PHY407 Lab4

Genevieve Beauregard (1004556045)
Anna Shirkalina (1003334448)

September 2020

Question 1

a)

Nothing to submit. We define a function `PartialPivot` which performs partial pivoting before Gaussian elimination. Testing this against equation 6.2 returns the correct result.

b)

We created a function `time_solve` which took in an N x N matrix input, A, and a column vector,v, of size N, and timed how long to solve the system Ax = v for a function that solved the system. It also returned the error of x, using the steps instructed in the handout. We did this for `GaussElim`, `PartialPivot` and `numpy.linalg.solve`, for a range of system sizes, collecting both the their times and errors in a matrix. The matrices A and v entries were randomly generated using `numpy.rand(N,N)` before being solved for each solver function call. Our plots can be seen in Fig 1 and 2.

We see that the times for `numpy.linalg.solve` is consistently faster that our other routines. `GaussElim` and `PartialPivot` are generally comparable trends, with `PartialPivot` being slightly slower. This is to be expected, the pivoting adds extra computing time. `numpy.linalg.solve` probably uses more efficient LU decomposition in a more efficient underlying language such as C and thus, is consistently faster and its computation time grows at a slower rate.

The errors for the `PartialPivot` and `GaussElim` are generally comparable as expected: `PartialPivot` solving has its basis in Gaussian elimination. The errors for `numpy.linalg.solve` are significantly lower though it increases with matrix at a similar pace as the other routines.

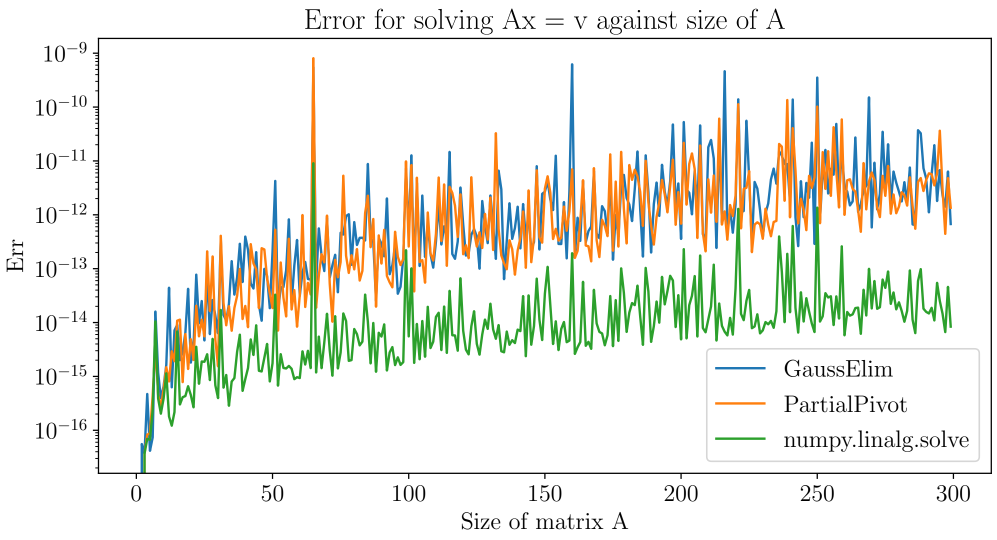

Figure 1: The relative error for solving system $Ax = v$ for the respective solving routines, where error was calculated to be the mean of the entries of $v - A \cdot x$.

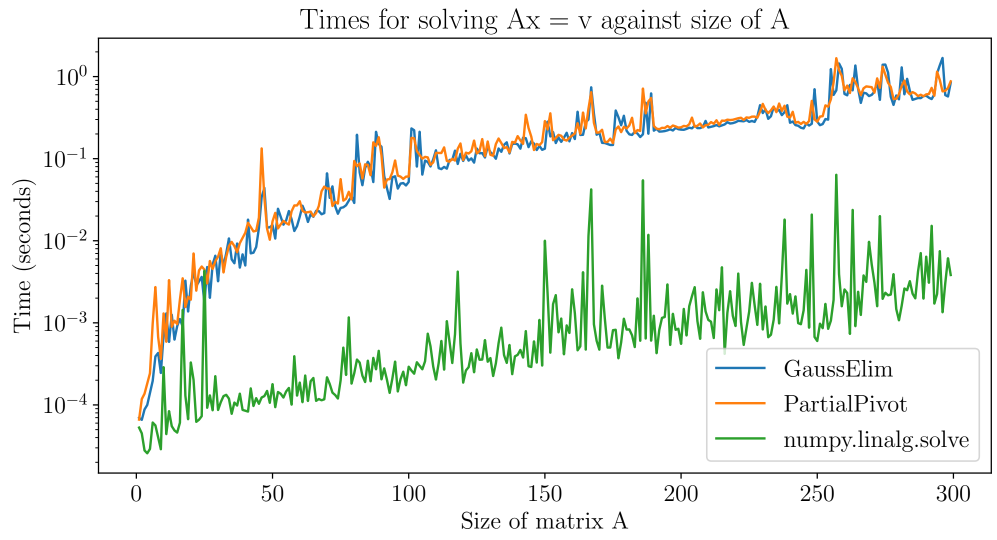

Figure 2: The times in solving system $Ax = v$ for the respective solving routines.

V_i (no inductor)	$\lvert V_i \rvert$ (V)	Phase (°)
V_1	1.70	-5.47
V_2	1.48	11.58
V_3	1.86	-4.16

Table 1: Voltages magnitudes and phase difference at $t = 0$, for a circuit with no inductor

V_i (inductor)	$\lvert V_i \rvert$ (V)	Phase (°)
V_1	1.56	-4.03
V_2	1.50	21.64
V_3	2.81	14.35

Table 2: Voltages magnitudes and phase difference at $t = 0$, for a circuit with $R_6 = i\omega L$ impedence.

c)

We utilised the our partial pivoting routine to solve the system as per the equation, with R_6 constant. Our system of equations is expressed as complex system of linear equations in the coefficient matrix `A` in our code and the column vector v. We obtain $\begin{pmatrix} x_1 x_2 x_3 \end{pmatrix}$ in an array using our `PartialPivot` function. We calculated the magnitudes and phase of the voltages: our values for $|V_I|$ and their respective phases can be seen in the output or in the tables 1 and 2.

We use this data to plot $V_i = x_3 e^{i\omega t}$ for each i. We created a time array spanning about two periods and plotted $V_i(t)$ - both for the constant resistance non-inductor circuit and for the circuit should R_6 have a complex impedance of $i\omega L$. Our plots can be seen in Fig 3 and 4. We plot the real part of V_i for the circuit with an inductor.

We see that replacing the R_6 with an imaginary impedance has the effect of increasing V_3's amplitude and shortening its period slightly. V_1 and V_2 relatively unchanged. We see that the amplitude of the voltage across R_6 increases drastically.

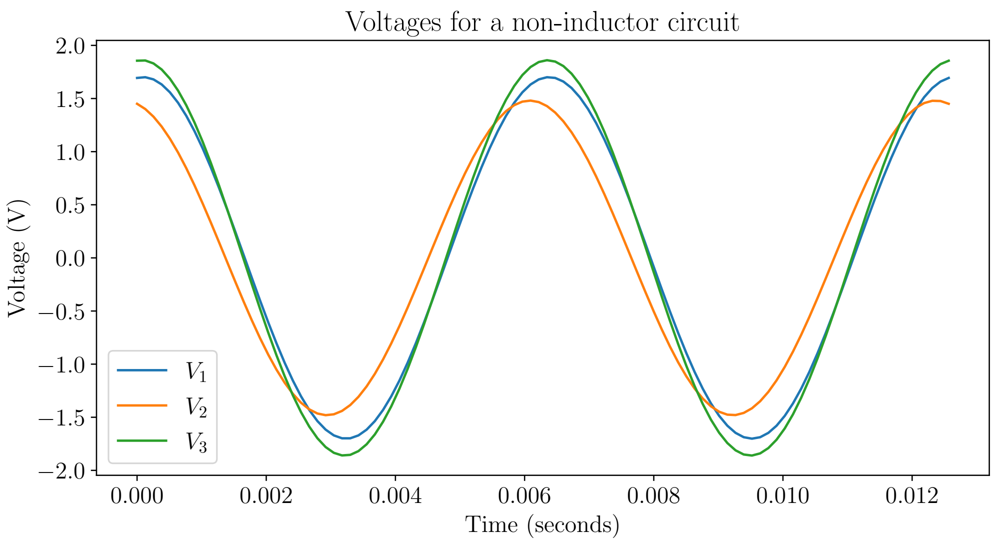

Figure 3: V_i against time.

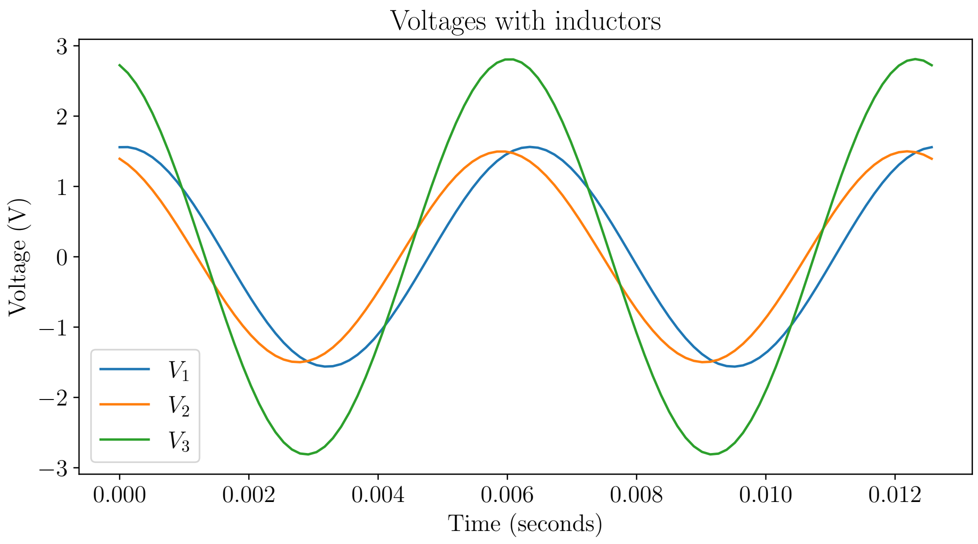

Figure 4: V_i with inductors as per the question.

Output

```
With no inductor,
|V_1| is 1.7014390658777336 V
|V_2| is 1.4806053465364062 V
|V_3| is 1.8607693200562132 V
At t = 0,
V_1 phase is, -5.469094970111944 degrees
V_2 phase is, 11.583418604687065 degrees
V_3 phase is, -4.164672651865924 degrees

With an inductor
|V_1| is 1.5621181940219633 V
|V_2| is 1.4994286802306562 V
|V_3| is 2.8112763903392537 V
At t = 0,
V_1 phase is, -4.025908819603362 degrees
V_2 phase is, 21.63928264257023 degrees
V_3 phase is, 14.352479528588603 degrees
```

Question 2

c) and d)

Pseudo-code for part c

```
    def Hmn(m, n):
    """
    Return the nm element of the matrix produced by the Hamiltonian operator.
    Use piece wise definition as provided in the physics introduction. And SI units
    ""


def H_matrix(N):
    """
    Return a N X N matrix H
    """
    initialize H
    loop for each row
        loop for each column
            H[row, column] = Hnm(row, column)


# use eigh to find eigenvectors and eigvalsh for eigenvalues
# (since H is symmetric)

# find eigenvalues for H of 10 X 10, and H of 100 x 100
```

```
values, eigenvectors = la.eigh(H_matrix(100))
eigenvalues_1 = la.eigvalsh(H_matrix(10))
eigenvalues_2 = la.eigvalsh(H_matrix(100))

# (convert to eV to compare with energy provided in book)
print("ground energy in eV for N = 10, E =", eigenvalues_1[0] / 1.602176634e-19,
      "for N = 100", eigenvalues_2[0] * 6.242e18)
```

We get the following printout:

”Ground energy in eV for N = 10, E = 5.8363731713281295 for N = 100, E = 5.836831830217487”

Where N is the size of the square matrix.
We can see that for a ten fold increase in matrix size we have the ground energy change by 0.0005 eV, thus even with an H matrix of size 10 we have a fairly accurate calculation of the energy in the states. (We have an 0.08% increase in accuracy when we increase the matrix size 10 times).

e)

We define two additional function, the wave_function and the wave_density_function. Where the wave function just calculates the wave at a point x given the excitation state (and the appropriate coefficients calculated earlier). And the wave_density_function just take the absolute value of the wave function and squares it. We then use the Gaussian integration as written in Lab 3, (found in the functions file) from 0 to L with a step size of 500, which is big enough to get an accurate calculation without being too computationally expensive.

We can then compute of $|\psi(x)|^2$ (normalized) for each energy state.

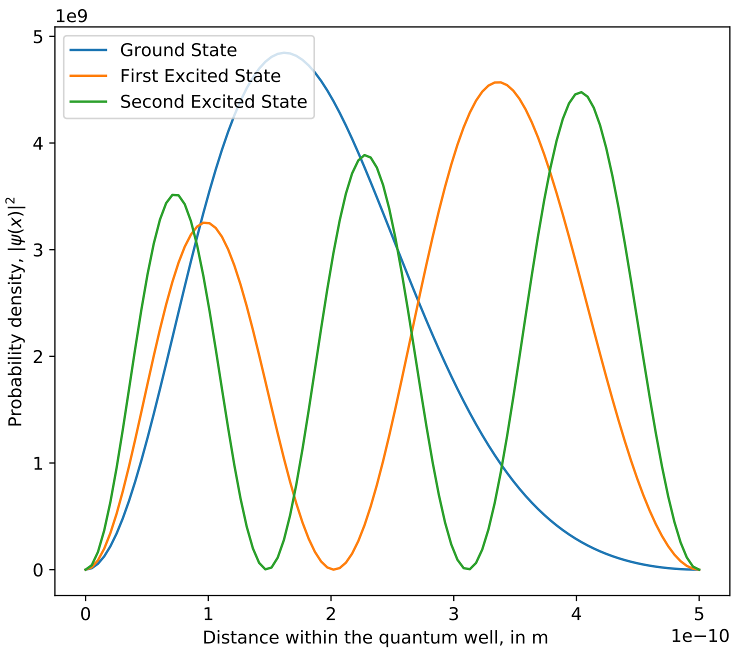

Figure 5: The probability density function for the first three energy levels in a quantum well

We can check that our probability density is normalized by recomputing the integral, and indeed we get $A_{ground} = 1.0000000000000002$, $A_{first} = 1.0000000000000004$, $A_{second} = 0.9999999999999991$, where $A = \int_0^L |\psi(x)|^2\,dx$. Which means that our function, $|\psi(x)|^2$ was normalized well enough.

Question 3

a)

We created a programme which specifically solved $x = 1 - e^{-cx}$ to an accuracy of 10^{-6}. We utilised the error function for method of relaxation as per the textbook. This code was later adapted for the general relaxation function, `relaxation_estimator` for later parts. We iterated over a range of values of c and obtained the plot as per Fig 6.

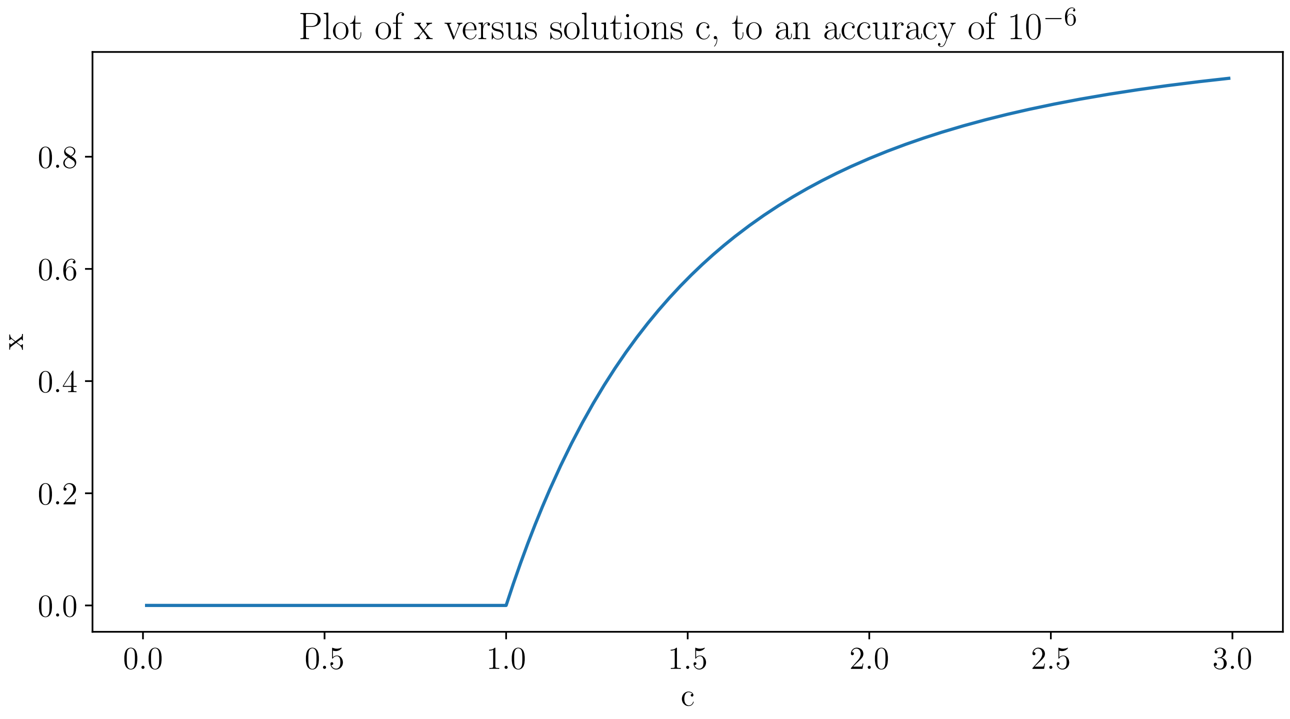

Figure 6: Solutions x to an accuracy of 10^{-6} from $x = 1 - e^{cx}$ against c.

b)

We wrote 2 functions `relaxation_estimator` and `error_relaxation` , found in `Functions.py` which solve equations using the relaxation method, given an initial guess, a desired accuracy, the function $f(x) = x$ and it's derivative $f'(x) = 1$. We found that the it took 13 iterations before we got `x = 0.7968111808647894` with the desired accuracy of 10^{-6}.

c)

We wrote a for loop which loops over values of w, starting from w = 0 to w = 1 to find the value of the optimal parmeter. The results are displayed below

ω	value of x	number of iterations
0.00	0.7968111809	13
0.05	0.7968111776	12
0.10	0.7968117689	12
0.15	0.7968117339	11
0.20	0.7968116525	10
0.25	0.7968114911	9
0.30	0.7968111689	8
0.35	0.7968118019	8
0.40	0.7968115701	7
0.45	0.7968119748	7
0.50	0.7968118378	6
0.55	0.7968114878	5
0.60	0.7968120464	5
0.65	0.7968120327	4
0.70	0.7968118544	3
0.75	0.7968123150	4
0.80	0.7968123494	4
0.85	0.7968123170	5
0.90	0.7968119669	6
0.95	0.7968113965	6

These results indicate that the optimal w values is 0.7, which reduces the number of iterations from 13 to 3. This makes sense because a bigger w, pushes x_{prime} to take a value closer to a multiple of f(x) if w becomes too big there starts too big a bigger influence from the $wf(x)$ term which starts to slow down the relaxation method.

d)

The circumstance where a negative ω value would come to a solution faster would be when your initial guess at the solution is already within the desired accuracy level. Than an ω close to -1, would give an $x_{prime} = (1-\omega)f(x_0) - \omega x_0 = (1-1)f(x_0) - (-1)x_0 = x_0$. The error would then have 0 in the numerator ($x = x_{prime}$). And to avoid a division by zero in the error we can have $\omega = -0.999$ which would still have give an error within the desired accuracy.

c)

We wrote 2 additional function `Binary_Search` and `Newton_method`, found in `Lab04_Q3c`.
Graphing the function $f(x) = x$, we see that the function has zeros at $x = 0$ and around $x = 5$. Since we are not interested in the trivial solution. We set the boundaries for the binary search at $x1 = 4$ and $x2 = 6$, for the relaxation method and Newton's method we have an initial guess at $x = 4$. We get the following print out:

```
The solution to the non-linear equation using binary x= 4.965114116668701
We use 21 iterations
The Wien displacement constant is equal to b = 0.002897772981687388
Therefore the temperature of the sun can be measured to be T =  5772.456138819498

The solution to the non-linear equation using relaxation x= 4.96511414525846
We use 5 iterations
The Wien displacement constant is equal to b = 0.0028977729650016422
Therefore the temperature of the sun can be measured to be T =  5772.45610558096

The solution to the non-linear equation using Newton's x= 4.965114231744276
We use 4 iterations
The Wien displacement constant is equal to b = 0.0028977729145262155
Therefore the temperature of the sun can be measured to be T =  5772.456005032302
```

We can see that Newton's method and the relaxation method used a quarter as many iterations as the binary search method. But that all three methods found the same solution within the desired accuracy.

We now construct a for loop to test how the speed of the methods varies with distance of the initial guess from the true value. For the binary method we will increase the distance from $x = 5$ for both x_1, x_2. For the relaxation and Newton's method we will subtract the distance from $x = 5$. We will iterate for distances from 0.5 to 4.5, since the binary method will not work for distances greater than or equal to 5 since the function has another root at $x = 0$
As shown in 7, we can see that the binary method is significantly slower than the relaxation and Newton's method. Newton's method dramatically increases the number of iterations it needs at an initial guess of $x = 1.5$. Figure 8,9, explains this because at an initial guess $x = 1.5$ to initial guesses $x = 0.5$, Newton's method give the solution of $x = 0$, which is the other solution (that we are not interested in), while the relaxation method and binary search method continue giving the solution $x = 4.965$.

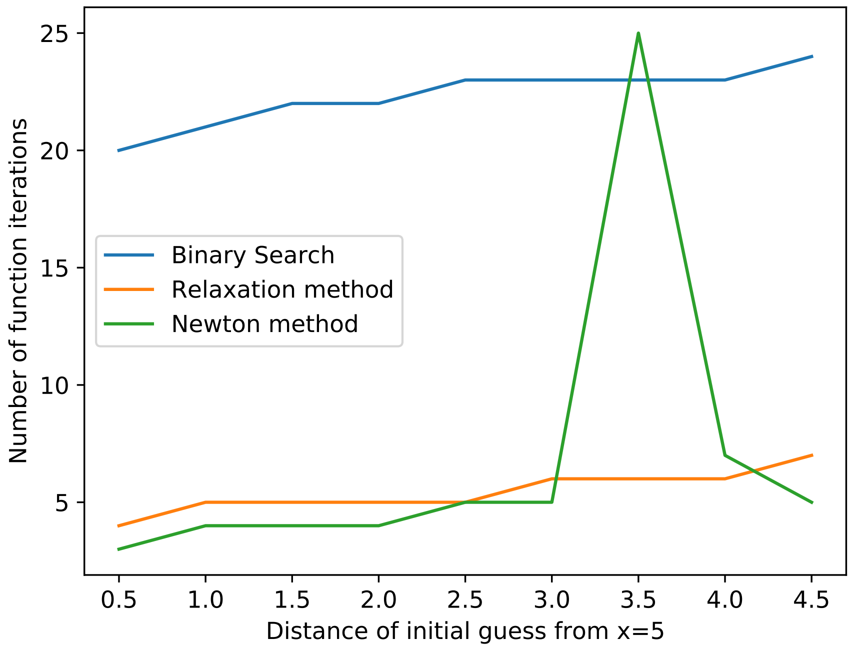

Figure 7: Number of iterations for each method

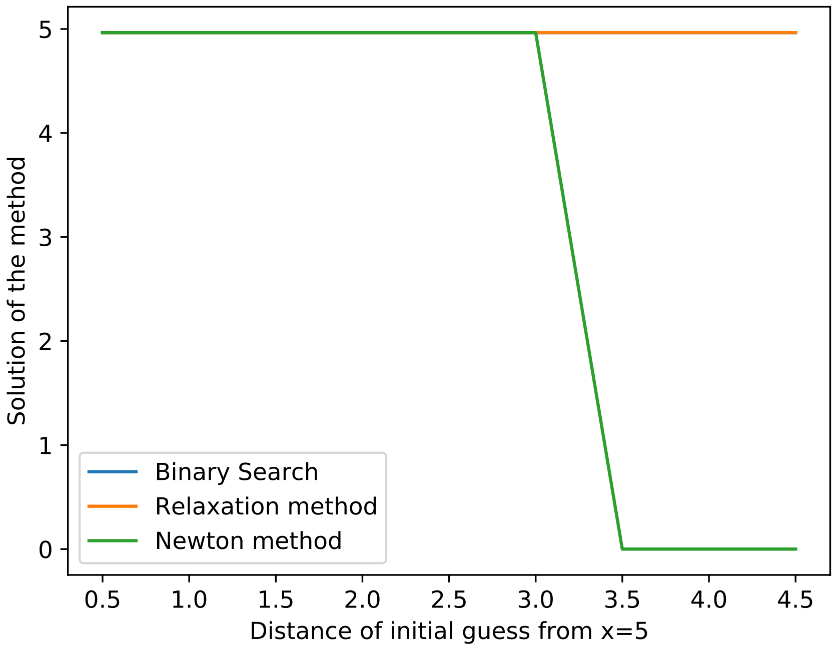

Figure 8: Solutions x to an accuracy of 10^{-6}.

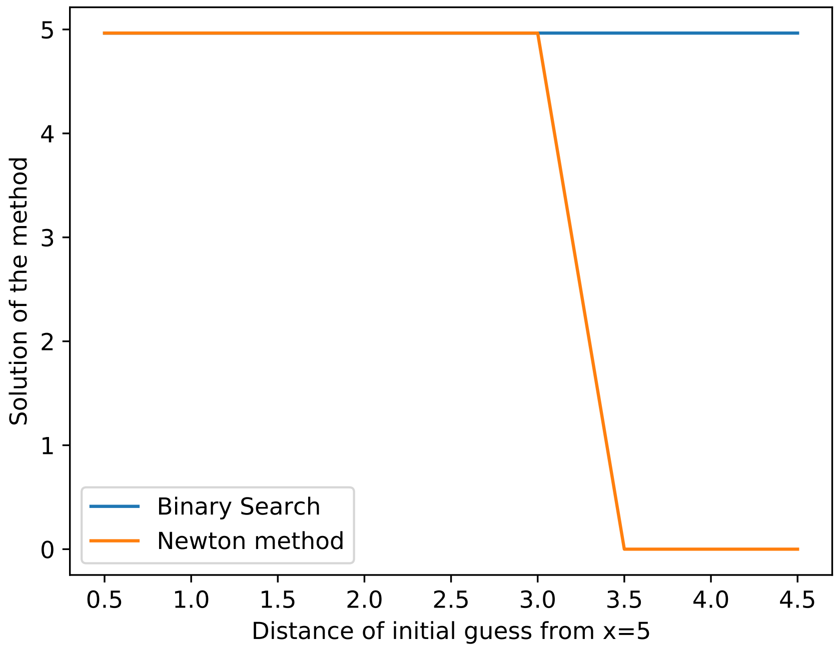

Figure 9: Solutions x to an accuracy of 10^{-6}.