Ordinary Differential Equations

Numerical Computing with Python

Alexey Fedoseev

November 12, 2019

Differential equations

The idea behind differential equations is fairly simple:

A differential equation states how a rate of change (a “differential”) in one variable is related to other variables.

An ordinary differential equation (ODE) is an equation that involves some ordinary derivatives (as opposed to partial derivatives) of a function.

There is one differential equation that everybody probably knows, that is Newton’s Second Law of Motion. If an object of mass m is moving with acceleration a and being acted on with force F then Newton’s Second Law tells us

$$F = ma$$

Differential equations

To see that this is in fact a differential equation we need to rewrite it. First, remember that we can rewrite the acceleration, a, in one of two ways:

$$a = \frac{dv}{dt} \quad \text{OR} \quad a = \frac{d^2u}{dt^2},$$

Where v is the velocity of the object and u is the position function of the object at any time t. So, with all these things in mind Newton's Second Law can now be written as a differential equation in terms of either the velocity, v, or the position, u, of the object as follows:

$$m\frac{dv}{dt} = F(t, v)$$

$$m\frac{d^2u}{dt^2} = F(t, u, \frac{du}{dt})$$

Initial-Value Problems (IVP)

- First-order Ordinary Differential Equation (ODE)

$$y'(t) = f(x, y)$$

  has infinite family of solution curves - the general solution.

- **Initial condition** - given two real numbers x_0 and y_0, we seek a solution for $x > x_0$ such that

$$y(x_0) = y_0$$

  The differential equation together with the initial condition is called an **initial value problem**.

- Systems of Differential Equations

$$\frac{d\mathbf{Y}}{dx} = \mathbf{F}(x, \mathbf{Y}), \quad \mathbf{Y}(x_0) = \mathbf{Y_0},$$

$$\mathbf{Y} = (y_1, y_2, \ldots, y_n)^T, \mathbf{F} = (f_1(x, \mathbf{Y}), f_2(x, \mathbf{Y}), \ldots, f_n(x, \mathbf{Y}))^T$$

Higher-Order ODEs

IVP for ODE of a higher order

$$y^{(n)} = F(x, y, y', y'', \dots, y^{(n-1)}),$$

$$y(x_0) = y_0, y'(x_0) = y_1, y''(x_0) = y_2, \dots, y^{(n-1)}(x_0) = y_{n-1}$$

could be converted into a first-order IVP by introducing extra variables.

Denote $z_1(x) = y(x), z_2(x) = y'(x), \dots, z_n(x) = y^{(n-1)}(x)$. The resulting first-order system is

$$\begin{cases} z_1' = z_2, \\ z_2' = z_3, \\ \dots \\ z'_{(n-1)} = z_n, \\ z_n' = F(x, z_1, z_2, \dots, z_n) \end{cases} \qquad \begin{cases} z_1(x_0) = y_0, \\ z_2(x_0) = y_1, \\ \dots \\ z_n(x_0) = y_{n-1} \end{cases}$$

Numerical Methods for ODEs

Let $a \le x \le b$. Consider the following IVP

$$y' = f(x,y), \quad y(a) = y_0,$$

Let $x_i = a + ih$, $h = (b-a)/N$, $i = 0, \dots, N$ be uniformly spaced points. Denote $y_n \approx y(x_n)$ as the numerical solution of IVP.

- Forward Euler Method (explicit, global error is $O(h)$)

$$y_{n+1} = y_n + hf(x_n, y_n)$$

- Backward Euler Method (implicit, global error is $O(h)$)

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

Numerical Methods for ODEs

- Trapezoidal Rule (implicit, global error is $O(h^2)$)

$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1}))$$

- Heun's Method (predictor-corrector, explicit, global error is $O(h^2)$)

$$\tilde{y}_{n+1} = y_n + hf(x_n, y_n)$$
$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, \tilde{y}_{n+1}))$$

Numerical Methods for ODEs

- Runge-Kutta 4 (explicit, global error is $O(h^4)$)

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$k_1 = f(x_n, y_n), \qquad k_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1),$$
$$k_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2), \qquad k_4 = f(x_n + h, y_n + hk_3),$$

Runge-Kutta method aims to achieve a higher accuracy by sacrificing the efficiency of Euler's method through re-evaluation f.

A numerical method is **convergent** if the numerical solution approaches the exact solution as $h \to 0$.

A numerical method is **stable** if for two different initial conditions y_0 and $\tilde{y}_0$ the computed solutions are close in the sense that $|y_n - \tilde{y}_n| < C|y_0 - \tilde{y}_0|$.

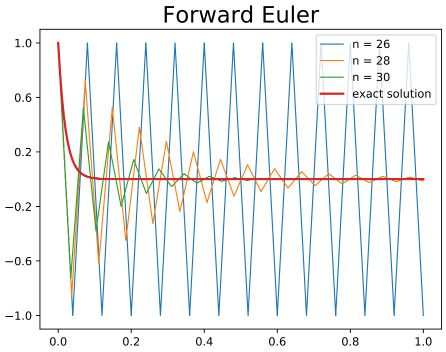
Stiff ODEs

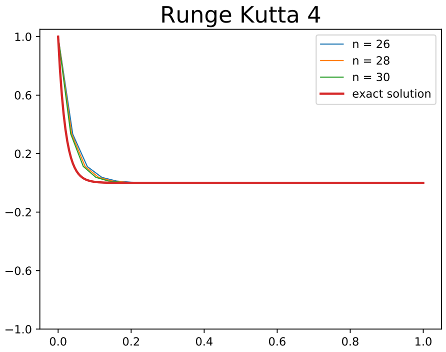
Some ODE terminology.

- A stiff ODE is one which is hard to solve. Why?
  - Because it requires a very small step size, h.
  - Or because it is prone to numerical instabilities.
- This usually happens when equation includes some terms that can lead to rapid variation in the solution.
- Not all methods are equally suited for stiff ODEs.
- Implicit, versus explicit, methods tend to be better for stiff problems.

Stiff ODE

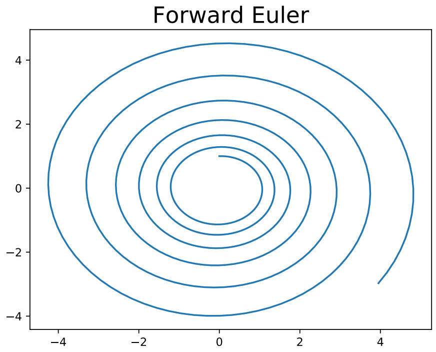
$$y'(x) = -50y, \quad y(0) = 1$$

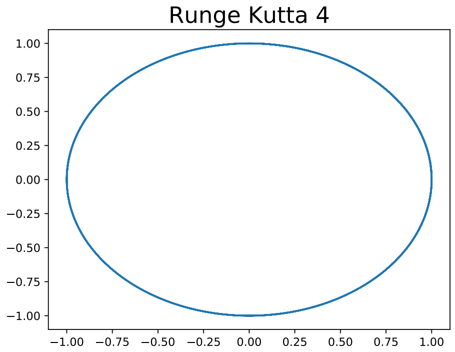




Harmonic Oscillator

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





Truncation Error

Local truncation error measures how well the method approximates the behaviour of a solution of the ODE over one step. Euler method gives us an approximation

$$y_{n+1} = y_n + hf(x_n, y_n)$$

Let $y(x)$ be the exact solution. Using the Taylor series we get

$$y(x_n + h) = y(x_n) + hy'(x_n) + O(h^2)$$

From here we get

$$LocalError = y(x_n + h) - y_n = O(h^2)$$

Truncation Error

Global truncation error is the cumulative error of the local truncation errors committed in each step.

$$GlobalError = N \cdot O(h^2) = \frac{b-a}{h} O(h^2) = O((b-a)h^2/h) = O(h)$$

It shows that the global truncation error is (approximately) proportional to h. For this reason, the Euler method is said to be *first order*.

The RK4 method is a fourth-order method, meaning that the *local* truncation error is of the order of $O(h^5)$, while the *total* accumulated error is of the order of $O(h^4)$.

Numerical Libraries

There are many different approaches for numerical integration and solving ODEs.

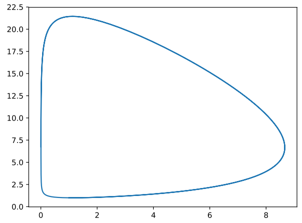
- Good schemes are implemented in packages such as `scipy.integrate.odeint`, `scipy.integrate.ode`.
- The `odeint` package uses an Adams integrator for non-stiff problems, and a backwards differentiation method for stiff problem.
- The `ode` package is a bit more flexible.

Lotka-Volterra equations

Predator–prey equations in which two species interact, one as a predator and the other as prey.

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# params (alpha, beta,  gamma, delta)
params = (0.1,   0.015, 0.0225, 0.02)
def system(z, t, alpha, beta, gamma, delta):
    x = z[0]; y = z[1]
    dxdt = x*(alpha - beta*y)
    dydt = -y*(gamma - delta*x)
    return [dxdt, dydt]
t = np.linspace(0, 300, 1000)
sol = odeint(system, [1.0, 1.0], t, args=params)
plt.plot(sol[:,0], sol[:,1])
plt.show()
```

$$\frac{dx}{dt} = x(\alpha - \beta y)$$
$$\frac{dy}{dt} = -y(\gamma - \delta x)$$



Infectious disease spreading

Consider a population consisting of two groups;

- the susceptibles (S), who can catch the disease
- the infectives (I), who have the disease and can transmit it.

$$\frac{dS}{dt} = -rSI$$
$$\frac{dI}{dt} = rSI - aI$$

where r is the infection rate and a is the recovery rate, and the initial conditions, $S(0)$ and $I(0)$, are known.

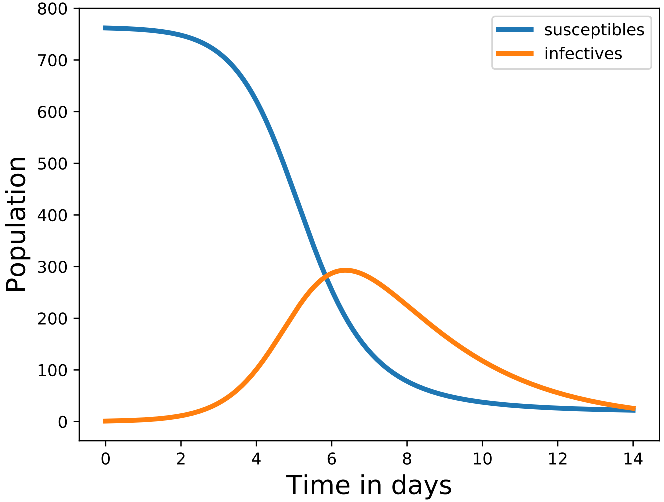
Let's apply this to real data:

- An influenza epidemic, which hit a British boarding school of 763 boys.
- The data were taken from the British Medical Journal (4 March 1978).
- The epidemic lasted 14 days.
- Let $S(0) = 762$ and $I(0) = 1$.
- Let $r = 0.00218$ per day, $a = 0.44$ per day.

Infectious disease spreading

```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
r = 0.00218; a = 0.44
def system(z, t, r, a):
    S = z[0]; I = z[1]
    dSdt = -r * S * I; dIdt = r * S * I - a * I
    return [dSdt, dIdt]
t = np.linspace(0, 14, 1000)
sol = odeint(system, [762, 1], t, args=(r, a))
plt.plot(t, sol[:,0], label='susceptibles', lw=3)
plt.plot(t, sol[:,1], label='infectives', lw=3)
plt.xlabel('Time in days', fontsize = 16)
plt.ylabel('Population', fontsize = 16)
plt.legend()
plt.show()
```

Infectious disease spreading


Population
Time in days
susceptibles
infectives