Chapter 23

Adaptive Methods and Stiff Systems

Chapter Objectives

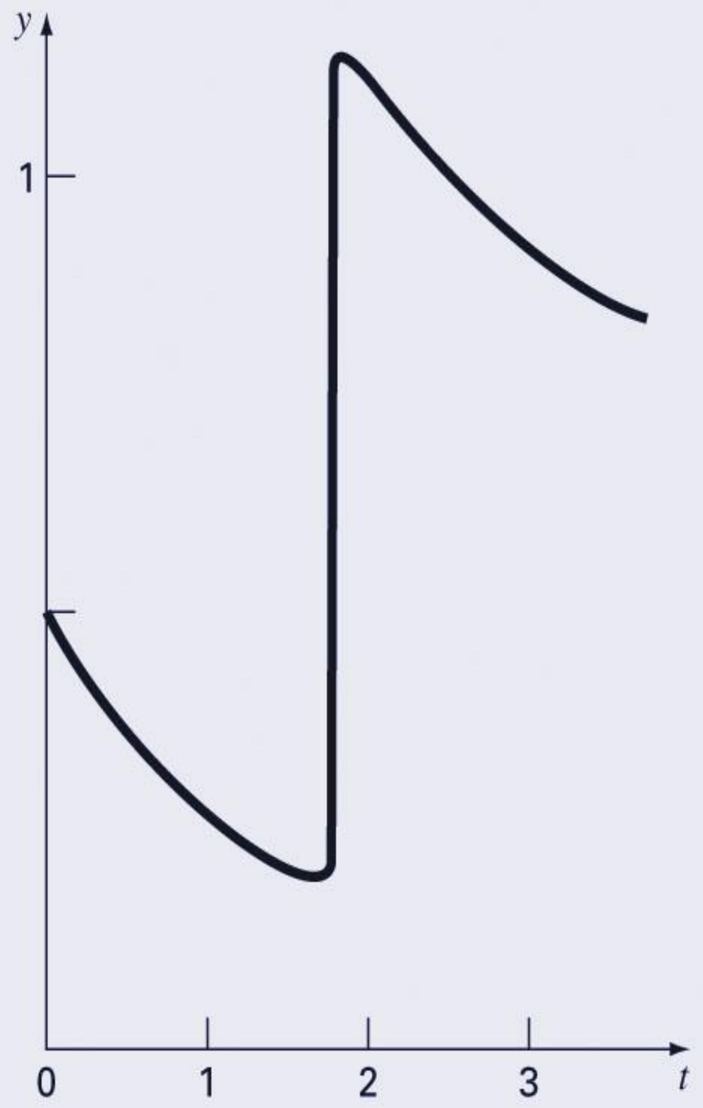
- Familiarizing yourself with the built-in Python functions for solving ODE available from the SciPy module’s integrate submodule.
- Learning how to adjust the options of the Python solve_ivp function.
- Understanding how to pass parameters to the solve_ivp function required by the function that evaluates the derivatives.
- Understanding what is meant by stiffness and its implications.

Adaptive Runge-Kutta Methods

- The solutions to some ODE problems exhibit multiple time scales - for some parts of the solution the variable changes slowly, while for others there are abrupt changes.

- Constant step-size algorithms would have to apply a small step-size to the entire computation, wasting many more calculations on regions of gradual change.

- Adaptive algorithms, on the other hand, can change step-size depending on the region.



Approaches to Adaptive Methods

- There are two primary approaches to incorporate adaptive step-size control:
  - *Step halving* - perform the one-step algorithm in two different ways, once with a full step and once with two half-steps, and compare the results.
  - *Embedded RK methods* - perform two RK iterations of different orders and compare the results. This is the preferred method.

Python Functions

- Python's solve_ivp function from the scipy.integrate module provides methods for solving ordinary differential equations (ODEs) with adaptive step sizes.
- The RK23 method uses second- and third-order Runge-Kutta functions to solve the ODE and adjust step sizes.
- The RK45 method uses fourth- and fifth-order Runge-Kutta functions to solve the ODE and adjust step sizes. This is recommended as the first method to use to solve a problem.

Using `ode` Functions

- The functions are generally called in the same way; ode45 is used as an example:

```
result =
solve_ivp(dydt,(ti,tf),y0,method='RK45',t_eval=tspan)
```

  - **`result.y`**: solution array, where each column represents one of the variables and each row corresponds to a time in the t vector
  - **`dydt`**: function returning a column vector of the right-hand-sides of the ODEs
  - **`tspan`**: time over which to solve the system
  - **`ti,tf`** are the initial and final values of the independent variable
  - **`y0`**: vector of initial values

Example - Predator-Prey

Solve

$$\frac{dy_1}{dt} = 1.2y_1 - 0.6y_1y_2$$

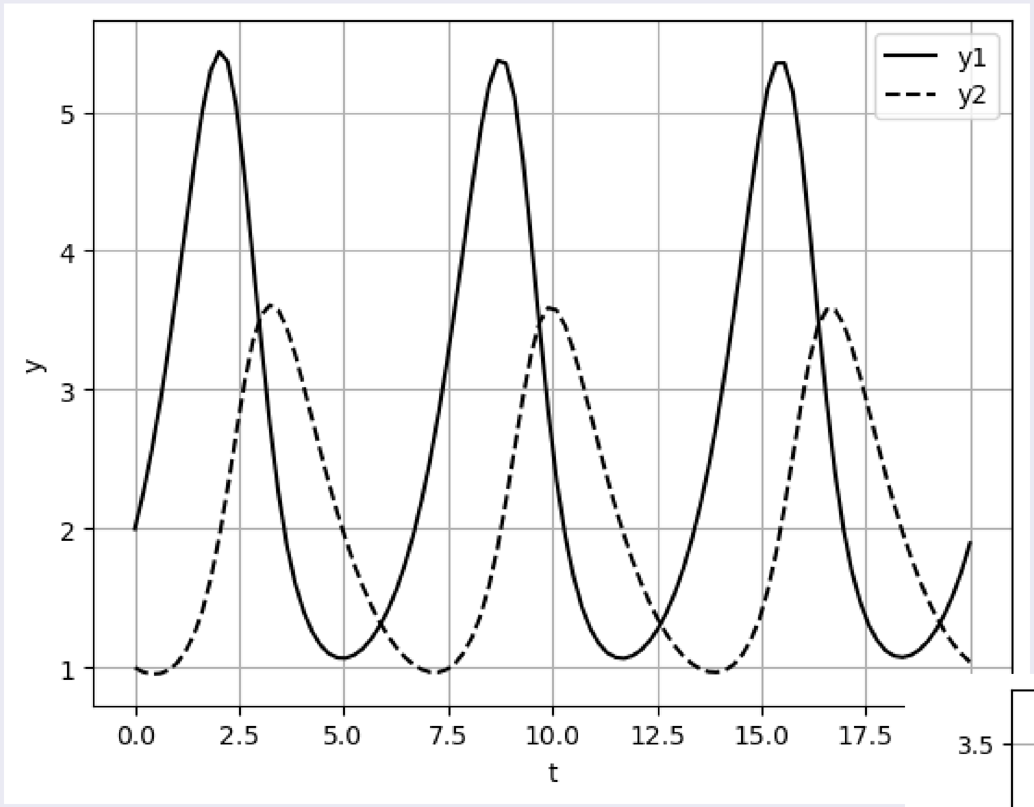
$$\frac{dy_2}{dt} = 0.8y_2 + 0.3y_1y_2$$

with $y_1(0)=2$ and $y_2(0)=1$ for 20 seconds.

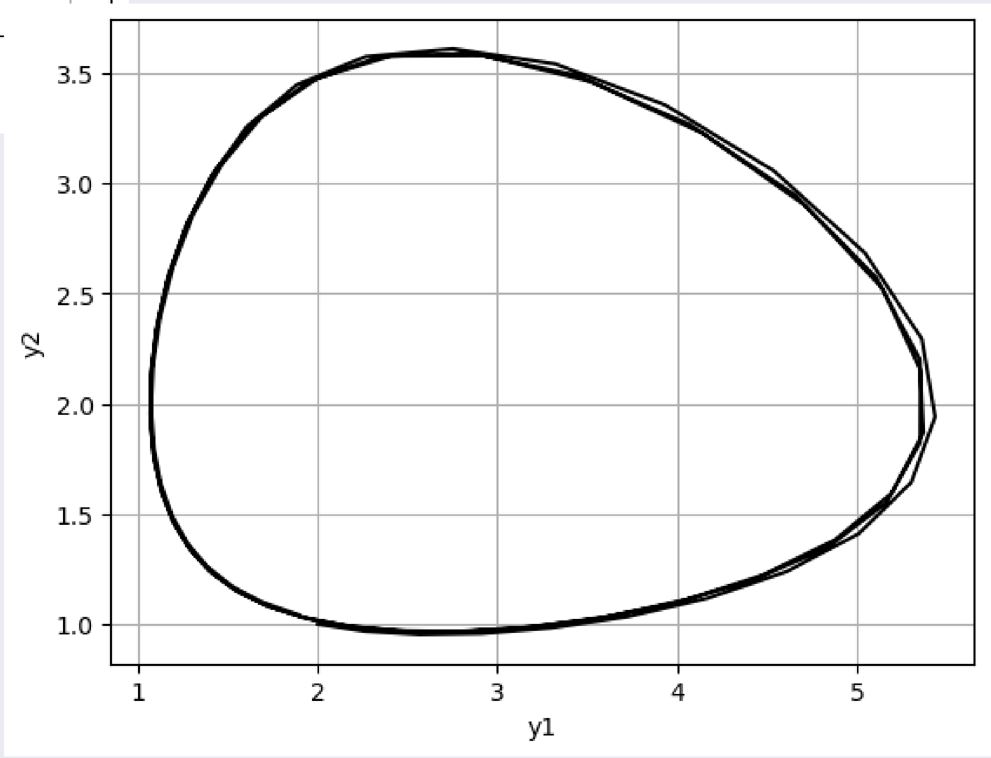
Plot in figure (1) $y_1 \; vs. \; t$ and $y_2 \; vs. \; t$
and in figure(2) $y_2 \; vs. \; y_1$

```
 def dydt(t,y):
     n = len(y)
     dy = np.zeros((n))
     dy[0] = 1.2*y[0] - 0.6*y[0]*y[1]
     dy[1] = -0.8*y[1] + 0.3*y[0]*y[1]
     return dy
```

```
import numpy as np
from scipy.integrate import solve_ivp
import pylab
ti = 0. ; tf = 20.
y0 = np.array([2.,1.])
tspan = np.linspace(ti,tf,100)
result = solve_ivp(dydt,(ti,tf),y0,t_eval=tspan)
t = result.t
y = result.y
pylab.plot(t,y[0,:],c='k',label='y1')
pylab.plot(t,y[1,:],c='k',ls='--',label='y2')
pylab.grid()
pylab.xlabel('t')
pylab.ylabel('y')
pylab.legend()
pylab.figure()
pylab.plot(y[0,:],y[1,:],c='k')
pylab.grid()
pylab.xlabel('y1')
pylab.ylabel('y2')
```



y1
y2
y
t



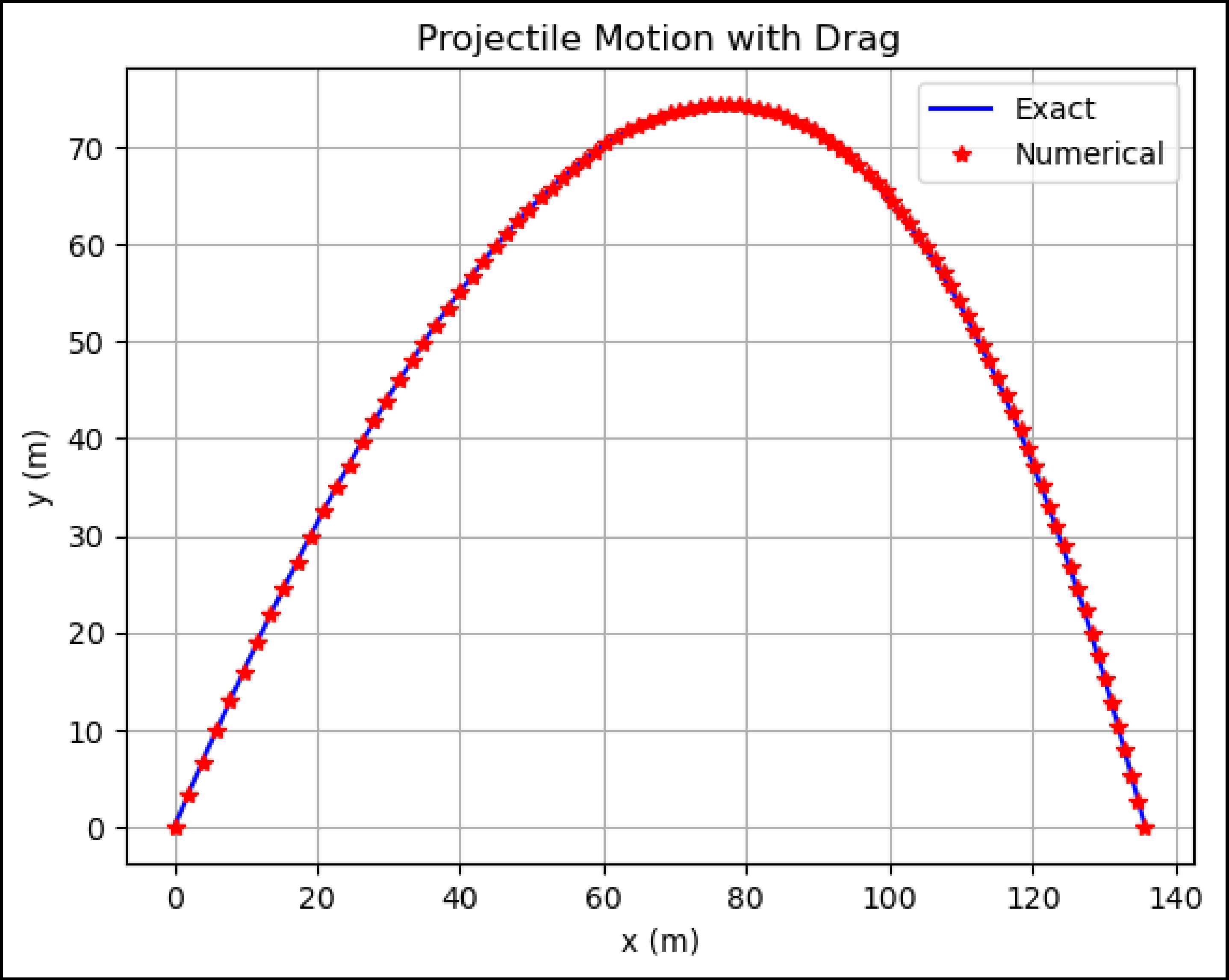
y2
y1

Passing parameters

Note in code projectile_linear how parameters are passed to the `solve_ivp` function

```python
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve
from scipy.interpolate import CubicSpline
def projectile_linear():
    v0 = 50  # Initial velocity (m/s)
    theta = 60  # Launch angle (degrees)
    k = 0.1  # Drag coefficient
    g = 9.8  # Gravitational acceleration (m/s^2)
    p = [k, g]  # Parameters to pass to ODE function
    tspan = (0, 9)  # Time span for simulation
    y0 = [0, v0 * np.cos(np.radians(theta)), 0, v0 * np.sin(np.radians(theta))]
    def f(t, y, k, g):
        dydt = np.zeros_like(y)
        dydt[0] = y[1]
        dydt[1] = -k * y[1]
        dydt[2] = y[3]
        dydt[3] = -g - k * y[3]
        return dydt
    sol = solve_ivp(f, tspan, y0, args=(k, g), dense_output=True)
    # Exact equations for projectile motion with drag
    xe = lambda t: v0 * np.cos(np.radians(theta)) * (1 - np.exp(-k * t)) / k
    ye = lambda t: -g * t / k + (v0 * np.sin(np.radians(theta)) + g / k) * (1 - np.exp(-k * t)) / k
    # To find the range (time when projectile hits the ground)
    def incsearch(func, a, b, step=0.1):
        t_vals = np.arange(a, b, step)
        for i in range(len(t_vals) - 1):
            if func(t_vals[i]) * func(t_vals[i + 1]) < 0:
                return (t_vals[i], t_vals[i + 1])
        return None

    t0 = incsearch(ye, 0.1, tspan[1])
    tg = fsolve(ye, t0[0])[0]  # Time to hit the ground (exact)
    Re = xe(tg)  # Exact range
    # Numerical results using cubic spline interpolation
    xs = CubicSpline(sol.t, sol.y[0, :])
    ys = CubicSpline(sol.t, sol.y[2, :])
    t0n = incsearch(lambda t: ys(t), 0.1, tspan[1])
    tgn = fsolve(lambda t: ys(t), t0n[0])[0]  # Time to hit the ground (numerical)
    Rn = xs(tgn)  # Range from numerical results
    # Time points for plotting
    T = np.linspace(0, tg, 100)
    # Plotting the results
    plt.plot(xe(T), ye(T), '-b', label='Exact')
    plt.plot(xs(T), ys(T), '*r', label='Numerical')
    plt.grid()
    plt.xlabel('x (m)')
    plt.ylabel('y (m)')
    plt.legend()
    plt.title('Projectile Motion with Drag')
    plt.show()
    print(f"Exact time to hit ground: {tg:.2f} s")
    print(f"Exact range: {Re:.2f} m")
    print(f"Numerical time to hit ground: {tgn:.2f} s")
    print(f"Numerical range: {Rn:.2f} m")
projectile_linear()
```



```
Exact time to hit ground: 7.83 s
Exact range: 135.70 m
Numerical time to hit ground: 7.83 s
Numerical range: 135.75 m
```

Stiffness

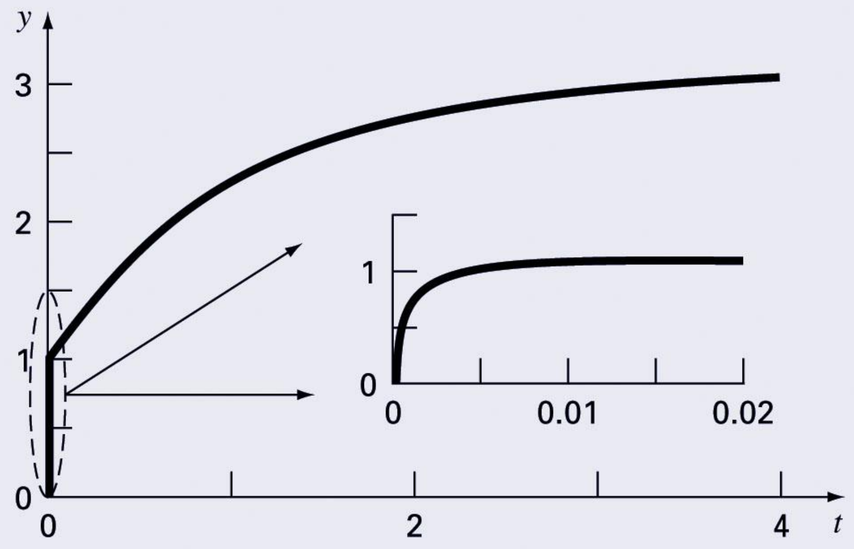
A *stiff system* is one involving rapidly changing components together with slowly changing ones.

An example of a single stiff ODE is:

$$\frac{dy}{dt} = -1000y + 3000 - 2000e^{-t}$$

whose solution if y(0)=0 is:

$$y = 3 - 0.998e^{-1000t} - 2.002e^{-t}$$



Python ODE Solvers for Stiff Systems

- Python via the SciPy integrate module that are effective for solving stiff systems of ODEs, including: **solve_ivp, LSODA, odeint, ode.**

- Using the first two of these are recommended because they represent the most current versions.

Example

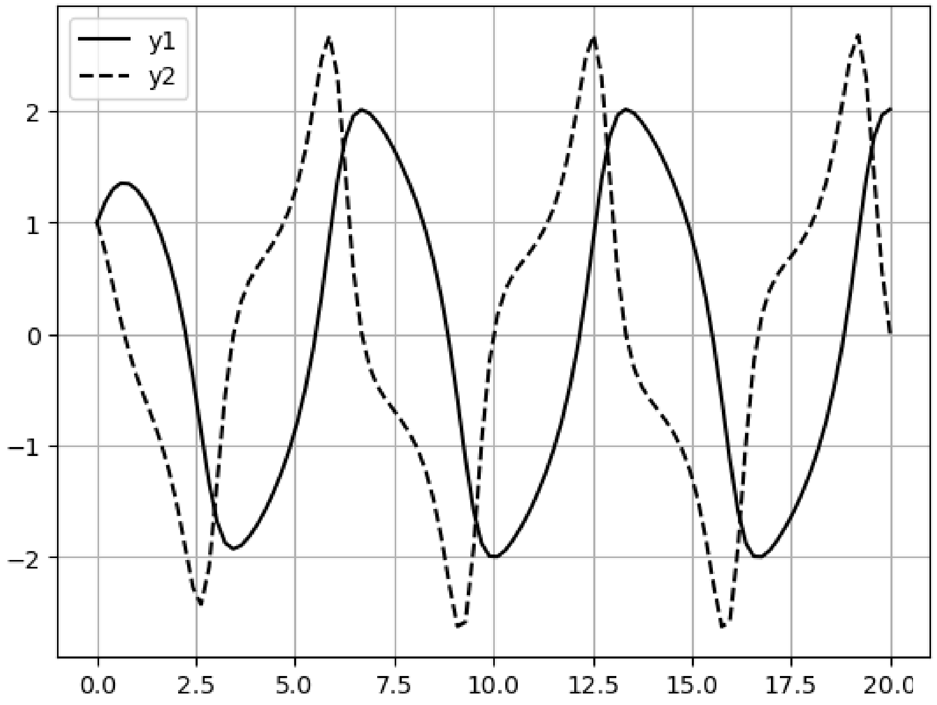
Compare the solutions of the stiff equation

$$\frac{d^2y}{dt^2} = 100(1 - y^2)\frac{dy}{dt} - y,$$
$$y(0) = 0, y'(0) = 1$$

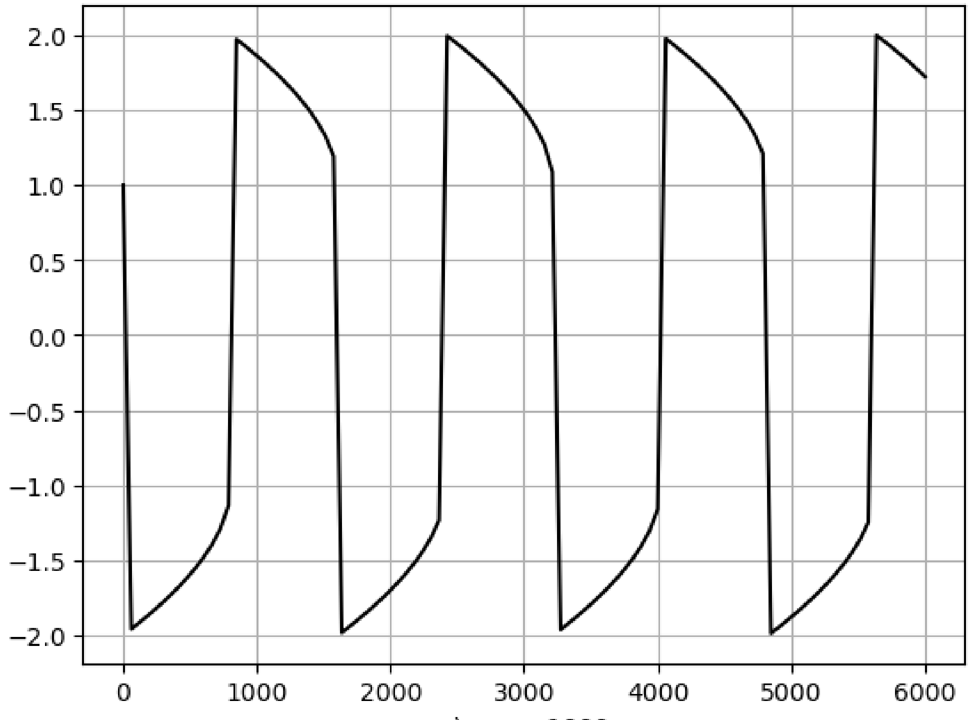
- Take time interval from 0 to 6000

```
  def dydt(t,y,mu):
      dy = np.zeros((2))
      dy[0] = y[1]
      dy[1] = mu*(1-y[0]**2)*y[1]-y[0]
      return dy

mu = 1.
ti = 0. ; tf = 20.
tspan = np.linspace(ti,tf,100)
y0 = np.array([1.,1.])
soln = solve_ivp(dydt,(ti,tf),y0,t_eval=tspan,args=(mu,))
# method RK45 is the default
t = soln.t
y = soln.y
pylab.plot(t,y[0,:],c='k',label='y1')
pylab.plot(t,y[1,:],c='k',ls='--',label='y2')
pylab.grid()
pylab.xlabel('a) mu = 1')
pylab.legend()
mu = 1000.
ti = 0. ; tf = 6000.
tspan = np.linspace(ti,tf,100)
y0 = np.array([1.,1.])
soln = solve_ivp(dydt,(ti,tf),y0,t_eval=tspan,method='LSODA',args=(mu,))
t = soln.t
y = soln.y
pylab.figure()
pylab.plot(t,y[0,:],c='k')
pylab.grid()
pylab.xlabel('a) mu = 1000')
```



a) mu = 1



a) mu = 1000