The University of Nottingham

SCHOOL OF MATHEMATICAL SCIENCES

SPRING SEMESTER 2024–2025

MTHS2008 ASSESSED COURSEWORK 2
NUMERICAL INTEGRATION AND FINITE DIFFERENCE METHODS

This coursework is for **Ahmed Ali** (student ID **20566668**) only.

Submission deadline: **5pm, Friday 9th May 2025**

This coursework contributes **20%** towards the overall grade for the module.

Rules:

Each student is to submit their own coursework.

You are allowed to work together and discuss in small groups (up to 4 people), but you must write all code by yourself.

Elements of your coursework questions have been randomised, so you will need to write code unique to this coursework sheet to answer questions correctly!

You must submit only the .py files requested in this question paper. Details of the required filenames are given within the questions. You are strongly encouraged to use the **Spyder IDE** (integrated development environment). Hence you should **not** write IPython Notebooks (.ipynb), and you should **not** use Jupyter. You should be using a version of Python 3 rather than Python 2.

You may adapt any code that we developed in class (i.e. you do not have to start from scratch).

Coursework template python files are available on the module homepage on Moodle. Download these first.

In each template file, the packages and libraries required to complete the question are listed. You are not permitted to use any libraries or packages beyond those included in each question's template file.

Marks breakdown:

This coursework is out of **100 marks**:

- **Outputs in Q1** – 20 marks;
- **Outputs in Q2** – 45 marks;
- **Outputs in Q3** – 25 marks;
- **Commenting and structure** – 10 marks.

To obtain the maximum marks for code commenting and structure:

- Your comments should be helpful to the reader; they should help make your program easier to navigate. Note that having too many comments is potentially as bad as having too few comments! The style of commenting we used in the example programs is what you should aim for.
- Your program structure should be: imports at the top (these should not need to change from the template files), followed by the requested function definitions, followed by anything else. You ought to remove or comment out any lines of code used to call and test your functions prior to submission.

Viewing Animations in Spyder:

Please see the first item on the module Moodle page "Installing Spyder" for details of how to alter settings to show matplotlib animations.

Guidelines for submitting the coursework correctly:

Take time to read the coursework questions in detail.

For full marks, your functions need to return the correct outputs for particular input arguments. Suggestions for tests you can use to check your code with are given but you are encouraged to test your code using your own examples also.

Please be aware that **we will test your functions** to check that they work and produce the desired output(s), both with the test given in the question and with different undisclosed data.

If your modules have filenames that differ to what we have explicitly asked for, then you risk losing marks.

- Therefore, please **do not** add your username or student ID number to your filename.

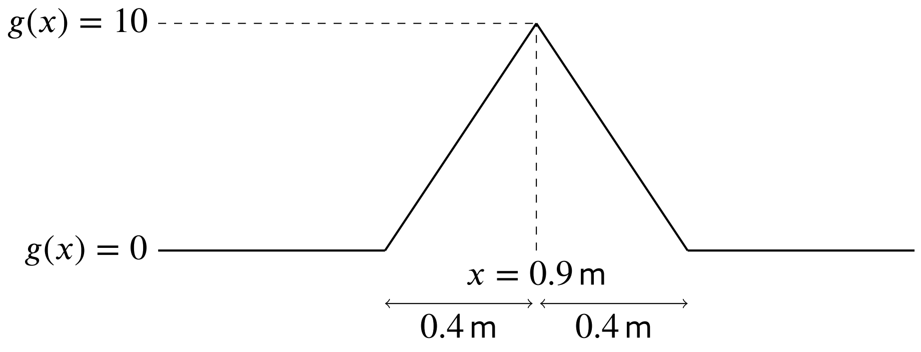
Your functions **must** have the **same** inputs and outputs as those specified in the questions (highlighted in blue), and in the **same order** as specified.

You can **check you have all the required methods and files** by running `test_call.py` and reading the output messages.

1. **Question 1: Double Integral**

   We are going to write a scalar function f of two variables, distance x (in metres) and time t (in seconds). The function $f(x,t)$ describes the output of a machine which behaves periodically, according to the following rules:

   - When time $t = 0.05$ the machine switches on (it is not on at $t < 0.05$).
   - It remains on for 0.05 seconds, then switches off (at $t = 0.1$).
   - This pattern repeats periodically every $T = 0.6$ seconds (i.e. the machine switches on again at $t = T + 0.05$, off at $t = T + 0.1$, on again at $t = 2T + 0.05$, off at $t = 2T + 0.1$ and so on ...)
   - When the machine is off then $f(x,t) = 0$.
   - When the machine is on then $f(x,t) = g(x)$, where $g(x)$ is the following 'triangular wave' pattern in space, symmetric around the given x value (not to scale):



   a) Begin with the template code `templateQ1.py` and modify it such that the function `f(x,t)` behaves as specified by the above rules.

      Note: you can use `f=np.vectorize(f)` to automatically convert your function so that it can be called with (and return) numpy vectors.

      Test: you can use the code at the bottom of the template file to plot $f(x,t)$ across space at different time points, and compare with the diagram above, to check that it is coded up correctly.

   b) Write a function using the **Composite (2D) Simpson's Rule** to calculate and return the value I of the following integral (for any function $f(x,t)$)

$$I = \int_0^{T_{\text{final}}} \int_0^{3.6} f(x,t)\,\mathrm{d}x\,\mathrm{d}t$$

      The function should be written as

      `calculate_double_simpsons_integral(Tfinal, f, Nx, Nt)`

      where the arguments are:

      – `Tfinal` — the upper limit of the time integral, T_{final};
      – `f` — the function, $f(x,t)$;
      – `Nx` — the number of strips to use in the x-direction;
      – `Nt` — the number of strips to use in the t direction.

The rest of the integral limits can be hardcoded into the function, rather than supplied as arguments.

Ensure your code raises a runtime error if either `Nx` or `Nt` is specified by the user as an odd number, using a python command like

```
raise RuntimeError("Nx must be even")
```

Test: your double integral should return 0.2 (or very close to it) when called with $T_{\text{final}} = T$ (the period of the machine activation), `f` $= f(x,t)$ as defined above, `Nx` $= 360$, and `Nt` $= 600$.

Be sure to use the template `templateQ1.py` which contains some clues and the method templates for you to fill in. **When it is complete, rename to `completeQ1.py` and submit it with this filename.**

[20 marks]

2. **Question 2: Time-dependent advection-diffusion equation**

We wish to solve the time-dependent advection-diffusion equation, as explored in Lectures 14 & 15.

Here, $u(x,t)$ represents the concentration of a chemical called *Unobtainium* (in units of moles per metre) dissolved in water, at position x along a pipe (measured in metres) at time t (measured in seconds). The concentration of unobtainium will obey the PDE

$$\frac{\partial u}{\partial t} - a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial u}{\partial x} = 0, \qquad (1)$$

where the diffusion coefficient $a = 0.15\,\text{m}^2\text{s}^{-1}$, the water in the pipe is flowing with velocity $b = -10.0\,\text{ms}^{-1}$, and our pipe covers the domain $\Omega : x \in [0, 3.6]$.

The concentration profile at time $t = 0$ is given as

$$u(x,0) = \begin{cases} 4x, & \text{for } 0 \leqslant< x < 0.5 \\ 4(1-x), & \text{for } 0.5 \leqslant x < 1.0 \\ 0, & \text{otherwise.} \end{cases} \qquad (2)$$

which should be coded into a method `u0(x)`.

Note: you can use `u0=np.vectorize(u0)` to automatically convert your initial condition function so that it can be called with (and return) a numpy vector.

Your aim in this question is to write code to run a finite difference scheme to solve Equation (1) up to time $t = T_{\text{final}}$, which will be called with a method

`advection_diffusion_backward_euler(Tfinal, Nx, Nt, u0)`.

The argument `Nx` specifies the number of space steps across the domain, `Nt` specifies the number of time steps, and `u0` specifies the initial condition.

- Your scheme should use second-order central differencing for the diffusion term. Your method should return a numpy 2d array (matrix) of the full solution $u(x,t)$ over each step in discretised time and space, as we did in the codes provided alongside Lectures 13 and 14. That is, it should be a matrix with $N_t + 1$ rows, and $N_x + 1$ columns, including the initial condition and values at both edges of the domain.

Note that we have a situation with a large Péclet Number, so you should use a first-order upwinding difference (in the appropriate direction for the value of b) for your advection term.

- Instead of our usual Dirichlet or Neumann boundary conditions, we are going to replace them with **periodic boundary conditions**. That is, the domain wraps around and joins back onto itself, so that $x_0 = x_N$ are the same point in space. You can think of this as $u(x)$ specifying the concentration as we go a distance x along the pipe, but now the pipe is flexible and we have joined the ends together.

- You should use the **Backward Euler Method** for time stepping.

- You may wish to approach this question by performing the following sub-tasks:
  - Write a semi-discretised equation for a point in the middle of the domain.
  - Work out how to represent periodic boundary conditions in the semi-discretised equations for each boundary, and thereby formulate the full matrix problem.
  - Work out how to fully discretise the matrix problem with the Backward Euler method for time steps.

    Note: since $x_0 = x_N$ then $u(x_0) = u(x_N)$ by definition, so u_0 and u_N will not both need to appear in the matrix problem. But, in order to plot the solution through the complete space, your method should still return a full solution for $(u_0, u_1, \dots, u_{N-1}, u_N)$.

  Tip: you may find it useful to refer to the `heat_equation_backward_euler.py` file from the Lectures section of the MTHS2008 Moodle page to get started.

Be sure to use the new template `templateQ2.py` which contains some clues and the method templates for you to fill in. **When it is complete, rename to `completeQ2.py` and submit it with this filename.**

Test: you should see the initial condition being smoothed out over time due to diffusion, and moving at a speed according to the advection term b. If things are coded up correctly, the simulation will settle to a steady state given by the black dashed line in the template animation when you run it for sufficient time with the values $N_x = 720$ and $N_t = 600$ (ten times fewer grid points in each direction should be sufficient for prototyping and checking your method is working).

[45 marks]

3. **Question 3: Time-dependent advection-diffusion equation with forcing**

   For this question, instead of Equation (1) we will instead solve a similar problem with a *forcing function* $f(x,t)$ on the right-hand side, representing the periodic injection of Unobtainium into a region of the pipe by a special machine:

$$\frac{\partial u}{\partial t} - a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial u}{\partial x} = f(x,t). \tag{3}$$

   We will use the same initial condition (2), values of a and b, domain Ω, and periodic boundary conditions as in Question 2, and also keep using the **Backward Euler** timestepping scheme.

a) Based upon your solution to Question 2, your task is to extend the finite difference scheme to deal with the forcing function, $f(x,t)$, and therefore to provide a method to solve Equation (3). The method must be called

`advection_diffusion_backward_euler_with_forcing(Tfinal, Nx, Nt, u0, f)`

and accept these arguments, where the first four arguments are the same as Question 2, and `f` is the new forcing function.

b) We will set the function $f(x,t)$ to be what you implemented in Question 1. Note that any mistakes in your forcing function `f(x,t)` $= f(x,t)$ from Question 1, or your initial condition `u0(x)` from Question 2, will not impact your score on this question, as we will run tests with model solutions for these.

Create a matplotlib plot of the total number of moles of Unobtainium contained in the domain over time, with t on the x-axis going from $t = 0$ up to $t = T_{\text{final}}$.

Return this plot from a function

`plot_total_unobtainium_over_time(u, Tfinal)`

(you should use 1D Simpson's Integration to calculate the number of moles in the spatial domain at any given time, where $N_x = 720$, and discretise in time such that $N_t = 600$).

Be sure to label your axes with variables and units, and provide an appropriate title.

c) Finally, have a look at your figure, when called with $T_{\text{final}} = 3.3$, using `plt.show()`. Think about the following:

- What do you notice about the total amount of Unobtainium?
- When does it increase, and why?
- When is it stable, and why?
- How does the number of moles at $T_{\text{final}} = 3.3$ relate to the double integral we did in Q1b) evaluated at $T_{\text{final}} = 3.3$?

Write the method

`comment_on_results()`

which will just return a string. In this string please write your thoughts on these questions in a few sentences.

Be sure to use the new template `templateQ3.py` which contains some clues and the method templates for you to fill in. **When it is complete, rename to `completeQ3.py` and submit it with this filename.**

[25 marks]

Before submission, be sure to **check you have all the required methods and files** by running `test_call.py` and reading the output messages.