Computer Science II
L2: ODE

Marco S. Nobile, Ph.D.

Bachelor's Degree in Engineering Physics

Ca' Foscari University of Venice

A.Y. 2022-2023

Today

- Implementare reazione autocatalitica di Robertson (stiff!)
  - Simulare mediante RK45, LSODA, Radau, BDF
  - Misurare il tempo di calcolo
  - Confrontare le performance di diversi metodi di integrazione

- Implementazione dello Jacobiano per metodi impliciti e adattivi

Preliminari

- Il modulo `time()` espone alcune funzionalità per **manipolare tempo** e date
- Possiamo utilizzarlo per **misurare il tempo di calcolo** dei nostri algoritmi
- Esempio:

```
from time import time
...
start = time()
# do something...
end = time()
print ("Running time: %.3f s" % end-start)
```

Metodi di integrazione

- Sfrutteremo la libreria `SciPy.integrate` già vista a lezione
  - Ricordate che possiamo passare l'argomento `"method"` alla classe `solve_ivp()`
  - Possiamo altresì indicare la **tolleranza di errore** (assoluto) mediante l'argomento facoltativo `atol`

- Alla fine dell'integrazione, l'oggetto `solve_ivp()` contiene diverse informazioni:
  - I tempi nell'attributo .t
  - Gli stati del sistema nel vettore `.y`
  - Il numero di step eseguiti nell'attributo `.nfev`

Robertson's autocatalytic chemical reaction

- Implement the following (stiff!) system of coupled ODEs:

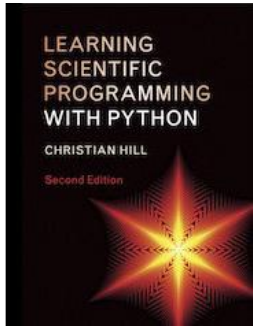
A classic example of a stiff system of ODEs is the kinetic analysis of Robertson's autocatalytic chemical reaction: H. H. Robertson, *The solution of a set of reaction rate equations*, in J. Walsh (Ed.), *Numerical Analysis: An Introduction*, pp. 178–182, Academic Press, London (1966).

The reaction involves three species, $x = [\mathrm{X}]$, $y = [\mathrm{Y}]$ and $z = [\mathrm{Z}]$ with initial conditions $x = 1, y = z = 0$:

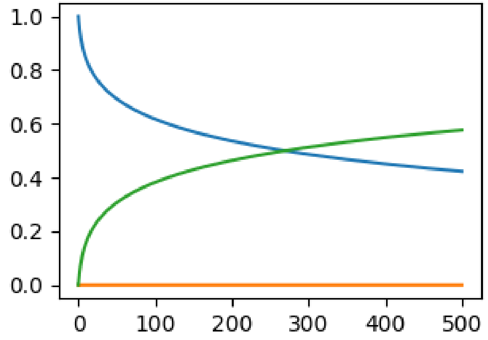
$$\dot{x} \equiv \frac{\mathrm{d}x}{\mathrm{d}t} = -0.04x + 10^4 yz,$$
$$\dot{y} \equiv \frac{\mathrm{d}y}{\mathrm{d}t} = 0.04x - 10^4 yz - 3 \times 10^7 y^2,$$
$$\dot{z} \equiv \frac{\mathrm{d}z}{\mathrm{d}t} = 3 \times 10^7 y^2.$$

(Note the very different timescales of the reactions, particularly for $[\mathrm{Y}]$.)

- Example taken from:



Should look like this:



Compare four different algorithms

- Create a four-panel figure with matplotlib
- In each panel, plot the result of the integration using the following methods
    1. RK45 (aka dopri)
    2. Radau (backward Runge-Kutta 5th order)
    3. LSODA
    4. BDF (variable-order implicit method)
- Compare the number of steps performed by each algorithm
- Compare the actual running time of each algorithm
- Which one is faster? Can you guess why?
- Check the impact of different error tolerances (atol)

Jacobian matrix

- Try to implement a **Jacobian matrix as a function** and pass it to the algorithms (use the optional argument `jac`)
  - The returned Jacobian matrix must have size (n, n) where n is the number of variables/ODEs
  - The function implementing the Jacobian is called as `jac(t, y)`
  - The element in position (i, j) corresponds to $\frac{\partial f_i}{\partial y_j}$

- You can either calculate the derivatives **by hand or by using SymPy** (see code snippet in the next slide)

- Try to simulate now. What happens?

```
def precalc_Jacobian():
    x = Symbol("x")
    y = Symbol("y")
    z = Symbol("z")
    mat_func = Matrix([
        -0.04 * x + 1.e4 * y * z,
        0.04 * x - 1.e4 * y * z - 3.e7 * y**2,
        3.e7 * y**2
       ])
    mat_vars = Matrix([x,y,z])
    return mat_func.jacobian(mat_vars)
```

ODEs

Variables for partial derivatives

```
precalculated_Jacobian = precalc_Jacobian()

def Jacobian(t,Y):
    x,y,z=Y
    n = int(sqrt(len(precalculated_Jacobian)))
    result = zeros(len(precalculated_Jacobian))
    for i, el in enumerate(precalculated_Jacobian):
        result[i]=eval(str(el))
    return result.reshape((n,n))
```

Implementation of Jacobian as function