ENGSCI 233

Assignment 5: Numerical Error and ODEs

Semester One 2024

Assignment Background and Objectives

The purpose of this assignment is to implement numerical methods for performing simple floating-point arithmetic and for the numerical solution of ordinary differential equations (ODEs), and to investigate their numerical error.

You have been provided the following files:

- `module_ass5.py` - a file in which to write the indicated functions. You can write additional helper functions in this file if required e.g. for plotting.
- `task1.py` - a "script" file in which to write any commands required to complete the various parts of Task 1.
- `task2.py` - a "script" file in which to write any commands required to complete the various parts of Task 2.
- `task3.py` - a "script" file in which to write any commands required to complete the various parts of Task 3.

Please complete all three tasks. After the first lecture on numerical error you should be able to write the **code** for Tasks 1 and 2.
After the second lecture on numerical error you should be able to answer the **theory** questions for Tasks 1 and 2 (i.e. the questions which ask you to comment on the cause of the numerical errors).
I recommend you leave Task 3 until after the material on RK methods has been presented in lectures.

You will be required to submit code, answers to posed questions, and mathematical workings as part of your submission (details at the end of this document).

Task 1: Numerical Error

1. The difference of two squares can be calculated by different methods:

$$z_1 = x^2 - y^2 \tag{1}$$
$$z_2 = (x-y)(x+y) \tag{2}$$

Though mathematically we would expect $z_1 = z_2$, this may not hold true in finite precision arithmetic with floating-point numbers.

Write a function `difference_squares` in `module_lab5.py` that takes as inputs two values, x and y, and calculates the difference between the squares using the two different methods in (1). The function should display to screen the value of both differences to 32 decimal places (dp).

Call your function `difference_squares` with the input values $x = 1+2^{-29}$ and $y = 1+2^{-30}$. Answer the following questions (which will form part of your submission):

1.1 Comment on how much the two values differ.

1.2 Explain why they differ.

1.3 Explain which result you think is more accurate.

2. Let $x = 1 + 1 \times 10^{-15}$ and $y = 1 + 2 \times 10^{-15}$. Now consider the value of

$$z = y - x \tag{3}$$

We can calculate this answer without the use of a computer

$$z = (1 + 2 \times 10^{-15}) - (1 + 1 \times 10^{-15}) \tag{4}$$
$$z = 2 \times 10^{-15} - 1 \times 10^{-15} \tag{5}$$
$$z = 1 \times 10^{-15} \tag{6}$$

Write a function `relative_error_subtraction` in `module_lab5.py` that calculates $z = x - y$ and displays numerical error information to the screen.

This function will have the IO:

- Input 1: `x` (float).
- Input 2: `y` (float).
- Input 3: `z_exact` (float). The exact value of $z = x - y$.
- No outputs

And will display the following to screen (e.g. using `print`):

- The value of x to 64 dp
- The value of y to 64 dp
- The approximate value of z to 64 dp
- The exact value of z to 64 dp
- The relative error to 16 dp

Make sure it is clear to the user what each displayed value relates to (rather than just displaying 5 values with no explanatory text).

Test your function with $x = 1 + 1e - 15$, $y = 1 + 2e - 15$. Answer the following questions (which will form part of your submission):

1.4 What is the relative error in z?

1.5 Explain what causes the relative error.

Task 2: Solving an ODE

Consider the initial value problem (IVP):

$$\frac{dy}{dt} + 5y = 2e^{-5t}\ , \qquad y\left(0\right) = 4 \tag{7}$$

The ODE is first-order, linear with constant coefficients, and non-homogeneous. It can therefore be solved using the mathematical analysis approach taught in ENGSCI 211 i.e. find the complementary function and particular integral, making sure to account for any resonance in the system, combine to form the general solution, and then apply the initial condition to find the exact solution.

Find the exact solution to this ODE mathematically (workings will form part of your submission).

Write a function `exact_solution_ode1` in `module_lab5.py` that calculates the exact solution y for a given value of t.

This function will have the IO:

- Input 1: `t` (float or array-like). Independent variable.
- Output 1: `y_exact` (float or array-like). Exact solution for given value of `t`.

Note that it should be able to handle either a float (e.g. a single value for t) or an array-like (e.g. list or 1D NumPy array) as an input.

With the exact solution available, it is possible to (fairly) accurately determine the numerical error of any approximate solution. An iterative numerical method for solving an ODE, such as the Euler method taught in ENGSCI 111, will incur numerical error in each solved step. A common approach to quantifying the numerical error in a multi-step solution is to average the absolute error across all steps. This is known as the *mean absolute error*, MAE:

$$MAE = \frac{\sum_{i=0}^{n-1} \left| y\left(t_i\right) - \tilde{y}\left(t_i\right) \right|}{n} \tag{8}$$

where n is the number of steps, y is the exact solution, and $\tilde{y}$ is the approximate solution. Note that in calculating the MAE for an ODE solution, the initial condition should **not** be included.

Write a function `mean_absolute_error` in `module_lab5.py` that can calculate the mean absolute error for an approximate solution.

This function will have the IO:

- Input 1: `y_exact` (1D NumPy array). Exact solution (excluding initial condition)
- Input 2: `y_approx` (1D NumPy array). Numerical approximation to the exact solution (excluding initial condition).
- Output 1: `mae` (float). The mean absolute error in the numerical solution.

Important: remember that when calling your function, you want to ensure that the input arrays do NOT contain the initial condition values at the start of the arrays, e.g. the first element of the `y_exact` array should be the first value calculated as part of the solution, rather than the initial known starting value.

The SciPy Python library has a built-in method, `solve_ivp`, for numerically solving ODEs. It uses a near-identical numerical method to that used by the MATLAB built-in function `ode45` encountered in ENGGEN 131 and ENGSCI 211. The method requires its first input to itself be a function that can be called to return $\frac{dy}{dt}$ for a given value of t and y. This works somewhat similarly to how a function handle can be passed in MATLAB, though recall that in Python everything is treated as an object. A function can therefore be passed as an argument to a function identically to a variable or any other kind of object.

Write a function `derivative_ode1` in `module_lab5.py` that can be used to calculate $\frac{dy}{dt}$ for the ODE defined in (7).

This function has the IO:

- Input 1: `t` (float). Value of the independent variable.
- Input 2: `y` (float). Value of the dependent variable.
- Output 1: `dydt` (float). Value of the first-order derivative.

Use `solve_ivp` and your `derivative_ode1` to solve the IVP in (7) over the time span $t = 0$ to $t = 2$ in `task2.py`. Note that by default the method will choose its own step size. Then, use your function `mean_absolute_error` to calculate the corresponding MAE (remember to exclude the initial condition in this calculation).

You may wish to consult the documentation page for `solve_ivp` to see how to use the method correctly. For example, note that the one output of `solve_ivp` is an object that has attributes `t` and `y` representing the independent and dependent variable, respectively.

Produce and save a plot of y against t, alongside the exact solution for comparison, on a single figure. Ensure that you include a legend that suitably labels each set of data points.

You may wish to consult the following matplotlib example of how to generate a simple plot with a legend.

Answer the following questions (which will form part of your submission):

2.1 What is the value of MAE. Based on this and a visual comparison of the exact and approximate solutions, comment on how accurate this solution is.

2.2 Note that the calculation of MAE may itself be subject to numerical error, regardless of the accuracy of the approximate solution. Briefly comment on two ways that numerical error may be incurred.

Task 3: Explicit Runge-Kutta Methods

In ENGSCI 211, you would have implemented the Euler and Improved Euler methods in MATLAB, which have the governing equations:

$$\begin{aligned}
y_E^{(k+1)} &= y^{(k)} + hf_0 && (9)\\
y_{IE}^{(k+1)} &= y^{(k)} + h\left(\frac{f_0}{2} + \frac{f_1}{2}\right) && (10)\\
f_0 &= f\left(t^{(k)}, y^{(k)}\right) && (11)\\
f_1 &= f\left(t^{(k)} + h, y^{(k)} + hf_0\right) && (12)
\end{aligned}$$

Both methods are members of the family of explicit Runge-Kutta (RK) methods. As both methods are fairly inaccurate, it is often necessary to solve an ODE with a higher-order explicit RK method, such as the "Classic RK4" method. Rather than hard-code a different solver for each method, the aim of this task is to implement a generalised solver that can implement any explicit RK method using the weights (α), nodes (β) and RK matrix (γ) that form its Butcher tableau:

$$\begin{array}{c|c} \beta^T & \gamma \\ \hline & \alpha \end{array} \tag{13}$$

The governing equation for an RK step is related to its Butcher tableau by:

$$y^{(k+1)} = y^{(k)} + h\sum_{i=0}^{n-1} \alpha_i f_i \tag{14}$$

$$f_i = f\left(t^{(k)} + \beta_i h, y^{(k)} + h\sum_{j=0}^{n-1} \gamma_{ij} f_j\right) \tag{15}$$

For example, the Butcher tableau of the Improved Euler method ($n = 2$) is given by:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \tag{16}$$

Write a function `explicit_rk_step` in `module_lab5.py` that performs one step of an explicit RK method based on its Butcher tableau.

The following assumptions can be made about this function:

- It only needs to work for a first-order ODE i.e. you do **not** need to account for having to solve a system of first-order ODEs.
- The RK method will **always** be explicit i.e. the step can be evaluated directly.

This function will have the following IO:

- Input 1: `f` (function). Function that can be called to return the first-order derivative.
- Input 2: `t` (float). Value of the independent variable at the start of the step.
- Input 3: `y` (float). Value of the dependent variable at the start of the step.
- Input 4: `h` (float). Step size along the independent variable.
- Input 5: `alpha` (1D NumPy array). Weights from the Butcher tableau.
- Input 6: `beta` (1D NumPy array). Nodes from the Butcher tableau.
- Input 7: `gamma` (2D NumPy array). RK matrix from the Butcher tableau.
- Output 1: `y_new` (float). Value of dependent variable at the end of the step.

It is recommended (though not assessed) to write unit test(s) to check that the step is calculated correctly. It is sensible to check at least two different explicit RK methods e.g. compare the results with the provided hard-coded RK method functions (no 100% guarantee they work correctly though...).

Write a function `explicit_rk_solver` in `module_lab5.py` that solves an ODE using an explicit RK method across a desired span of the independent variable.

This function may call your previously written function, `explicit_rk_step`, and will have the following IO:

- Input 1: `f` (function). Function that can be called to return the first-order derivative.
- Input 2: `tspan` (list). Two floats representing the start and end values of the independent variable, respectively.
- Input 3: `y0` (float). Initial condition for the dependent variable.

- Input 4: `h` (float). Step size along the independent variable.
- Input 5: `alpha` (1D NumPy array). Weights from the Butcher tableau.
- Input 6: `beta` (1D NumPy array). Nodes from the Butcher tableau.
- Input 7: `gamma` (2D NumPy array). RK matrix from the Butcher tableau.
- Output 1: `t` (1D NumPy array). Values of the independent variable for each step. The first and last values of this array should be equal to those in `tspan`.
- Output 2: `y` (1D NumPy array). Value of dependent variable for each step. The first value should be equal to the initial condition, the second value the first solved step, and so on, with the final value equal to the solution at the last value of the independent variable.

The following assumptions can be made about the function:

- It only needs to work for a first-order ODE.
- The span of the independent variable will be exactly divisible by the step size i.e. an integer number of steps with equal spacing.

It is recommend (though not assessed) to write unit test(s) to check that the solver is working correctly. A common mistake here is not returning the solution at **all** steps of the independent variable. For example, if `tspan = [0., 1.]` and `h=0.1`, each of the returned arrays should contain eleven values, with the final value of `t` being equal to `1` (not `0.9` as a lot of students accidentally do). Fortunately, this and other mistakes are something that can be tested for.

Investigating Numerical Error and Stability

You can now investigate the behaviour of the numerical error for several different explicit RK methods.

In `task3.py`, solve the initial value problem given in (7), over a time span $t = 0$ to $t = 2$, with three different RK methods (their Butcher tableau's were provided in lectures):

- Euler method.
- Improved Euler method.

- Classic RK4 method.

For each method, calculate the MAE (not including the initial condition) for the following step sizes:

```
h = [0.01, 0.025, 0.05, 0.1, 0.2, 0.25]
```

Produce and save a plot of MAE against h for all three methods on a single figure, ensuring that a legend is included that specifies which data points belong to which RK method.

Answer the following questions (which will form part of your submission):

3.1 Estimate from your plot the proportionality relationship between MAE and h for the Euler method. Comment on how this compares to the expected truncation error of the method.

3.2 Comment on an advantage and a disadvantage of using the Classic RK4 method over the Euler or Improved Euler methods.

This plot of MAE should give an indication as to the relative accuracy of the three different methods for solving this ODE. Note that you may also wish to produce a plot of the numerical solutions for one particular value of h, to investigate how each solution behaves relative to the exact solution.

Also in `task3.py`, solve the initial value problem again with all three methods, but this time with a step size of $h = 0.5$. Produce and save a plot of y against t for all three methods on a single figure, alongside the exact solution for comparison, ensuring that an appropriate legend is included.

3.3 Comment on the numerical stability of each method. How does this impact on the behaviour of the numerical error in each case?

3.4 In the context that we are unable to mathematically determine the stability condition for a given explicit RK method and ODE, briefly comment on how might we determine numerically whether or not the solution is numerically stable? *Hint:* you may want to look at the MAE for $h = 0.5$ in addition to the previously evaluated steps.

Submission Instructions

Your submission for this assignment on Canvas should include your code files:

- `task1-3.py` including any code used as part of Tasks 1–3.
- `module_ass5.py` including all of your functions written for Tasks 1–3. Ensure that you have included a docstring for each function.
- (optional) `test_ass5.py` including any unit tests you may have written as part of your quality control process. Note that no unit tests are assessed in this assingment, but you are encouraged to write and submit them.
- A word or pdf file containing your answers to each question (please label as 1.1, 1.2, etc...), and your mathematical workings for Task 2.