University of Nottingham
UK | CHINA | MALAYSIA

MATH2019 Introduction to Scientific Computation

— Coursework 4 (10%) —

Submission deadline: ***3pm, 2 May 2023***

Note: This is currently **version 2.3** of the PDF document. (1st May, 2023)

No more questions will be added to this PDF in future weeks.

This coursework contributes **10%** 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 (2 to 3 people), but *you must write your own coursework and program all code by yourself*.
- Please be informed of the UoN Academic Misconduct Policy (incl. plagiarism, false authorship, and collusion). This includes collusion/plagiarism with students from our Chinese and Malaysia campuses.

Coursework Aim and Coding Environment:

- In this coursework you will develop Python code related to algorithms that solve *numerical integration and numerical ODE problems* and you will study some of the algorithms' behaviours.
- As discussed in lectures, you should write (and submit) plain Python code (.py), and 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).

Timetabled Support Sessions (Tue 9am-10am, Coates C20 and Thu 1pm-2pm, ESLC C13):

- You can work on the coursework whenever you prefer.
- We especially encourage you to work on it during the (optional) timetabled computing sessions.

Piazza (`https://piazza.com/class/l7z5sbys7m74fq`):

- You are allowed and certainly encouraged(!) to also ask questions using Piazza to obtain clarification of the coursework questions, as well as general Python queries.
- However, when using Piazza, please ensure that you are not revealing any answers to others. Also, please ensure that you are not directly revealing any code that you wrote. Doing so is considered Academic Misconduct.
- When in doubt, please simply attend a drop-in session to meet with a PGR Teaching Assistant or the Lecturer.

Helpful Resources:

- Python 3 Online Documentation
- Spyder IDE (integrated development editor)
- NumPy User Guide
- Matplotlib Usage Guide

Some Further Advice:

- Write your code as neatly and readable as possible so that it is easy to follow. Add some comments to your code that indicate what a piece of code does. Frequently run your code to check for (and immediately resolve) mistakes and bugs.

Submission Procedure:

- Submission will open after Tuesday, 28 April 2023.
- To submit, simply upload the requested .py-files on Moodle. (Your submission will be checked for plagiarism using *turnitin*.)

Getting Started:

• Download the contents of the "Coursework 4 Pack" folder from Moodle into a single folder. (More files will be added in later weeks.)

Marking Notes (please read carefully):

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

• In all questions, the correctness of the following may be checked when marking:
  – The `type` of any requested function outputs;
  – The `numpy.shape` of any `numpy.ndarray` outputs;
  – The values of any numerical outputs;
  – Any plots produced (these will be marked manually);
  – Function "docstrings" (these will be marked manually);
  – Where asked for, the quality of the explanation of the results seen (these will be marked manually);
  – Comments/structure of code (these will be marked manually).

• `.py` files that include template functions, with correct inputs/outputs for all functions can be found on Moodle.

• The template `.py` files also contain some simple tests of your functions. You are encouraged to write your own tests of your functions in a separate file.

• Once the final part of the coursework has been set, `test_student_code.py` will be made available on Moodle. Download and save this file in the same folder as your solution `.py` files. Run `test_student_code.py` to generate in your folder the file `StudentCodeTestOutput.html`. Open that file with a browser to see exactly what tests were performed on your code. If your code fails to produce valid outputs using this facility, then it will fail to run through the automarker, and you risk scoring low output/plot marks.

• If your modules have filenames that differ to what has been explicitly asked for, then your modules will not run through the automarker, and you risk scoring low output/plot marks. Therefore, please do not add your username or student ID number to your filename.

• Every function you write should have a "docstring", the contents of which could be manually marked. In some questions you will use the docstring to explain your results.

• Every function you write should have appropriate comments and a clear structure, which could be manually marked.

• The automated tests for all questions will be 'timed out ' if they take too long to run. All questions should only need at most a few seconds to run. You are encouraged to run your code with much larger parameter values than in the tests shown in this document, thus ensuring your code does not take an excessive amount of time.

• When setting up a figure inside a function, you should first use the following syntax before any other plotting commands:

```
fig = plt.figure()
```

`fig` can then be returned from the function. An example of this is given in `approximations.py`.

• All figures produced should have a title, axis labels and, where appropriate, a legend.

▶ Numerical Integration

Suppose $f \in C^{\infty}[a,b]$, and we have a set of nodal points $x_j = a + jh$, for each $j = 0, 1, \ldots, n$, with $h = (b-a)/n$. Then, the Composite Trapezium rule for n subintervals ($\mathrm{CTR}(n)$) can be written as

$$\int_a^b f(x)\,\mathrm{d}x \approx \underbrace{\frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right]}_{\mathrm{CTR}(n)}. \tag{1}$$

1 **This is a warm up exercise. No marks will be awarded for it.**

- Its solution will be made available on the 21st March. The solution to this problem will be required in the question that follows.
- The file `numerical_integration.py` contains an unfinished function with the following definition:

```
def composite_trapezium(a,b,n,f)
```

The function should return as output:

– `integral_approx` - a floating point number of type `numpy.float64`.

- Complete the `composite_trapezium` function so that it evaluates $\mathrm{CTR}(n)$. By referring to equation (1), make sure input variable names correspond to the mathematical notation.
- Once you have completed `composite_trapezium`, test it by running `numerical_integration.py`. You should obtain:

```
integral_approx =

1.9835235375094544
```

2

(a) The file `numerical_integration.py` contains an unfinished function with the following definition: **[10 / 40]**

```
def romberg_integration(a,b,n,f,level)
```

This function returns:

– `integral_approx` - a floating point number of type `numpy.float64`.

- Complete the function to compute an approximation to the definite integral of a function, f over the interval $[a,b]$.
- The function should apply Richardson extrapolation to $\mathrm{CTR}(n)$ with the specified level of extrapolation (`level`), as detailed in lectures. Hence, if `level=1`, it should return the composite trapezium rule approximation with n strips. While,

if `level=2`, it should return the Romberg integration approximation based on $\mathrm{CTR}(n)$ and $\mathrm{CTR}(2n)$, *etc.* That is, it should return $R_{\text{level,level}}$ from the notes.

- Once you have written `romberg_integration`, test it by running `numerical_integration.py`. In this case, you should obtain:

```
level = 1, integral_approx = 1.5707963267948966
level = 2, integral_approx = 2.0045597549844207
level = 3, integral_approx = 1.9999831309459861
level = 4, integral_approx = 2.0000000162880416
```

(b) The file `numerical_integration.py` contains an unfinished function with the following definition

```
def compute_errors(N,no_n,levels,no_levels,f,a,b,true_val)
```

The function should return as output:

- `error_matrix` - a `numpy.ndarray` of shape `(no_levels, no_n)`;
- `fig` - a `matplotlib.figure.Figure`.

- Complete the `compute_errors` function to compute the errors of Romberg integration for different levels of extrapolation (stored in `levels`) and different numbers of subintervals (stored in `N`) where the true value of the integral is known and provided in `true_val`. Hence, your function should call `romberg_integration`.
- `error_matrix` stores the approximation errors computed for different values of `n` and `level`. Hence, the kth row of `error_matrix` contains the errors for `level = levels[k]` for all values `n` in `N`.
- The function should also plot the errors against the number of subintervals, `n` for each `level` using a log-log scale and return this figure in `fig`.
- Add a "docstring" at the top of your function definition, that describes and explains the results seen for the function $f(x) = 1/x$ integrated between 1 and 2.
- There is no need to include the usual function/input variables description in the docstring.
- Once you have written `compute_errors`, test it by running `numerical_integration.py`. In this case, you should obtain:

```
error_matrix =

[[2.00000000e+00 4.29203673e-01 1.03881102e-01 2.57683981e-02]
[9.43951024e-02 4.55975498e-03 2.69169948e-04 1.65910479e-05]
[1.42926818e-03 1.68690540e-05 2.47545428e-07 3.80915521e-09]
[5.54997967e-06 1.62880416e-08 5.96744876e-11 2.29150032e-13]]
```

► Numerical ODEs - One-step and Multistep methods

One-step methods are numerical techniques for approximating solutions to differential equations, where the value y_{i+1} is computed solely based on the information from the previous time level t_i. However, there are also multistep methods that use more historical information to determine y_{i+1}. A general m-step method, with step-length h is shown below

$$y_{i+1} = y_i + h \sum_{k=0}^{m-1} a_k f(t_{i-k}, y_{i-k}),$$

for some coefficients a_k.

The Adams-Bashforth family of methods is a group of multistep methods. The 1-step Adams-Bashforth method coincides with the Euler method, while the 2-step method utilizes information from two previous time points, resulting in a formula for y_{i+1} that includes terms y_i and y_{i-1}:

$$y_{i+1} = y_i + \frac{h}{2}[3f(t_i, y_i) - f(t_{i-1}, y_{i-1})],$$

where h is the step size and $y' = f(t, y)$ is the first order differential equation being solved.

3 In `numerical_ODEs_first_order.py` you will find the following function definition: **[15 / 40]**

```
adams_bashforth(f,a,b,ya,n,method)
```

The function should return as output:

- `t` - a `numpy.ndarray` of shape `(n+1,)`;
- `y` - a `numpy.ndarray` of shape `(n+1,)`.

• Complete the function so that it approximates the solution $y(t)$ for an initial value problem, over the interval `[a,b]`, with RHS function given in `f` and initial condition given in `ya`.

• `n` steps should be performed, of length $h = (b - a)/n$, so there should be `n+1` solution values, including the initial condition.

• The input `method` acts as a selector. If `method=1`, then the Euler method should be used, if `method=2` then the 2-step Adams-Bashforth should be used.

• The 2-step Adams-Bashforth method needs initial approximations y_0 and y_1. Hence, your implementation should use the Euler method to obtain y_1.

• Once you have written `adams_bashforth`, test it by running `numerical_ODEs.py`. In this case, you should obtain:

```
  t       True       Euler     Adams-Bashforth
-------------------------------------------------
1.85    4.942590    4.827726    4.936837
1.90    5.067053    4.947987    5.061173
1.95    5.188156    5.064887    5.182153
2.00    5.305472    5.178006    5.299348
```

► **Numerical ODEs - Second-order ordinary differential equations**

Second-order IVPs may be solved numerically by first converting them into a system of first-order IVPs. This method is called reduction of order. Let

$$y'' = f(t, y, y'), \quad a < t \leq b, \tag{2}$$
$$y(a) = \alpha, \quad y'(a) = \beta, \tag{3}$$

where $y(t)$ is to be found. We can rewrite the original IVP as a system of two first-order IVPs by introducing a new function $z(t) = y'(t)$:

$$y' = z, \quad y(a) = \alpha; \tag{4}$$
$$z' = f(t, y, z), \quad z(a) = \beta. \tag{5}$$

This system of first-order IVPs can be solved numerically using the Euler and the 2-step Adams Bashforth methods which were implemented in the previous question.

4

(a) In `numerical_ODEs_second_order.py` you will find the following function definition: **[15 / 40]**

```
def adams_bashforth_2(f, a, b, alpha, beta, n, method):
```

The function should return as output:

- `t` - a `numpy.ndarray` of shape `(n+1,)`;
- `y` - a `numpy.ndarray` of shape `(n+1,)`.

• Complete the function so that it approximately solves (2)-(3) by applying either the Euler method or the two-step Adams Bashforth method to (4) and (5) simultaneously.

• As before, the input `method` acts as a selector, so that, if `method=1`, the Euler method is used and if `method=2` the 2-step Adams-Bashforth method is used.

• `n` steps of the method should be applied and all other inputs to the function are as per (2)-(3)

• *Hint*: within `adams_bashforth_2` you could define a new function `f_system` that sets up as a vector the right hand sides of the differential equations in (4) and (5). Then, code from Q3 can be copied and used with only minimal modification.

(b) In `numerical_ODEs_second_order.py` you will find the following function definition:

```
def compute_ode_errors(n_vals,no_n,a,b,alpha,beta,f,true_y):
```

The function should return as output:

– `error_y` - `numpy.ndarray` of shape `(2,no_n)`;

• Complete the function so that it computes the final error $e_n = |y(b) - y_n|$ for each `n` in `n_vals` for both the Euler method and the 2-step Adams-Bashforth method. These errors should be stored in the zeroth row of `error_y` for the Euler method and in the first row of `error_y` for the 2-step Adams-Bashforth method.
• The exact solution $y(t)$ should be provided in the function `true_y` and other inputs are as in part (a).
• In the docstring, comment on the rates of convergence of the two methods. There is no need to include the usual function/input variables description in the docstring.

• Once you have written `adams_bashforth_2` and `calculate_ode_errors`, test them by running `numerical_ODEs_second_order.py`. In these cases, you should obtain:

```
  t       True       Euler     Adams-Bashforth
-------------------------------------------
0.93    0.956506    0.967227    0.956963
0.95    0.975196    0.986426    0.975651
0.98    0.993226    1.004962    0.993679
1.00    1.010584    1.022823    1.011035
```

and

```
errors_y =

[[1.23076169e-01 6.14728437e-02 3.06584716e-02 1.53039426e-02
  7.64501626e-03 3.82069566e-03]
 [4.70866112e-02 1.14384283e-02 2.83426094e-03 7.05050880e-04
  1.75778316e-04 4.38805209e-05]]
```

End of questions.