University of Nottingham
UK | CHINA | MALAYSIA

MATH2019 Introduction to Scientific Computation

— Coursework 3 (10%) —

Submission deadline: ***3pm, Tuesday, 7 Mar 2023***

Note: This is currently **version 4.1** of the PDF document. (20th February, 2023)
No more questions will be added to this PDF.

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 *polynomial interpolation and numerical differentiation problems* and you will study some of the algorithms' behaviour.
- 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, 21st February 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 3 Pack" folder from Moodle into a single folder. (More files may 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.

▶ Lagrange Polynomial Interpolation

Recall, given $p+1$ distinct points $\{\hat{x}_i\}_{i=0}^{p}$, the *Lagrange interpolating polynomials* are:

$$L_i(x) = \frac{\Pi_{\substack{j=0 \\ j\neq i}}^{p}(x-\hat{x}_j)}{\Pi_{\substack{j=0 \\ j\neq i}}^{p}(\hat{x}_i-\hat{x}_j)}$$

A function $f:\mathbb{R}\to\mathbb{R}$ can then be *interpolated* by the pth order polynomial function $p_p(x)$ given by

$$p_p(x) = \sum_{i=0}^{p} f(\hat{x}_i)L_i(x). \tag{1}$$

0 **Introductory question, not for credit**

- Let $f(x)=\cos(\pi x)+x$ and let $\hat{x}_0=-\frac{1}{2}$, $\hat{x}_1=0$ and $\hat{x}_2=\frac{1}{2}$.
- Find the (quadratic) Lagrange polynomials $L_0(x)$, $L_1(x)$ and $L_2(x)$ based on the points $\hat{x}_0$, $\hat{x}_1$ and $\hat{x}_2$. (You can do this either by hand, or by using Python). Use Python to plot these 3 Lagrange polynomials on one figure and make sure they look correct.
- Use (1) to construct the polynomial interpolant $p_2(x)$ of $f(x)$. (Again, you can do this either by hand, or using Python). Once you have this interpolant, use Python to plot it and compare against $f(x)$ over the interval $[-3,3]$.
- Go to the Geogebra demonstration https://www.geogebra.org/m/bwmpekfa and compare your answers with the one obtained there when $a=-\frac{1}{2}$ and $b=\frac{1}{2}$. Investigate using higher polynomials degrees and try different functions $f(x)$ over different intervals.

1 (**This is a warm up exercise. No marks will be awarded for it.** Its solution will be made available on the 7th February, however, you are thoroughly encouraged to attempt this question in advance of the questions for credit being released. The solution to this problem will be required in questions that follow.)

The file `lagrange_polynomials.py` contains an unfinished function with the following definition:

```
def lagrange_poly(p,xhat,n,x,tol)
```

This function returns:

- `lagrange_matrix` - a `numpy.ndarray` of shape `(p+1,n)`
- `error_flag` - an integer.

- Complete the function so that given a set of $p+1$ distinct nodal points $\{\hat{x}_i\}_{i=0}^{p}$ (stored in `xhat`) and a set of n evaluation points $\{x_j\}_{j=0}^{n-1}$ (stored in `x`), it will return a $(p+1)\times n$ matrix stored in `lagrange_matrix`, where the ijth entry of the matrix is $L_i(x_j)$.

• In addition, your function should also perform a check to make sure that the nodal points $\{\hat{x}_i\}_{i=0}^{p}$ are distinct. If the points are distinct, then the output variable `error_flag` should be set to 0, otherwise `error_flag` should be set to 1. Floating point numbers x and y should be considered equal if $|x - y| <$ `tol`.

• **No other checks on the inputs are required.**

• Once you have written `lagrange_poly`, test it by running `lagrange_polynomials.py`. You should obtain:

```
lagrange_matrix =

[[  6.5625   2.1875   0.3125  -0.0625   0.0625  -0.3125  -2.1875]
 [-11.8125  -2.1875   0.9375   0.5625  -0.3125   1.3125   8.4375]
 [  8.4375   1.3125  -0.3125   0.5625   0.9375  -2.1875 -11.8125]
 [ -2.1875  -0.3125   0.0625  -0.0625   0.3125   2.1875   6.5625]]

error_flag = 0
```

---

2 In `approximations.py` you will find the following function definition: **[6 / 40]**

```
def poly_interpolation(a,b,p,n,x,f,produce_fig)
```

The function should return as output:

– `interpolant` - a `numpy.ndarray` of shape `(n,)`;
– `fig` - a `matplotlib.figure.Figure` when the Boolean input `produce_fig` is `True` and `None` if `produce_fig` is `False`.

• Complete the function so that it evaluates the pth order polynomial interpolant $p_p(x)$ of a function f at a set of points $\{x_j\}_{j=0}^{n-1}$. The nodal interpolation points should be *uniformly spaced* over the interval $[\mathtt{a}, \mathtt{b}]$, including the endpoints.

• The function *must* call `lagrange_poly` from Q1 with `tol = 1.0e-10`.

• If `produce_fig` is `True` then the function should also plot (on the same set of axes) the function f evaluated at the points $\{x_j\}_{j=0}^{n-1}$ and the interpolant $p_p(x)$ evaluated at the same points. (Note, when testing this, it is assumed that the points in `x` form an increasing sequence.)

• Once you have written `poly_interpolation` test it by running `approximations.py`. Your output should be:

```
interpolant =

[2.64872127 2.79438712 2.83153269 2.80097589 2.74353466 2.7000269
 2.71127054 2.81808351 3.06128371 3.48168907]
```

► Lagrange Interpolation Errors

Recall, if polynomial interpolation of order p is used to approximate a function $f : [a, b] \to \mathbb{R}$, with nodal points $[\hat{x}_0, \ldots, \hat{x}_p]$, then for sufficiently smooth f , we have

$$\max_{x\in[a,b]} |p_p(x) - f(x)| \leq \max_{\xi\in[a,b]} \left| \frac{f^{(p+1)}(\xi)}{(p+1)!} \right| \max_{x\in[a,b]} \left| \Pi_{j=0}^{p}(x - \hat{x}_j) \right| .$$

3 In `compute_errors.py` you will find a function with definition **[6 / 40]**

```
def interpolation_errors(a,b,n,P,f)
```

The function should return as output:

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

• Complete the function to compute (approximations to) the error $\max_{x\in[a,b]} |p_{p_j}(x) - f(x)|$ for a range of polynomial degrees $\mathbf{P} = \{p_j\}_{j=0}^{n-1}$. The `numpy.ndarray error_matrix` should contain the errors when a uniform set of interpolating nodes is used. Hence, your function should call `poly_interpolation`.

Hint: In order to find an approximation to the error, evaluate the error $|p_p(x) - f(x)|$ for 2000 equally spaced points over $[a, b]$ and take the maximum of those.

• The function should also plot the errors against $\{p_j\}_{j=0}^{n-1}$, using `plt.semilogy` and return this figure in `fig`.

• Add a description at the top of your `interpolation_errors` function as a "docstring" that when `help(interpolation_errors)` is run will comment and explain the results when $\texttt{P} = \{1, 2, 3, \ldots, 10\}$ and

(a) $f(x) = \mathrm{e}^{2x}$, $[a, b] = [0, 1]$;

(b) $f(x) = \frac{1}{1+25x^2}$, $[a, b] = [-5, 5]$;

• There is no need to include the usual function description in `docstring`.

• Once you have written `interpolation_errors` test it by running `compute_errors.py`. You should obtain:

```
error_matrix =

[0.17863278 0.04634435 0.01368204 0.00431683 0.0014182 ]
```

► **Interpolation of Multivariate functions**

Lagrange interpolation can also be used for interpolation of multivariate functions. Suppose we have a function $f : \mathbb{R}^2 \to \mathbb{R}$ that we would like to approximate over the rectangular region $[a, b] \times [c, d]$. To interpolate with a polynomial of degree p, we can define a set of $p + 1$ distinct nodal points in both the x and y directions: $\{\hat{x}_i\}_{i=0}^{p}$ and $\{\hat{y}_i\}_{i=0}^{p}$, respectively. f can then be interpolated by the function

$$p_p(x, y) = \sum_{i=0}^{p} \sum_{j=0}^{p} f(\hat{x}_i, \hat{y}_j) L_{x,i}(x) L_{y,j}(y),$$

where $\{L_{x,i}(x)\}_{i=0}^{p}$ are the Lagrange polynomials associated with x nodal points and $\{L_{y,j}(y)\}_{j=0}^{p}$ are the Lagrange polynomials associated with the y nodal points.

4 In `approximations.py` you will find a function with the following definition: **[8 / 40]**

```
def poly_interpolation_2d(p,a,b,c,d,X,Y,n,m,f,produce_fig)
```

The function should return as output:

- `interpolant` - a `numpy.ndarray` of shape `(m,n)`;
- `fig` - a `matplotlib.figure.Figure` when the Boolean input `produce_fig` is `True` and `None` if `produce_fig` is `False`.

• Complete the function so that it evaluates the pth order polynomial $p_p(x, y)$ of a function at a set of grid points stored in the $m \times n$ arrays `X` and `Y`. Here, `X` and `Y` are assumed to have been created using the `numpy.meshgrid` function. Hence, the ijth component of `interpolant` should contain the value of $p_p(X_{ij}, Y_{ij})$. The nodal interpolating points in x and y should be uniformly spaced over the intervals $[a, b]$ and $[c, d]$ respectively, including the endpoints.

• The function *must* call `lagrange_poly` from Q1 with `tol = 1.0e-10`.

• If `produce_fig` is `True` then the function should produce a contour plot of the interpolant $p_p(x)$, which is returned in `fig`.

• Once you have written `poly_interpolation_2d` test it by running `approximations.py`. You should obtain:

```
interpolant =

[[2.71828183 3.03773178 4.23949621 7.3890561 ]
 [0.99999993 1.11751899 1.55962339 2.71828164]
 [2.71828183 3.03773178 4.23949621 7.3890561 ]]
```

▶ Numerical Differentiation

Recall that, given a point x and a set of distinct points $\{\hat{x}_j\}_{i=0}^{p}$, such that $\hat{x}_i = x$ for some $0 \leq i \leq p$ we can find an approximation to $f'(x)$ as follows:

$$f'(x) \approx p_p'(x) = \sum_{i=0}^{p} f(x_i) L_i'(x),$$

where $p_p(x)$ is the interpolant and $\{L_i\}_{i=0}^{p}$ are the usual Lagrange polynomials associated with the points $\{\hat{x}_i\}_{i=0}^{p}$.

5 In `lagrange_polynomials.py` you will find a function with the following definition: **[8 / 40]**

```
def deriv_lagrange_poly(p,xhat,n,x,tol)
```

The function returns:

- `deriv_lagrange_matrix` - a numpy.ndarray of shape `(p+1,n)`
- `error_flag` - an integer

• Complete the function so that given a set of $p+1$ distinct nodal points $\{\hat{x}_i\}_{i=0}^{p}$ (stored in `xhat`) and a set of n evaluation points $\{x_j\}_{j=0}^{n-1}$ (stored in `x`), it will return a $(p+1) \times n$ matrix stored in `deriv_lagrange_matrix`, where the ijth entry of the matrix is $L_i'(x_j)$.

• *Hint:* Derive a formula for $L_i'(x)$ by hand using the product rule then implement this. **Do not** use `SymPy`.

• In addition, your function should also perform a check to make sure that the nodal points $\{\hat{x}_i\}_{i=0}^{p}$ are distinct. If the points are distinct, then the output variable `error_flag` should be set to 0, otherwise `error_flag` should be set to 1. Floating point numbers x and y should be considered equal if $|x-y| <$ `tol`.

• **No other checks on the inputs are required.**

• Once you have written `deriv_lagrange_poly` test it by running `lagrange_polynomials.py`. You should obtain:

```
deriv_lagrange_matrix =

[[-17.875  -7.435  -1.315   0.485  -2.035  -8.875]
 [ 41.625  13.905  -0.855  -2.655   8.505  32.625]
 [-32.625  -8.505   2.655   0.855 -13.905 -41.625]
 [  8.875   2.035  -0.485   1.315   7.435  17.875]]

error_flag = 0
```

6 In `approximations.py` you will find a function with the following definition: **[6 / 40]**

```
def approximate_derivative(x,p,h,k,f)
```

The function returns:

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

• Complete the function so that it will evaluate the derivative of the pth order polynomial interpolant of a function f at a point x. This value is an approximation to $f'(x)$. The interpolating points should be equally separated so that $\{\hat{x}_j\}_{j=0}^{p}$ are such that

$$\hat{x}_j = \hat{x}_0 + jh, \quad j = 0, \ldots, p,$$

for $h > 0$ and should be positioned so that $\hat{x}_k = x$, where $0 \leq k \leq p$. That is, x coincides with one of the nodal points.

• The function should make use of `deriv_lagrange_poly` from Q**5** with `tol = 1.0e-10`.

• Once you have written `deriv_approx`, test it by running `approximations.py`. The outputs should be

```
k = 0, deriv_approx = -2.150454361444488
k = 1, deriv_approx = -2.1405845045118443
k = 2, deriv_approx = -2.1405845045118417
k = 3, deriv_approx = -2.1504543614444867
```

---

7 In `compute_errors.py` you will find a function with the following definition: **[6 / 40]**

```
def derivative_errors(x,P,m,H,n,f,fdiff)
```

The function returns:

– `E` - a `numpy.ndarray` of shape `(m,n)`

– `fig` - a `matplotlib.figure.Figure`

• Complete the function so that for a set of m **even** polynomial degrees $\mathbf{P} = \{p_i\}_{i=0}^{m-1}$ and a set of n widths $\mathbf{H} = \{h_j\}_{j=0}^{n-1}$, it will return an $m \times n$ matrix E, where

$$E_{ij} = |f'(x) - p'_{p_i,h_j}(x)|, \quad 0 \leq i \leq m-1,\, 0 \leq j \leq n-1$$

and $p_{p_i,h_j}(x)$ is the polynomial interpolant of order p_i with interval width h_j such that $x = \hat{x}_{p_i/2}$. *i.e.* $p'_{p_i,h_j}(x)$ is the **centred** $p_i + 1$ point approximation to $f'(x)$.

• Input `fdiff` is a function that computes the exact derivative $f'(x)$.

• The function *must* call `derivative_approximation` from Q**6**.

• The function should also plot $\{E_{ij}\}_{j=0}^{n-1}$ against $\{h_j\}_{j=0}^{n-1}$ for each p_i. A single set of axes with a logarithmic scale on both axes should be used.

• Add a "docstring" at the top of your `derivative_errors` function that comment on the results when

– `P = np.array([2,4,6,8])`
– `H = np.array([1/4,1/8,1/16,1/32,1/64,1/128,1/256])`

in the following two cases

(a) $f(x) = \mathrm{e}^{2x}$ and $x = 1$.

(b) $f(x) = \begin{cases} 0, & x \leq 0, \\ x^2, & x > 0 \end{cases}$ and $x = 0$.

• There is no need to include the usual function description in the docstring.

• Once you have written `derivative_errors`, test it by running `compute_errors.py`. You should obtain:

```
E =

[[3.96825397e-03 9.80392157e-04 2.44379277e-04]
 [2.64550265e-04 1.55617803e-05 9.58350104e-07]
 [4.32900433e-05 5.67028431e-07 8.49768522e-09]]
```

End of questions