ENGSCI 331 - Computational Techniques 2

Lab: Nonlinear Equations

Department of Engineering Science & Biomedical Engineering

Due: Tuesday October 14 2025, 6:00pm

Introduction

Download `NLELab.zip` from `Canvas > Modules > Nonlinear Equations` and unzip it into the folder that you've set aside for labs in this course.

Submission Instructions:

Upload your code:

Please combine all **code** files for your submission into a single zipped directory and upload to the relevant Canvas assignment. Please do **not** modify the original file names or include any additional code files. Please do **not** modify any existing class, method, or function names, and use these functions where provided.

Please ensure that all code submitted is your own work. Each student's submissions will be cross-checked for evidence of plagiarism.

You may assume that the person marking your code has knowledge of what you are attempting to do, and therefore you can avoid e.g. excessive comments in the code providing step-by-step explanation of how the numerical methods work. Some amount of commenting is still appropriate (e.g. to help structure any complicated scripts or functions), but the point of the assignment is not to assess coding style. However, at least make sure that your functions include header comments / docstrings.

Hand-In Items:

The tasks have specific workings, answers, figures etc that you should hand-in, **in addition to your code**. These are clearly indicated in this lab document. Your hand-in items should be compiled into a brief report named report_[your upi].pdf/doc/docx. A single report containing material for all tasks is appropriate. Please upload this **in addition to your zipped code** as part of your assignment submission.

Formatting your plots: Plots should have their axes labelled, a title or caption, and a legend if there is more than one data set. Some tasks are supplied with code to produce plots; these plots may or may not already contain this information. Modify the code where appropriate.

Task 1 - Iterative Algorithms

Background and Aim: Implement the algorithms discussed in lectures to iteratively find one root of the nonlinear function $f_1(x) = 2x^2 - 8x + 4$, starting from either an initial interval $\left(x^{(0)}, x^{(1)}\right)$ or an initial guess $x^{(0)}$.

Methodology: In the extracted zip archive you find a file called `algorithms.py` containing the incomplete function:

- `def bisection(f, xl, xr, max_iter, tol)`

and headers for functions:

- `def secant(f, x0, x1, max_iter, tol)`
- `def regula_falsi(f, xl, xr, max_iter, tol)`
- `def newton(f, g, x0, max_iter, tol)`
- `def combined(f, g, xl, xr, max_iter, tol)`

The initial comments for each function have already been written, specifying the function input/output required for tasks 1 and 2, with details provided about each variable.

Pseudo-code is not provided, so it may be useful to write this yourself, based on the course notes.

Complete each function, such that it:

- produces a sequence of root estimates using the appropriate method and stores them in an array alongside the input initial root estimates;
- applies the following simple root finding test each iteration: $|f(x^{(k)})| < \Delta$ where Δ is some numerical tolerance value (`tol`);
- outputs the total number of iterations undertaken;
- returns a termination flag, using the `ExitFlag enum`. (**HINT:** It is a good idea to check the contents of all supplied files to be aware of the expected exit-flag types available).

Note: each function should seek to ***minimise the number of function evaluations***.

In `Newton` and `Combined`, you should use the algebraic derivative $g(x) = f'(x)$.

For `Combined`, you should combine the bisection method with Newton's method:

- Use either end of the provided bracket $\left(x^{(0)}, x^{(1)}\right)$ as the starting estimate for Newton's method.

- Attempt to use Newton's method to find the next root estimate, $x^{(k+1)}$.
- If $x^{(k+1)}$ from Newton's method lies outside of the current bracket, $\left(x^{(L)}, x^{(R)}\right)$, use the bisection method for this iteration to get a better estimate for $x^{(k+1)}$.
- Each iteration, use the latest root estimate $x^{(k+1)}$ to update the bracket, $\left(x^{(L)}, x^{(R)}\right)$.
- This algorithm should have both *guaranteed* and *fast* convergence.

Verification: The script `task1.py` calls `bisection`, `secant`, `regula_falsi`, `newton` and `combined` one-by-one to find a single root of $f(x)$ and constructs a table outlining the root estimates, and performance of the methods. You should not need to modify this script, except perhaps to comment out calls to incomplete functions.

1. Submit the output from `task1.py`, and comment on the performance of each of the various methods when applied to this function (both in terms of iterations and function / derivative evaluations).

Task 2 - Algorithm Comparison

Background and Aim: In Task 1, your algorithms were tested on a simple polynomial, $f(x) = 2x^2 - 8x + 4$, which has two real roots at $x = 2 \pm \sqrt{2}$. The aim of this task is to compare the behaviour of your algorithms for the following three nonlinear functions:

$$\begin{aligned} f_2(x) &= x^2 - 1, \\ f_3(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}}, \\ f_4(x) &= \cos(x) + \sin(x^2) - \frac{1}{2}. \end{aligned}$$

Methodology: Your Task 1 functions should each output a sequence of estimates for the root in a one-dimensional array:

$$\vec{x} = \left\{x^{(0)}, x^{(1)}, \ldots, x^{(k)}\right\}$$

Note that the script `task2.py` includes starting estimates of the roots, which you should not modify, as they have been chosen so as to exhibit specific behaviour in each method. It also produces a plot of $f(x)$ vs. x for your reference – you can set variable `disp_func = false` when you no longer wish to produce/view this plot.

Once you have run `task2.py`, answer the following questions:

2. Explain what is being depicted in the various graphs, and comment of the performance of the algorithms on the different functions. For example, ensure that you discuss the behaviour of Newton's method for each of the three functions. If Newton's method fails to find a root for any of the three functions, explain why this is the case and how this issue is fixed in your implementation of `Combined`.

   To help you structure your answer, discuss each function separately:

   (a) Comment on the methods' performance on function $f_2(x)$.
   (b) Comment on the methods' performance on function $f_3(x)$.
   (c) Comment on the methods' performance on function $f_4(x)$.

3. You have little control over which root your algorithms converge to, other than by modifying the initial interval/estimate. Briefly outline a strategy for systematically finding multiple roots of a general continuous nonlinear function $f(x)$. State any assumptions that need to be made in order to find all such roots. (Note: There is no answer that is guaranteed to find all roots in all circumstances).

Task 3 - Sensitivity of Newton's Method

Background and Aim: In this task you will adapt your Newton's method code from Task 1 to explore how damping factors can improve the stability and sensitivity of Newton's method.

Methodology: In the extracted zip archive, you will find the file `task3.py`. This script calls a function `newton_damped()` multiple times to produce a visualisations of the root that is converged to from various starting points, alongside the number of iterations. You will need to create this function; you should start by copying your `newton` function, and then add an extra (final) argument `beta`.

Implement an adaptive damping factor within the update step of the `newton_damped()` function, as described in the notes.

4. Create two pairs of plots using `task3.py`: one pair without damping, and one with damping, and describe what is shown in the plots and discuss the differences. (You can use the included equation, $x^6 = 1$, or change this to something else with at least two roots in the domain.)

Task 4 - Newton's Method in Two Dimensions

Background and Aim: Having created a number of algorithms to find one root of a single-variable nonlinear function $f(x)$ in Task 1, the aim of this task is to extend Newton's method to find a solution to a system of nonlinear equations. We will first try to find a solution to the following system of two nonlinear functions of two independent variables:

$$\mathbf{f}_5(x,y) = \begin{bmatrix} x^2 - 2x + y^2 + 2y - 2xy + 1 = 0 \\ x^2 + 2x + y^2 + 2y + 2xy + 1 = 0 \end{bmatrix}$$

The vector function for this task is defined as `f5` in `functions2.py`. To determine $\mathbf{f}_5(x,y)$ you can call the function as follows: `f5([x,y])` and it will return a list. Recall from the lectures that Newton's method in two dimensions can be expressed as:

$$\begin{bmatrix} \dfrac{\partial f_1\left(x^{(k)},y^{(k)}\right)}{\partial x} & \dfrac{\partial f_1\left(x^{(k)},y^{(k)}\right)}{\partial y} \\ \dfrac{\partial f_2\left(x^{(k)},y^{(k)}\right)}{\partial x} & \dfrac{\partial f_2\left(x^{(k)},y^{(k)}\right)}{\partial y} \end{bmatrix} \begin{bmatrix} {\delta_x}^{(k+1)} \\ {\delta_y}^{(k+1)} \end{bmatrix} + \begin{bmatrix} f_1\left(x^{(k)},y^{(k)}\right) \\ f_2\left(x^{(k)},y^{(k)}\right) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

which can also be written in matrix form $J^{(k)}\,\vec{\delta}^{(k+1)} + \vec{f}^{(k)} = 0$.

Newton's method requires that you calculate the Jacobian, J, each iteration. A simple method to find J is to estimate the first-order partial derivatives using finite differences:

$$\frac{\partial f\left(x,y\right)}{\partial x} \approx \frac{f\left(x+h,y\right) - f\left(x-h,y\right)}{2h}, \qquad \frac{\partial f\left(x,y\right)}{\partial y} \approx \frac{f\left(x,y+h\right) - f\left(x,y-h\right)}{2h}.$$

You will need to implement this for general $n \times n$ matrices by completing the code in `Jacobian.py`.

Methodology: In the extracted zip archive, you will find the incomplete function `newton_multi` in `newton_multi.py`. The definition of this function has already been written, and specifies the function input/output. Complete this function, ensuring that it:

– applies the convergence test on the L_∞-norm of the residuals to check if a root has been found.

– applies a finite limit to the number of iterations in case no root is found.

– outputs a sequence of root estimates as a list of lists:

$$\vec{x} = \begin{bmatrix} \left[{x_1}^{(0)} \;\; {x_2}^{(0)} \;\; {x_3}^{(0)} \;\; \ldots \;\; {x_n}^{(0)}\right] \\ \left[{x_1}^{(1)} \;\; {x_2}^{(1)} \;\; {x_3}^{(1)} \;\; \ldots \;\; {x_n}^{(1)}\right] \\ \ldots \\ \left[{x_1}^{(k)} \;\; {x_2}^{(k)} \;\; {x_3}^{(k)} \;\; \ldots \;\; {x_n}^{(k)}\right] \end{bmatrix}$$

You may wish to hard-code the model for only two equations / variables for Tasks 4 and 5, but Task 6 will need a general implementation for n equations. You only need to submit the n-dimensional Newton method code.

Verification: The initial part of the script `task4.py` calls `newton_multi` to find a root of $\mathbf{f}_5(x, y)$, and verifies that a root has been located.

Task 5 - Visualising Newton's Method

Background and Aim: The function `newton_multi` from Task 4 was verified on a system of two nonlinear functions, $\mathbf{f}_6(x, y)$ with a root at $(0, -1)$. The aim of this task is to examine the behaviour of the algorithm applied to a system of equations with multiple solutions. This system of equations is defined as `f6` in `functions2.py`. We will converge from four different starting root estimates: $(-1, 3)$, $(2, 3)$, $(2, 0)$.

Methodology: Script `task5.py` produces these two subplots in a figure:

– Top: plot of $f_6[0]\left(x^{(k)}, y^{(k)}\right)$ vs k for all starting locations.

– Bottom: plot of $f_6[1]\left(x^{(k)}, y^{(k)}\right)$ vs k for all starting locations.

It then produces another figure which shows 2D-plots of $f_6[0](x, y)$ and $f_6[1](x, y)$, and overlaid are the sequence of solutions found for each starting point.

Answer the following three questions on Newton's method in two dimensions:

5. Briefly outline two *different* reasons why `newton_multi` could fail to converge to a solution; give an example of a starting point that fails to converge to a solution for `f6`.

6. Explain what the convergence plots are showing about the speed of convergence. You will need to change the code in `task5.py` (around lines 38-42) to show the log of the (absolute) function values.

7. Now change the function to `f7` (line 17) and comment on and explain the convergence for this in comparison to that of `f6`.

Task 6 - Newton's Method in n Dimensions

If you have completed the `newton_multi` function correctly it should also be able to solve a system of 3 equations and 3 unknowns. Use `task6.py` to test your implementation.

The code benchmarks the performance of your function compared to the `scipy` function `fsolve`. You can also compare the performance with a parallel implementation of your `newton_multi` function.

No specific submission is needed for this task; but you should check your `newton_multi` function is working correctly, and submit it (along with `Jacobian.py`).

Task 7 - Laguerre's Method

Background and Aim: Here we wish to implement Laguerre's method to find roots for polynomials. Within the Laguerre's folder of the lab files, are four files: `deflate.py`, `horner.py`, `laguerre.py` and `task7.py`. `laguerre.py` is the only file that is incomplete. Complete `laguerre.py` and find the roots of the following polynomial (you will need to modify the coefficients in `task7.py`):

$$f(x) = x^6 + 2x^5 - 4x^3 + 2x^2 + 4.$$

8. Provide the output of `task7.py` for the polynomial above.

9. Comment on the number of iterations it takes to converge to the various roots of the polynomial, compared to the other methods you have implemented. (Of course this method is finding all the roots, not just one.)