Lesson 9: Machine Error and the Runge–Kutta Method

1 Radioactive decay again

The Euler method has a discretization (truncation) error on the order of Δt. This error can be reduced by increasing the number of timesteps (which decreases the timestep Δt). Can we make the error arbitrarily small?

> Exercise 1: Write a new code for the radioactive decay problem. Make this code as simple as possible. Do not collect data into arrays. Start with $N_0 = 100$ nuclei and a mean lifetime of $\tau = 5.0\,\mathrm{s}$. Have your code compute the number of nuclei that remain after $10\,\mathrm{s}$. Do this for $10, 100, 1000, \ldots$ timesteps. What is the relative error in each case? How small can you make the error?

One of the factors that prevents you from obtaining an arbitrarily small error is runtime. For example, with 10 million timesteps, the relative error is probably around 10^{-7}, but your code might take 10 seconds or more to run. For a complicated code, it might take minutes or hours or even days for the code to run at a resolution that gives you the desired level of accuracy. In this lesson we will introduce the second order Runge–Kutta method for solving ordinary differential equations. Second order Runge–Kutta is more efficient than the Euler method in the sense that it requires less runtime to achieve a given level of accuracy.

2 Machine Error

In addition to truncation error, there is another source of error that is always present in numerical computation. This error is due to computers having only a finite amount of memory available. To investigate these errors we need to look more closely at floating point numbers.

Rational numbers can potentially be represented exactly by a computer, but irrational numbers cannot. For example, π and e are both irrational, and would require infinite precision, and therefore infinite memory, to be stored exactly on a computer. Interestingly, there are many more irrational numbers than there are rational numbers, but even many rational numbers require a large number of digits to be represented correctly. Computers represent numbers using a finite number of digits, or bits. Integers can be represented exactly, up to the limit of the computer's memory. Other real numbers can only be represented approximately, and we must accept this fact. One way to represent a large range of values over many

orders of magnitude efficiently is to use floating point numbers. Floating point numbers are represented somewhat like scientific notation.

As an example, let's take a few digits of e:

$$2.7182818 = 2712818 \times 10^{-7}$$

On a 32-bit system, a number like this is represented using up to 32 bits of memory. The bits used to represent the number are broken up into three parts:

1. 1 bit is used to represent the sign of the number
2. 8 bits are used to represent the exponent
3. 23 bits are used to represent the mantissa

This type of floating point representation is known as single precision. There is also a double precision representation where numbers are represented by:

1. 1 bit is used to represent the sign of the number
2. 11 bits are used to represent the exponent
3. 52 bits are used to represent the mantissa

for a total of 64 bits to represent the whole number.

Due to the finite representation of floating point numbers, operations involving them are subject to errors. Often, these errors won't be significant. However, we should be aware of such errors, and how they arise.

One consequence of machine error is that the addition of floating point numbers is not associative. That is, the order in which we perform operations matters, so that

$$x + (y + z) \neq (x + y) + z$$

For example, consider the following code:

```
x = 0.1
y = 0.2
z = 0.3

sum1 = x + (y + z)
sum2 = (x + y) + z
```

Exercise 2: Execute the above python code and print the results. Use: `print 'sum1 = {!r}' .format(sum1)` for `sum1`, and similarly for `sum2`. This format insures that the full number is printed. Observe the difference in the outputs.

The results you receive will depend on the precision of your computer, and your version of Python. If we try checking the equivalence of the two numbers with `sum1 == sum2`, we

receive `False`. When you compare two floating point numbers, you should check that the numbers are within some small value of each other. For example, use `abs(sum1 - sum2) < 1e-10`.

From our recent work with the radioactive decay code, we can infer that for sufficiently small timestep, machine error can affect the results. If the timestep is too small the simulation can become less accurate with increasing resolution.

Homework: Using your code from Exercise 1, find the relative error with the number of timesteps ranging from nts = 10^1 to, say, 10^9, and place the data in a file. Create a python code to read the data and make a plot of log(error) versus log(nts). Explain the differences between your plot and the plot that you would expect to find, assuming the error is proportional to Δt.

3 Second order Runge–Kutta

Consider the ordinary differential equation

$$du/dt = f(u,t) \tag{1}$$

The dependent variable is u and the independent variable is t. For the Euler method, we approximate the equation at time $t = t_i \equiv i\Delta t$ by $(u_{i+1} - u_i)/\Delta t = f(u_i, t_i)\Delta t$, where $u_i \equiv u(t_i)$. Rearranging, we have the result

$$u_{i+1} = u_i + f(u_i, t_i)\,\Delta t$$

which allows us to solve for u_{i+1} given u_i. The Taylor series expansion of $u(t_i + \Delta t)$ shows that the exact result is

$$u_{i+1} = u_i + f(u_i, t_i)\,\Delta t + \mathcal{O}(\Delta t^2)$$

where $\mathcal{O}(\Delta t^2)$ denotes terms of order Δt^2. Thus, each step of the Euler method introduces errors of order Δt^2. Since the number of timesteps required to reach a finite time is proportional to $1/\Delta t$, the cumulative error in the Euler method is order Δt. We say that the Euler method is a *first order* method.

The second order Runge–Kutta method (RK2) is defined by the following two–substep process:

$$u_h = u_i + f(u_i, t_i)\,\Delta t/2 \tag{2a}$$
$$u_{i+1} = u_i + f(u_h, t_h)\,\Delta t \tag{2b}$$

where $t_h \equiv t_i + \Delta t/2$. In the first substep, the Euler method is used to estimate the value of u at the *half* timestep. That is, the value of u at time $t_h \equiv t_i + \Delta t/2$. Here, this value is denoted u_h; sometimes it is denoted $u_{i+1/2}$. The second substep looks just like the Euler 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.

Let's compute the truncation error for RK2. Note that we can insert u_h from Eq. (2a) into Eq. (2b), with the result

$$u_{i+1} = u_i + f(u_i + f(u_i, t_i)\,\Delta t/2, t_i + \Delta t/2)\,\Delta t \tag{3}$$

The function f, with its complicated arguments, can be expanded in a Taylor series about $\Delta t = 0$:

$$f(u_i + f(u_i, t_i)\,\Delta t/2, t_i + \Delta t/2) = f(u_i, t_i) + \left.\frac{\partial f}{\partial u}\right|_{t_i} f(u_i, t_i)\,\Delta t/2 + \left.\frac{\partial f}{\partial t}\right|_{t_i} \Delta t/2 + \mathcal{O}(\Delta t^2)$$

The coefficient of the terms proportional to $\Delta t/2$ are

$$\left.\frac{\partial f}{\partial u}\right|_{t_i} f(u_i, t_i) + \left.\frac{\partial f}{\partial t}\right|_{t_i} = \left.\left[\frac{\partial f}{\partial u}\frac{du}{dt} + \frac{\partial f}{\partial t}\right]\right|_{t_i} = \left.\frac{df}{dt}\right|_{t_i}$$

where Eq. (1) has been used to replace f with du/dt. Thus, RK2 is equivalent to

$$u_{i+1} = u_i + f(u_i, t_i)\,\Delta t + \frac{1}{2}\left.\frac{df}{dt}\right|_{t_i} \Delta t^2 + \mathcal{O}(\Delta t^3) \tag{4}$$

Now consider the Taylor series for $u(t_i + \Delta t)$ about $\Delta t = 0$:

$$u_{i+1} = u_i + \left.\frac{du}{dt}\right|_{t_i} \Delta t + \frac{1}{2}\left.\frac{d^2u}{dt^2}\right|_{t_i} \Delta t^2 + \mathcal{O}(\Delta t^3) \tag{5}$$

Using the differential equation (1), we see that

$$\left.\frac{du}{dt}\right|_{t_i} = f(u_i, t_i) \quad \text{and} \quad \left.\frac{d^2u}{dt^2}\right|_{t_i} = \left.\frac{df}{dt}\right|_{t_i}$$

Therefore the second order Runge–Kutta method (4) agrees with the exact Taylor series (5) through terms of order Δt^2.

A more complete analysis shows that the order Δt^3 terms in Eqs. (4) and (5) do not agree. Thus, the error in each timestep of RK2 is proportional to Δt^3. The number of timesteps required to reach any given finite time is proportional to $1/\Delta t$, so the accumulated error in RK2 is order Δt^2. Second order Runge–Kutta is a second order method. As you might guess, there are higher–order Runge–Kutta methods. Fourth order Runge–Kutta is very popular.

Homework:

- Write a code to solve the radioactive decay problem using second order Runge–Kutta. Choose a mean lifetime of $\tau = 5\,\mathrm{s}$ and a simulation time of $10\,\mathrm{s}$.
- Compute the error for at least four different resolutions, and show that the relative errors are proportional to Δt^2.
- Compute the relative error in your RK2 code when the number of timesteps is set to `nts = 100`. Use your Euler code to find the approximate number of timesteps required for the Euler method to give a result with comparable error.