ENGSCI 331 - Computational Techniques 2

Ordinary Differential Equations - Lab Part 1

Introduction

Background and Lab Objective

This lab will focus on the numerical solution of ordinary differential equations (ODEs) using Runge-Kutta methods. There are multiple tasks, each of which should be completed. A marking rubric will be made available prior to the deadline, which will indicate the marks available for each task.

Lab Files and Code:

The lab files are all available from the Canvas lab assignment:

- `task[...]_ode.py`

  A script file for each Task.

- `functions_ode.py`

  Contains the functions used throughout the Tasks.

Submission Instructions:

Upload your code:

Please combine all code files for your submission into a single zipped directory and upload to the relevant Canvas assignment. Please do **not** modify the original file names or include any additional code files. Please do **not** modify any existing class, method, or function names. You are allowed to create additional functions or classes and methods in `ode_functions.py` if you would like e.g. helper functions to produce plots or similar.

Your code will be run through software to detect similarities with other student code, so please ensure that all code submitted is your own work.

Hand-In Items:

Some tasks have specific workings that you should hand-in, in addition to your code. These are clearly labelled in this lab document. Your hand-in items should be compiled into a brief report named `report_upi.pdf/doc/docx`. Please upload this alongside your zipped code as part of your assignment submission.

Any written comments in your report can be targeted at a reader who has a complete understanding of the methods and techniques, and knows what you have been asked to do. It should detail anything interesting, unexpected, or complex in the implementation; present and very briefly discuss interesting results; and present any suitable conclusions if appropriate.

Formatting your plots: Plots should have their axes labelled, a title or caption, and a legend if there is more than one data set.

Coding Tips

The following tips may be helpful as you work through this lab.

Coding Tip #1: Initialise a NumPy array with a list:

```
y = np.array([0, 0]) # correct
y = np.array(0, 0)   # incorrect
```

Coding Tip #2: Python code will pause when a plot is shown on screen. You will need to close the plot before the code will continue. Boolean variable(s) can be used to control whether a particular piece of code, for example the code that produces and shows a figure on screen, should be executed. This may be easier than manually commenting/uncommenting commands.

Coding Tip #3: The `*args` given in the argument of a function can be used to package together a number of *optional* input parameters. This is useful when one or more other model parameters appear in the differential equation.

Coding Tip #4: The command `ax.invert_xaxis()` can be used to reverse/invert the direction of the x-axis. A similar command can be used for the y-axis.

Coding Tip #5: It is often easier to work with NumPy arrays than nested Python lists:

```
v = np.array(v) # convert Python list to numpy array
print(v[:,0])   # print first column of 2d array
```

Task 1: Bungy Jumper Model

Background and Objective

The Kawarau bungy jump near Queenstown was the world's first commercial Bungy operation and involves jumping from a suspension bridge that is 43 m above the Kawarau river.

An engineering student wants to jump and have their entire body dunked in the river. As the operator, you need to select an appropriate Bungy cord. There are 12 cords to choose from, each with a specific length and stretchiness. *Short* and *Regular* cords have a length of 16 and 21 m, respectively. The cord spring constants range from 50 to 100 N m^{-1}, inclusive, in steps of 10 N m^{-1}. The cords are named via their length and stretchiness, for example:

- *SHORT60* is a cord of length 16 m with a spring constant of 60 N m^{-1}.
- *REG90* is a cord of length 21 m with a spring constant of 90 N m^{-1}.

The vertical dynamics of a jumper can be modelled by the second-order ODE:

$$\frac{d^2y}{dt^2} = \frac{F}{m}$$

where y is the vertical displacement measured downward from the jump platform (m), F is the sum of forces acting on the jumper (N), m is the mass of the jumper (kg), and t is time (s). The jumper will experience different forces throughout their jump:

- When the cord is slack, the jumper only experiences the forces of gravity and drag:

  $$\frac{d^2y}{dt^2} = g - \operatorname{sgn}(v)\,\frac{c_d\,v^2}{m}$$

  where g is gravitational acceleration (m s^{-2}), c_d is the drag coefficient of the jumper as they travel through the air (kg m^{-1}) and v is the vertical velocity (m s^{-1}) of the jumper. Note the inclusion of the signum function, which impacts how the drag force acts depending on whether the jumper is falling downward or bouncing back upward. It is a piecewise function defined here as:

  $$\operatorname{sgn}(v) = \begin{cases} -1 & v < 0 \\ 0 & v = 0 \\ 1 & v > 0 \end{cases}$$

  It is available in NumPy as `np.sign()` or can be implemented manually.

- When the cord is stretched, spring and damping forces must also be included:

  $$\frac{d^2y}{dt^2} = g - \operatorname{sgn}(v)\,\frac{c_d\,v^2}{m} - \frac{k}{m}(y - L) - \frac{\gamma\,v}{m}$$

where k is the cord's spring constant (N m^{-1}), L is the length of the unstretched cord (m), and γ is the damping coefficient (N s m^{-1}).

- Our simple model will **not** account for any forces experienced in the water.

The objective of this task is to find the cord that will fully dunk the engineering student by the smallest amount, and to assess whether this can be done safely.

Steps to Complete

1. Write the second-order ODE as a system of first-order ODEs. You will be solving this system for two dependent variables, the vertical displacement y and vertical velocity v.

2. Complete the function `def derivative_bungy` in the provided functions file.

   This function should return the first-order derivatives of your system of ODEs as a 1D NumPy array. You may optionally want to write a test function to check that it works as expected.

3. Complete the function `def explicit_solver_fixed_step` in the provided functions file.

   This function should be able to implement **any** explicit RK method using the input weights, nodes and RK matrix (α, β, γ) from its Butcher tableau.

   Ensure that the function solves all time steps - a common mistake is to not solve the final time step of `t1`. You will also need to ensure that it can receive any optional parameters required by the derivative function i.e. it will need to work for other models than just our Bungy model. This can be achieved by packaging multiple arguments together.

   This function can be challenging to implement. If you are struggling and want to progress with finishing the task, you can alternatively hard-code the classic RK4 solver (though you will not receive full marks for this function). Another alternative approach is to solve only a first-order ODE, check that it works, and then move on to solving a system of first-order ODEs.

   You may optionally want to write a test function to check that this function works as expected. You could compare these results with that of a hard-coded classic RK4 solver.

4. Solve the Bungy model in the provided task file using the classic RK4 method for each of the twelve different cords.

   Assume the following parameters for your explicit RK solver: $h = 0.05$ (s), $t_0 = 0$ (s), $t_1 = 50$ (s). Additionally, assume the following ODE parameter values: $g = 9.8$ (m s^{-2}), $m = 67$ (kg), $c_d = 0.75$ (kg m^{-1}) and $\gamma = 8$ (N s m^{-1}).

   You may further assume that the jumper has zero initial vertical displacement, but jumps downward with an initial velocity of 2 m s^{-1}, and that they have a height of 1.8 m and must therefore reach a maximum vertical displacement of at least 44.8 m to be fully dunked in the river.

5. Produce a plot that appropriately compares the maximum vertical displacement reached for each of the 12 cords. This plot should allow you to easily identify the best bungy cord e.g. include a visual indicator for what vertical displacement represents the jumper being fully dunked.

   This can either be done directly in the provided task file, or you can alternatively create an additional function to assist with the plotting.

6. For your selected cord, produce two additional plots showing:

   - The vertical displacement **and** velocity against time.
   - The vertical displacement against velocity i.e. a phase plot.

   This can either be done directly in the provided task file, or you can alternatively create an additional function to assist with the plotting.

Hand-In

- Plot from Step 5. Write a brief comment to justify which cord should be chosen.
- Plots from Step 6. At approximately what time after jumping and with what velocity does the jumper impact the water?
- Consider a situation where the scales were not read correctly, and the true mass of the jumper is actually 85 kg, rather than 67 kg. Comment on what impact this will have on the jump (e.g. velocity at point of impact with water).

Task 2: Atmospheric Convection Model

Background and Objective

The Earth's atmosphere can be treated like a fluid. As the sun heats up the ground, the air directly above it will warm up. As warm air expands, it becomes less dense than its surroundings and will begin to rise in a thermal column. Colder air displaced by this thermal column, now more dense than its surroundings, will begin to descend. Atmospheric convection, driven by horizontal and vertical temperature and moisture variations, can lead to many atmospheric phenomenon, such as sea-breezes and thunderstorms.

The Lorenz system can be derived from a simplified model of atmospheric convection:

$$\frac{dx}{dt} = \sigma\,(y - x)$$
$$\frac{dy}{dt} = x\,(\rho - z) - y$$
$$\frac{dz}{dt} = x\;y - \beta\;z$$

This system of first-order ODEs can be solved for the co-dependent solutions:

- x is related to the rate of convection.
- y is related to the horizontal temperature variation.
- z is related to the vertical temperature variation.

There are three system parameters: σ, ρ and β. In the Lorenz model of atmospheric convection, these system parameters relate to the fluid dynamics of the Earth's atmosphere. This model has been widely studied as one of the first well known chaotic systems. It is highly sensitive to its initial conditions and exhibits seemingly random behaviour - somewhat unexpected for a deterministic system.

The Lorenz and other similar systems of ODEs are commonly used in modelling various physical systems, such as:

- Lasers
- Brushless DC motors
- Electric circuits
- Chemical reactions

The objective of this task is to use the Dormand-Prince embedded RK method to calculate the solution to the Lorenz system with an adaptive step size.

Steps to Complete

1. Complete the function `def derivative_lorenz` in the provided functions file.

2. Complete the function `def dormand_prince_solve` in the provided functions file.

   This should be your own implementation of the Dormand-Prince embedded RK method with an adaptive step size. It will need to store both the independent variable and the solution vector for each successful iteration. Be sure to store the time step used on a successful iteration **before** it is updated for the next iteration.

   If struggling to implement an adaptive step size method, and you want to progress with this task, you can instead solve using the classic RK4 method with a fixed time step of $h = 10^{-3}$ (though you will not receive full marks for this function).

3. Use the Dormand-Prince method to solve the Lorenz system with the initial conditions: $x(0) = y(0) = z(0) = 1$, system parameters $\sigma = 10$, $\rho = 28$ and $\beta = \frac{8}{3}$, time parameters $t_0 = 0$ and $t_1 = 40$, an error tolerance of $\epsilon_0 = 10^{-5}$ and an initial time step of $h = 10^{-3}$.

4. Produce a phase plot of the solution in the xz-plane, which will show the famous "Lorenz butterfly". There are two distinct states to which the solution trends, known as the Lorenz attractor. This is an example of a *strange* attractor as it has a fractal structure i.e. every path around an attractor state is unique.

5. Produce a timeseries plot of each solution variable i.e. $x(t)$, $y(t)$ and $z(t)$. You should notice that the solution settles into an irregularly oscillating pattern. The solution will spend a seemingly random amount of time oscillating around each of two clear possible states. This randomness from a deterministic system (i.e. one that has no inherent noise or randomness in its governing equations) is an indicator of a *chaotic* solution.

6. Solve the system with a second set of initial conditions, $x(0) = y(0) = z(0) = 1.001$. Produce a new set of plots that overlay these new results with your previous results. While initially very similar, the two solutions will reach a point of randomness where $x(t)$, $y(t)$ and $z(t)$ look very different. This high degree of sensitivity to the initial conditions is a further indicator of a chaotic system.

Hand-In

- Plots from Step 4 and 5. Write a brief comment describing whether you expect the solution to settle down towards a steady-state in the long-term.

- Plots from Step 6. Write a brief comment on whether you expect the solution to be highly sensitive to the numerical accuracy of the numerical solution method. Therefore, is it possible to accurately predict that exact values of x, y and z at a given time in a chaotic system such as this?