University of Nottingham
UK | CHINA | MALAYSIA

MATH2019 Introduction to Scientific Computation

— Coursework 1 (10%) —

Submission deadline: ***3pm, Monday, 26 Oct 2020***

Note: This is currently **version 3** of the PDF document.

No more new questions will be added anymore.

This version has some updated text on page 2, 3 and 4; See notes in the margin.

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 that solve nonlinear equations, and you will study some of the algorithms' behaviour.
- A total of 40 points can be obtained in this coursework. The weighting of each question will appear in a later version of this PDF.

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 1 Pack" folder from Moodle.

(More files may be added in later weeks.)

(!) New info (!) **Submission Procedure:**

- Submission will open after 15 October 2020.
- To submit, simply upload your 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 `GenerateYourOutputsCW1.m`, which can be found in folder "Coursework 1 Pack" from Moodle.

- **Bisection method**

  Let $f(x) = x^3 + x^2 - 2x - 2$. Note that this is the same function as in Lecture 1. Consider the bisection method for finding the root of f in the interval $[1, 2]$.

1 Open the *script* m-file `bisection_script.m`, and add your code to it:

- First simply plot the function to get an idea of what it looks like.
- Then try implementing the bisection algorithm as explained in Lecture 1.

Hint: The Lecture 1 SlidesAndNotes PDF contains a pseudo-code algorithm for bisection, as well as a simple Matlab code. This was also demonstrated in Lecture 1 (see Moodle for links to Echo360 Engage for the recordings, and to MS Teams for the live-lecture recording.)

- Verify that the first few approximations are correct. What value do your approximations seem to converge to?

2 Next, you will repeat what you have done above, but using a *function* m-file. To help you get started, open the function m-file `bisection_func.m`. Note that this function has the following prototype:

```
function p_vec = bisection_func(f,a,b,Nmax)
```

where the output `p_vec` is the column vector of approximations p_n $(n = 1, 2, \ldots)$ provided by the bisection method. The input `f` is a function handle, `a` and `b` define the initial interval $[\mathtt{a}, \mathtt{b}]$ of interest, and `Nmax` is the maximum number of iterations.

- Complete this function so that it implements the bisection algorithm, and provides the output as required above.
- Test your function by typing, for example, in the command window:

```
f = @(x)  x.^3 + x.^2 - 2*x - 2
a = 1
b = 2
Nmax = 5
p_vec = bisection_func(f,a,b,Nmax)
```

- (The below has been added on 8 Oct 2020.)

Marks can be obtained for your `bisection_func` m-file for correctly generating the ~~following~~ **required** output, for certain set(s) of inputs for {`f,a,b,Nmax`}. The correctness of the following will be checked: **[5 / 40]**

← Sentence corrected on **15 Oct 2020**

- The size (number of columns and rows) of output `p_vec`
- The values of output `p_vec`

(Run the file `GenerateYourOutputsCW1.m` to see these outputs generated using the set of inputs from Part 2. Note that in marking your work, different input(s) may be used.)

- **Bisection method with stopping criterion**

3 • Next, write a new *function* m-file called `bisectionStop_func.m`, which implements the bisection method with a stopping criterion. This function *must* have the following prototype:

```
function p_vec = bisectionStop_func(f,a,b,Nmax,TOL)
```

where p_vec, f, a, b and Nmax are as before, and TOL is a tolerance value. Ensure that the bisection method stops prematurely by using a stopping criterion based on the absolute value of the last two computed approximations, in other words, stop when $|p_{i+1} - p_i| < \mathtt{TOL}$.

Hint: Use the command "break" to prematurely exit "for" or "while" loops.

- Test your function by typing, for example, in the command window:

```
f = @(x)  x.^3 + x.^2 - 2*x - 2
a = 1
b = 2
Nmax = 50
TOL = 10^(-4)
p_vec = bisectionStop_func(f,a,b,Nmax,TOL)
```

- Also add a brief description at the top of your file. This description should become visible, whenever one types: help bisectionStop_func.

Hint: See also Section 9.5 of the Matlab Guide for writing a "help" for user-defined functions.

▶

Marks can be obtained for your bisectionStop_func m-file for generating the ~~following~~ **required** output, for certain set(s) of inputs for {f,a,b,Nmax, TOL}. The correctness of the following will be checked: **[10 / 40]**

← Sentence corrected on **15 Oct 2020**

- The length of p_vec
- The values of p_vec
- As well as for the output of "help bisectionStop_func".

(Run the file GenerateYourOutputsCW1.m to see these outputs generated using the set of inputs from Part 3. Note that in marking your work, different input(s) may be used.)

▶ Fixed-point iteration method

To solve the root-finding problem $f(x) = 0$, but using fixed-point iteration, we define

$$g(x) = x - c\, f(x)$$

and consider the corresponding fixed-point problem $g(p) = p$. (Note that if $g(p) = p$, then indeed $f(p) = 0$.)

$\boxed{4}$ • Write a new *function* m-file fpiter_func.m, which implements fixed-point iteration. The function has the following prototype:

```
function p_vec = fpiter_func(f,c,p0,Nmax)
```

and needs to provide the output p_vec, which is the column vector of approximations p_n $(n = 1, 2, \ldots)$ provided by fixed-point iteration. The input f is a function handle, c is a parameter (used in the definition of g), p0 is the initial guess, and Nmax is the maximum number of iterations.

- Test your function by typing, for example, in the command window:

```
f = @(x)  x.^3 + x.^2 - 2*x - 2
c = 1/10
p0 = 1
Nmax = 5
p_vec = fpiter_func(f,c,p0,Nmax)
```

- As before, don't forget to add a brief description at the top of your file.

▶

Marks can be obtained for your fpiter_func m-file for generating the ~~following~~ **required** output, for certain set(s) of inputs for $\{$f,c,p0,Nmax$\}$. The correctness of the following will be checked: **[10 / 40]**

← Sentence corrected on **15 Oct 2020**

- The size of p_vec
- The values of p_vec
- As well as for the output of "help fpiter_func".

(Run the file GenerateYourOutputsCW1.m to see these outputs generated using the set of inputs from Part 4. Note that in marking your work, different input(s) may be used.)

▶ (The below has been added on 15 Oct 2020.)

▶ **Newton's method**

We now consider Newton's method to solve the root-finding problem $f(x) = 0$.

5 • Write a new *function* m-file newton_func.m, which implements Newton's method. The function has the following prototype:

```
function p_vec = newton_func(f,dfdx,p0,Nmax)
```

and needs to provide the output p_vec, which (as before) is the column vector of approximations p_n $(n = 1, 2, \ldots)$ provided by Newton's method. The input f is a function handle to $f(x)$, dfdx is a function handle to $f'(x)$, p0 is the initial guess, and Nmax is the maximum number of iterations.

- Test your function by considering, for example, the problem studied in Lecture 3 $(f(x) = x^2 - 2,\ p_0 = 1)$ and typing in the command window:

```
f = @(x)  x.^2 - 2
dfdx = @(x) 2*x
p0 = 1
Nmax = 5
```

```
p_vec = newton_func(f,dfdx,p0,Nmax)
```

- As before, don't forget to add a brief description at the top of your file.

▶

Marks can be obtained for your `newton_func` m-file for generating the required output, for certain set(s) of inputs for {`f`,`dfdx`,`p0`,`Nmax`}. The correctness of the following will be checked: **[9 / 40]**
- The size of `p_vec`
- The values of `p_vec`
- As well as for the output of "help newton_func".

(Run the file `GenerateYourOutputsCW1.m` to see these outputs generated using the set of inputs from Part 5. Note that in marking your work, different input(s) may be used.)

▶ ⋆ **Optimized fixed-point iteration method** (this is a challenging question!)
In this last part, we return to the fixed-point iteration method (Part 4), and aim to optimise the parameter c used in the method.

$\boxed{6}$ • Write a new *function* m-file `optParamFPiter_func.m`, which determines an optimal value of parameter c, in the following sense: For an optimal c, the fixed-point iteration method is the *fastest* to converge to a given tolerance value `TOL`. In other words, the absolute value of the error reaches `TOL` in the *least* amount of iterations. For simplicity, you may assume:

- The exact solution p is given,
- The optimal value of c is within the interval $[0.001, 1]$,
- You only need to find an *approximate* optimal value of c within the set $\{0.001, 0.002, 0.003, \ldots, 0.999, 1\}$.

Your function must have the following prototype:

```
function [c_opt,N_opt] = optParamFPiter_func(f,p0,p,TOL,Nmax)
```

and needs to provide as output the optimal value `c_opt`, and corresponding number of iterations `N_opt`. The input `f` is a function handle to $f(x)$, `p0` is the initial guess, `p` is the exact value of the fixed point, `TOL` a given tolerance to be attained by the error, and `Nmax` is a maximum number of iterations (which can be assumed to be larger than `N_opt`).

Hint: One idea to determine the optimal value of c is by the following trial-and-error: Within `optParamFPiter_func`, simply call your `fpiter_func` using some value of c, and determine the required number of iterations; Then try a different value of c, etc.

Hint 2: You may find useful the following Matlab inbuilt functions: `find`, `max`, `min` (see e.g. their help / documentation).

- Test your function by considering, for example, the data used in Part 4:

```
f = @(x)  x.^3 + x.^2 - 2*x - 2
p0 = 1
p = sqrt(2)
TOL = 10^(-6)
Nmax = 20
[c_opt,N_opt] = optParamFPiter(f,p0,p,TOL,Nmax)
```

▶

Marks can be obtained for your `optParamFPiter_func` m-file for generating the required outputs, for certain set(s) of inputs for {`f`,`p0`,`p`,`TOL`,`Nmax`}. The correctness of the following will be checked: **[6 / 40]**

- The value of `c_opt` and of `N_opt`
- Also, it is required that your function completes its run within 1 minute.

(Run the file `GenerateYourOutputsCW1.m` to see these outputs generated using the set of inputs from Part 6. Note that in marking your work, different input(s) may be used.)

▶ Finished!