Chapter 18

Ordinary Differential Equations II

18.1 Systems of equations

Euler's method can be applied to any system of ordinary differential equations (ODEs). Consider the following example:

$$dx/dt = xy - z + t \tag{18.1a}$$
$$dy/dt = 2z + x \tag{18.1b}$$
$$dz/dt = t^2 - yz \tag{18.1c}$$

This set of equations has one independent variable (namely t) and three dependent variables (namely, x, y and z). The goal is to determine the functions $x(t)$, $y(t)$ and $z(t)$.

We can apply the same reasoning as in the previous chapter and replace the differential equations (18.1) with a set of discrete equations. This process is called *discretization*.

Begin by evaluating the differential equations at time t_i, then replace the derivatives such as dx/dt with the two–point forward difference approximation $(x_{i+1} - x_i)/\Delta t$. Rearrange the results to obtain

$$x_{i+1} = x_i + (x_i y_i - z_i + t_i)\Delta t \ , \tag{18.2a}$$
$$y_{i+1} = y_i + (2z_i + x_i)\Delta t \ , \tag{18.2b}$$
$$z_{i+1} = z_i + (t_i^2 - y_i z_i)\Delta t \ . \tag{18.2c}$$

These equations are solved iteratively. Step 1: Insert the initial values x_0, y_0, z_0 into the right–hand sides of Eqs. (18.2) with $i = 0$. This yields x_1, y_1, z_1. Step 2: Insert x_1, y_1, z_1 into the right–hand sides of Eqs. (18.2) with $i = 1$. This yields x_2, y_2, z_2. Continue for N timesteps to reach x_N, y_N, z_N.

Exercise 18.1a

Solve the system (18.1) numerically using Euler's method (18.2) with initial data $x(0) = 0$, $y(0) = 0$, and $z(0) = 1$. Make a plot of the dependent variables (x, y, and z) versus t for $0 \leq t \leq 2.0$.

Exercise 18.1b

Use Euler's method to solve the system of ODEs

$$dx/dt = yt/x \ ,$$
$$dy/dt = x - y + t \ ,$$

with initial conditions $x(0) = -20$ and $y(0) = 5$. Plot x and y versus t on a single graph with $0 \leq t \leq 25$. Experiment with different numbers of timesteps N and determine the final values $x(25)$ and $y(25)$ to 5 significant figures.

18.2 Euler's method in general

Any system of first order ODEs can be described using the shorthand notation

$$\frac{du}{dt} = F(u,t) \ . \tag{18.3}$$

Here, u denotes the dependent variables (the unknown functions) and $F(u,t)$ denotes the right–hand sides. For example, consider the system (18.1) from the previous section. The dependent variables x, y and z are collected into a "vector" of unknowns,

$$u(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \ , \tag{18.4}$$

and the right–hand sides are written as

$$F(u,t) = \begin{pmatrix} xy - z + t \\ 2z + x \\ t^2 - yz \end{pmatrix} \ . \tag{18.5}$$

With these identifications for $u(t)$ and $F(u,t)$, Eq. (18.3) is equivalent to Eqs. (18.1).

Euler's method is obtained by replacing the derivatives du/dt with the two–point forward difference formula $(u_{i+1} - u_i)/\Delta t$ and evaluating $F(u,t)$ at time t_i. Solving for u_{i+1}, we find

$$u_{i+1} = u_i + F(u_i, t_i)\Delta t \ . \tag{18.6}$$

Here, $u_i = u(t_i)$ denotes the dependent variables at the discrete times t_i. Also recall that $\Delta t = (t_N - t_0)/N$ where N is the number of timesteps.

Exercise 18.2

Use Euler's method to solve the system of ODEs $du/dt = F(u,t)$ with

$$u = \begin{pmatrix} x \\ y \end{pmatrix} \ , \quad F(u,t) = \begin{pmatrix} y + t \\ -4(x - t) \end{pmatrix} \ .$$

Choose initial conditions $x(0) = 0$, $y(0) = 12$ and time interval $0 \le t \le 10$. Apply the three–point convergence test of Eq. (17.23) to each of the variables x and y.

18.3 Second order equations

Up to this point we have focused on first order ODEs. Higher order ODEs can be treated using the same numerical techniques by defining new variables. For example, consider the second order ODE

$$\ddot{x} = 2xt - x\dot{x} \ , \tag{18.7}$$

with initial conditions $x(0) = 1$ and $\dot{x}(0) = 0$. (Dots denote time derivatives.) We can write this as a system of first order ODEs by defining a new variable $y = \dot{x}$. Equation (18.7) is equivalent to

$$\dot{x} = y \ , \tag{18.8a}$$

$$\dot{y} = 2xt - xy \ , \tag{18.8b}$$

with initial conditions $x(0) = 1$ and $y(0) = 0$.

Exercise 18.3a

Solve the system (18.8) with the given initial conditions using Euler's method. Plot a graph of x versus t for the domain $0 \le t \le 3$, and find the value of $x(3)$ accurate to 3 significant figures. (Justify your answer.)

Exercise 18.3b

Use Euler's method to solve the second order differential equation

$$\ddot{x} = 1 - 2x(\dot{x})^2$$

with initial conditions $x(0) = -1$, $\dot{x}(0) = 0$. Plot x and $\dot{x}$ as functions of t for $0 \leq t \leq 10$.

18.4 Second–order Runge–Kutta

Euler's method is first order, meaning its errors are proportional to N^{-1}. The error is (approximately) cut in half when we double the resolution. This is not particularly efficient. It generally takes a large number of timesteps N to achieve a modest amount of accuracy. There are other numerical algorithms that are more efficient.

Second order Runge–Kutta (RK2) is a second order method, with errors proportional to N^{-2}. With RK2, the error is (approximately) reduced by a factor of $1/4$ when the resolution is doubled. Here is the RK2 method:

$$u_h = u_i + F(u_i, t_i)\Delta t/2 \ , \tag{18.9a}$$

$$t_h = t_i + \Delta t/2 \ , \tag{18.9b}$$

$$u_{i+1} = u_i + F(u_h, t_h)\Delta t \ . \tag{18.9c}$$

In the first equation (18.9a), Euler's method is used to approximate the value of u at the *half* timestep $t_h \equiv t_i + \Delta t/2$. The values of the dependent variables at the half timestep are denoted u_h. The third equation (18.9c) looks just like Euler's method, but instead of using u_i and t_i in the function F, we use the half–timestep values u_h and t_h.

The second–order Runge–Kutta method is not difficult to implement in a numerical code. After importing NumPy and defining parameters, create an array for the discrete times:

```
t = np.linspace(tinitial,tfinal,N+1)
```

Next, define an array for each of the dependent variables using (for example) `np.zeros(N+1)`. The RK2 Eqs. (18.9) should appear in a loop, such as

```
for i in range(N):
```

This implements the N timesteps.

Comments:

- Remember, Eq. (18.9a) is a *set* of equations, one for each of the dependent variables. For example, for the system (18.1), Eq. (18.9a) represents

```
x[i+1] = x[i] + (x[i]*y[i] - z[i] + t[i])*Deltat/2
y[i+1] = y[i] + (2*z[i] + x[i])*Deltat/2
z[i+1] = z[i] + (t[i]**2 - y[i]*z[i])*Deltat/2
```

  Likewise, Eq. (18.9c) is a set of equations. All of the equations (18.9) should be included in the single `for` loop.
- Be careful to implement Eq. (18.9c) properly. The function $F(u_h, t_h)$ is evaluated at the half–timestep values u_h and t_h, but the leading term on the right–hand side is u_i, not u_h.
- The variables u_h and t_h do not need to be defined as arrays. Their values can be reassigned with each cycle through the `for` loop.

Since the error for second order Runge–Kutta is proportional to N^{-2}, we have $|\text{error}| = CN^{-2}$ for some positive constant C. Take the logarithm of this relation to obtain

$$\log(|\text{error}|) = \log(C) - 2\log(N) \ . \tag{18.10}$$

A plot of $\log(|\text{error}|)$ versus $\log(N)$ should be (approximately) a straight line with slope -2.

Exercise 18.4a

Use RK2 to numerically solve the system of ODE's

$$\dot{x} = -y \ ,$$
$$\dot{y} = x \ ,$$

with initial conditions $x(0) = 1$, $y(0) = 0$, over the time interval $0 \le t \le 10$. Carry out a convergence test: Use the exact solution $x(t) = \cos t$, $y(t) = \sin t$ to compute the error in $x(10)$ for various numbers of timesteps. Plot $\log(|\text{error}|)$ versus $\log(N)$ and show that as N becomes large the slope approaches -2.

Second order Runge–Kutta is better than Euler's method because, most often, it requires fewer timesteps and less compute time to achieve a comparable level of accuracy.

Exercise 18.4b

Use RK2 to solve the system

$$\dot{x} = \sin(tx)$$
$$\dot{y} = y^2 - x$$

with initial conditions $x(0) = 1.0$ and $y(0) = 0.0$, for $0 \leq t \leq 10$. Find the value of $x(10)$ accurate to 5 significant figures. About how many timesteps are required to achieve this level of accuracy? Compare with the number of timesteps required for Euler's method.

18.5 Fourth–order Runge–Kutta

Runge–Kutta methods are a family of numerical algorithms for solving systems of ODEs. The most popular member of this family is fourth–order Runge–Kutta (RK4). Applied to the system $du/dt = F(u,t)$, the RK4 algorithm is

$$u_a = u_i + F(u_i, t_i)\Delta t/2 \ , \tag{18.12a}$$
$$u_b = u_i + F(u_a, t_h)\Delta t/2 \ , \tag{18.12b}$$
$$u_c = u_i + F(u_b, t_h)\Delta t \ , \tag{18.12c}$$
$$u_d = u_i + F(u_c, t_{i+1})\Delta t \ , \tag{18.12d}$$
$$u_{i+1} = \frac{1}{3}(u_a + 2u_b + u_c + u_d/2) - \frac{1}{2}u_i \ . \tag{18.12e}$$

As with RK2, $t_h \equiv t_i + \Delta t/2$ is the time half–way between t_i and t_{i+1}. The variables u_a, u_b, u_c and u_d are intermediate values of $u(t)$ that are used to evaluate the right–hand side function $F(u,t)$.

Although it is far from obvious, RK4 is a *fourth–order* scheme; that is, the errors are proportional to N^{-4}. A plot of $\log(|\text{error}|)$ versus $\log(N)$ should be (approximately) a straight line with slope -4.

Exercise 18.5a

Use RK4 to solve the system of ODEs

$$\dot{x} = \ln y + \sin x \ ,$$
$$\dot{y} = \ln x + \cos y \ ,$$

with initial conditions $x(0) = y(0) = 1$. Plot graphs of x and y versus t. Find $x(10)$ and $y(10)$ to 6 significant figures.

Exercise 18.5b

Use RK4 to solve the system of ODEs

$$\dot{x} = y \ ,$$
$$\dot{y} = x \ ,$$

with initial conditions $x(0) = 1$, $y(0) = 0$ over the time interval $0 \leq t \leq 2$. Carry out a convergence test: Use the exact solution $x(t) = \cosh t$, $y(t) = \sinh t$, to compute the errors in $x(2)$ and $y(2)$ for various numbers of timesteps N. Plot a graph of $\log(|\text{error}|)$ versus $\log(N)$ and show that as N becomes large the slope approaches -4.

18.6 `solve_ivp()`

The function `solve_ivp()` from the `scipy.integrate` library solves systems of first order ordinary differential equations. The acronym "ivp" stands for *initial value problem*. This refers to the types of problems we have been solving all along—a system of differential equations with initial conditions. See the discussion in Sec. 20.1.

Consider the initial value problem

$$\ddot{x} + ax - (\dot{x})^2/x = 0 \ , \tag{18.13}$$

with $x(0) = 1$, $\dot{x}(0) = 0$. The equivalent system of first order equations is

$$\dot{x} = y \ , \tag{18.14a}$$
$$\dot{y} = -ax + y^2/x \ , \tag{18.14b}$$

with $x(0) = 1$ and $y(0) = 0$. The following code solves this system numerically for times $0.0 \leq t \leq 8.0$ and parameter value $a = 0.1$:

```
import numpy as np
import scipy.integrate as si
import matplotlib.pyplot as plt

# right-hand sides of ODEs
def F(t,variables,a):
    x,y = variables
    dxdt = y
    dydt = -a*x + y**2/x
    return dxdt, dydt

# parameters
```

```
a = 0.1
ti = 0.0    # initial time
tf = 8.0    # final time

# initial conditions
xi = 1.0
yi = 0.0

# solve the ODEs
u = si.solve_ivp(F,[ti,tf],[xi,yi],args=[a])
t = u.t       # independent variable
x = u.y[0]    # first dependent variable
y = u.y[1]    # second dependent variable

# plot results
plt.close()
plt.plot(t,x,'r',t,y,'b')
plt.show()
```

Take a close look at the arguments of `solve_ivp()`.

- The first argument `F` is the name of the right–hand sides function.
- The second argument `[ti,tf]` is a list containing the initial and final times.
- The third argument `[xi,yi]` is a list of initial conditions.
- The fourth argument `args=[a]` is an optional list of parameters to pass to the function definition.

Also note the following:

- `solve_ivp()` returns the results as a dictionary, which we call `u`. Discrete values of the independent variable are stored in `u.t`. The corresponding values of the dependent variables are stored in `u.y`. Thus, the first dependent variable is `u.y[0]`, the second dependent variable is `u.y[1]`, *etc.*
- In the function definition the line `x,y = variables` assigns the dependent variables to the more familiar names `x` and `y`.

Exercise 18.6a

Run the code. How does the graph look? What are the values of x and y at the final time?

By default, `solve_ivp()` only saves the data at a few discrete times, so

the resulting graph is not very smooth. This has nothing to do with the accuracy of the code. You can tell `solve_ivp()` to keep more data points by specifying the option `t_eval`. For example,

```
tee = np.linspace(ti,tf,101)
u = si.solve_ivp(F,[ti,tf],[xi,yi],args=[a],t_eval=tee)
```

instructs `solve_ivp()` to keep `101` data points beginning with the initial time `ti` and ending with the final time `tf`.

Exercise 18.6b

Specify the option `t_eval` with `101` data points. Now how does the graph look? Have the values of x and y at the final time changed? Do x and y change if you replace `101` with `11`?

Exercise 18.6c

Use `solve_ivp()` to solve

$$\dot{x} = \sqrt{x} - ax/y \ ,$$
$$\dot{y} = t/b - y/x \ ,$$

with $a = 1.3$, $b = 2.0$, and initial conditions $x(0) = 3$, $y(0) = 1$. Plot x and y as functions of t for $0 \leq t \leq 10$.

18.7 ODE solvers with `solve_ivp()`

By default, `solve_ivp()` uses the "RK45" method to solve ODEs. This method uses a combination of fourth and fifth–order Runge–Kutta algorithms to solve the ODEs, monitor the error, and adjust the timestep as needed. The `solve_ivp()` function can access other solvers, including "RK23" (second and third order Runge–Kutta), "DOP853" (eighth–order Runge–Kutta), "Radau" (an order 5 Runge–Kutta type method), and others. You can specify the solver to use by adding an option such as `method = 'DOP853'` as an argument in `solve_ivp()`.

Exercise 18.7a

Solve the system (18.14) using various ODE solvers in `solve_ivp()`. Is there a difference in the results?

How accurate is `solve_ivp()`? If we ask Python to print out the value of a dependent variable at any particular time, it gives us the answer to 16 or 17 significant figures. Should we trust all of the digits? It can be difficult to judge the accuracy of a "canned" solver like those in `solve_ivp()`. This is the main reason why research scientists often prefer to write their own numerical routines.

For the system (18.14) the exact solution is known:

$$x(t) = e^{-at^2/2} \,, \tag{18.15a}$$

$$y(t) = -ate^{-at^2/2} \,. \tag{18.15b}$$

In this case we can determine the accuracy of the ODE solvers by comparing with the exact results. Keep in mind that the accuracy of any given algorithm is highly problem–dependent.

Exercise 18.7b

Solve Eqs. (18.14) using your RK4 code and compute the errors in the numerical values of $x(8)$ and $y(8)$. How many decimal places are correct? Does RK45 do any better? How about DOP853?

Exercise 18.7c

Use your own RK4 code to solve

$$\dot{x} = \sin^2(tx)$$

$$\dot{y} = y^2 - x$$

with initial conditions $x(0) = 1.0$ and $y(0) = 1.0$, for $0 \leq t \leq 10$. Plot the results. Solve the same system using `solve_ivp()` with the RK45 method. Do your code and RK45 agree? Compare using both low and high resolution with your RK4 code.

There are optional arguments that can be given to `solve_ivp()` to increase the accuracy of RK45, DOP853, *etc.* These options are discussed in Sec. 19.6.