Computational Methods and Modelling

Antonio Attili

antonio.attili@ed.ac.uk

School of Engineering
University of Edinburgh
United Kingdom

Lecture 4
Ordinary Differential Equations (ODEs), Euler and Runge-Kutta methods

THE UNIVERSITY of EDINBURGH

Introduction

In the present and following lectures, we will study how to deal with Differential Equations using numerical methods

- How to solve Ordinary Differential Equations using numerical methods
- How to approximate derivatives of different orders that appear in Ordinary and Partial Differential Equation
- Learn about the stability of numerical schemes

- In this lecture, we will discuss
  - Derivatives and Taylor series
  - Euler Method
  - Runge-Kutta Methods

Motivations

Differential equations are omnipresent in mathematical models of natural phenomena and engineering applications.

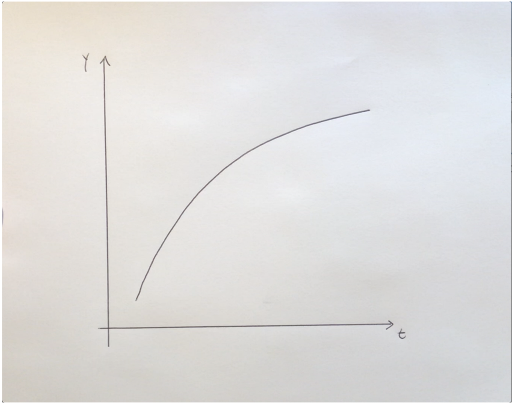
Examples are countless and with complexity that might range from

- Simple, linear Ordinary Differential Equations (ODE):
  Pendulum equation for in the small-angle approximation:
  $$\frac{d^2\theta}{dt^2} + \frac{g}{l}\theta = 0$$
- To non-linear Ordinary Differential Equations:
  Logistic differential equation (applications in machine learning, population dynamics, virus spread)
  $$\frac{df}{dt} = rf - \frac{rf^2}{k}$$
  where f is the population (or number of infected)
- And non-linear systems of Partial Differential Equations (PDE):
  Navier-Stokes equations of fluid-dynamics (weather forecasting, energy production devices like gas and wind turbines)
  $$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial \tau_i j}{\partial x_j}$$
  where u_i is the velocity vector, p the pressure, $\nu =$ the kinematic viscosity, and τ_{ij} the stress tensor.

Common aspect: All these equations contain derivatives.

Approximation of a derivative

- We want to compute the derivative of a function $y(t)$ in the point t_i.



See video on Learn: "Lecture 4 - Differential Equations, Euler and Runge-Kutta methods - Video 1" in Course Material - Lecture Slides

- The derivative (the slope of the function) can be approximated as:

$$\frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y(t_{i+1}) - y(t_i)}{t_{i+1} - t_i}$$

Approximation of a derivative

- The approximate expression for the slope

$$\frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y(t_{i+1}) - y(t_i)}{t_{i+1} - t_i}$$

  is called a **Finite Divided Difference**

- The expression

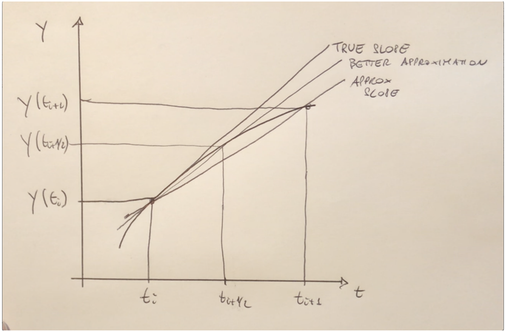
$$\frac{dy}{dt} \approx \frac{\Delta y}{\Delta t}$$

  is approximate because Δ is finite.

- From calculus:

$$\frac{dy}{dt} = \lim_{\Delta t \to 0} \frac{\Delta y}{\Delta t}$$

  So if $\Delta t \to 0$ the approximate slope converges to the real one



See video on Learn: "Lecture 4 - Differential Equations, Euler and Runge-Kutta methods - Video 2" in Course Material - Lecture Slides

Taylor's theorem and Taylor series

- Taylor's theorem: If the function $f(x)$ of the independent variable x and its $n+1$ derivatives are continuous in an interval containing the two points x_i and $x_{i+1} = x_i + h$, $f(x)$ can be expanded in the following series:

$$
\begin{aligned}
f(x_{i+1}) = & f(x_i) + \frac{f'(x_i)}{1!}(x_{i+1} - x_i) + \\
& + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2 + \frac{f'''(x_i)}{3!}(x_{i+1} - x_i)^3 + \\
& + \ldots + \\
& + \frac{f^{(n)}(x_i)}{n!}(x_{i+1} - x_i)^n + R_n
\end{aligned}
$$

where

$$R_n = \int_{x_i}^{x_{i+1}} \frac{(x_i - t)^n}{n!} f^{(n+1)}(t)dt$$

- R_n can be also expressed in the **Lagrangian form** (the derivation of R_n is not important for our purpose).

$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x_{i+1} - x_i)^{n+1}$$

where $x_{i+1} < \xi < x_i$

- R_n is often called **Truncation Error**

The Taylor series term-by-term

- Defining $h = x_{i+1} - x_i$ the Taylor series can be written as:

$$f(x_{i+1}) = f(x_i) + \frac{f'(x_i)}{1!}h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + ... + \frac{f^{(n)}(x_i)}{n!}h^n + R_n$$

- It can be used to approximate a function in a point in terms of the value of the function and its derivatives in another point
- Depending on the number of terms we keep in the series, we have different levels of approximation:
  - *zero-order approximation*:

    $$f(x_{i+1}) = f(x_i)$$

    If $f(x)$ is constant, this is a perfect estimate
  - first-order approximation

    $$f(x_{i+1}) = f(x_i) + f'(x_i)h$$

    This can predict a change in the function, but it is exact only if the function is linear
  - order n approximation

    $$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + ... + R_n \quad \text{with} \quad R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!}h^{n+1}$$

Truncation error

- The Truncation Error
$$R_n = \frac{f^{(n+1)}(\xi)}{(n+1)!} h^{n+1}$$
cannot be determined since ξ is not known. We only know that it lies between x_i and x_{i+1}

- However, we have control over h. For different orders (equivalently, different values of n), the error decreases in different ways if we decreases h.

- We often write $R_n = \mathcal{O}(h^{n+1})$.
The expression $\mathcal{O}(h^{n+1})$ states that the error is of order of h^{n+1}, which means that the error is proportional to h^{n+1}.

Numerical derivative

- Let's consider the Taylor series, truncated after the first derivative:

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + R_1$$

- This equation can be solved for:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} - \frac{R_1}{x_{i+1} - x_i}$$

- Which is the same expression we wrote in the graphical excercise we did before, but now we have also an estimate of the error:

$$\frac{R_1}{x_{i+1} - x_i} = \frac{f''(\xi)(x_{i+1} - x_i)^2}{2!} \frac{1}{x_{i+1} - x_i} = \mathcal{O}(x_{i+1} - x_i)$$

- In the usual more compact form:

$$f'(x_i) = \frac{\Delta f_i}{h} + \mathcal{O}(h)$$

  where Δf_i is called **first forward difference** and h is called the **step size**.
  It is called forward since we used data in i and $i+1$
- Then, $\Delta f_i / h$ is the **first forward divided difference**

The first forward divided difference is one of many ways to approximate the derivative using the Taylor series.

One-step methods for ODEs

- We want to solve equations in the form

$$\frac{dy}{dx} = f(x, y)$$

  with a given initial condition $y_0 = y(x_0)$
- To solve this equation with a numerical method on a set of discrete points, we need to be able to extrapolate from a value y_i to a new value y_{i+1} over a step h (usually starting from the initial condition y_0).
- In a general mathematical form, this translates to:

$$y_{i+1} = y_i + \phi h$$

  where ϕ is an estimate of an appropriate slope of the function y over the step h.
- Then, this formula can be applied step-by-step to compute on an interval as large as we want.
- This formula is the general way to express all **one-step** methods.
- The difference for different methods could be in the way we specify the slope ϕ.
- The simplest approach is to estimate the slope from the differential equation itself as the first derivative of y at the point x_i, which is nothing else than $f(x_i, y_i)$.
- This approach is called the **Euler Method**.
- Alternative ways to estimate the slope ϕ can result in more accurate predictions, for example in **one-step Runge-Kutta methods**.

Euler Method

- For the equation:

$$\frac{dy}{dx} = f(x, y) \quad \text{with} \quad y_0 = y(x_0)$$

  The formula

$$y_{i+1} = y_i + f(x_i, y_i)h$$

  is referred to as **Euler Method** or sometimes the **Euler-Cauchy Method**.
- A new value of y is computed extrapolating **linearly** over the step h using a slope approximated with the derivative in the original point x_i, where the solution and its derivatives are know.

Python code, Euler method

Euler method

```python
# ----------------------------------------------------
# importing modules
import numpy as np
import matplotlib.pyplot as plt
import math
# ----------------------------------------------------

# ----------------------------------------------------
# inputs

# functions that returns dy/dx
# i.e. the equation we want to solve: dy/dx = - y
def model(y,x):
    k= - 1
    dydx = k * y
    return dydx

# initial conditions
x0 = 0
y0 = 1
# total solution interval
x_final = 1
# step size
h = 0.2
# ----------------------------------------------------
```

```python
# ----------------------------------------------------
# Euler method

# number of steps
n_step = math.ceil(x_final/h)

# Definition of arrays to store the solution
y_eul = np.zeros(n_step+1)
x_eul = np.zeros(n_step+1)

# Initialize first element of solution arrays
# with initial condition
y_eul[0] = y0
x_eul[0] = x0

# Populate the x array
for i in range(n_step):
    x_eul[i+1]  = x_eul[i]  + h

# Apply Euler method n_step times
for i in range(n_step):
    # compute the slope using the differential equation
    slope = model(y_eul[i],x_eul[i])
    # use the Euler method
    y_eul[i+1] = y_eul[i] + h * slope
# ----------------------------------------------------
```


THE UNIVERSITY of EDINBURGH

Python code, output example

Output

```python
# -----------------------------------------------------
# super refined sampling of the exact solution c*e^(-x)
# n_exact linearly spaced numbers
# only needed for plotting reference solution

# Definition of array to store the exact solution
n_exact = 1000
x_exact = np.linspace(0,x_final,n_exact+1)
y_exact = np.zeros(n_exact+1)

# exact values of the solution
for i in range(n_exact+1):
    y_exact[i] = y0 * math.exp(-x_exact[i])
# -----------------------------------------------------

# -----------------------------------------------------
# print results on screen
print ('Solution: step x y-eul y-exact error%')
for i in range(n_step+1):
    print(i,x_eul[i],y_eul[i], y0 * math.exp(-x_eul[i]),
            (y_eul[i]- y0 * math.exp(-x_eul[i]))/
            (y0 * math.exp(-x_eul[i])) * 100)
# -----------------------------------------------------
```

```python
# -----------------------------------------------------
# print results in a text file (for later use if needed)
file_name= 'output_h' + str(h) + '.dat'
f_io = open(file_name,'w')
for i in range(n_step+1):
    s1 = str(i)
    s2 = str(x_eul[i])
    s3 = str(y_eul[i])
    s4 = s1 + ' ' + s2 + ' ' + s3
    f_io.write(s4 + '\n')
f_io.close()
# -----------------------------------------------------

# -----------------------------------------------------
# plot results
plt.plot(x_eul, y_eul , 'b.-',x_exact, y_exact , 'r-')
plt.xlabel('x')
plt.ylabel('y(x)')
plt.show()
# -----------------------------------------------------
```

Solution obtained with the Euler method

- We consider the ODE:

$$\frac{dy}{dx} = -y \quad \text{with} \quad y(x=0) = 1$$

- which has the analytical (exact) solution

$$y(x) = e^{-x}$$

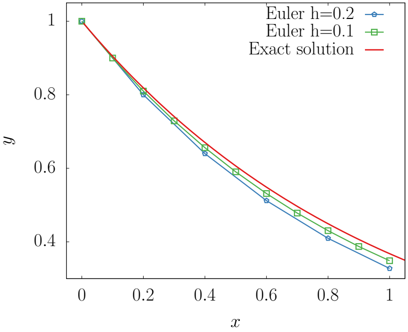
- We run the Euler methods to compute the solution in the interval $0 \leq x \leq 1$ for two different steps $h = 0.2$ and $h = 0.1$:

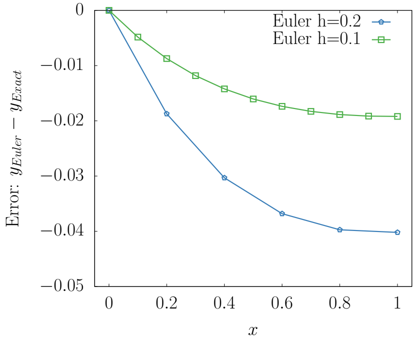
x	y exact	y Euler $h = 0.1$	error % $h = 0.1$	y Euler $h = 0.2$	error % $h = 0.2$
0.0	1.0	1.0	0.0	1.0	0.0
0.1	0.904	0.9	-0.534	NA	NA
0.2	0.818	0.81	-1.066	0.800	-2.287
0.3	0.740	0.729	-1.595	NA	NA
0.4	0.670	0.656	-2.121	0.640	-4.523
0.5	0.606	0.590	-2.644	NA	NA
0.6	0.548	0.531	-3.165	0.512	-6.707
0.7	0.496	0.478	-3.682	NA	NA
0.8	0.449	0.43	-4.197	0.409	-8.841
0.9	0.406	0.387	-4.709	NA	NA
1.0	0.367	0.348	-5.219	0.327	-10.92

Table: Solution and error for the Euler method with two two different steps (NA = not available)

Solution obtained with the Euler method

- Plotting the two solutions in the interval $0 \leq x \leq 1$ with different steps $h = 0.2$ and $h = 0.1$ we obtain:





- The error decreases by approximately a factor of 2 if the step h is halved
- Note that we used very large values of h in the example to highlight the numerical error

Runge-Kutta methods

- **Runge-Kutta** methods achieve high accuracy without the use of higher order derivatives like in the case of the Taylor saries
- Many versions exist, but all can be cast as: $$\boxed{y_{i+1} = y_i + \phi(x_i, y_i, h)h}$$

  where $\phi(x_i, y_i, h)$ is usually called an **increment function**.
- The increment function $\phi(x_i, y_i, h)$ is written, in general form as:

$$\phi = a_1 k_1 + a_1 k_1 + ... + a_n k_n$$

  where the coefficients a_j are constant and:

$$\begin{aligned}
k_1 &= f(x_i, y_i) \\
k_2 &= f(x_i + p_1 h, y_i + q_{11} k_1 h) \\
k_3 &= f(x_i + p_2 h, y_i + q_{21} k_1 h + q_{22} k_2 h) \\
&. \\
&. \\
k_n &= f(x_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + ... + q_{n-1,n-1} k_n - 1h)
\end{aligned}$$

  where also the coefficients p_j and q_j are constant.
- The $k_1 ... k_n$ can be computed in a cascade from k_1 to k_2 to k_n (recurrence relations)
- Once n is selected, which is the order of the method, all the constants are computed by equating $y_{i+1} = y_i + \phi(x_i, y_i, h)h$ to the terms of the Taylor series.
- Finally, it is worth noting that the Runge-Kutta method with $n = 1$ is the Euler method.

Computation of the coefficients of the second-order Runge-Kutta method

- With $n = 2$, the Runga-Kutta method is written as:

$$y_{i+1} = y_i + (a_1 k_1 + a_2 k_2)h$$

  where

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + p_1 h, y_i + q_{11} k_1 h) \end{aligned}$$

  and we need to compute a_1, a_2, p_1, and q_{11}.

- To do this, we start from the Taylor series:

$$y_{i+1} = y_i + f(x_i, y_i)h + \frac{f'(x_i, y_i)}{2} h^2$$

  We use chain rule to compute:

$$f'(x_i, y_i) = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{dy}{dx}$$

  and substituting this in the Taylor series we get

$$y_{i+1} = y_i + f(x_i, y_i)h + \left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{dy}{dx} \right) \frac{h^2}{2}$$

- Now, we have to equate this with $y_{i+1} = y_i + (a_1 k_1 + a_2 k_2)h$ to find the coefficients.

Computation of the coefficients of the second-order Runge-Kutta method

- Interpreting $k_2 = f(x_i + p_1 h, y_i + q_{11} k_1 h)$ as a Taylor series expansion with respect to both variables x and y, we can write:

$$f(x_i + p_1 h, y_i + q_{11} k_1 h) = f(x_i, y_i) + p_1 h \frac{\partial f}{\partial x} + q_{11} k_1 h \frac{\partial f}{\partial y} + \mathcal{O}(h^2)$$

- Inserting this and the expression $k_1 = f(x_i, y_i)$ in $y_{i+1} = y_i + (a_1 k_1 + a_2 k_2) h$:

$$y_{i+1} = y_i + \boxed{\left[a_1 f(x_i, y_i) + a_2 f(x_i, y_i)\right] h} + \left[a_2 p_1 \frac{\partial f}{\partial x} + a_2 q_{11} f(x_i, y_i) \frac{\partial f}{\partial y}\right] h^2 + \mathcal{O}(h^2)$$

- We equate term-by-term this equation with

$$y_{i+1} = y_i + \boxed{f(x_i, y_i) h} + \left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} \frac{dy}{dx}\right) \frac{h^2}{2}$$

- In order for the two highlighted terms to be equal, we must have

$$a_1 + a_2 = 1$$

- Equating also the other terms, we have

$$a_2 p_1 = \frac{1}{2} \quad \text{and} \quad a_2 q_{11} = \frac{1}{2}$$

Computation of the coefficients of the second-order Runge-Kutta method

- So the second-order Runga-Kutta method is:

$$y_{i+1} = y_i + (a_1 k_1 + a_2 k_2)h$$

where

$$k_1 = f(x_i, y_i)$$
$$k_2 = f(x_i + p_1 h, y_i + q_{11} k_1 h)$$

with

$$a_1 + a_2 = 1$$
$$a_2 p_1 = \frac{1}{2}$$
$$a_2 q_{11} = \frac{1}{2}$$

- Since we have 3 equations and 4 unknowns, one of the coefficients needs to be specified arbitrarily. Therefore, there is an infinite number of second-order Runge-Kutta methods, which will give different results if the equation is more complicate than a simple linear or quadratic one.

Computation of the coefficients of the second-order Runge-Kutta method

- If we assume $a_2 = 1/2$, we can calculate the other coefficients to obtain $a_1 = 1/2$, $p_1 = 1$, and $q_{11} = 1$. Then:

  $$y_{i+1} = y_i + (\frac{1}{2}k_1 + \frac{1}{2}k_2)h$$

  with

  $$k_1 = f(x_i, y_i)$$
  $$k_2 = f(x_i + h, y_i + k_1 h)$$

  This is the predictor-corrector method described before, since k_1 and k_2 are the derivatives in x_i and x_{i+1}
- If $a_2 = 1$, we obtain the midpoint method also described before
- If $a_2 = 2/3$, we obtain the **Ralston method**, which is the second-order Runge-Kutta method with the minimum truncation error:

  $$y_{i+1} = y_i + (\frac{1}{3}k_1 + \frac{2}{3}k_2)h$$

  with

  $$k_1 = f(x_i, y_i)$$
  $$k_2 = f(x_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1 h)$$

Fourth order Runge-Kutta methods

- The most popular Runge-Kutta methods are the fourth order
- The most used version is:

$$y_{i+1} = y_i + \left(\frac{1}{6}k_1 + \frac{2}{6}k_2 + \frac{2}{6}k_3 + \frac{1}{6}k_4\right)h$$

  with

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h) \\ k_3 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h) \\ k_4 &= f(x_i + h, y_i + k_3 h) \end{aligned}$$

- The slope

$$\frac{1}{6}k_1 + \frac{2}{6}k_2 + \frac{2}{6}k_3 + \frac{1}{6}k_4$$

  is a weighted average of four slopes with
  - K_1 computed in the first point x_i (like in the Euler method)
  - K_2 and K_3 computed in the middle $x_{i+1/2}$ (like in the mid-point method)
  - K_4 computed in the end-point x_{i+1} (like in the predictor-corrector method)

Python code, Euler vs Runge-Kutta methods

- As usual, we want to solve

$$\frac{dy}{dx} = f(x, y) \quad \text{with} \quad y_0 = y(x_0)$$

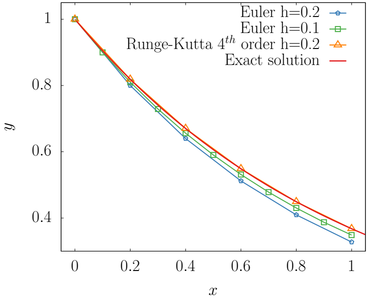
Euler method

```
# ------------------------------------------------------
# Euler method



# Apply Euler method n_step times
for i in range(n_step):
    # compute the slope using the differential equation
    slope = model(y_eul[i],x_eul[i])
    # use the Euler method
    y_eul[i+1] = y_eul[i] + h * slope
# ------------------------------------------------------
```

Runge-Kutta fourth order method

```
# ------------------------------------------------------
# Fourth Order Runge-Kutta method


# Apply RK method n_step times
for i in range(n_step):

    # Compute the four slopes
    x_dummy = x_rk[i]
    y_dummy = y_rk[i]
    k1 =  model(y_dummy,x_dummy)

    x_dummy = x_rk[i]+h/2
    y_dummy = y_rk[i] + k1 * h/2
    k2 =  model(y_dummy,x_dummy)

    x_dummy = x_rk[i]+h/2
    y_dummy = y_rk[i] + k2 * h/2
    k3 =  model(y_dummy,x_dummy)

    x_dummy = x_rk[i]+h
    y_dummy = y_rk[i] + k3 * h
    k4 =  model(y_dummy,x_dummy)

    # compute the slope as weighted average of four slopes
    slope = 1/6 * k1 + 2/6 * k2 + 2/6 * k3 + 1/6 * k4

    # use the RK method
    y_rk[i+1] = y_rk[i] + h * slope
# ------------------------------------------------------
```

Solution obtained with the Runge-Kutta method



x	y exact	error Euler % $h = 0.1$	error R-K4 % $h = 0.2$
0.0	1.0	0.0	0.0
0.2	0.818	-1.066	3.2×10^{-4}
0.4	0.670	-2.121	6.3×10^{-4}
0.6	0.548	-3.165	9.4×10^{-4}
0.8	0.449	-4.197	1.2×10^{-3}
1.0	0.367	-5.219	1.5×10^{-4}

Table: Error for the Euler and Runge-Kutta methods

- The error for the Runge-Kutta method is remarkably smaller compared to the error obtained with the Euler method
- It is worth noting that the error is small even with the rather large step $h = 0.2$ used