EOSC 453 Assignment 0: Solving ODEs numerically in MATLAB or Python

Figures only: Due September 9, 2024

0.1 Introduction

This assignment is aimed at getting you acquainted with three things: MATLAB or PYTHON (if they are new to you), the course website and simple approaches to the numerical solution of ordinary differential equations (ODEs). We will be solving ODEs and PDEs on a computer. This assignment simply guides you through some basic examples and techniques that are on the course website. I expect you to work through this material and the corresponding example scripts on the website by class 2 (sept 9). I will hand out assignment 1 at the end of class 2, which will reinforce the material here through solving a very topical problem. Assignment 1 will be the focus of the subsequent 2 or 3 classes. Much or all of the material below is excerpted from: *Notes on the numerical solution of differential equations* on the course website.

0.2 The Euler Method: An "intuitive" approach using the definition of the derivative

Suppose you want to solve numerically the simple first-order ordinary differential equation

$$\frac{dY}{dt} = -\lambda\, Y \tag{1}$$

with the initial condition $Y(0) = Y_0$. Equation (1) is the well known equation governing the decay of a radioactive isotope where λ is the decay constant. The analytic solution for this equation happens to be $Y(t) = Y_0 \exp(-\lambda t)$ but our aim is to obtain this result numerically rather than by analytical methods. We start by assuming that the time variable t varies discretely, rather than continuously, so that $t = [t_1, t_2, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_{N-1}, t_N]$ where the sample interval is constant at Δt, e.g. $t_{i+1} - t_i = \Delta t$. The values of $Y(t)$ at times t_i are denoted $Y_i = Y(t_i)$. Referring back to Equation (1), at a particular time t_i the equation could be written

$$\left(\frac{dY}{dt}\right)_i = -\lambda\, Y_i \tag{2}$$

where $(dY/dt)_i$ denotes the time derivative dY/dt evaluated at time t_i. From the definition of a derivative we might write this time-derivative as

$$\left(\frac{dY}{dt}\right)_i \approx \frac{Y_{i+1} - Y_i}{\Delta t}, \tag{3}$$

which allows (2) to be approximated by the finite-difference equation

$$\frac{Y_{i+1} - Y_i}{\Delta t} = -\lambda\, Y_i. \tag{4}$$

This procedure leads to the recursion relation

$$Y_{i+1} = Y_i - \lambda \Delta t\, Y_i. \tag{5}$$

(Note that this exercise is equivelant to approximating the derivative with a truncated Taylor series: $y(t+\Delta t) = y(t) + \Delta t y\prime(t) + ((\Delta t)^2/2) y\prime\prime(t) + \ldots \approx y(t) + \Delta t y\prime(t)$) To start the integration, use the initial condition $Y_1 = Y_0$ and march the solution forward in time, by repeated application of the recursion relation (5):

$$\begin{array}{lll} Y_1 & = Y_0 & (6a) \\ Y_2 & = Y_1 - \lambda \Delta t Y_1 & (6b) \\ Y_3 & = Y_2 - \lambda \Delta t Y_2 & (6c) \\ \ldots & = \ldots & \end{array}$$

A MATLAB or PYTHON script to perform this "step-by-step" integration and plot the result can be downloaded from the course website. It is called "ODEexample1main.m" or "ODEexample1main.ipynb". Download and play with either example script. How do you determine the "best" time step for this class of calculation? Be able to explain your answer from looking at the results of your calculation AND from the form of the equation iteslf (hint: do some dimensional analysis to work out what is the characteristic time scale over which the solution evolves).

A more interesting application of similar thinking is to consider the case of two coupled ordinary equations, for example,

$$\begin{array}{lll} \frac{dY_1}{dt} & = \alpha Y_2 & (7a) \\ \frac{dY_2}{dt} & = -\alpha Y_1. & (7b) \end{array}$$

A distinctive feature of the above system is that the solution of (7a) depends on that of (7b) and *vice versa*. Following the steps that lead to Equation (5), I approximate the time derivatives dY_1/dt and dY_2/dt by finite difference approximations and arrive at the recursion relations

$$\begin{array}{lll} Y_{i+1}^{[1]} & = Y_i^{[1]} + \alpha \Delta t Y_i^{[2]} & (8a) \\ Y_{i+1}^{[2]} & = Y_i^{[2]} - \alpha \Delta t Y_i^{[1]}. & (8b) \end{array}$$

Assuming initial conditions $Y_1(0) = Y_0^{[1]}$ and $Y_2(0) = Y_0^{[2]}$, one can march the solution forward as for the previous example. A MATLAB or PYTHON script to perform this integration and plot the result can be downloaded from the course website. It is called "ODEexample2main.m" or "ODEexample2main.ipynb".

0.3 Runge-Kutta Integration

The foregoing examples are chosen for their simplicity and, in fact, are not representative of standard approaches to the numerical solution of initial value problems. The problem with the simple approach is that it is not necessarily accurate—you can confirm this by experimenting with the value of Δt in solving Example 2. In terms of a Taylor series the method is accurate only to order Δt and thus is appropriate only where the time steps are sufficiently small that the crude linear approximation in Equation 3 holds. This tricky issue raises the important issues of the *quality* and *suitability* of the integrator. The recursion schemes of Examples 1 and 2 illustrate the use of a low-quality integrator that is rarely suitable for the accurate solution of systems of differential equations. Usefully, MATLAB and PYTHON offer a range of integrators of varying quality (note: there are more options in MATLAB). You might wonder what the value of low- and intermediate-quality integrators might be: a disadvantage of high-quality integrators is that they execute relatively slowly—sometimes prohibitively slowly if the problem is large. So, the challenge of finding the balance between accuracy and computational efficiency often arises at the beginning of any analysis.

Improving the accuracy of an integrator in a practical sense is not as simple as adding more terms to a Taylor expansion (as the order of the derivative increases the time steps become vanishingly small such that it takes forever to get a solution). A popular method of great practical importance and that is fourth order ($O((\Delta t)^4)$) accurate (i.e., the same as a 5 term Taylor expansion) is the classical Runge-Kutta formula in the interval $t_n \leq t \leq t_{n+1}$:

$$y_{(n+1)} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + 0.5\Delta t, y_n + 0.5\Delta t k_1) \\ k_3 &= f(t_n + 0.5\Delta t, y_n + 0.5\Delta t k_2) \\ k_4 &= f(t_n + \Delta t, y_n + \Delta t k_3). \end{aligned}$$

Here, for equation (1), $f(t, y) = dY/dt = -\lambda Y$ with $Y(0) = Y_0$. The sum $\frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ is then essentially an average slope (just a better description of the derivative, really). That is, k_1 is the slope in the left-hand end of the interval, k_2 and k_3 are the slopes at the midpoint found with Equation 3 and k_4 is the slope at $t_n + \Delta t$ using Equation 3 and the slope k_3 to go from t_n to $t_n + \Delta t$.

In MATLAB or PYTHON, the implementation of an integrator is the same regardless of the quality of the integrator. In this class, we will mostly rely on Runge-Kutta integration. The general Runge-Kutta problem is to integrate a coupled system of ODEs such as

$$\begin{aligned} \frac{dY_1}{dt} &= F_1(t, Y_1, Y_2, \ldots, Y_{M-1}, Y_M, F_2, F_3, \ldots, F_M) \\ \frac{dY_2}{dt} &= F_2(t, Y_1, Y_2, \ldots, Y_{M-1}, Y_M, F_1, F_3, \ldots, F_M) \\ \frac{dY_3}{dt} &= F_3(t, Y_1, Y_2, \ldots, Y_{M-1}, Y_M, F_1, F_2, \ldots, F_M) \quad (9) \\ \ldots &= \ldots. \end{aligned}$$

Note in (9) that the form of the left-hand side terms is predictable and problem-specific details are all on the right-hand side. In setting up to apply the Runge-Kutta method, all that is required is to specify the exact form of the right-hand side terms. Thus, for the previously-analyzed coupled system,

$$\begin{aligned} \frac{dY_1}{dt} &= \alpha Y_2 \quad (10a) \\ \frac{dY_2}{dt} &= -\alpha Y_1 \quad (10b) \end{aligned}$$

and it is only necessary to define the function

$$\begin{aligned} F_1 &= \alpha Y_2 && (11a) \\ F_2 &= -\alpha Y_1 && (11b) \end{aligned}$$

and pass this information to a standard Runge-Kutta solver.

0.4 *Solution of higher-order differential equations*

The Runge-Kutta approach is far more powerful than might first be imagined. It is not entirely obvious that the method leads immediately to an approach to solving higher-order differential equations. As an example, consider the nonlinear fourth-order differential equation

$$Y \frac{d^4Y}{dt} + \left(\frac{d^2Y}{dt^2}\right)^2 + \exp(-Y) = G(t). \tag{12}$$

Through "reduction of order" or just trying it out, it is readily confirmed that, taking $Y_1(t) = Y(t)$, the following system of first-order differential equations is equivalent to (12)

$$\begin{aligned} \frac{dY_1}{dt} &= \tfrac{dY}{dt} = Y_2 && (13a) \\ \frac{dY_2}{dt} &= \tfrac{d^2Y}{dt^2} = Y_3 && (13b) \\ \frac{dY_3}{dt} &= \tfrac{d^3Y}{dt^3} = Y_4 && (13c) \\ \frac{dY_4}{dt} &= \tfrac{d^4Y}{dt^4} = \tfrac{1}{Y_1}\left(G(t) - Y_3^2 - \exp(-Y_1)\right). && (13d) \end{aligned}$$

Thus the Runge-Kutta setup for this problem simply entails the definition of the vector function F_k

$$\begin{aligned} F_1 &= Y_2 && (14a) \\ F_2 &= Y_3 && (14b) \\ F_3 &= Y_4 && (14c) \\ F_4 &= \tfrac{1}{Y_1}\left(G - Y_3^2 - \exp(-Y_1)\right) && (14d) \end{aligned}$$

and integration subject to appropriate 4 initial conditions on $[Y_1, Y_2, Y_3, Y_4]$.

1 Assignment

Install MATLAB and/or PYTHON (see, e.g., https://www.anaconda.com/). Make the material above and the example scripts on the website yours. If MATLAB or PYTHON are new to you, work together or email Luke Brown to make an appointment. Either platform is most easily learned by example. So, run the example scripts for solving one or two coupled ODEs using home-grown Euler and Runge-Kutta methods, change them, rewrite them in your own way. Break them. Fix them. Make them better. Add more evolution equations. Make them yours. This is one way to build understanding and get quickly past any reservations you might have with solving ODEs on a computer. Unlike every other assignment, *nothing is due in to me but come to class with figures related to whatever you did.*

To start, compare numerical solutions of equation 1 or 7 with time steps that are $\gg 1/\alpha$, $\approx \alpha$, $\ll \alpha$ to analytical solutions. Play around. A lot. **I will assume that you have worked through these examples and that you understand how the scripts work and why they are constructed in the way you will see**.

As a last comment, please also pay special attention to the structure of the scripts `odeRKexamplemain.m` and `odeRKexamplemain.ipynb`, which can call RK4 or any other similar integrator (such as ode45 and ode15s in MATLAB or odeint and solvers from solve_ivp in Python). Complex calculations (and simple ones) are most effectively structured to have a “Main” program that calls functions to do the business at hand: define equations, do calculations, plot results, etc.. You might also use a separate 'input' file that defines constants and parameters that will be used in your calculations. The goal is to write 'clean code' and to that end, your final Main program should rarely be longer than 1 page. You won't listen to me on this point but at least think about it.