ENGSCI 331 - Computational Techniques 2

Ordinary Differential Equations - Lab Assignment

Semester Two 2025

Introduction

Background and Lab Objective

This lab will focus on the numerical solutions of ordinary differential equations (ODEs). There are multiple tasks, all of which should be completed.

Lab Files and Code:

The lab files are all available from Canvas:

- `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, and use these functions where provided. 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.

Please ensure that all code submitted is your own work. Each student's submissions will be cross-checked for evidence of plagiarism.

You may assume that the person marking your code has knowledge of what you are attempting to do, and therefore you can avoid e.g. excessive comments in the code providing step-by-step explanation of how the numerical methods work. Some amount of commenting is still appropriate (e.g. to help structure any complicated scripts or functions), but the point of the assignment is not to assess coding style. However, at least make sure that your functions include header comments / docstrings.

Hand-In Items:

The tasks have specific workings, answers, figures etc that you should hand-in, **in addition to your code**. These are clearly indicated in this lab document. Your hand-in items should be compiled into a brief report named `report_[your upi].pdf/doc/docx`. A single report containing material for all three tasks is appropriate. Please upload this **in addition to your zipped code** as part of your assignment submission.

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: Execution of Python code will pause when a plot is shown on screen. You will need to close the plot before the code will continue. Control statements with Boolean variable(s) can be used to control whether a particular optional block of code should be executed, for example code that produces and shows a figure on screen. This may be easier than commenting/uncommenting lines.

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

Coding Tip #4: Be careful when assigning large numerical values using scientific or 'E-notation'. `10e6` in Python represents 10×10^6 (i.e. 10^7), not 10^6.

Coding Tip #5: 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 #6: 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 Jump

Difficulty: ★★☆☆☆

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 material stiffness. *Short* and *Regular* cords have a length of 15 and 20 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 stiffness, for example:

- *SHORT60* is a cord of length 15 m with a spring constant of 60 N m^{-1}.
- *REG90* is a cord of length 20 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 you can implement it manually.

- When the cord is taut (i.e., stretched, not slack) spring and damping forces will also

apply:

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

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}).

- For simplicity, the model does **not** account for any physical interaction between the bungy jumper and the water. (Buoyancy, additional drag, etc.)

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

   You will need to code 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.

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

2. Complete the function `def explicit_rk_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. Remember that other model systems may also involve a different number of equations.

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

3. 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: $h = 0.1$ (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}.

4. 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.

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

   - The vertical displacement **and** velocity against time.
   - The vertical velocity against displacement 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: Task 1

- Plot from Step 4. Write a brief comment to justify which cord should be chosen.
- Plots from Step 5. Colliding with water at high speeds can be dangerous. If the jumper has a height of 1.8m and lands in the water head-first, at what time after jumping and with what velocity does the jumper first impact the water when using the chosen cord? (Use your simulation output to attempt a precise answer rather than eyeballing the plot).
- 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 effect this will have on the jump. (Again, refer to specific numerical results).
- *Remember to hand in your code!* Refer to the hand-in instructions at the beginning of the assignment.

Task 2: Three-Body Problem

Difficulty: ★★★★☆

Background and Objective

Bodies in space orbit around each other according to the gravitational forces described by Newton's laws. Closed-form solutions for this motion only exist for two-object models. To solve examples of the so-called "three-body problem", numerical methods must be used.

The "three-body" problem generalises to a 3D system with any number of objects. For simplicity, we consider motion of exactly three bodies in two dimensions only. (I.e., the bodies are assumed to lie within the x-y plane).

Newton's second law relates the forces acting on body i to its acceleration:

$$F_i = m_i r''$$

Where F_i is the net force acting on the body i, m_i is the mass of body i and r_i is the position of object i. (r_i'' is thus the second derivative of r_i with respect to time, i.e. the acceleration),

Newton's law of gravitation states that the force acting on body i by body j is given by:

$$F_{ij} = g\frac{m_i \cdot m_j}{||r_i - r_j||_2^2} d_{ij}$$

Where $d_{ij} = \frac{r_j - r_i}{||r_j - r_i||_2}$ is a unit vector in the direction from body i to body j, $m_{i,j}$ is the mass of the respective body, and g is a physical constant. We will assume $g = 1$.

By the principle of superposition, the total force acting on body i is the sum of the interactions between body i and all other bodies:

$$F_i = \sum_{i \neq j} F_{ij}$$

Steps to Complete

1. Complete the function `def derivative_threebody` in the provided functions file. Each second-order ODE for each of the three bodies can be decomposed into separate second-order equations for the forces in the x and y directions, and each of these second-order equations can be expressed as first-order systems. The cluster of 3 bodies can thus be modelled by a total of 12 first-order ODEs.

   Observe that each additional body within the system not only requires four additional first-order equations representing the forces the other bodies exert on that body, but also

adds terms to the other equations in the system representing forces the body exerts on the other bodies. This nonlinear increase in computation is typical of real-world phenomena, where the derivative calculation f for the first-order system can require an arbitrarily large number of operations, and is why we think about the computational cost of RK methods in terms of the number of f evaluations per iteration. It may have been difficult to appreciate this point by considering only simple ODEs where f requires few operations and can easily be calculated even by hand. (Such the lecture example, or Task 1).

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

   This function should be your own implementation of the Dormand-Prince embedded RK method with an adaptive step size. (The Dormand-Prince method is a popular general-purpose Runge-Kutta method, adapted for use in other software packages such as MATLAB's `ode45` function). The function 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. Note that unlike Task 1, this function only needs to implement the specific Dormand-Prince embedded method. It does not need to accept parameters for other embedded RK methods as function inputs. (However, it should still be able to solve a general first-order system with any number of equations).

   Because the timestep sizes of embedded methods are controlled automatically by the error tolerance of the algorithm, in a naive implementation there is no guarantee that the final solution step is solved exactly on the t_1 end-time specified by the function inputs. You should make sure that your code does ensure this result, without compromising the accuracy specified by the automatic timestep controller.

3. Use your implementation of the Dormand-Prince method to solve the three-body system with the following system parameters and initial conditions (provided in the task file): Initial Body Positions (x, y): B1 (1,3), B2 (-2,-1), B3 (1,-1). All initial velocities are zero. $g = 1$, M1 = 3, M2 = 4, M3 = 5. Time parameters $t_0 = 0$ and $t_1 = 18$, an error tolerance of $\epsilon_0 = 0.015$ and an initial time step of $h = 10^{-2}$. Use the timestep controller described in the lectures with a safety factor of 0.9. Additionally, modify this controller to incorporate a minimum timestep length $h_{min} = 10^{-7}$. (In practice, it would be sensible to pass this minimum value as an argument to the solver, but for the purposes of this exercise you can hard-code it into the function, i.e. do not modify the function definition provided).

Hand-In: Task 2

1. Because gravitational forces pull the three bodies towards each other, an intuitive outcome of the situation might be that the three bodies eventually collapse together at a single point and stop moving. Is this what we expect to happen in the formulation used in this

task? Explain why. (Hint: It may be useful to think in terms of the total energy of the system).

2. Produce a static plot showing the positions of the three bodies over the specified time period. Then, re-run the simulation with slightly different system parameters; increase the mass of body 3 by 0.001. Produce a plot of this solution also. (Remember that all plots should include appropriate labels / annotations).

   Observe that even a slight variation in the initial configuration produces a very different solution, and consider how this situation differs from the Bungy Jump model. Systems with this property are described as 'chaotic' and are difficult to model accurately. You can experiment with altering the error tolerance and/or removing the minimum-step-size limit and see that trying to achieve even modest levels of accuracy requires extremely small step-sizes. (You do not need to hand in any results from this exploration).

3. You have used generative AI to produce an animation function to further illustrate your results (`def My_Gen_AI_3BP_Animation_Tool`), and you plan to show this output to your manager at *SatelliteFab* as part of a project depicting motion of asteroids. Your senior colleague suggests that you do not submit this, because the output of the animation tool is visually misleading. Explain your colleague's concern. (You do not need to alter this function or create an improved visualisation tool).

4. *Remember to hand in your code!* Refer to the hand-in instructions at the beginning of the assignment.

Task 3: Van der Pol Oscillator

Difficulty: ★★★☆☆

Background and Objective

The following ODE is known as the Van der Pol oscillator:

$$\frac{d^2y}{dt^2} - \mu\left(1 - y^2\right)\frac{dy}{dt} + y = 0$$

It describes a non-conservative oscillator with non-linear damping. Originally, the equation described the behaviour of a specific electrical circuit involving vacuum tubes, but it has found modelling applications in other domains. It becomes increasingly stiff as the parameter μ (which indicates the strength of the non-linear damping) increases, making it difficult to solve efficiently with explicit RK methods.

An alternative option would be to solve the ODE with the implicit backward Euler method. While not hightly accurate (only first-order accurate), the method is unconditionally stable and will therefore never suffer from numerical instability issues, regardless of the step size chosen.

As the ODE is non-linear, we are unable to formulate an explicit update equation from our implicit backward Euler method (as shown in lecture for a linear ODE). Instead, we must iteratively evaluate $\vec{y}_{(i+1)}^{(k+1)}$ in the backward Euler update equation, a process known as Fixed Point Iteration:

$$\vec{y}_{(i+1)}^{(k+1)} = \vec{y}^{(k)} + h\vec{f}\left(t^{(k+1)}, \vec{y}_{(i)}^{(k+1)}\right)$$

Note that the Van der Pol oscillator is also second-order, so we must solve it as a system of first-order ODEs. We must also decide on a way of estimating the initial value of $\vec{y}_{(0)}^{(k+1)}$ to begin our fixed point iteration, such as through the use of a single step of an explicit RK method. (Forward Euler is good enough). We must then iterate across i until convergence, which can be evaluated through use of a uniform test:

$$\frac{|\vec{y}_{(i+1)}^{(k+1)} - \vec{y}_{(i)}^{(k+1)}|}{1 + |\vec{y}_{(i+1)}^{(k+1)}|} < \epsilon$$

The objective of this task is to implement a backward Euler solver and use it to investigate the solution behaviour of the Van der Pol oscillator.

Steps to Complete

1. Write a function `def derivative_vanderpol` in the provided functions file. As no template code has been provided, you will need to write your own suitable header comments.

The function definition should be:

```
def derivative_vanderpol(t, y, mu):
```

and it should return an array of first-order derivatives, similar to the previous derivative functions.

You may find it helpful to first write by hand the second-order Van der Pol oscillator as a system of first-order ODEs.

2. Write a function `def backward_euler_solver` in the provided functions file. The function definition should be:

   ```
   def backward_euler_solver(func, y0, t0, t1, h, tol, max_iter, *args):
   ```

   and it should return t and $\vec{y}$, similar to the solvers from the previous tasks. The arguments `func, y0, t0, t1, h, *args` are defined similarly to the previous tasks and you may assume a fixed step size. The argument `tol` corresponds to the error tolerance when convergence testing on your estimate of $\vec{y}_{(i+1)}^{(k+1)}$. The argument `max_iter` corresponds to the maximum number of permitted iterations across i (for each k) when estimating $\vec{y}_{(i+1)}^{(k+1)}$. You can adjust the values of `tol` and `max_iter` based on system performance, to help prevent excessive computation time. Suggested values are `tol=1.e-3` and `max_iter=10`.

3. In `task3_ode.py`, use the backward Euler method to solve the Van der Pol oscillator with the initial conditions $y(0) = 1$ and $\frac{dy}{dt}(0) = 0$, a system parameter $\mu = 0$, and time span $t_0 = 0$ to $t_1 = 20$.

   Solve for two different fixed time steps of $h_a = 0.1$ and $h_b = 0.01$. Produce a phase plot of $\frac{dy}{dt}$ against y for each time step used. By considering the expected behaviour of the Van der Pol oscillator equation when $\mu = 0$, as we are solving in this case, consider the relative accuracy of these two solutions.

4. Using a fixed time step of $h = 0.01$ and the same initial conditions and time span as above, solve the Van der Pol oscillator for three different values of its free parameter: $\mu = 0$, $\mu = 2$ and $\mu = 4$. Overlay the phase plot ($\frac{dy}{dt}$ against y) for each solution onto a single figure, allowing for easier comparison between them. While the backward Euler method will not be perfectly accurate, this comparison should still indicate the impact of increasing μ on the non-linear oscillator.

   This system is an example of a relaxation oscillator i.e. one that has a repeating signal that is nonsinusoidal. You can observe this nonsinusoidal repeating behaviour in the plot of $y(t)$. (You do not need to submit this plot).

Hand-In: Task 3

- Plot(s) from Step 3. Write a brief comment describing which time step provides a more accurate solution. Additionally, explain what issue both solutions have by considering the theoretical result for when $\mu = 0$.
- Plot from Step 4. Based on the phase plot, comment on why the Van der Pol oscillator becomes stiffer, and therefore more difficult to solve explicitly, as μ increases.
- *Remember to hand in your code!* Refer to the hand-in instructions at the beginning of the assignment.