1 Introduction

In this project you will explore the problems of numerical function evaluation and root finding under various pathological conditions. You will consider the root solving problem \( f(x) = 0 \), where

\[
f(x) = e^{k(x-c)} - \cos(k(x-c)) - k(x-c)
\]

for \( k, c \) both positive real numbers. Convince yourself that there is a root at \( x = c \). However, without a mathematical understanding of this function (as part of an automated analysis using a CAS, for instance) a computer will not know there should be a root at \( x = c \) and will have to resort to the kinds of “black box” methods we have looked at in class. We will assume that the computer has explicit derivatives of \( f \) available to it without knowing \( c \) itself.

2 General Instructions

Please do not work together on this project in groups – everyone must submit their own unique report.

Read the whole document through carefully before you begin.

This project requires you to work in a programming language that uses double precision IEEE arithmetic, which is the standard in Python and Matlab. The project is split into three main parts, and into a total of 14 small tasks to help you organize your work effectively. As you proceed you should collect your work into a brief report. I prefer you to prepare this document using a word processor, but you may also hand write it and attach printouts.

Hand in or email me your report on or before Wednesday Oct 22. This will contain all your code, output, graphs, observations and explanations. The professionalism of your technical writing is one of the assessment criteria, and includes being able to state ideas concisely (i.e., within a couple of sentences), to use clear logic, and to take advantage of mathematical concepts you have learned in this course when appropriate.

You are welcome to ask me any questions to clarify your understanding of these tasks, the math, or the programming during office hours or by email, but use your common sense and always attempt to solve a problem before coming to me with it. Get started early, and don’t wait until a few days before the deadline to realize that you need clarification on a range of issues.

Specific instructions below will assume you are working in Python. In your program code, make sure to import the appropriate libraries to access the standard math functions. In Python, just use from numpy import * at the top of your script, or start with from solve_1D import * to get both the solvers, the numpy and plotting libraries all in one go. The latter assume that your solve_1D.py file is in the same directory as your project code file.

You can save a python plot by using the disk icon on the plot window’s menu, and then choosing either EPS, PNG, or SVG format (there may be others available, but make sure you are able to print those out or successfully embed them in your report). You may also print the figure directly to paper.
You are provided with a new version of the `solve_1D.py` file that has a few additional features, which is available to download from the course web page. For instance, the functions now have an optional “quiet” argument that is `True` by default (so that verbose output is displayed). When this option is set to `False` in a call, print statements will be executed that show more details of each iteration. e.g. `secant(0, 1, f, 1e-5, quiet=False).

2.1 Zen and the Art of Numerical Analysis

The nature of this project is to help you realize that real-world mathematical and computational problems require ingenuity and exploration, for instance because they can show up limitations in standard techniques. You will learn that mathematically similar problems may require quite different numerical approaches, and that there is rarely a unique or obvious best approach to take. In other words, it is an art. You must learn to think, try an experiment, accept its failure, learn what went wrong, and try again. You will be led through the tasks as if on a guided tour of what it’s like to tackle an unknown problem for the first time, much like I did when I put together this project. As such, some tasks have long descriptions, but all require relatively little work.

3 Project Tasks

3.1 Part I

In this part we study a root solving problem that only exhibits a little pathological behavior.

1) Either by adapting the `newton_raphson` function given in `solve_1D.py`, or from writing your own, create a function that performs modified Newton steps, as defined on page 80–81 of the book.

2) Find the first and second derivatives of $f(x)$ and write functions that evaluate $f$ and these derivatives given $x$ for the choice $k = 1$ and $c = \sqrt{2}$. Under these conditions, a reasonable bracketing interval which the computer might identify for the root is $[-1, 4]$. Compare how many steps need to be taken to reach $\pm 10^{-5}$ accuracy with the un-modified and modified Newton solvers, starting from the left-hand endpoint. Show your full output (i.e., using the `quiet=False` option).

3) With the plotting functions imported from matplotlib, plot a graph of this $f(x)$ using the values of $k$ and $c$ from (2) over the bracketing interval. For hints, you can use the example plot in python introduction script you were given at the beginning of the course (e.g. use `linspace(xlo, xhi, N)` to generate N uniform points in the $x$ domain). Choose $N = 100$. You do not need to save this graph or use it again.

4) From your understanding of the theory behind this modification, why do you think there has been an improvement with the modified solver?

3.2 Part II

Now you will study an example of this root-finding problem that is unfortunately ill-suited to the double precision arithmetic your computer uses. We do this by changing $k$ to be $10^{-8}$ but keeping $c$ the same. We will use these values for the rest of the project tasks.

5) Use the `figure()` command to start a new figure window. Plot a graph of $f(x)$ using the new values of $k$ and $c$ over the same bracketing interval. Choose $N = 100$ points again. Now use the `hold()` command after creating this plot in order to keep subsequent plots appearing in the same figure.

6) Why won’t any of the 1D solving algorithms we have studied solve this root-finding problem in its current form?
7) As is typical behavior for a numerical analyst, let us replace the study of \( f \) with that of a function that approximates it in the bracketing interval, and explore whether this helps us numerically find the root. Taylor series approximations are one of the most popular choices of approximating function. By whatever means, calculate a third order Taylor approximation for \( f(x) \) centered at \( x = x_0 \) before substituting in the values for \( k \) or \( c \) (the algebra is much less messy that way). You’ll need the first derivative of \( f(x) \) again later on, so make a function to calculate its value (you could call it \( df \)).

8) Now you have a cubic polynomial which we’ll call \( p(x) \). Find its first two derivatives. By substituting \( k = 10^{-8}, c = \sqrt{2} \) and \( x_0 = 1.3 \), write code for three functions, \( p, dp \) and \( ddp \), which can evaluate these values at a given value of \( x \). (Note: Any value for the centre \( x_0 \) in the bracketing interval that is not \( c \) would work in the following tasks – you’re welcome to verify that yourself.)

9) Plot \( p(x) \) over the same range as you did for \( f(x) \). This graph should appear overlaid on your plot from (5). Comment on the graphical differences between \( p(x) \) and \( f(x) \). Assess by eye whether the graph looks accurate, given what you know mathematically about \( f(x) \) (from calculus and your answer to (4)) and comment on this. To do this, zoom in on your graph around the minimum of the \( p(x) \) by clicking on the magnifying glass icon and then drawing a bounding box around the area you wish to zoom in on. The back icon returns to the previous view.

10) By considering the computer arithmetic operations involved in the evaluation of \( f(x) \) versus \( p(x) \), suggest a likely explanation for the striking difference in their graphs.

11) Try Newton’s method on \( p(x) \) using \( xtol=1e-5 \) and describe what happens. Has this helped us solve the original root-finding problem for \( f(x) \)? Use the graphs you plotted to help you comment on why.

Unfortunately your modified Newton solver will not make this issue go away (convince yourself why). The problem you now see stems from error in evaluating the coefficients of the Taylor series themselves! This is because to get those coefficients we had to evaluate \( f(x) \) at some values of \( x \) close to \( c \), which is exactly where we have seen that \( f \) misbehaves when \( k \) is as small as \( 10^{-8} \). So we fixed one problem with our approximation but replaced it with another. In fact, with no additional help, a computer would not ultimately help us solve the root-finding problem in Part II. This is what we would label an “ill-posed” numerical problem. But in the meantime we’ve gathered some insights into the pathology of \( f \) and a way in which approximations can be flawed.

3.3 Part III

Take a look at the graphs again. What do you notice about the position of the minimum of \( p(x) \) relative to \( c = \sqrt{2} \)? By a stroke of algebraic coincidence, the symmetry in \( f \) that causes the catastrophic numerical behavior near the root also ensures that the Taylor polynomials we derive for any \( x_0 \approx c \) will have a turning point very close to \( c \). (We don’t need to go into the technical reasons why this is true, but sometimes we get lucky when we’re exploring uncharted territory, provided we keep our eyes open for leads.)

12) Numerically validate the assertion made in (11) by creating a new function \( q \) that is identical to the \( p \) that you made earlier except derived using the alternative center value \( x_0 = 0 \) (any other value in the interval would also be fine). Create an array containing 100 \( y \)-axis values for plotting \( q \) using the same \( x \)-values you used for your earlier graphs. Plot your function \( q \), and observe that the graphs of \( p \) and \( q \) appear to be merely vertically offset from each other. (For this new center you’ll actually see that there are no real roots in this neighborhood.)

13) Use the built-in \( \max \) and \( \min \) functions to verify that the minimum and maximum difference between the graphs of \( p \) and \( q \) is almost constant at approximately \( 8.5152 \times 10^{-17} \), which is an
order of magnitude smaller than the vertical scale of these graphs. In other words, we’ve discovered that the error in computing the different Taylor polynomials (as the center $x_0$ varies) is almost exclusively in the first (constant) term, so let’s change the numerical task to avoid that error.

14) From the above observation, you can instead use numerical techniques to find where that turning point is, and maybe use this as a starting approximation for the solution to $f(x) = 0$ in Part II. The turning point is given by $\frac{dp}{dx} = 0$, which now becomes your root-finding problem for Part III. By considering what kind of function is $\frac{dp}{dx}$, choose a suitable numerical method for this problem (and for which you already have a code) and solve it using your $dp$ with $xtol=1e-5$. Comment on what you find: in particular, evaluate $f(x)$ on your result and hence determine whether you have satisfactorily solved the $f(x)=0$ problem.

4 Code ideas

To give you some idea about what to put in your test code, here is part of the python code that I used to compare these solvers when I explored my solution to this project’s problem:

```python
# None of these methods find the right solution!
# with f(x) and its derivative df
x1 = newton_raphson(1.4, f, df, 1e-5)
x2 = newton_raphson(4, f, df, 1e-5)
x3 = secant(0.5, 0.75, f, 1e-5)
x4 = bisection(-1, 4, f, 1e-5)

# with p(x) and its derivatives dp and ddp
x5 = secant(0.5, 0.75, p, 1e-5)
x6 = newton_raphson(0.5, p, dp, 1e-5)
x7 = mod_newt(0.5, p, dp, ddp, 1e-5)

abs_errors = abs(sqrt(2) - array([x1, x2, x3, x4, x5, x6, x7]))
print abs_errors
```

The above code displays an assessment of seven different approximate solutions for the value of $c$ using absolute error. (Calculating relative error is not very helpful here, because $x \approx c \approx 1$, so dividing by $x$ or $c$ near $c$ gives values almost identical to the absolute error.)

I used code like the following to produce my graphs:

```python
xs = linspace(-1,4,100) # array
ys_f = f(xs)
ys_p = p(xs)
ys_q = q(xs)

plot(xs, ys_f)
plot(xs, ys_p)
plot(xs, ys_q)
show()
```