form $(-1)^s \times c \times b^q$.
- Smallest positive nonzero number that can be represented is $b^{1+\text{emin}-p}$ (corresponding to the smallest value of the exponent) and largest is $b^{1+\text{emax}} - 1$.

$$(-1)^s \times c \times b^q$$

- Representation isn't unique: (sometimes) could make significand smaller and exponent bigger

- A unique representation is fixed by choosing the exponent to be as small as possible.
- Representing numbers smaller than b^{emin} involves a loss of precision, as number of digits in significand $< p$ and exponent takes its minimum value (subnormal numbers)
- If we stick with normal numbers and a p-bit significand, leading bit will be 1 and so can be dropped from the representation: only requires $p - 1$ bits.

- Specification for floating point numbers used by Python (and many other languages) is contained in the IEEE Standard for Floating Point Arithmetic IEEE 754
- Default Python `float` uses 64 bit *binary64* representation (often called *double precision*)
- Here's how those 64 bits are used:
  - $p = 53$ for the significand, encoded in 52 bits
  - 11 bits for the exponent
  - 1 bit for the sign

- Another common representation is 32 bit *binary32* (*single precision*) with:
  - $p = 24$ for the significand, encoded in 23 bits
  - 8 bits for the exponent
  - 1 bit for the sign

Floating point numbers in NumPy

- NumPy's finfo function tells all machine precision

```
np.finfo(np.float64)
```

```
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308, dtype=floa
```

- Note that $2^{-52} = 2.22 \times 10^{-16}$ which accounts for resolution 10^{-15}
- This can be checked by finding when a number is close enough to treated as 1.0.

```
x=1.0
while 1.0 + x != 1.0:
    x /= 1.01
```

```
print(x)
```

```
1.099427563084686e-16
```

- For binary32 we have a resolution of 10^{-6}.

```
np.finfo(np.float32)
```

```
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

- Taking small differences between numbers is a potential source of rounding error

A3 The Apollo 11 spacecraft took 76 hours to travel from the Earth to the Moon, a distance of 384,400 km. Estimate the difference between the Earth-Moon distance as measured in the rest frame of the spacecraft during the transit and the Earth-Moon distance as measured in the Earth's rest frame. You may assume the spacecraft moves with constant velocity and you may ignore the Moon's motion relative to the Earth.

- Solution: $x - x' = x(1 - \gamma^{-1}) \sim x\beta^2/2 \sim 4.2\text{mm}$.

```
import numpy as np
from scipy.constants import c
beta = 384400e3 / (76 * 3600) / c
gamma = 1/np.sqrt(1 - beta**2)
print(1 - np.float32(1/gamma), 1 - np.float64(1/gamma))
```

```
0.0 1.0981660025777273e-11
```

The dreaded NaN

- As well as a floating point system, IEEE 754 defines `Infinity` and `NaN` (Not a Number)

```
np.array([1, -1, 0]) / 0
```

```
/var/folders/9y/4kfs30yd1rz98737_h9jpn9h0000gn/T/ipykernel_13279/2604490398.py:1: RuntimeWarn
  np.array([1, -1, 0]) / 0
/var/folders/9y/4kfs30yd1rz98737_h9jpn9h0000gn/T/ipykernel_13279/2604490398.py:1: RuntimeWarn
  np.array([1, -1, 0]) / 0
```

```
array([ inf, -inf,  nan])
```

- They behave as you might guess

```
2 * np.inf, 0 * np.inf, np.inf > np.nan
```

```
(inf, nan, False)
```

- NaNs propagate through subsequent operations

```
2 * np.nan
```

```
nan
```

- If you get a NaN somewhere in your calculation, you'll probably end up seeing it somewhere in the output
- (this is the idea)

Differential equations with SciPy

> Newton's fundamental discovery, the one which he considered necessary to keep secret and published only in the form of an anagram, consists of the following: *Data aequatione quotcunque fluentes quantitates involvente, fluxiones invenire; et vice versa.* In contemporary mathematical language, this means: "It is useful to solve differential equations".
>
> Vladimir Arnold, *Geometrical Methods in the Theory of Ordinary Differential Equations*

- Solving differential equations is *not possible in general*

$$\frac{dx}{dt} = f(x,t)$$

- Cannot be solved for general $f(x,t)$
- Formulating a system in terms of differential equations represents an important first step
- Numerical analysis of differential equations is a colossal topic in applied mathematics
- Important thing is to access existing solvers (and implement your own if necessary) and to *understand their limitations*

- Basic idea is to *discretize* equation and solution $x_j \equiv x(t_j)$ at time points $t_j = hj$ with some *step size* h

---

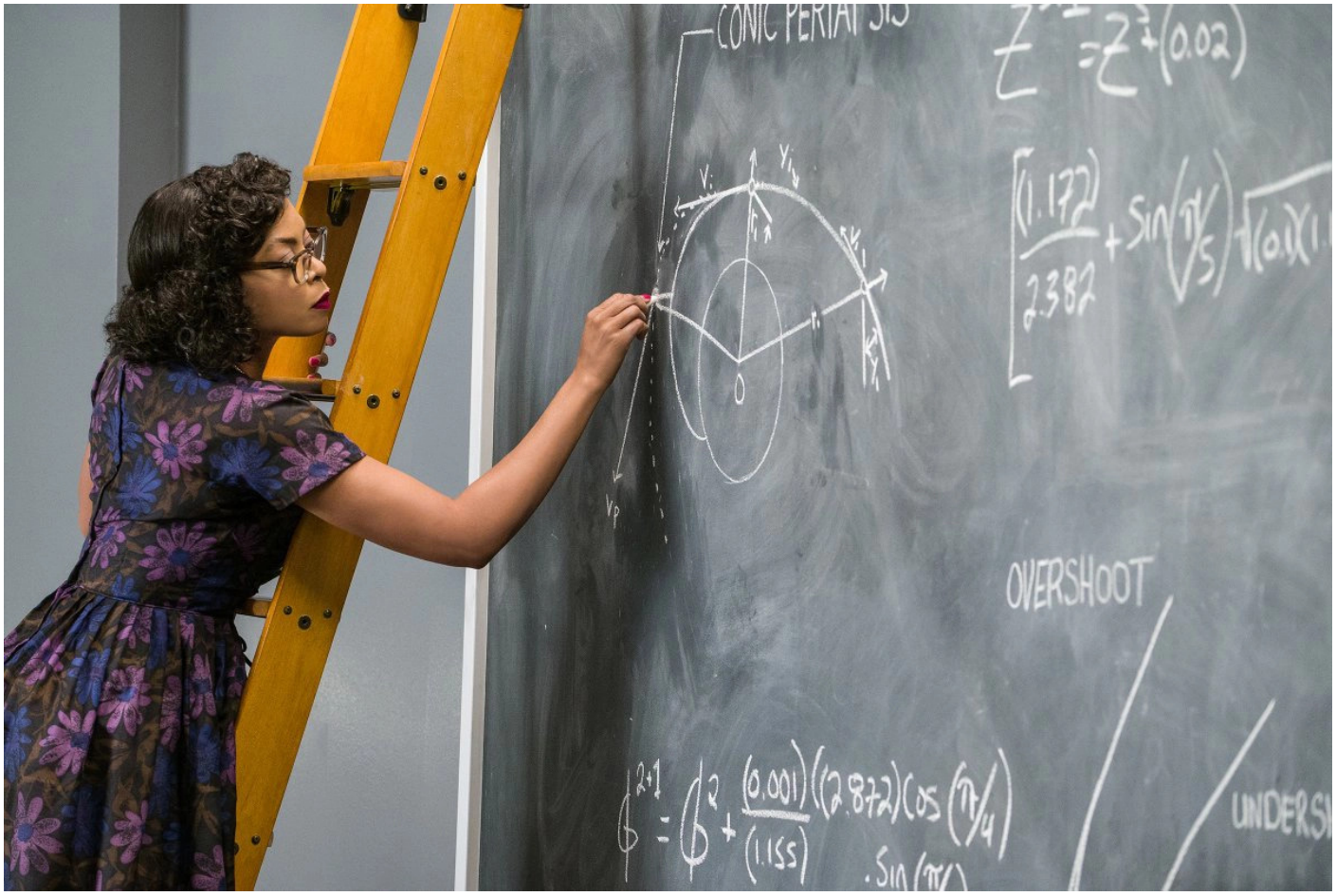


Figure 1: Taraji P. Henson as Katherine Johnson in *Hidden Figures*

Euler's method

$$\frac{dx}{dt} = f(x,t)$$

- Simplest approach: approximate LHS of ODE

$$\left.\frac{dx}{dt}\right|_{t=t_j} \approx \frac{x_{j+1} - x_j}{h}$$

$$x_{j+1} = x_j + hf(x_j, t_j)$$

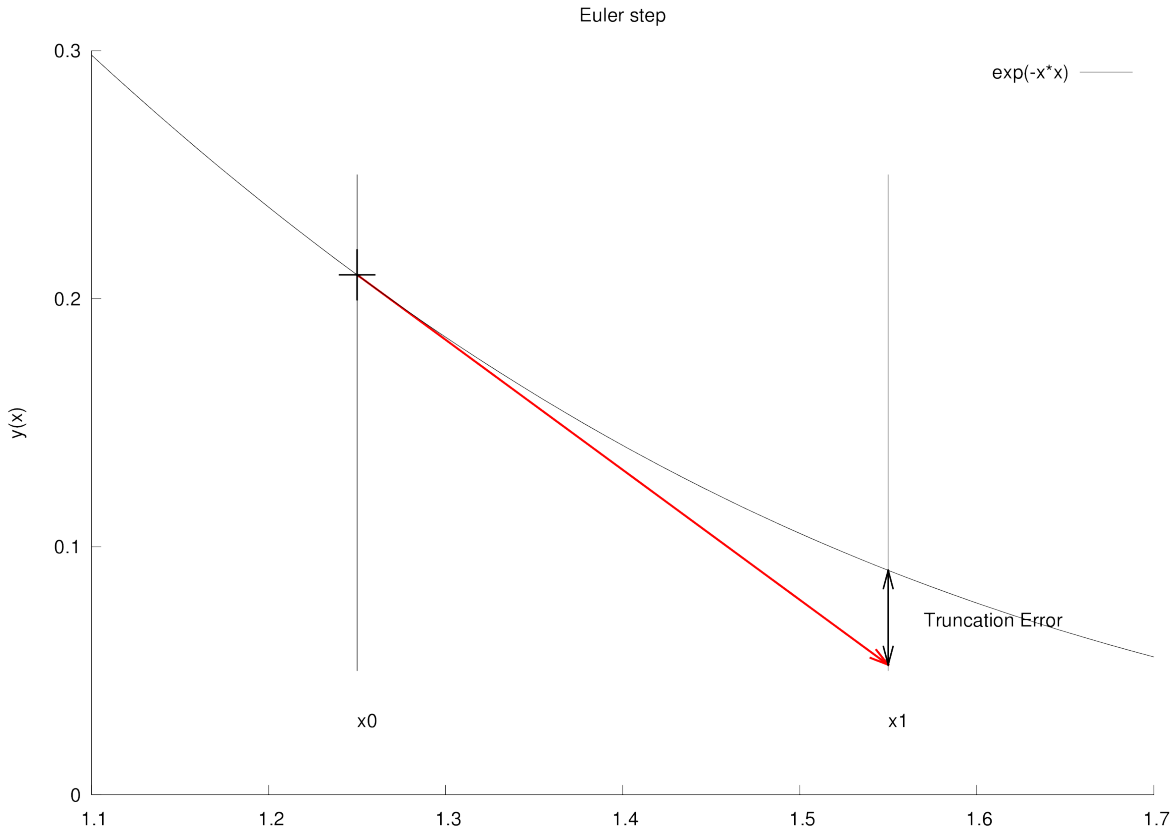
$$\boxed{x_{j+1} = x_j + hf(x_j, t_j)}$$

- Once *initial condition* x_0 is specified, subsequent values obtained by iteration

---

The Euler method approximates the function as a straight line over the distance h.

Illustrated for the ODE: $y' = -2\,x\,y$ (which has a solution $A\exp(-x^2)$)



There is a "truncation error" associated with the finite step size

A Taylor expansion yields

$$y(x_i + h) = y(x_i) + h\, y'(x_i) + \frac{h^2}{2} y''(x_i) + \mathcal{O}(h^3)$$

In Euler's method we **truncate** the series after the linear term. The **truncation error** is $\mathcal{O}(h^2)$ in each step.

How many steps do we need?

- If we want to integrate over a range of order unity ($x = 0$ to 1 for example), then we need $\mathcal{O}(h^{-1})$ steps, so the **total truncation error** is $\mathcal{O}(h)$ if we assume (pessimistically) that the errors accumulate.
- So an accuracy of 1 part in 1 million needs of order a million steps. (Of course this is not true if the function is actually a straight line…)
- Euler's Method is called **first order** since its error over a finite scale goes as h^1. An n^{th} order method has a truncation error per step $\mathcal{O}(h^{n+1})$.

Can we take an infinite number of small steps and get a perfect answer?

- No, because **round-off error due to finite precision arithmetic** also occurs. At each step we get a round-off error of some value ϵ, which depends on the computer's binary representation of numbers.
- Integrating over a finite range we accumulate a total round-off error $\sim \epsilon/h$, for a total error of
$$E \sim \frac{\epsilon}{h} + h$$
- **Moral:** if we use small steps, round-off error dominates, but if we use large steps, truncation error dominates

---

Higher order methods

- More sophisticated methods typically higher order: the SciPy function scipy.integrate.solve_ivp uses fifth order method by default

---

Midpoint method

- Midpoint method is a simple example of a higher-order integration scheme

$$\begin{aligned} k_1 &\equiv h f(x_j, t_j) \\ k_2 &\equiv h f(x_i + k_1/2, t_j + h/2) \\ x_{j+1} &= x_j + k_2 + O(h^3) \end{aligned}$$

- $\mathcal{O}(h^2)$ error cancels!
- Downside is that we have two function evaluations to perform per step, but this is often worthwhile

---

We can achieve much higher accuracy in many fewer steps by using higher-order methods

The total error E in an n^{th} order method is

$$E \sim \frac{\epsilon}{h} + h^n$$

for which h is minimised for

$$h_{\min} = \left(\frac{\epsilon}{n}\right)^{1/(n+1)}$$

The minimum error is then

$$E_{\min} \sim \left(\frac{\epsilon}{n}\right)^{n/(n+1)}$$

- For a fourth-order method and double precision arithmetic we get a step size of $h_{\min} \sim 6 \times 10^{-4}$ and a corresponding error of $E_{\min} \approx 6 \times 10^{-13}$.
- Compare this with the values of $h \sim 10^{-8}$ (implying 10^4 times as many steps) and $E \sim 10^{-8}$ we obtain for a first-order method like Euler's method.

Stability

- Euler method may be unstable, depending on equation
- Simple example:

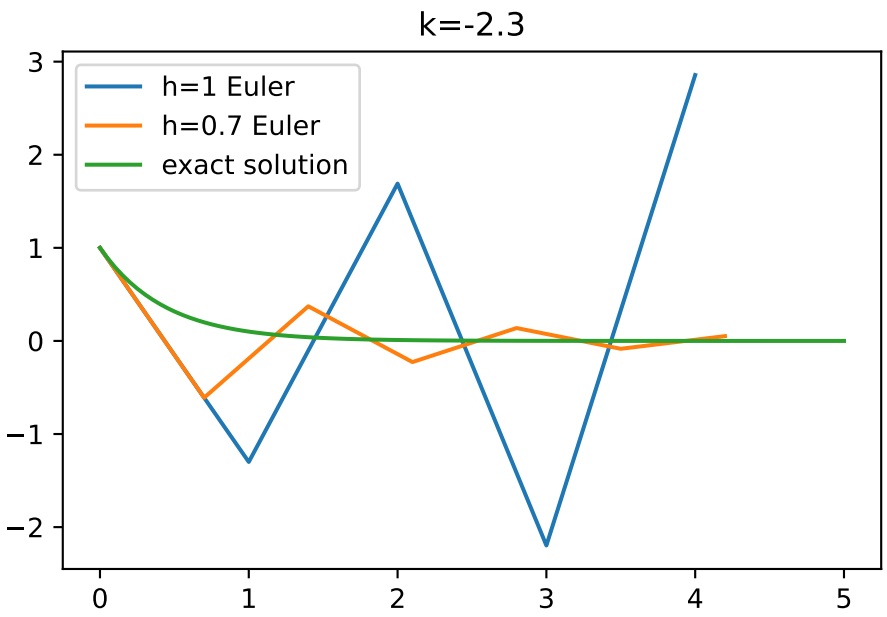
$$\frac{dx}{dt} = kx$$

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

def euler(h, t_max, k=1):
    """
    Solve the equation x' = k x, with x(0) = 1 using
    the Euler method.

    Integrate from t=0 to t=t_max using stepsize h for
    num_steps = t_max / h.

    Returns two arrays of length num_steps: t, the time coordinate, and x_0, the position.
    """
    num_steps = int(t_max / h)
    # Allocate return arrays
    x = np.zeros(num_steps, dtype=np.float32)
    t = np.zeros(num_steps, dtype=np.float32)
    x[0] = 1.0  # Initial condition
    for i in range(num_steps - 1):
        x[i+1] = x[i] + k * x[i] * h
        t[i+1] = t[i] + h  # Time step
    return t, x
```

Plot the result as a function of step size

```
k = -2.3
t_max = 5
t, x = euler(1, t_max, k)
plt.plot(t, x, label="h=1 Euler")
t, x = euler(0.7, t_max, k)
plt.plot(t, x, label="h=0.7 Euler")
t = np.linspace(0, t_max, 100)
plt.plot(t, np.exp(k * t), label="exact solution")
plt.title("k=-2.3")
plt.legend()
plt.show()
```

```
k = -2.3
t_max = 5
t, x = euler(1, t_max, k)
plt.plot(t, x, label="h=1 Euler")
t, x = euler(0.7, t_max, k)
plt.plot(t, x, label="h=0.7 Euler")
t = np.linspace(0, t_max, 100)
plt.plot(t, np.exp(k * t), label="exact solution")
plt.title("k=-2.3")
plt.legend()
plt.show()
```



- For a linear equation, the Euler update is a simple rescaling

$$x_{j+1} = x_j(1 + hk)$$

- Region of stability is $|1 + hk| \leq 1$
- In general, numerical methods need to be tested for both **accuracy** and **stability**.

Using SciPy

- Coming up with integration schemes is best left to the professionals
- Try integrate module of the SciPy library
- scipy.integrate.solve_ivp provides a versatile API

Reduction to first order system

- All these integration schemes apply to systems of *first order* differential equations
- Higher order equations can always be presented as a first order system

---

- We are often concerned with Newton's equation

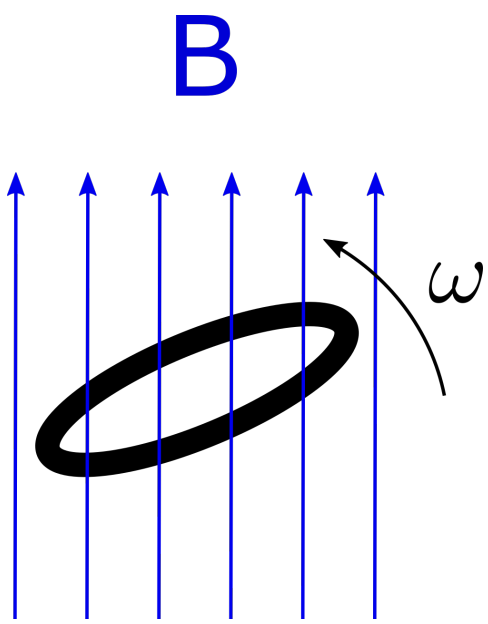
$$m\frac{d^2\mathbf{x}}{dt^2} = \mathbf{f}(\mathbf{x}, t)$$

which is three second order equations

- Turn this into a first order system by introducing the velocity $\mathbf{v} = \dot{\mathbf{x}}$, giving six equations

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}$$
$$m\frac{d\mathbf{v}}{dt} = \mathbf{f}(\mathbf{x}, t)$$

Worked example: spinning ring in a magnetic field



Old IB problem:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{2}{\tau}\sin^2\theta\,\frac{\mathrm{d}\theta}{\mathrm{d}t}$$

with approximate solution (for light damping)

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} \approx \omega_0 e^{-t/\tau}$$

We set

$$Y_0 \equiv \theta, Y_1 \equiv \dot{\theta}$$

to obtain

$$\begin{aligned}\dot{Y_0} &= Y_1 \\ \dot{Y_1} &= -\frac{2}{\tau}\sin^2(Y_0)\,Y_1\end{aligned}$$

- Solving using SciPy requires defining a function which returns the RHS of the equations

```
import scipy.integrate

def derivatives(t, y, tau):
    """
    Return the derivatives for the spinning ring equation at t,y

    The equation is
      d^2 theta/dt^2 = - (2/tau) * sin^2(theta) * d theta/dt
    and we work in the transformed variables y[0] = theta, y[1] = d(theta)/dt
    """
    return [y[1], -(2.0 / tau) * np.sin(y[0]) ** 2 * y[1]]
```

Then call `solve_ivp`

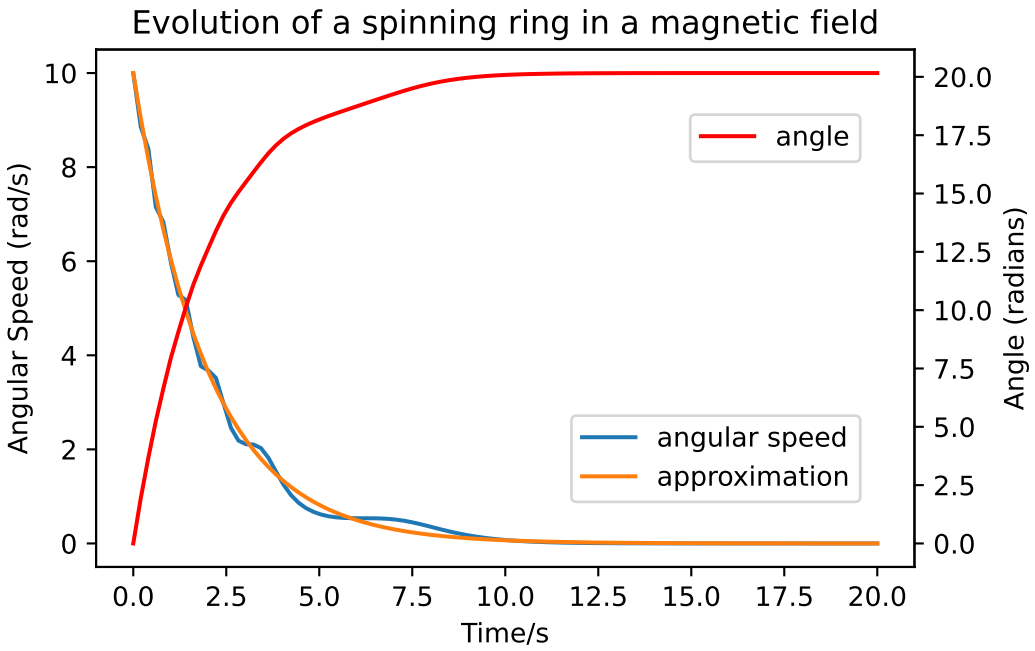
```
solution = scipy.integrate.solve_ivp(
    fun=derivatives,
```

```
    t_span=(0, 20),
    y0=(0.0, 10.0),
    args=(2.0,),
    t_eval=np.linspace(0, 20, 100),
)
x, y, dydx = solution.t, solution.y[0], solution.y[1]
```

We can now plot the results

```
fig, ax1 = plt.subplots()
ax1.plot(x, dydx, label="angular speed")
ax1.plot(x, 10 * np.exp(-x / 2.0), label="approximation")
ax1.set_xlabel("Time/s")
ax1.set_ylabel("Angular Speed (rad/s)")
ax1.set_title("Evolution of a spinning ring in a magnetic field")
ax1.legend(loc="lower right", bbox_to_anchor=(0.95, 0.1))
ax2 = ax1.twinx()  # Use second set of axes for angular position
ax2.plot(x, y, label="angle", color="red")
ax2.set_ylabel("Angle (radians)")
ax2.legend(loc="upper right", bbox_to_anchor=(0.95, 0.9));
```

```
fig, ax1 = plt.subplots()
ax1.plot(x, dydx, label="angular speed")
ax1.plot(x, 10 * np.exp(-x / 2.0), label="approximation")
ax1.set_xlabel("Time/s")
ax1.set_ylabel("Angular Speed (rad/s)")
ax1.set_title("Evolution of a spinning ring in a magnetic field")
ax1.legend(loc="lower right", bbox_to_anchor=(0.95, 0.1))
ax2 = ax1.twinx()  # Use second set of axes for angular position
ax2.plot(x, y, label="angle", color="red")
ax2.set_ylabel("Angle (radians)")
ax2.legend(loc="upper right", bbox_to_anchor=(0.95, 0.9));
```



- We did not have to specify a time step
- This is determined *adaptively* by solver to keep estimate of local error below `atol + rtol * abs(y)`
- Default values of 10^{-6} and 10^{-3} respectively
- Monitoring conserved quantities (e.g. energy, momentum, angular momentum) is a good experimental method for assessing the accuracy of integration