University of Nottingham
UK | CHINA | MALAYSIA

MATH2019 Introduction to Scientific Computation

— Coursework 4 (10%) —

Submission deadline: ***2pm, Monday, 26th April 2021*** + ***grace period***

Note: This is currently **version 2** of the PDF document.

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).

Piazza and Live Computing Classes:

- You are allowed to ask questions on Piazza to obtain clarification of the coursework questions, as well as general Matlab queries. This is certainly encouraged!
- 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 arrange a 1-on-1 meeting with the demonstrators or lecturers during the Live Computing Classes.**

Coursework Aim & Assessment:

- In this coursework you will develop Matlab code related to algorithms to perform polynomial interpolation & numerical differentiation and you will study some of the algorithms' behaviour.
- A total of 40 points can be obtained in this coursework.

Some Advice:

- You are expected to have basic familiarity with Matlab (in particular: vectors and matrices, plotting, logic and loops, function handles, writing functions and m-files).
- Helpful resources: Matlab Guide (by Tom Wicks), Matlab's Online Documentation
- Write your code as neatly and concisely as possible so that it is easy to follow.
- Add some comments to your scripts that indicate what a piece of code does.

Getting Started:

- Download the contents of the "Coursework 4 Pack" folder from Moodle. Now available

---

Submission Procedure:

- Submission will open: TBC.
- To submit, simply upload the required m-files on Moodle. (Your submission will be checked for plagiarism using *turnitin*.)
- Your m-files will be marked (mostly) automatically: This is done by running your functions and comparing their output against the true results.
- You can check the outputs that your m-files generate by running the file `GenerateYourOutputsCW4.m`, which can be found in folder "Coursework 4 Pack" from Moodle.

► **Numerical Differentiation**

Recall that, given a point x and a set of distinct points $\{x_j\}_{i=0}^p$, such that $x_i = x$ 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 $\{L_i\}_{i=0}^p$ are the Lagrange polynomials through the points $\{x_i\}_{i=0}^p$

1 • Write a *function* m-file called `derivLagrangePoly.m`, which, given a set of $p+1$ distinct nodal points $\{x_i\}_{i=0}^p$ and a set of n evaluation points $\{\hat{x}_j\}_{j=1}^n$, will return a $(p+1) \times n$ matrix, called `LDiff`, where the ijth entry of the matrix is $L_{i-1}'(\hat{x}_j)$.

- **No checks** on the inputs are required.
- Also add a brief description at the top of your `derivLagrangePoly` m-file. This description should become visible whenever one types: `help derivLagrangePoly`.
- *Hint:* Derive a formula for $L_i'(x)$ by hand involving sums and products and then implement this. **Do not** use the symbolic toolbox in Matlab.
- The function *must* have the following prototype:

```
function LDiff = derivLagrangePoly(p,x,n,xhat)
```

- Test your function (for example) by typing the following into the command window.

```
p = 3;
x = linspace(-0.5,0.5,4);
n = 6;
xhat = linspace(-1,1,6);
LDiff = derivLagrangePoly(p,x,n,xhat)
```

In this case you should obtain the matrix

$$\texttt{LDiff} = \begin{array}{rrrrrr} -17.8750 & -7.4350 & -1.3150 & 0.4850 & -2.0350 & -8.8750 \\ 41.6250 & 13.9050 & -0.8550 & -2.6550 & 8.5050 & 32.6250 \\ -32.6250 & -8.5050 & 2.6550 & 0.8550 & -13.9050 & -41.6250 \\ 8.8750 & 2.0350 & -0.4850 & 1.3150 & 7.4350 & 17.8750 \end{array}$$

Marks can be obtained for your `derivLagrangePoly` m-file for generating the required output, for certain set(s) of inputs. The correctness of the following will be checked: **[8 / 40]**

- The size and values of `L`
- The output of "help derivLagrangePoly".

(Run the file `GenerateYourOutputsCW4.m` to see these outputs generated for a particular set of inputs. Note that in marking your work, different input(s) may be used.)

---

2 • Write a *function* m-file called `polyDerivative.m` that will evaluate the derivative of the pth order polynomial interpolant of a function f at a point x. The interpolating points should be equally separated so that $\{x_j\}_{j=0}^{p}$ are such that

$$x_j = x_0 + jh, \quad j = 0, \ldots, p,$$

for $h > 0$ and should be positioned so that $x_k = x$, where k is an additional input to `polyDerivative.m`.

- The function should make use of `derivLagrangePoly.m` from Q1
- Also add a brief description at the top of your `polyDerivative` m-file. This description should become visible whenever one types: `help polyDerivative`.
- The function should have the following prototype:

```
function dInterp = polyDerivative(x,p,h,k,f)
```

Here, `dInterp` is a single value containing the approximation to the derivative at x.

- Test your function (for example) by typing the following into the command window.

```
p = 3;
h = 0.1;
x = 0.5
for k=0:3
        dInterp = polyDerivative(x,p,h,k,@(x) cos(pi*x)+x)
end
```

In this case, your output should be (shown horizontally for conciseness):

```
dInterp =          dInterp =          dInterp =          dInterp =

         -2.1505         -2.1406            -2.1406           -2.1505
```

Marks can be obtained for your `polyDerivative` m-file for generating the required output, for certain set(s) of inputs. The correctness of the following will be checked: **[4 / 40]**

- The correctness of `dInterp` for various inputs `x`, `p` and `k`.
- The output of "help polyDerivative".

(Run the file `GenerateYourOutputsCW4.m` to see these outputs generated for a particular set of inputs. Note that in marking your work, different input(s) may be used.)

3 • Write a *function* m-file called `derivativeErrors.m` that given a set of **even** polynomial degrees $\mathbf{P} = \{p_i\}_{i=1}^{l}$ and a set of widths $\mathbf{H} = \{h_j\}_{j=1}^{m}$, will return an $l \times m$ matrix E, where

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

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

• The function should also plot $\{E_{ij}\}_{j=1}^{m}$ against $\{h_j\}_{j=1}^{m}$ for each p_i. A single set of axes with a logarithmic scale on both axes should be used. The plot should have all relevant labels and legends.

• Add a description at the top of your `derivativeErrors.m` that when 'help derivativeErrors' is typed will comment on the results when $\mathbf{P} = \{2, 4, 6, 8\}$, $\mathbf{H} = \{1/4, 1/8, 1/16, 1/32, 1/64, 1/128, 1/256\}$, $f(x) = \mathrm{e}^{2x}$ and $x = 1$.

• The function must call `polyDerivative` from Q2 and should have the following prototype

```
function E = derivativeErrors(x,P,H,f,fdiff)
```

Here `fdiff` is a function that is the exact derivative of `f`.

• Test your function (for example) by typing the following into the command window.

```
P = [2,4,6];
H = [1/4,1/8,1/16];
x = 0;
format long
E = derivativeErrors(x,P,H,@(x)1./(x+2),@(x) -1./((x+2).^2))
```

In this case you should obtain

$$\mathrm{E} = \begin{matrix} 0.003968253968254 & 0.000980392156863 & 0.000244379276637 \\ 0.000264550264550 & 0.000015561780268 & 0.000000958350105 \\ 0.000043290043290 & 0.000000567028432 & 0.000000008497685 \end{matrix}$$

[4 / 40]

Marks can be obtained for your `derivativeErrors` m-file for generating the required output, for certain set(s) of inputs, as well as the plots and comments. The correctness of the following will be checked:

• The size and values of `E`;

• The error plots produced;

• The output of "help derivativeErrors" and in particular the quality of explanation for the outputs seen.

(Run the file `GenerateYourOutputsCW4.m` to see these outputs generated for a particular set of inputs. Note that in marking your work, different input(s) may be used.)

► Solving ODEs Numerically - θ-schemes

Suppose we wish to solve the Initial value problem:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y), \quad a \le t \le b,$$
$$y(a) = y_0.$$

The *Euler method* is part of a more general set of methods called θ-*schemes*, that take the form:

$$y_{i+1} = y_i + h\left[\theta f(t_i, y_i) + (1-\theta) f(t_{i+1}, y_{i+1})\right], \quad i = 0, 1, \ldots, N-1,$$

where $0 \le \theta \le 1$ and $h = \frac{b-a}{n}$. If $\theta = 1$, we obtain the Euler method. For every other choice of θ, we have an implicit equation for y_{i+1}. Setting $\theta = 0$ yields the *backward Euler* method.

4 • Write a *function* m-file called `thetaODESolver.m` that will implement a θ-scheme, given inputs a, b, f, N, y_0, θ and **optionally** $\dfrac{\partial f}{\partial y}$ and return a vector of the time-points $\{t_i\}_{i=0}^{N}$ and a vector of approximate solutions $\{y_i\}_{i=0}^{N}$.

• Root finding algorithms should be used to solve the implicit equations at each time step:

- If $\dfrac{\partial f}{\partial y}$ **is not** provided, the **natural** fixed point iteration should be used to solve the implicit equation at each time step and should be stopped when the difference between successive iterates is below a tolerance of 10^{-12}.
- If $\dfrac{\partial f}{\partial y}$ **is** provided, Newton's method should be used to solve the implicit equation at each time step. Again a tolerance of 10^{-12} between successive iterates should be used to stop Newton's method.
- You should write the fixed point and Newton iterations yourself, within the `thetaODESolver` function.
- A maximum of 1000 iterations should be performed at each time-step.
- If the maximum number of iterations is reached, then the code should stop immediately and return an `errorFlag = 1`. Otherwise, the code should return `errorFlag = 0`.

• Also add a brief description at the top of your `thetaODEsolver` m-file. This description should become visible whenever one types: `help thetaODEsolver`. As well as the usual requirements, the description should give the user a sufficient condition on f under which the **fixed point iterations** will converge.

- The function should have the following prototype

```
function [t,y,errorFlag] = thetaODESolver(a,b,f,N,y0,theta,df)
```

Here, `df` is an **optional** argument that provides $\frac{\partial f}{\partial y}$.

- Test your function (for example) by typing the following into the command window:

```
a = 0; b = 2;
N = 6;
theta = 0.25;
f = @(t,y) y-t.^2+1;
df = @(t,y) ones(size(t));
y0=0.5;
[t,y,errorFlag] = thetaODESolver(a,b,f,N,y0,theta)
[t,y,errorFlag] = thetaODESolver(a,b,f,N,y0,theta,df)
```

In both cases you should obtain:

$$\mathtt{t} = \quad 0 \quad 0.3333 \quad 0.6667 \quad 1.0000 \quad 1.3333 \quad 1.6667 \quad 2.0000$$

$$\mathtt{y} = \quad 0.5000 \quad 1.1296 \quad 1.9156 \quad 2.8288 \quad 3.8267 \quad 4.8485 \quad 5.8058$$

$$\mathtt{errorFlag} = 0$$

[12 / 40]

Marks can be obtained for your `thetaODESolver` m-file for generating the required output, for certain set(s) of inputs. Tests with and without `df` will be performed. The correctness of the following will be checked:
- The size and values of `t` and `y`
- The values of `ErrorFlag`
- The output of ”help thetaODESolver”, **including the sufficient condition for convergence of the fixed point iterations.**

(Run the file `GenerateYourOutputsCW4.m` to see these outputs generated for a particular set of inputs. Note that in marking your work, different input(s) may be used.)

---

5 • Write a *function* m-file called `thetaSchemeErrors` that given a set of N-values $\mathbf{N} = \{N_j\}_{j=1}^m$, a set of θ-values $\mathbf{T} = \{\theta_i\}_{i=1}^l$, a function $f(t, y)$ and true solution $y(t)$, will return an $l \times m$ matrix called E of the errors at the final time point t_{N_j}, so that

$$E_{ij} = |y(b) - y_{N_j}^{\theta_i}|,$$

where $y_{N_j}^{\theta_i}$ is the approximation to $y(b)$ using θ_i and N_j time steps.

• The function should also plot $\{E_{ij}\}_{j=1}^m$ against $\{N_j\}_{j=1}^m$ for each θ_i, $i = 1, \ldots, l$. A single set of axes with a logarithmic scale on both axes should be used. The plot should have all relevant labels and legends.

• Add a description at the top of your `thetaSchemeErrors.m` file that when 'help thetaSchemeErrors' is typed will comment on the results when $\mathbf{T} = \{0, 0.25, 0.5, 0.75, 1\}$ and $\mathbf{N} = \{4, 8, 16, 32, 64, 128, 256\}$, in the following case:

$$f(t, y) = \frac{\cos(t)}{3y^2}, \quad a = 0, \quad b = \frac{\pi}{2}, \quad y(a) = 1.$$

• The function must call `thetaODESolver` from Q4. (without the need for `df`)

• The function should have the following prototype:

```
function E = thetaSchemeErrors(a,b,f,true_y,y0,N,T)
```

• Test your function (for example) by typing the following into the command window.

```
a = 0; b = 2;
N = [4, 8, 16, 32];
T = [0.2,0.4,0.6,0.8];
f = @(t,y) y-t.^2+1;
true_y = @(t) (t+1).^2-exp(t)/2;
y0=0.5;
E = thetaSchemeErrors(a,b,f,true_y,y0,N,T)
```

$$E = \begin{matrix} 1.1351 & 0.4623 & 0.2147 & 0.1040 \\ 0.1400 & 0.1045 & 0.0600 & 0.0318 \\ 0.3892 & 0.1639 & 0.0746 & 0.0355 \\ 0.6896 & 0.3683 & 0.1921 & 0.0984 \end{matrix}$$

[4 / 40]

Marks can be obtained for your `thetaSchemeErrors` m-file for generating the required output, for certain set(s) of inputs, as well as the plots and comments. The correctness of the following will be checked:

- The size and values of `E`;
- The error plots produced;
- The output of "help thetaSchemeErrors" and in particular the quality of explanation for the outputs seen.

(Run the file `GenerateYourOutputsCW4.m` to see these outputs generated for a particular set of inputs. Note that in marking your work, different input(s) may be used.)

► Solving ODEs Numerically - Runge-Kutta Schemes

6 • Write a *function* m-file called `rungeKutta.m` that will implement Runge-Kutta methods to solve the following system of m first-order Initial Value Problems:

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \boldsymbol{f}(t, \boldsymbol{y}), \quad a \leq t \leq b,$$
$$\boldsymbol{y}(a) = \boldsymbol{y}_0.$$

• The function should take as input a, b, $\boldsymbol{y}_0$, the number of time steps to perform N, the number of equations in the system m, the function $\boldsymbol{f}$ and the `method` to be used, which can take the following values:

- `method = 1` - Forward Euler Method;
- `method = 2` - Modified Euler Method;
- `method = 4` - The RK4 method.

Note, $\boldsymbol{y}_0$ should be assumed to be a **column** vector of length m and $\boldsymbol{f}$ should be assumed to be a function that returns a **column** vector of length m.

• As output, the function should return a row-vector of length $N+1$ containing the nodal points $\{t_i\}_{i=0}^{N}$ and a matrix of size $m \times (N+1)$, where the jth row contains the approximations to the jth component of $\boldsymbol{y}$ at all the time points.

• Add a brief description at the top of your `rungeKutta` m-file. This description should become visible whenever one types: `help rungeKutta`.

• The function should have the following prototype:

```
function [t,y] = rungeKutta(a,b,f,N,y0,m,method)
```

• Test your function (for example) by typing the following into the command window.

```
a = 0; b = 1; N = 5; m = 3;
f = @(t,y) [y(1)-2*y(2)+t*y(3);-y(1)+y(2)+y(3)-t^2;...
        t*y(2)-y(2)+t^3];
y0 = [1;0;1];
method = 1;
[t,y] = rungeKutta(a,b,f,N,y0,m,method)
method = 2;
[t,y] = rungeKutta(a,b,f,N,y0,m,method)
method = 4;
[t,y] = rungeKutta(a,b,f,N,y0,m,method)
```

In all cases you should obtain:

$$t = 0 \quad 0.2000 \quad 0.4000 \quad 0.6000 \quad 0.8000 \quad 1.0000$$

For `method = 1`, you should obtain:

$$y = \begin{matrix} 1.0000 & 1.2000 & 1.4800 & 1.8753 & 2.4469 & 3.2950 \\ 0 & 0 & -0.0480 & -0.1853 & -0.4654 & -0.9602 \\ 1.0000 & 1.0000 & 1.0016 & 1.0202 & 1.0782 & 1.1992 \end{matrix}$$

For `method = 2`, you should obtain:

$$y = \begin{matrix} 1.0000 & 1.2400 & 1.5998 & 2.1562 & 3.0421 & 4.4788 \\ 0 & -0.0240 & -0.1319 & -0.3873 & -0.8841 & -1.7691 \\ 1.0000 & 1.0008 & 1.0150 & 1.0632 & 1.1666 & 1.3355 \end{matrix}$$

For `method = 4`, you should obtain:

$$y = \begin{matrix} 1.0000 & 1.2463 & 1.6209 & 2.2087 & 3.1570 & 4.7132 \\ 0 & -0.0272 & -0.1433 & -0.4170 & -0.9523 & -1.9148 \\ 1.0000 & 1.0018 & 1.0180 & 1.0694 & 1.1770 & 1.3491 \end{matrix}$$

Marks can be obtained for your `rungeKutta` m-file for generating the required output, for certain set(s) of inputs. ~~as well as the plots and comments.~~ The correctness of the following will be checked: **[8 / 40]**

- The size and values of `t` and `y`;
- The output of ”help rungeKutta”.

(Run the file `GenerateYourOutputsCW4.m` to see these outputs generated for a particular set of inputs. Note that in marking your work, different input(s) may be used.)