ENGSCI 331 - Computational Techniques 2

Department of Engineering Science

Introduction

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:

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

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.)