MATH2019 Introduction to Scientific Computation

— Coursework 3 (10%) —

Submission deadline: ***3pm, Wednesday, 6th Mar 2024***

Note: This is currently **version 3.2** of the PDF document. (22nd February, 2024)

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).

Coursework Aim and Coding Environment:
- In this coursework you will develop Python code related to algorithms that solve *polynomial interpolation 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 (Thu 5pm-6pm, Pope A16, and Fri 2pm-3pm, Coates C20):
- 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/llxk0wzpvnpn3`):
- You are allowed (and 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 readably 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 Wednesday 21st February 2024.
- 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. These will be updated with each major update of the coursework.

• 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. These will be updated with each major update of the coursework.

• 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 from 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 `polynomial_interpolation.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:

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

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

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

$\boxed{0}$ **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 polynomial degrees and try different functions $f(x)$ over different intervals.

$\boxed{1}$ The file polynomial_interpolation.py contains an unfinished function with the definition **[8 / 40]**

```
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 `main.py`. You should obtain

```
lagrange_matrix =

[[ 3.0078125   0.7469375 -0.0154375 -0.0083125  0.0393125 -0.6015625]
 [-3.8671875   0.3954375  0.9725625  0.0511875 -0.1816875  2.4609375]
 [ 2.4609375 -0.1816875  0.0511875  0.9725625  0.3954375 -3.8671875]
 [-0.6015625  0.0393125 -0.0083125 -0.0154375  0.7469375  3.0078125]]

error_flag = 0
```

---

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

```
def uniform_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 $[\mathtt{a}, \mathtt{b}]$.

• 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 `uniform_poly_interpolation`, test it by running `main.py`. You should obtain

```
interpolant =

[ 1.64872127  0.42811005 -0.06148329 -0.01954368  0.35444396  0.8609947
  1.30062361  1.47384578  1.18117627  0.22313016]
```

$\boxed{3}$ Consider the following set of nonuniform nodal points $\{\hat{x}_i\}_{i=0}^{p+1}$ on the interval $(-1, 1)$:

$$\hat{x}_i = \cos\left(\frac{2i+1}{2(p+1)}\pi\right), \quad i = 0, \ldots, p \tag{2}$$

In `polynomial_interpolation.py` you will find a function with definition **[4 / 40]**

```
def nonuniform_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 to evaluate 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 *nonuniformly spaced* over the interval $[\mathtt{a}, \mathtt{b}]$ and be a *linearly* scaled and shifted version of those shown in (2), using a mapping $M : [-1, 1] \to [a, b]$ such that $M(-1) = a$ and $M(1) = b$.

• The function *must* call `lagrange_poly` from Q1 and use `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.

• Once you have written `nonuniform_poly_interpolation`, test it by running `main.py`. You should obtain

```
interpolant =

[ 1.84005546  0.52779343 -0.01656702 -0.00313432  0.35798309  0.85667679
  1.28283834  1.42635931  1.07713126  0.02504578]
```

▶ **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

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

$\boxed{4}$ In `polynomial_interpolation.py` you will find a function with definition **[9 / 40]**

```
def compute_errors(a,b,n_p,P,f)
```

The function should return as output:

- `error_matrix` - a `numpy.ndarray` of shape $(2, n_p)$.
- `fig` - a `matplotlib.figure.Figure`.

• Complete the function to compute (approximations to) the error $\max_{x\in[a,b]} |p_{p_k}(x) - f(x)|$ for a range of polynomial degrees $\mathbf{P} = \{p_k\}_{k=0}^{n_p-1}$. Row zero of `error_matrix` should contain the errors when a uniform set of interpolating nodes is used, while row one should contain the errors when the nonuniform set of interpolating nodes is used. Hence, your function should call both `uniform_poly_interpolation` and `nonuniform_poly_interpolation`.

In order to find an approximation to the error, evaluate the error $|p_p(x) - f(x)|$ for 2500 equally spaced points over $[a,b]$ and take the maximum of those.

• The function should also plot on the same axes the uniform and nonuniform errors against $\{p_k\}_{k=0}^{n_p-1}$, using `plt.semilogy` and return this figure in `fig`.

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

```
error_matrix =

[[0.96153446 0.64622887 0.70700961 0.43835605 0.43268858]
 [0.92592192 0.60059757 0.75029616 0.4020164  0.55590751]]
```

• Add a description at the top of your `compute_errors` function as a "docstring" that, when `help(p_int.compute_errors)` is run, will briefly comment on and explain the results when `P` $= \{1,2,3,\ldots,40\}$ and

(a) $f(x) = e^x$, $[a,b] = [-1,1]$.

(b) $f(x) = \sin(\pi x) + \frac{1}{10}\cos(8\pi x)$, $[a,b] = [-1,1]$.

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

▶ **Piecewise Polynomial Interpolation**

Recall, given a function $f : [a,b] \to \mathbb{R}$, we can construct its piecewise polynomial interpolant of order p by splitting $[a,b]$ up into uniform subintervals and applying the Lagrange interpolant of order p on each subinterval. Hence, using m subintervals $\{[\tilde{x}_{i-1}, \tilde{x}_i]\}_{i=1}^m$, the piecewise interpolant $S_p^m(x)$ satisfies:

$$S_p^m(x)|_{[\tilde{x}_{i-1},\tilde{x}_i]} = p_p^i(x), \quad i = 1, \dots, m$$

where $p_p^i(x)$ is the polynomial interpolant of $f(x)$ on $[\tilde{x}_{i-1}, \tilde{x}_i]$.

$\boxed{5}$ In `polynomial_interpolation.py` you will find a function with definition **[5 / 40]**

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

The function should return as output:

- `p_u_interpolant`, `p_nu_interpolant` - two `numpy.ndarrays`, each 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 to compute two **continuous** piecewise polynomial approximations of order p of a function $f(x)$ with m uniformly spaced subintervals of width $(b-a)/m$. The two approximations should be constructed by making calls to both `uniform_poly_interpolation` and `nonuniform_poly_interpolation` on each subinterval.

• Here, `x` is a vector of `n` points (assumed in the interval $[a,b]$) at which the piecewise interpolant $S_p^m(x)$ should be evaluated. The array `p_u_interpolant` should contain the interpolant values when a uniform set of interpolating nodes is used, while the array `p_nu_interpolant` should contain the interpolant values when the nonuniform set of interpolating nodes is used.

• 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 both interpolants $S_p^m(x)$ evaluated at the same points.

• Once you have written `piecewise_interpolation`, test it by running `main.py` with your own tests. The expected output of the test in `main.py` is not given here, but it is simple to replace the function `f` with one for which it is simple to check the correctness of your answer. **You do not have to submit your tests with your coursework.**

• It is strongly advised that you write and test your new code using `uniform_poly_interpolation` before modifying it so that it can also be used with `nonuniform_poly_interpolation`.

6 In `polynomial_interpolation.py` you will find a function with definition **[8 / 40]**

```
def compute_piecewise_errors(a,b,n_m,M,n_p,P,f)
```

The function should return as output:

- `u_error_matrix`, `nu_error_matrix` - two `numpy.ndarray`s, each of shape `(n_m,n_p)`.
- `u_fig`, `nu_fig` - two `matplotlib.figure.Figure`s.

• Complete the function to compute (approximations to) the error $\max_{x\in[a,b]} |S^{m_l}_{p_k}(x) - f(x)|$ for a range of polynomial degrees $\mathbf{P} = \{p_k\}_{k=0}^{n_p-1}$ and numbers of subintervals $\mathbf{M} = \{m_l\}_{l=0}^{n_m-1}$. Row l of `u_error_matrix` should contain the errors when a uniform set of interpolating nodes is used in each subinterval, while row l of `nu_error_matrix` should contain the errors when the nonuniform set of interpolating nodes is used in each subinterval.

Compute the approximation to the error by evaluating the error $|S^{m_l}_{p_k}(x) - f(x)|$ for 2500 equally spaced points over $[a,b]$ and take the maximum of those.

• The function should create two plots, one for the uniform errors and one for the nonuniform errors. Each plot should be of the errors against $\{m_l\}_{l=0}^{n_m-1}$, have a separate line for each $\{p_k\}_{k=0}^{n_p-1}$, and use `plt.loglog`. The figures should be returned in `u_fig` and `nu_fig`.

• Add a description at the top of your `compute_piecewise_errors` function as a "docstring" that, when `help(p_int.compute_piecewise_errors)` is run, will briefly comment on and explain the results when `P` $= \{1,2,3,4,5,6\}$ and `M` $= \{1,2,4,8,16,32,64,128,256,512,1024\}$, with

(a) $f(x) = \sin(\pi x) + \frac{1}{10}\cos(8\pi x)$, $[a,b] = [-1,1]$.

(b) $f(x) = \sqrt{|x - \frac{1}{3}|}$, $[a,b] = [-1,1]$.

You should also comment briefly in the "docstring" on the relative effectiveness of increasing p and m at reducing the error.

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

• Once you have written `compute_piecewise_errors`, test it by running `main.py` with your own tests. You do not have to submit these tests with your coursework but you should choose functions $f(x)$, which help you to confirm that your code is interpolating correctly.

END OF QUESTIONS