MATH3036 Scientific Computation and Numerical Analysis

— Coursework 1 (20%) —

Submission deadline: ***3pm,*** ***~~Monday, 3 March~~*** ***Wednesday 5 March*** ***2025***

Note: This is currently **version 4** of the PDF document. (19th February, 2025)

No more new questions will be added anymore.

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 (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 the (Petrov–) Galerkin method.
• You should write (and submit) plain Python code (.py), and you are encouraged to use the **Spyder IDE** (integrated development environment) or other IDE. Hence you should *not* write IPython Notebooks (.ipynb), and you should *not* use Jupyter.

How and Where to run Spyder:
• Spyder comes as part of the Anaconda package (recall that Jupyter is also part of Anaconda). Here are three options on how to run Spyder:
(1) You can choose to install Anaconda on you personal device (if not done already).
(2) You can open Anaconda on any University of Nottingham computer.
(3) You can open a UoN Virtual Desktop on your own personal device, which is virtually the same as logging onto a University of Nottingham computer, but through the virtual desktop. Then simply open Anaconda. Here is further info on the UoN Virtual Desktop.
• The A18 Computer Room in the Mathematical Sciences building has a number of computers available, as well as desks with dual monitors that you can plug into your own laptop.

Support Sessions:
• Fridays 10–11am, ESLC C13 Computer Room (Time-tabled drop-in)
• Fridays 11–12noon, ESLC C13 Computer Room (Lecturer's office hour drop-in)
• You can work on the coursework whenever you prefer.
• We especially encourage you to work on it during the (optional) drop-in sessions.

Moodle Forum/Blog (https://moodle.nottingham.ac.uk/mod/forum/view.php?id=8053188):
• You are allowed and certainly encouraged(!) to also ask questions to obtain clarification of the coursework questions, as well as general Python queries.
• However, when posting a query/response, please ensure that you are not revealing any answers. 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 the Lecturer.

Helpful resources:

• Python 3 Online Documentation – Spyder IDE (integrated development editor) – NumPy User Guide – Matplotlib Quick Start Guide – Moodle: Core Programming 2022-2023 – Moodle: Core Programming 2021-2022

• You are expected to have basic familiarity with Python, in particular: logic and loops, functions, NumPy and matplotlib.pyplot. Note that it will always be assumed that the package numpy is imported as np, and matplotlib.pyplot as plt.

Some Further Advice:

• Write your code as neatly and read-able 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.

• Coursework Questions with a "$\star$" are more tedious, but can be safely skipped, as they don't affect follow-up questions.

Submission Procedure:

• Submission will open after 14 Feb 2025.

• To submit, simply upload the requested .py-files on Moodle. (Your submission will be checked for plagiarism using *turnitin*.)

• Your work will be marked (mostly) automatically: This is done by running your functions and comparing their output against the true results.

▶ Petrov–Galerkin discretization for first-order differential equation

(**The questions 0, 1, 2 and 3 below are warm-up exercises. No marks will be awarded for them. Their solution will be available on Moodle as file `warmup_solution.py`.**)

Recall the problem (strong form):

$$u'(x) = f(x) \qquad \forall x \in \Omega := (0,1)\,, \qquad u(0) = 0\,,$$

for some given function f.

Consider the global polynomial approximation of degree N (with N an integer ≥ 1) given by the following Petrov–Galerkin discretization (see the end of Lecture 1):

$$\text{Find } u_h \in U_h \subset H^1_{(0}(\Omega): \qquad \int_0^1 u_h' v_h \,\mathrm{d}x = \int_0^1 f v_h \,\mathrm{d}x \qquad \forall v_h \in V_h \subset L^2(\Omega)\,.$$

where $U_h = \Big\{\text{polynomials of degree at most } N, \text{ and equal to 0 at } x = 0\Big\}$,

and $V_h = \Big\{\text{polynomials of degree at most } N-1\Big\}$.

Consider the following basis functions:

$U_h = \mathrm{Span}\{x, \frac{1}{2}x^2, \frac{1}{3}x^3, \ldots, \frac{1}{N}x^N\}$ and $V_h = \mathrm{Span}\{1, x, x^2, \ldots, x^{N-1}\}$.

$\boxed{0}$ Download from Moodle (Coursework 1 pack) the Python files (main.py and DEtools.py). Move the files into a folder, and open them in, e.g, Spyder (Anaconda)

Note: The file `DEtools.py` will be a *module* (recall that a module can be imported and contains Python definitions and statements). The idea is that `DEtools.py` contains your function definitions (algorithms), while `main.py` is a testing *script* that is used to test your module by importing it and calling its functions.

$\boxed{1}$ The file `DEtools.py` already contains code that will correctly compute the Petrov-Galerkin (PG) approximation for the case $N = 2$.

• Run main.py (the two parts related to Q1) to compute this PG approximation $(N = 2)$, and to plot this approximation as well as the exact solution.

• Compare the obtained results with the hand calculated approximation (see Problem at end of Lecture 1).

• Study how the PG approximation depends on the number of subintervals n used in the numerical integration (composite mid-point rule) of integrals defining the components of the right-hand vector $\underline{\mathbf{b}}$.

• Next change the data $f(x)$ and $u_{ex}(x)$. Does it work as expected?

$\boxed{2}$ Next, implement the case $N = 3$, by completing the following unfinished functions in DEtools.py:
AssemblePGmatrixN3() and AssemblePGvectorN3(f,n).
Test your functions by suitably adding code in the `main.py` file.

$\boxed{3}$ ($\star$) Next, implement the general case (N) by completing the unfinished functions:
AssemblePGmatrix(N) and AssemblePGvector(N,f,n).
Test your functions by suitably adding code in the `main.py` file.

▶ Petrov–Galerkin for a general first-order differential equation

(**Assessed questions start here.** Do make sure you've looked at Questions 0–3)

Next, consider the more general 1st-order problem:

$$u'(x) + c\,u(x) = f(x) \qquad \forall x \in \Omega := (0,1)\,, \qquad u(0) = 0\,,$$

where $c \in \mathbb{R}$ is a parameter. The corresponding Petrov–Galerkin discretization is:

$$\text{Find } u_h \in U_h \subset H^1_{(0}(\Omega) : \quad \int_0^1 \Big(u_h'\, v_h + c\, u_h\, v_h\Big)\,\mathrm{d}x = \int_0^1 f v_h\,\mathrm{d}x \quad \forall v_h \in V_h \subset L^2(\Omega)\,.$$

where the discrete spaces are the same as before, i.e., $U_h = \mathrm{Span}\{\varphi_1, \ldots, \varphi_N\}$ with $\varphi_j(x) = \frac{1}{j}x^j$, and $V_h = \mathrm{Span}\{\psi_1, \ldots, \psi_N\}$ with $\psi_i(x) = x^{i-1}$.

Let $u_h(x) = \sum_{j=1}^{N} u_j\, \varphi_j(x)$. The resulting $N \times N$ linear system is:

$$A\underline{\mathbf{u}} = \underline{\mathbf{b}} \quad \text{with} \quad A_{ij} = \frac{1}{i+j-1} + \frac{c}{(i+j)j}$$

in other words,

$$A := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} + c \begin{bmatrix} \frac{1}{2} & \frac{1}{6} & \frac{1}{12} & \cdots \\ \frac{1}{3} & \frac{1}{8} & \frac{1}{15} & \cdots \\ \frac{1}{4} & \frac{1}{10} & \frac{1}{18} & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix},$$

and the components b_i of vector $\underline{\mathbf{b}}$ are the same as before (see Question 1), i.e., composite mid-point rule approximations of the involved integrals $\int_0^1 f(x)\, \psi_i(x)\, \mathrm{d}x$ using n subintervals.

$\boxed{4}$ Note that the (updated) file DEtools.py contains the unfinished function: **[8 / 40]**

```
def AssemblePGmatrixGenDE(N,c):
    """
    DESCRIPTION

    Parameters
    ----------
    N : TYPE
        DESCRIPTION.
    c : TYPE
        DESCRIPTION.

    Returns
    -------
    A: TYPE
       DESCRIPTION.

    """

    return A
```

The function AssemblePGmatrixGenDE is to return the `numpy.ndarray A`, shape (`N`, `N`).

- Complete this function so that it provides the output as required.
- Test your function by running the `main.py` file (part Q4), which will run your function, then solve the system (using `AssemblePGvector` for $f(x) = 1$), and plot the approximation and exact solution. Try different N. (Do approximations seem to converge as $N \to \infty$?)

Hint: When $N = 3$ and $c = 4$ the output for `A` should be:

```
[[3.         1.16666667 0.66666667]
 [1.83333333 0.83333333 0.51666667]
 [1.33333333 0.65       0.42222222]]
```

and $\underline{\mathbf{u}} =$ `[ 0.89593831 -2.33651299 1.55717532]`

- Also complete the function documentation at the top of the function definitions, the so-called `doc_string`. Simply replace `DESCRIPTION` and `TYPE` by appropriate ones. This documentation should become visible, whenever one types e.g. `help(DE.AssemblePGmatrixGenDE)` or `DE.AssemblePGmatrixGenDE?`.

Hint: See some of the docstrings already given in DEtools.py. See also the useful online pandas guide to writing a docstring.

► Assessment

When submitting your coursework, you will only be asked to upload your `DEtools.py` file.

Marks can be obtained for your function AssemblePGmatrixGenDE for generating the required output, for certain set(s) of inputs. The correctness of the following may be checked:

- The `type` of outputs
- The `np.shape` (number of columns and rows) of outputs
- The values of outputs
- The output of "help(. . .)" on your function(s).

Note that in marking your work, different sets of input(s) may be used.

► Galerkin FEM for second-order differential equation

Next consider the following 2nd-order differential equation, with Dirichlet boundary conditions:

$$-u'' + c\,u = f \quad \text{in } \Omega := (0,1)\,, \qquad u(0) = 0\,, \quad u(1) = 0\,,$$

where $c \geq 0$ is a parameter. Consider the Galerkin FEM ($U_h = V_h$) based on the weak form:

Find $u_h \in V_h \subset H^1_{(0}(\Omega)$:

$$\underbrace{\int_0^1 \big(u_h' v_h' + c\,u_h\,v_h\big)\,\mathrm{d}x}_{=:b(u_h,v_h)} = \underbrace{\int_0^1 f v_h\,\mathrm{d}x}_{=:\ell(v_h)} \qquad \forall v_h \in V_h \subset H^1_{(0}(\Omega)\,.$$

with $V_h :=$ {continuous piecewise linears, equal to 0 at $x=0$ and $x=1$}.
Consider a uniform mesh with N equal-sized elements of width $h = 1/N$, nodes $x_0, x_1, \ldots, x_N$ (i.e., $x_i = ih$), and a basis of the usual hat-functions φ_i, i.e., $V_h = \mathrm{Span}\{\varphi_i\}_{i=1}^{N-1}$, see Lecture 2 SlidesAndNotes. (Note that there are $N-1$ basis functions in V_h, each centered at an interior node. This ensures that functions in V_h satisfy the boundary conditions.)

Let $u_h(x) = \sum_{j=1}^{N-1} u_j \varphi_j(x)$. It can be shown that the resulting $(N-1) \times (N-1)$ linear system is:

$$A\boldsymbol{u} = \boldsymbol{b} \quad \text{with} \quad A := \frac{1}{h}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} + \frac{c\,h}{6}\begin{bmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 \end{bmatrix},$$

and the components b_i of vector $\boldsymbol{b}$ are approximations of $\ell(\varphi_i)$ using the (composite) Mid-point rule:

$$b_i := \frac{h}{2}\Big(f(x_{i-\frac{1}{2}}) + f(x_{i+\frac{1}{2}})\Big) \qquad i = 1, 2, \ldots, N-1\,.$$

where $x_{i-\frac{1}{2}} = \frac{1}{2}(x_{i-1} + x_i)$ and $x_{i+\frac{1}{2}} = \frac{1}{2}(x_i + x_{i+1})$.

(turn page)

\boxed{5} Note that the file DEtools.py contains the following two unfinished functions: **[10 / 40]**

```python
def AssembleFEMmatrix(N,c):
    """
    DESCRIPTION

    Parameters
    ----------
    N : TYPE
        DESCRIPTION.
    c : TYPE
        DESCRIPTION.

    Returns
    -------
    A: TYPE
       DESCRIPTION.

    """

    return A
```

```python
def AssembleFEMvector(N,f):
    ...

    return b
```

The function AssembleFEMmatrix is to return the `numpy.ndarray A`, shape $(\mathtt{N}-1, \mathtt{N}-1)$, while AssembleFEMvector is to return the `numpy.ndarray b`, shape $(\mathtt{N}-1,)$.

• Complete these functions so that they provide the output as required.

Hint: See Lecture 2 SlidesAndNotes for useful python code (FEM approximations for a first-order differential equation).

• Test your function by running the `main.py` file (part Q5), which will run your functions, then solve the system, and plot the approximation and exact solution. Try different N. (Do approximations seem to converge as $N \to \infty$?)

Hint: When $N = 4$, $f(x) = \sin(\pi x)$, the output for `b` should be:
`[0.16332037 0.23096988 0.16332037].`

← Hint added on **14 Feb, 11:20am**

Hint: When $N = 4$, $c = 6$, $f(x) = 1$, the output for `u` should be:
`[0.06028369 0.07801418 0.06028369].`

• Also complete the function documentation for AssembleFEMmatrix at the top of the function definitions, the so-called `doc_string`. Simply replace `DESCRIPTION` and `TYPE` by appropriate ones. This documentation should become visible, whenever one types e.g. `help(FEM.AssembleFEMmatrix)` or `FEM.AssembleFEMmatrix?`.

▶ Assessment

When submitting your coursework, you will only be asked to upload your `DEtools.py` file.

Marks can be obtained for your functions AssembleFEMmatrix, AssembleFEMvector for generating the required output, for certain set(s) of inputs. The correctness of the following may be checked:

- The `type` of outputs
- The `np.shape` (number of columns and rows) of outputs
- The values of outputs
- The output of "help(. . .)" on your function(s).

Note that in marking your work, different sets of input(s) may be used.

(Turn page)

▶ **The heat equation: Backward Euler in time and Galerkin FEM in space**

Consider the PDE (in space x and time t) for $u = u(t,x)$, initial condition and boundary conditions:

$$\begin{aligned} u_t - \alpha u_{xx} &= 0 && \text{for } (t,x) \in (0,T) \times (0,1)\,, \\ u(0,x) &= u_0(x) && \forall x \in (0,1)\,, \\ u(t,0) = u(t,1) &= 0 && \forall t \in (0,T)\,. \end{aligned}$$

where $u_0 \in L^2(0,1)$ and $\alpha > 0$ are given. Note that the partial derivatives of u are denoted by:

$$u_t = \frac{\partial u}{\partial t} \quad \text{and} \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}\,.$$

Consider the Backward Euler method in time, with time-step τ. Hence consider approximations $u^k(x) \approx u(t^k, x)$, where $t^k := k\tau$, and $k = 0,1,2,\ldots,N_{\text{time}}$ (with $\tau N_{\text{time}} = T$).

Applying backward Euler's method to $u_t - \alpha u_{xx} = 0$ then gives the following differential equation with boundary conditions, for each $k = 1,2,3,\ldots$:

$$\begin{aligned} &\frac{u^k(x) - u^{k-1}(x)}{\tau} - \alpha u^k_{xx}(x) = 0 \quad \text{for } x \in (0,1)\,, \\ &u^k(0) = u^k(1) = 0\,, \end{aligned}$$

and where $u^0(x) = u(0,x) = u_0(x)$.

This differential equation (and boundary conditions) has the following weak formulation, to be solved for $u^k \in H^1_0(0,1)$ such that

$$\int_0^1 u^k v \,\mathrm{d}x + \tau\alpha \int_0^1 u^k_x v_x \,\mathrm{d}x = \int_0^1 u^{k-1} v \,\mathrm{d}x\,, \qquad \forall v \in H^1_0(0,1)\,.$$

The initial condition has the weak formulation:

$$\int_0^1 u^0(x) w(x) \,\mathrm{d}x = \int_0^1 u_0(x) w(x) \,\mathrm{d}x$$

for suitable test functions w.

To numerically solve these problem, we now consider a standard Galerkin FEM in space: That is, we use a mesh with nodes $x_i = ih$, $i = 0,1,2,\ldots,N$, (with $Nh = 1$) and a standard basis of *hat* functions $\{\varphi_j\}_{j=1}^{N-1}$ (i.e., the FE space consists of continuous piecewise linear approximations, which vanish for $x = 0$ and $x = 1$). Our approximation $u^k_h(x)$, at each k, is then finally defined by

$$u(t_k, x) \approx u^k(x) \approx u^k_h(x) := \sum_{j=1}^{N-1} u^k_j \varphi_j(x)\,,$$

where the coefficients $\underline{\mathbf{u}}^k = (u_1^k, u_2^k, \dots, u_{N-1}^k)$ are computed via:

$$(M + \tau\alpha K)\underline{\mathbf{u}}^k = M\underline{\mathbf{u}}^{k-1} \qquad \text{for } k = 1, 2, \dots, N_{\text{time}} \tag{1}$$

$$M\underline{\mathbf{u}}^0 = \underline{\mathbf{b}} \tag{2}$$

with $(N-1) \times (N-1)$ matrices

$$M = \frac{h}{6}\begin{bmatrix} 4 & 1 & & & \\ 1 & 4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & 4 & 1 \\ & & & 1 & 4 \end{bmatrix} \quad \text{and} \quad K = \frac{1}{h}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix},$$

and $b_i = \int_0^1 u_0(x)\varphi_i(x)\,\mathrm{d}x$ for $i = 1, 2, \dots, N-1$, which is to be approximated using the (composite) mid-point rule, as in Question 5.

$\boxed{6}$ The (updated) file DEtools.py contains four unfinished functions: **[12 / 40]**

```
def AssembleMatrix_M(N):

    return M
```

```
def AssembleVector_u0(N,u0_func):

    return u0_vec
```

```
def AssembleMatrix_K(N):

    return K
```

```
def HeatEqFEM(tau,alpha,Ntime,N,u0_func):
    u_array = np.zeros((Ntime+1, N-1))

    return u_array
```

- Complete these functions so that they provide the output as required:

The function AssembleMatrix_M is to return the above-defined `numpy.ndArray` `M` of shape $(\mathtt{N}-1, \mathtt{N}-1)$.

The function AssembleMatrix_K is to return the above-defined `numpy.ndArray` `K` of shape $(\mathtt{N}-1, \mathtt{N}-1)$.

The function AssembleVector_u0 is to return the `numpy.ndArray` `u0_vec` of shape $(\mathtt{N}-1,)$, which is the solution of Eq. (2) (hence requires the implementation of vector $\underline{\mathbf{b}}$ and the use of matrix M).

The function HeatEqFEM is to return the `numpy.ndArray` `u_array`, shape $(\mathtt{Ntime}+1, \mathtt{N}-1)$ of all approximations computed by the method, i.e., $\mathtt{u\_array}[k,:] = \underline{\mathbf{u}}^k$. The inputs are the time-step size `tau`, number `alpha`, the number of time steps `Ntime`, the number of elements `N`, and the initial-condition function `u0`.

Hint: Your function HeatEqFEM should call the other functions.

Hint 2: Use np.linalg.solve for solving linear systems. Recall that matrix-vector multiplication is done using np.matmul or the @ symbol.

- Test your function by running the `main.py` file (part Q6), which will run your functions, prints their output, and also plots all the computed approximations $u_h^k(x)$ in a figure.

Hint 3: When running the main file, with $\alpha = 1/2$, $N = 8$, $\tau = 0.02$, `Ntime` $= 16$, and $u_0(x) = 3x(1-x) + \sin(2\pi x)$, the output for `u0_vec` should be:

```
u0_vec = [ 1.05704103 1.58996228 1.43107453 0.75386598
  -0.01685095 -0.45771357 -0.39088446]
```

and the last approximation $\underline{\mathbf{u}}^k$, for $k = N_{\text{time}}$, is:

```
u_array[-1,:] = array([0.06769343, 0.12387186, 0.15948201, 0.16960386,
   0.15390558, 0.11598558, 0.062117 ])
```

▶ **Assessment**

When submitting your coursework, you will only be asked to upload your `DEtools.py` file.

Marks can be obtained for your functions AssembleVector_u0, AssembleMatrix_M, AssembleMatrix_K, HeatEqFEM for generating the required output, for certain set(s) of inputs. The correctness of the following will be checked:

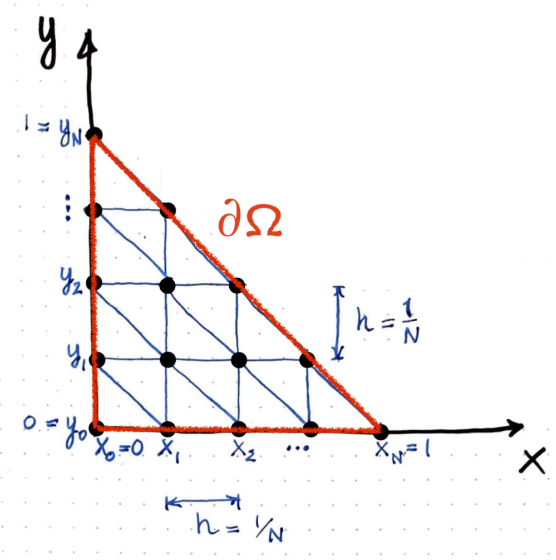
- The `type` of the outputs
- The `np.shape` (number of columns and rows) of these outputs
- The values of these outputs

Note that in marking your work, different sets of input(s) may be used.

(Turn page)

► **2D Poisson problem on a triangle-shaped domain:** Galerkin FEM on triangular meshes

(Note: There is no need to fully understand how to do Galerkin FEM for the below 2-D partial differential equation (PDE). What matters for the coursework is that you should be able to understand the description of the resulting linear system. If you wish to immediately get started on the relevant coursework questions, simply start reading from the next page onwards, starting at "*It turns out...*". In case you wish to read more about 2-D FEM, see, e.g., [Johnson, 1987, Section 1.4] or [Larson, Bengzon, 2013, Section 4.2].)



Consider the following PDE posed in a 2-D triangle-shaped domain $\Omega := \Big\{(x, y) \in \mathbb{R}^2 \,\Big|\, x > 0, y > 0, x + y < 1\Big\}$, subject to boundary conditions on the boundary $\partial\Omega$ (the boundary $\partial\Omega$ is indicated in red in the Figure):

$$\begin{aligned} -u_{xx} - u_{yy} &= 1 \qquad \text{for } (x, y) \in \Omega\,, \\ u &= 0 \qquad \text{on boundary } \partial\Omega\,. \end{aligned}$$

where the subscript indicates a partial derivative, e.g., $(\cdot)_x = \frac{\partial}{\partial x}(\cdot)$, $(\cdot)_y = \frac{\partial}{\partial y}(\cdot)$, $(\cdot)_{xx} = \frac{\partial^2}{\partial x^2}(\cdot)$, etc.
A suitable weak formulation is:

$$\text{Find } H_0^1(\Omega) : \qquad \int_\Omega \big(u_x\, v_x + u_y v_y\big)\, \mathrm{d}x\, \mathrm{d}y = \int_\Omega v\, \mathrm{d}x\, \mathrm{d}y \qquad \forall v \in H_0^1(\Omega)\,,$$

where $H_0^1(\Omega)$ is a 2-D Sobolev space, i.e.,

$$H_0^1(\Omega) := \left\{ w : \Omega \to \mathbb{R} \,\middle|\, \int_\Omega (w^2 + w_x^2 + w_y^2)\, \mathrm{d}x\, \mathrm{d}y < \infty\,, \text{ and } w(x, y) = 0 \text{ for } (x, y) \in \partial\Omega \right\}.$$

Consider its Galerkin FEM discretization on a mesh with N^2 triangles as illustrated in the Figure. One can assume $N \geq 4$. The triangular mesh is structured and has nodes (x_i, y_j), $i = 0, 1, 2, \ldots, N$, $j = 0, 1, 2, \ldots, N - i$, with $x_i = ih$, $y_j = jh$ and $h = 1/N$.

There are $\frac{1}{2}(N-2)(N-1)$ *interior* nodes (i.e., nodes *not* on the boundary $\partial\Omega$). These are counted starting from the bottom left, moving right, then continuing with the row of nodes above it. In other words, the order of interior nodes is: (x_1, y_1), $(x_2, y_1), \ldots, (x_{N-2}, y_1)$, (x_1, y_2), $(x_2, y_2), \ldots, (x_{N-3}, y_2), \ldots, \ldots, (x_1, y_{N-3})$, (x_2, y_{N-3}), (x_1, y_{N-2}).
The FEM approximation is then given by

$$u_h(x, y) = \sum_{k=1}^{\frac{1}{2}(N-2)(N-1)} u_k\, \varphi_k(x, y)\,,$$

where $\varphi_k(x, y)$ is the 2-D hat function associated to the k^{th} interior node (which $= 1$ at its node, and $= 0$ at the other nodes). Note: The coefficient u_k equals the value of the approximation $u_h(x, y)$ at the k^{th} interior node.

It turns out that the linear system for $\underline{\mathbf{u}} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \end{pmatrix}$ is (no need to verify this):

$$A\underline{\mathbf{u}} = \underline{\mathbf{b}}$$

where A has size $\frac{1}{2}(N-2)(N-1) \times \frac{1}{2}(N-2)(N-1)$ and is defined by the following block structure:

$$A = \begin{bmatrix} T_{N-2} & -I_{N-2,N-3} & O & \cdots & O \\ -I_{N-3,N-2} & T_{N-3} & -I_{N-3,N-4} & \ddots & \vdots \\ O & -I_{N-4,N-3} & \ddots & \ddots & O \\ \vdots & \ddots & \ddots & T_2 & -I_{2,1} \\ O & \cdots & O & -I_{1,2} & T_1 \end{bmatrix}$$

with T_m the following tridiagonal matrix of size $m \times m$, and $I_{k,\ell}$ is the matrix of size $k \times \ell$ with ones on its diagonal, and O is a matrix with zeros (and suitable size):

$$T_m = \begin{bmatrix} 4 & -1 & 0 & \cdots \\ -1 & 4 & -1 & \ddots \\ 0 & \ddots & \ddots & \ddots \\ \vdots & \ddots & -1 & 4 \end{bmatrix},$$

$$I_{k,\ell} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{bmatrix},$$

$$O = \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 0 & 0 & \ddots & \ddots \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & 0 & 0 \end{bmatrix}.$$

The vector $\underline{\mathbf{b}}$ is of size $\frac{1}{2}(N-2)(N-1)$ and $b_i = h^2$ for all $i = 1, 2, \ldots, \frac{1}{2}(N-2)(N-1)$.

(Turn page)

$\boxed{7}$ The (updated) file DEtools.py contains these two unfinished functions: **[6 / 40]**

```
def Assemble2dFEMvector(N):

    return b
```

```
def Assemble2dFEMmatrixN6():

    return A
```

• Complete these functions so that they provide the output as required:
The function Assemble2dFEMvector is to return the `numpy.ndArray b` of shape $(\frac{1}{2}(N{-}2)(N{-}1),)$ with components equal to $h^2 = 1/N^2$.

The function Assemble2dFEMmatrixN6 is to return the `numpy.ndArray A`, when $N = 6$, hence it has shape $(10, 10)$.

Hint 1: When $N = 6$, the matrix A is given by:

$$\begin{bmatrix} 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 4 \end{bmatrix}$$

Hint 2: You are allowed to "hard-code" the above matrix (i.e., simply input the entire matrix as a given array), instead of generating the matrix using existing numpy functions. If you wish to use existing functions, the following could be useful to you: `np.diag`, `np.eye`, `np.zeros`,, `np.block`, `np.vstack`, `np.hstack`, etc.
• Test your functions by running the `main.py` file (part Q7), which will run your functions, prints their output, and also solves the linear system. The output for $\underline{\mathbf{u}}$ should be:

```
[0.01896745 0.02404602 0.0208193 0.01214927 0.02404602 0.02861953 0.01930415 0.0208193 0.01930415 0.01214927]
```

$\boxed{8(\star)}$ **Note:** This question is very difficult, but worth only few marks.
The (updated) file DEtools.py also contains the unfinished function: **[4 / 40]**

```
def Assemble2dFEMmatrix(N):

    return A
```

This function is to return the `numpy.ndArray A` for *arbitrary* integer input N ($\geq$ 4), hence it should have shape $(\frac{1}{2}(N{-}2)(N{-}1)\,,\,\frac{1}{2}(N{-}2)(N{-}1))$.
• Complete this function.
Hint: You could use `for` loops to generate the block matrix. The following functions could also be useful to you: `np.diag`, `np.eye`, `np.zeros`,, `np.block`,

`np.vstack`, `np.hstack`, etc.

• Test your function by running the `main.py` file (part Q8), which will solve the system for some large value of N, and also plots the FEM approximation (using a scatter plot).

▶ **Assessment**

Marks can be obtained for your functions Assemble2dFEMvector, Assemble2dFEMmatrixN6, Assemble2dFEMmatrix for generating the required output, for certain set(s) of inputs. The correctness of the following will be checked:

- The `type` of outputs
- The `np.shape` (number of columns and rows) of these of outputs
- The values of the outputs

Note that in marking your work, different sets of input(s) may be used.

▶ Finished!