

10.6: Integrating ODEs with Scipy

Except for educational purposes, it is almost always a bad idea to implement your own ODE solver; instead, you should use a pre-written solver.

10.6.1 The `scipy.integrate.odeint` Solver

In Scipy, the simplest ODE solver to use is the `scipy.integrate.odeint` function, which is in the `scipy.integrate` module. This is actually a wrapper around a low-level numerical library known as LSODE (the **L**ivermore **S**olver for **O**DEs"), which is part of a widely-used ODE solver library known as **ODEPACK**. The most important feature of this solver is that it is "adaptive": it can automatically figure out (i) which integration scheme to use (choosing between either a high-order Adams-Moulton method, or another implicit method known as the **B**ackward **D**ifferentiation **F**ormula which we haven't described), and (ii) the size of the discrete time steps, based on the behavior of the solutions as they are being worked out. In other words, the user only needs to specify the derivative function, the initial state, and the desired output times, without having to worry about the internal details of the solution method.

The function takes several inputs, of which the most important ones are:

1. `func` , a function corresponding to the derivative function $\vec{F}(\vec{y}, t)$.
2. `y0` , either a number or 1D array, corresponding to the initial state $\vec{y}(t_0)$.
3. `t` , an array of times at which to output the ODE solution. The first element corresponding to the initial time t_0 . Note that these are the "output" times only—they do *not* specify the actual time steps which the solver uses for finding the solutions; those are automatically determined by the solver.
4. (optional) `args` , a tuple of extra inputs to pass to the derivative function `func` . For example, if `args=(2, 3)` , then `func` should accept four inputs, and it will be passed 2 and 3 as the last two inputs.

The function then returns an array `y` , where `y[n]` contains the solution at time `t[n]` . Note that `y[0]` will be exactly the same as the input `y0` , the initial state which you specified.

Here is an example of using `odeint` to solve the damped harmonic oscillator problem $m\ddot{x} = -\lambda\dot{x} - kx(t)$, using the previously-mentioned vectorization trick to cast it into a first-order ODE:

```
from scipy import *
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def ydot(y, t, m, lamdb, k):
    x, v = y[0], y[1]
    return array([v, -(lamdb/m) * v - k * x / m])

m, lamdb, k = 1.0, 0.1, 1.0 # Oscillator parameters
y0 = array([1.0, 5.0]) # Initial conditions [x, v]
t = linspace(0.0, 50.0, 100) # Output times

y = odeint(ydot, y0, t, args=(m, lamdb, k))

## Plot x versus t
plt.plot(t, y[:,0], 'b-')
plt.xlabel('t')
plt.ylabel('x')
plt.show()
```

There is an important limitation of `odeint` : it **does not handle complex ODEs**, and always assumes that \vec{y} and \vec{F} are real. However, this is not a problem in practice, because you can always convert a complex first-order ODE into a real one, by replacing the complex vectors \vec{y} and \vec{F} with double-length real vectors:

$$\vec{y}' \equiv \begin{bmatrix} \text{Re}(\vec{y}) \\ \text{Im}(\vec{y}) \end{bmatrix}, \quad \vec{F}' \equiv \begin{bmatrix} \text{Re}(\vec{F}) \\ \text{Im}(\vec{F}) \end{bmatrix}. \quad (10.6.1)$$

10.6.2 The `scipy.integrate.ode` Solvers

Apart from `odeint`, Scipy provides a more general interface to a variety of ODE solvers, in the form of the `scipy.integrate.ode` class. This is a much more low-level interface; instead of calling a single function, you have to create an ODE "object", then use the methods of this object to specify the type of ODE solver to use, the initial conditions, etc.; then you have to repeatedly call the ODE object's `integrate` method, to integrate the solution up to each desired output time step.

There is an extremely aggravating inconsistency between the `odeint` function and this `ode` class: the expected order of inputs for the derivative functions are reversed! The `odeint` function assumes the derivative function has the form $F(y, t)$, but the `ode` class assumes it has the form $F(t, y)$. Watch out for this!

Here is an example of using `ode` class with the damped harmonic oscillator problem $m\ddot{x} = -\lambda\dot{x} - kx(t)$, using a Runge-Kutta solver:

```
from scipy import *
import matplotlib.pyplot as plt
from scipy.integrate import ode

## Note the order of inputs (different from odeint)!
def ydot(t, y, m, lambd, k):
    x, v = y[0], y[1]
    return array([v, -(lambd/m) * v - k * x / m])

m, lambd, k = 1.0, 0.1, 1.0    # Oscillator parameters
y0 = array([1.0, 5.0])        # Initial conditions [x, v]
t = linspace(0.0, 50.0, 100) # Output times

## Set up the ODE object
r = ode(ydot)
r.set_integrator('dopri5')    # A Runge-Kutta solver
r.set_initial_value(y0)
r.set_f_params(m, lambd, k)

## Perform the integration. Note that the "integrate" method only integrates
## up to one single final time point, rather than an array of times.
x = zeros(len(t))
x[0] = y0[0]
for n in range(1, len(t)):
    r.integrate(t[n])
    assert r.successful()
    x[n] = (r.y)[0]

## Plot x versus t
```

```
plt.plot(t, x, 'b-')  
plt.xlabel('t')  
plt.ylabel('x')  
plt.show()
```

See the [documentation](#) for a more detailed list of options, including the list of ODE solvers that you can choose from.

This page titled [10.6: Integrating ODEs with Scipy](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.