Modular ODE Solvers

Modular Solver for a Single, 1st Order ODE

Euler Integration Scheme

This code was presented in the previous chapter. It performs an Euler integration of the exponential growth equation $dy/dt = ay$.

pros: This example is simple, linear and easy to understand.

cons: This approach works less well for more complex ODEs with higher-order integration schemes.

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

#########  Parameters  #########

a    = -0.2          # decay constant
tmax = 100           # maximum time
dt   = 1             # time step
y0   = 1             # initial value of y

#########  Create Arrays  #########

N = int(tmax/dt)+1   # number of steps
y = np.zeros(N)      # array to store y values
t = np.zeros(N)      # array to store times

y[0] = y0            # assign initial value


#########  Euler Integration  #########

for n in range(N-1):
    f = a*y[n]              # derivative
    y[n+1] = y[n] + f*dt    # Euler rule
    t[n+1] = t[n] + dt
```

exponential growth derivative → `f = a*y[n]`

Break Code into Functions

Functions make your code modular and easy to modify.

Euler Function: Perform the numerical integration for a given ODE and return the solution $y(t)$

Derivative Function: Calculate the derivative dy/dt given the model parameters.

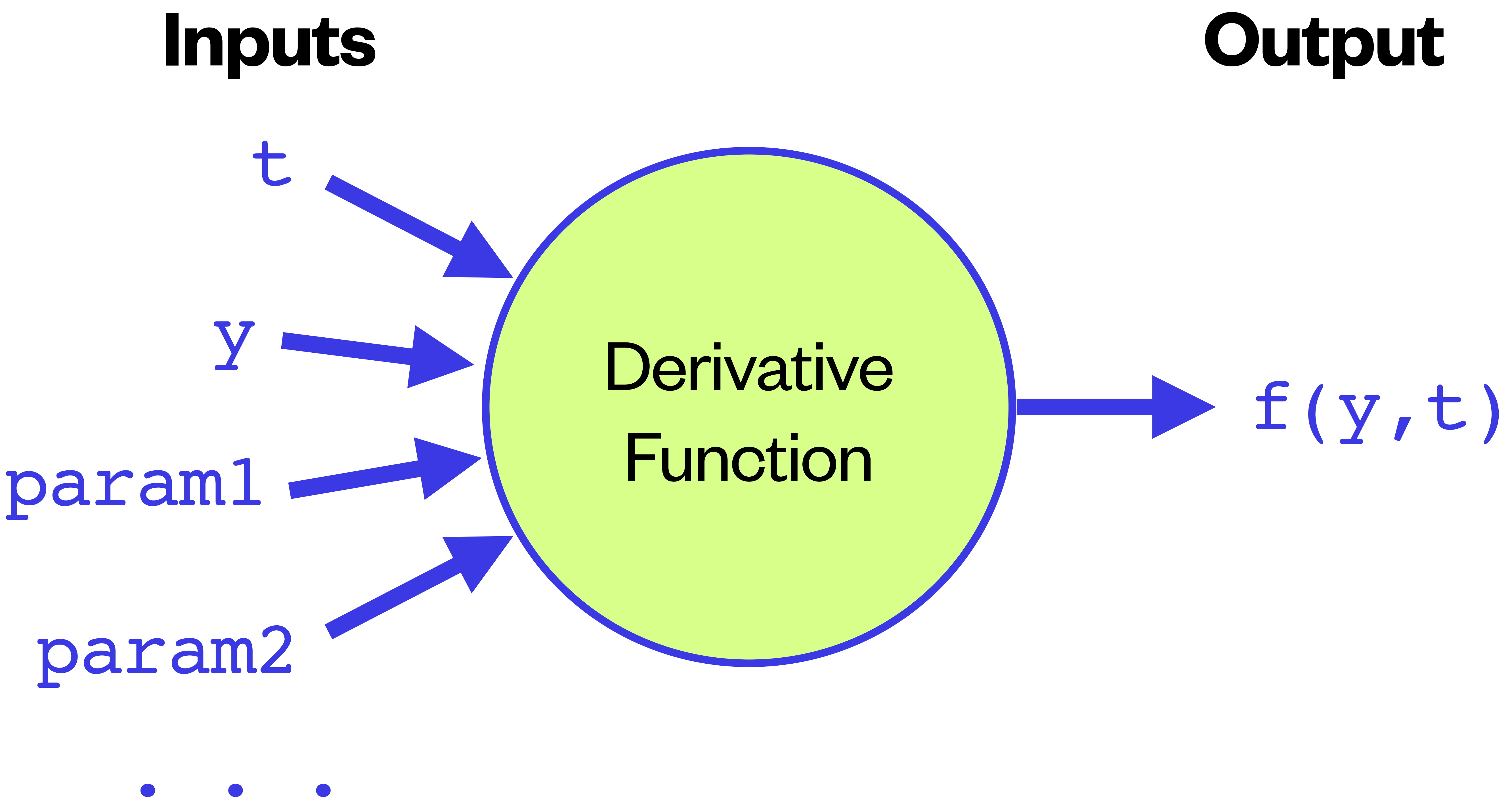
```
import numpy as np
import matplotlib.pyplot as plt

#########  Parameters  #########

a    = -0.2            # decay constant
tmax = 100             # maximum time
dt   = 1               # time step
y0   = 1               # initial value of y

#########  Create Arrays  #########

N = int(tmax/dt)+1     # number of steps
y = np.zeros(N)        # array to store y values
t = np.zeros(N)        # array to store times

y[0] = y0              # assign initial value


#########  Euler Integration  #########

for n in range(N-1):
    f = a*y[n]               # derivative
    y[n+1] = y[n] + f*dt     # Euler rule
    t[n+1] = t[n] + dt
```

Derivative function

Calculates and returns the derivative $f(y, t)$ for first-order ODE:

$$\frac{dy}{dt} = f(y, t)$$

- Passed parameters:
  - t = time
  - y = dependent variable
  - param1 = parameter
  - param2 = another parameter
- Returned value:
  - derivative dy/dt



Example: Exponential Growth Function

Derivative function

Calculates and returns the derivative $f(y, t)$ for the first-order ODE:

$$\frac{dy}{dt} = ay = f(y, t)$$

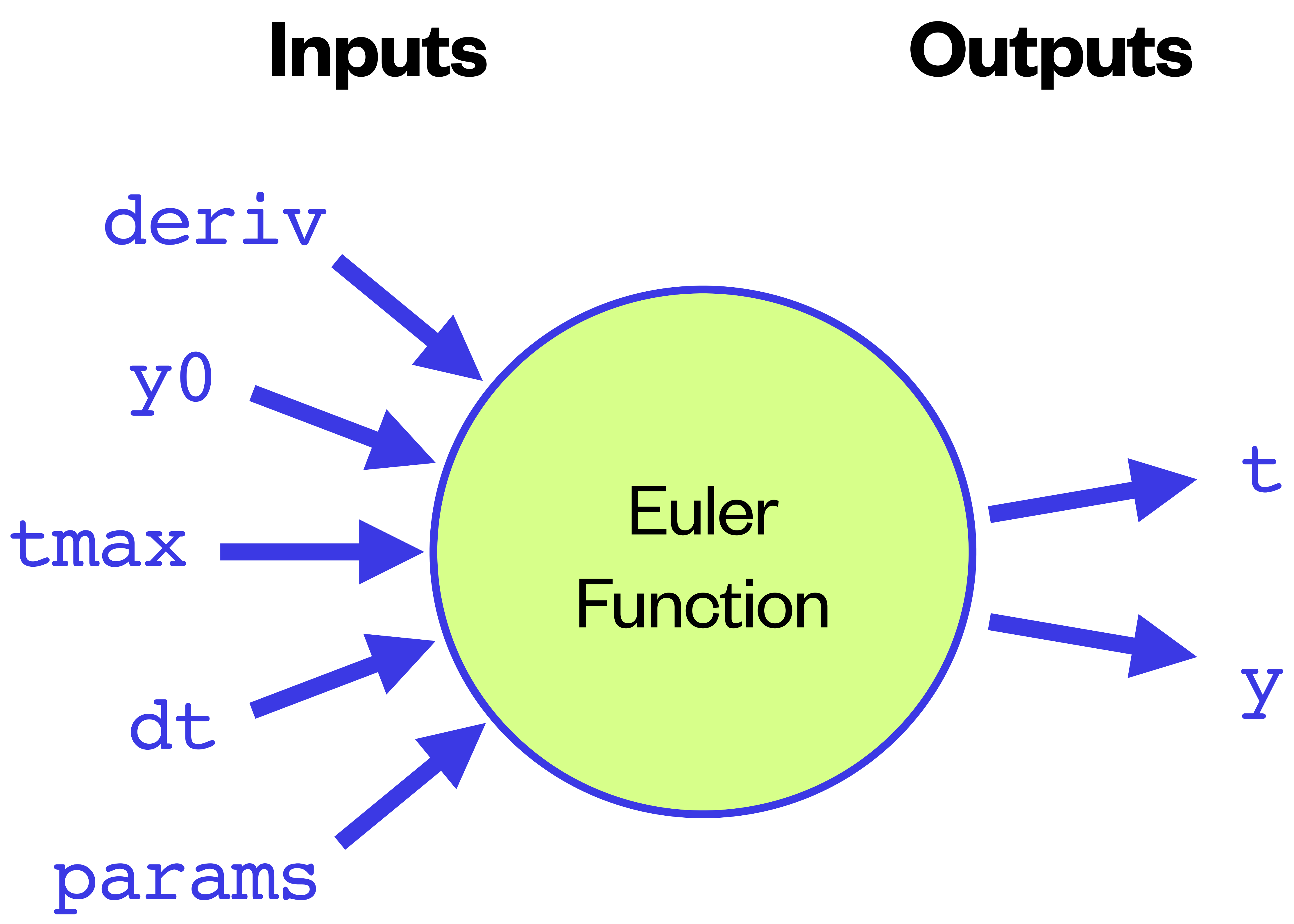
```
##########  Derivative Function  ##########

def deriv_exp(t, y, a):

    dydt = a*y
    return dydt
```

- Passed parameters:
  - `t` = time
  - `y` = dependent variable
  - `a` = growth rate parameter
- Returned value:
  - derivative dy/dt

Euler function

Performs the numerical integration using Euler's method and a derivative function.

- Passed parameters:
  - `deriv` = derivative function
  - `y0` = initial condition
  - `tmax` = maximum time
  - `dt` = time step
  - `params` = array of parameters
- Returned value:
  - `t` = array of times
  - `y` = array containing solution



Euler function

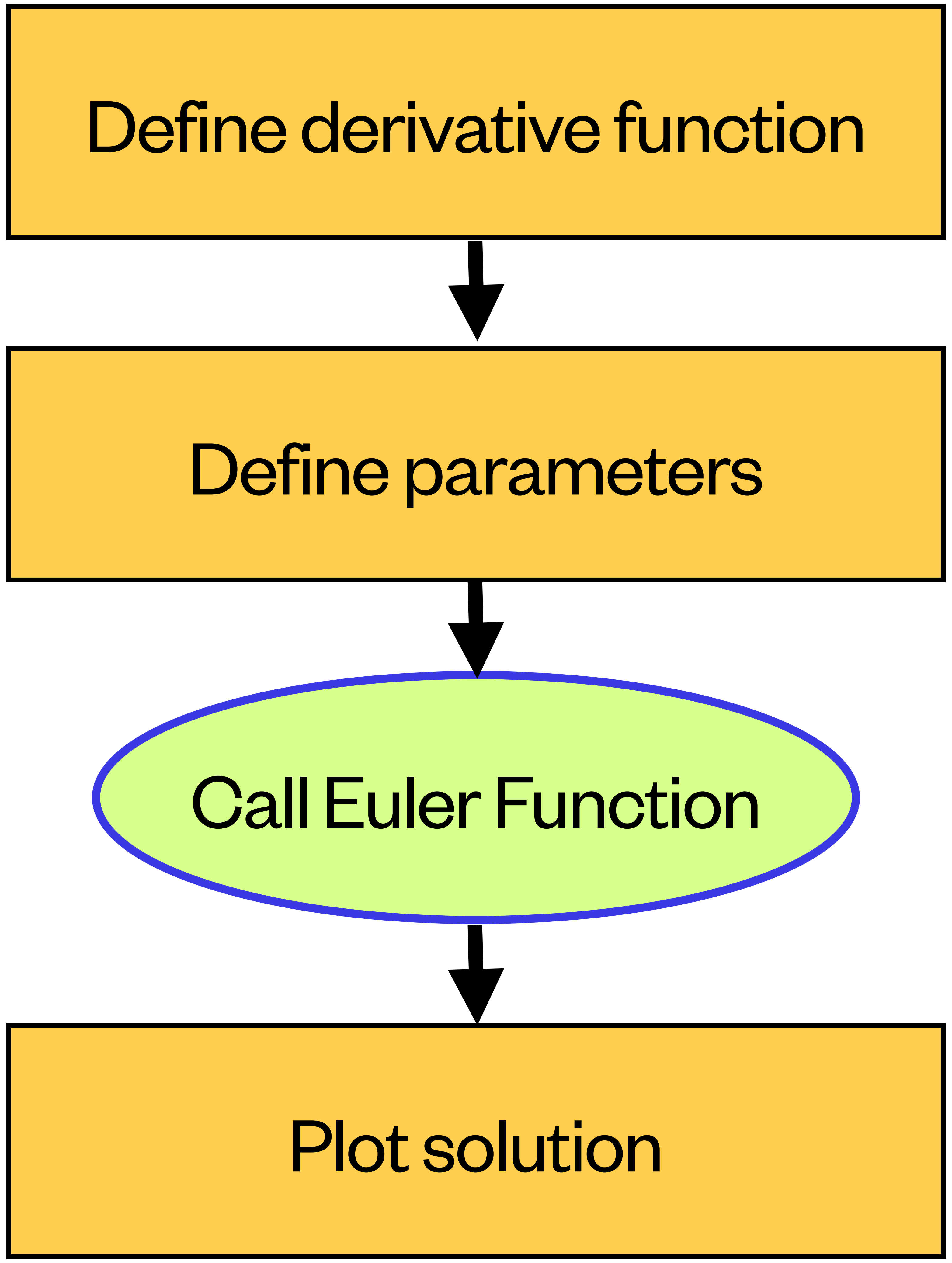
Performs the numerical integration using Euler's method and a derivative function.

- Passed parameters:
  - `deriv` = derivative function
  - `y0` = initial condition
  - `tmax` = maximum time
  - `dt` = time step
  - `params` = array of parameters
- Returned value:
  - `t` = array of times
  - `y` = array containing solution

```
#########  Euler Integration  #########

def Euler(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    N = int(tmax/dt)+1     # number of steps in simulation
    y = np.zeros(N)        # array to store y values
    t = np.zeros(N)        # array to store times

    y[0] = y0              # assign initial value

    #########  Loop to implement the Euler update rule  ####

    for n in range(N-1):
        f = deriv(t[n], y[n], *params)  # use "*" to unpack
        y[n+1] = y[n] + f*dt
        t[n+1] = t[n] + dt

    return t, y
```

don't forget the *

Put it all together: Derivative and Euler Functions in action



Put it all together: Derivative and Euler Functions in action

call to **Euler()** function

get solution (**t** and **y**)

pass **deriv_exp()** function defining the ODE to integrate

```
#########  Parameters  #########

a    = 0.2          # decay constant
tmax = 100          # maximum time
dt   = 0.5          # time step
y0   = 1            # initial value of y

params = [a]        # bundle parameters in array

#########  Perform Euler Integration  #########

t, y =  Euler(deriv_exp, y0, tmax, dt, params)

#########  Plot Solution  #########

plt.plot(t, y, label='Euler')
```

Summary

The modular approach to numerical integration code has the following advantages:

- The **`Euler()`** function can be used to solve **any** first-order ODE using the Euler method. This function does not need to be changed when a new ODE is solved.
- The **`derivs_exp()`** function contains all the information about the ODE being solved. It can be used with a different numerical integration solver (e.g. midpoint, Runga-Kutta, etc.)

Modular Solver for a System of 1st Order ODEs

All 2nd Order ODEs Can be Written as a System of Two, 1st-Order ODEs

For example, we can write $F = ma$ as a system of two 1st order ODEs:

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

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

Writing a System of ODEs as a Generalized Vector Equation

We introduce this approach through an example. Let's solve the simple harmonic oscillator problem:

$$F = -kx \longrightarrow \frac{d^2x}{dt^2} = -\frac{k}{m}x$$

We write this 2nd-order ODE as a system of coupled 1st-order ODEs:

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = -(k/m)x$$

We want to solve for the variables $x(t)$ and $v(t)$.

Writing a System of ODEs as a Generalized Vector Equation

Introduce a generalized vector $\vec{y}$ whose components are x and v, where $y^{(0)}(t) = x(t)$ and $y^{(1)}(t) = v(t)$, i.e.

$$\vec{y} = \begin{pmatrix} y^{(0)}(t) \\ y^{(1)}(t) \end{pmatrix} = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}.$$

Our system of coupled 1st-order ODEs

$$\frac{dx}{dt} = v \qquad \frac{dv}{dt} = -(k/m)x$$

may be written in terns of y_0 and y_1 as

$$\frac{dy^{(0)}}{dt} = y^{(1)} \qquad \frac{dy^{(1)}}{dt} = -(k/m)y^{(0)}.$$

Writing a System of ODEs as a Generalized Vector Equation

The introduction of the generalized vector $\vec{y} = (y^{(0)}, y^{(1)})$ allows us to write our system of ODEs as a single differential equation:

$$\frac{d\vec{y}}{dt} = \vec{a}(\vec{y}, t)$$

where

$$\vec{y} = \begin{pmatrix} y^{(0)} \\ y^{(1)} \end{pmatrix} \quad \text{and} \quad \vec{a}(\vec{y}, t) = \begin{pmatrix} y^{(1)} \\ -(k/m)y^{(0)} \end{pmatrix}.$$

We can solve this ODE using Euler or any other method.

Writing a System of ODEs as a Generalized Vector Equation

Applying the Euler method to solve this system gives

$$\vec{y}_{n+1} = \vec{y}_n + \vec{a}_n \Delta t$$

This is amazing! We can solve a system of potentially hundreds of ODEs using a single Euler update equation!

In component form, this is equivalent to:

$$\begin{pmatrix} y^{(0)}_{n+1} \\ y^{(1)}_{n+1} \end{pmatrix} = \begin{pmatrix} y^{(0)}_{n} \\ y^{(1)}_{n} \end{pmatrix} + \begin{pmatrix} y^{(1)}_{n} \\ -(k/m) y^{(0)}_{n} \end{pmatrix} \Delta t$$

or, n terms of x and v

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} x_{n} \\ v_{n} \end{pmatrix} + \begin{pmatrix} v_{n} \\ -(k/m) x_{n} \end{pmatrix} \Delta t$$

Arrays used in the Multi-Variable Code

Initial conditions (1x2 array): `y0 =`

x0	v0

Solution (Nx2 array): `y =`

x[0]	v[0]
x[1]	v[1]
x[2]	v[2]
x[3]	v[3]
x[4]	v[4]

```
x = y[:,0]
v = y[:,1]
```

`derivs_sh0()` returns 1x2 array:

dxdt	dvdt

Multi-Variable Derivative Function

y is a 1x2 array containing x and v

y =

x	v

derivs_sho returns a 1x2 array containing derivatives dx/dt and dv/dt:

dxdt	dydt

```
#########  Derivative Function  #########
#
# This function returns the derivatives for the
# Simple Harmonic Oscillator ODE

def deriv_sho(t, y, m, k):

    # extract variables from y array
    x = y[0]              # position
    v = y[1]              # velocity

    # calculate derivatives
    dxdt = v
    dvdt = -k/m*x

    # return derivatives in a numpy array
    return np.array([dxdt, dvdt])
```

Multi-Variable Euler Function

This line determines the number of variables in the system by checking to see if the initial conditions variable `y0` is a float or an array.

If `y0` is a float, there's only a single variable.

If `y0` is a NumPy array, the number of variables = the number of elements in `y0`.

```
#########  Multi-Variable Euler Integration  #########

def Euler_Vec(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    # determine the number of variables in the system from initial
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1          # number of steps in simulation
    y = np.zeros((N,nvar))      # array to store y values
    t = np.zeros(N)             # array to store times

    if nvar == 1:
        y[0] = y0               # assign initial value if single var
    else:
        y[0,:] = y0             # assign vector initial values if mu

    #########  Loop to implement the Euler update rule  #########

    for n in range(N-1):
        f = deriv(t[n], y[n], *params)
        y[n+1] = y[n] + f*dt
        t[n+1] = t[n] + dt

    return t, y
```

Multi-Variable Euler Function

`y` is a (N)×(nvar) array, with the columns storing the solution for each variable.

If `y` is a 2D array, we must use slicing to copy the initial condition array `y0` to the top row of the solution array `y`. If `y` is a 1D array, we just set `y[0]` to the initial value `y0`.

```
#########  Multi-Variable Euler Integration  #########

def Euler_Vec(deriv, y0, tmax, dt, params):

    #########  Create Arrays  #########

    # determine the number of variables in the system from initial
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1          # number of steps in simulation
    y = np.zeros((N,nvar))      # array to store y values
    t = np.zeros(N)             # array to store times

    if nvar == 1:
        y[0] = y0               # assign initial value if single var
    else:
        y[0,:] = y0             # assign vector initial values if mu

    #########  Loop to implement the Euler update rule  #########

    for n in range(N-1):
        f = deriv(t[n], y[n], *params)
        y[n+1] = y[n] + f*dt
        t[n+1] = t[n] + dt

    return t, y
```

Multi-Variable Euler Function

The loop implementing the Euler method for our system of ODEs looks exactly like the loop when we had only a single ODE.

```
######### Multi-Variable Euler Integration #########

def Euler_Vec(deriv, y0, tmax, dt, params):

    ######### Create Arrays #########

    # determine the number of variables in the system from initial
    nvar = 1 if not isinstance(y0, np.ndarray) else y0.size

    N = int(tmax/dt)+1            # number of steps in simulation
    y = np.zeros((N,nvar))        # array to store y values
    t = np.zeros(N)               # array to store times

    if nvar == 1:
        y[0] = y0                 # assign initial value if single var
    else:
        y[0,:] = y0               # assign vector initial values if mu

    ######### Loop to implement the Euler update rule #########

    for n in range(N-1):
        f = deriv(t[n], y[n], *params)
        y[n+1] = y[n] + f*dt
        t[n+1] = t[n] + dt

    return t, y
```

Put it all together: Derivative and Euler Functions in action

Initial conditions are now stored in a 1×2 array

We have to extract the solution for each variable from the returned y array.

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

#########  Parameters  #########

m    = 1          # mass
k    = 1          # spring constant
tmax = 10         # maximum time
dt   = 0.001      # time step
x0   = 1          # initial position
v0   = 0          # initial velocity

params = np.array([m,k])     # bundle parameters together
y0 = np.array([x0,v0])       # bundle initial conditions


#########  Perform Euler Integration  #########

t, y =  Euler_Vec(deriv_sho, y0, tmax, dt, params)

x = y[:,0]        # extract positions
v = y[:,1]        # extract velocities


#########  Plot Solution  #########

plt.plot(t, x, label='x')          # plot position
```