Assignment Three

November 24, 2019

MCSC 6020G
Fall 2019
Submitted by Derick Smith

Question One:

Analysis of matrix $T_N \in \mathbb{R}^{\text{nxn}}$,

$$T_N = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}$$

a) Find L_N such that $L_N L_N^T = T_N$

Note: All matrices in $\mathbb{R}^{\text{nxn}}$ and no indices out of bounds.

First using cholesky decomposition, $T_N = PDP^T$,

$$P = \{p_{i,j}\}_{\forall i,j} \tag{1}$$

$$p_{i,j} = \begin{cases} 1 & i = j \\ \dfrac{1}{i+1} - 1 & j = i-1 \\ 0 & o/w \end{cases} \tag{2}$$

$$P^T = \{p_{j,i}\}_{\forall i,j} \tag{3}$$

$$D = \{d_{i,j}\}_{\forall i,j} \tag{4}$$

$$d_{i,j} = \begin{cases} 1 + \dfrac{1}{i} & i = j \\ 0 & o/w \end{cases} \tag{5}$$

Next split the diagonal matrix D and distribute among P and P^T,

$$D' = \{d'_{i,j} = \sqrt{d_{i,j}}\}_{\forall i,j} \tag{6}$$

$$PDP^T = PD'D'P^T \tag{7}$$

$$= (PD')\left(D'P^T\right) \tag{8}$$

$$L_N = PD' \tag{9}$$

$$l_{i,j} = \begin{cases} \sqrt{1 + \dfrac{1}{i}} & i = j \\ \left(\dfrac{1}{i} - 1\right)\sqrt{1 + \dfrac{1}{i-1}} & j = i-1 \\ 0 & o/w \end{cases} \tag{10}$$

$$L_N^T = \{l_{j,i}\}_{\forall i,j} \tag{11}$$

$$\boldsymbol{L_N L_N^T = T_N}$$

Note: The equations were developed and verified experimentally using Python. The code can be found in the file <choleskyDecomp.py>.

(b) All eigenvectors and eigenvalues of T_N.

Through eigendecomposition:

$$\boldsymbol{T_N = Q\Lambda Q^T}$$

The diagonal matrix of eigenvalues[1]:

$$\Lambda = \{\lambda_{i,j}\}_{\forall i,j} \tag{1}$$

$$\lambda_{i,j} = \begin{cases} 4\left(sin\left[\frac{\pi i}{2(n+1)}\right]\right)^2 & i = j \\ 0 & o/w \end{cases} \quad {}^1(2)$$

The column matrix of eigenvectors[1]:

$$Q = \{q_{i,j}\}_{\forall i,j} \tag{3}$$

$$q_{i,j} = \left\{\sqrt{\frac{2}{n+1}} sin\left(\frac{\pi ij}{n+1}\right)\right\}_{\forall i,j} \quad {}^1(4)$$

$$Q^T = \{q_{j,i}\}_{\forall i,j} \tag{5}$$

Note: The equations were found in the textbook, Matrix Computations by Van Loan and verified experimentally using Python. The code can be found in the file <eigDecomp.py>.

Question Two:

(a) & (b) Implementation of Newton-like optimizer

While experimenting with the helper function findXzero(), the classic Newton method with findXzero() was found to be substantially faster than any other combination. This includes the Newton-like method with or without findXzero().

The following are the frequency plots at the different combinations demonstrating their overall efficiencies at various δ's and number of σ's after 2000 iterations per combination.

Figure 2.a.1:

[1] Matrix Computations, Van Loan, p 229

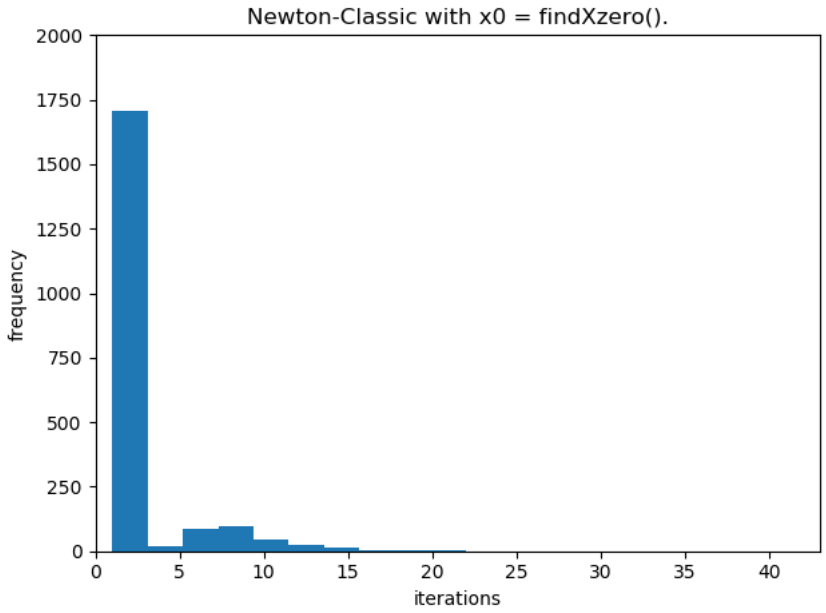


Figure 2.a.2:

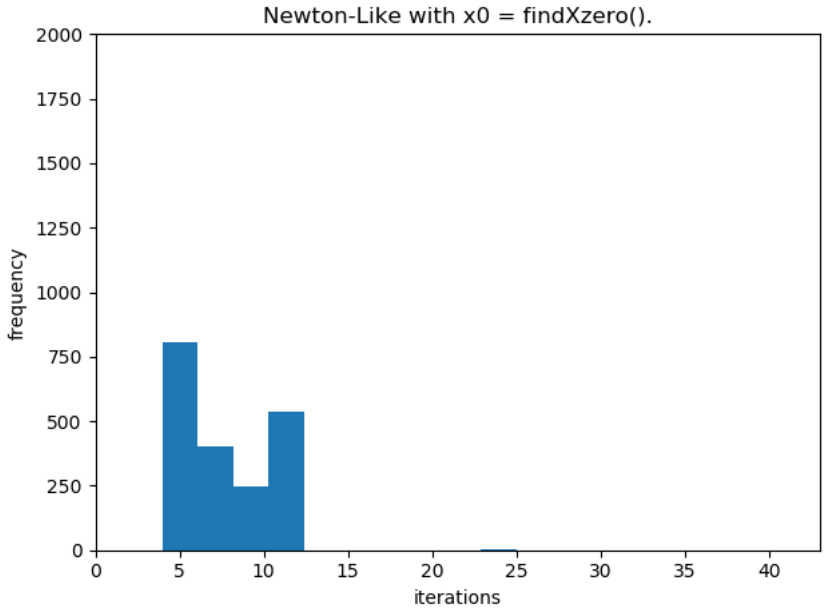


Figure 2.a.3:

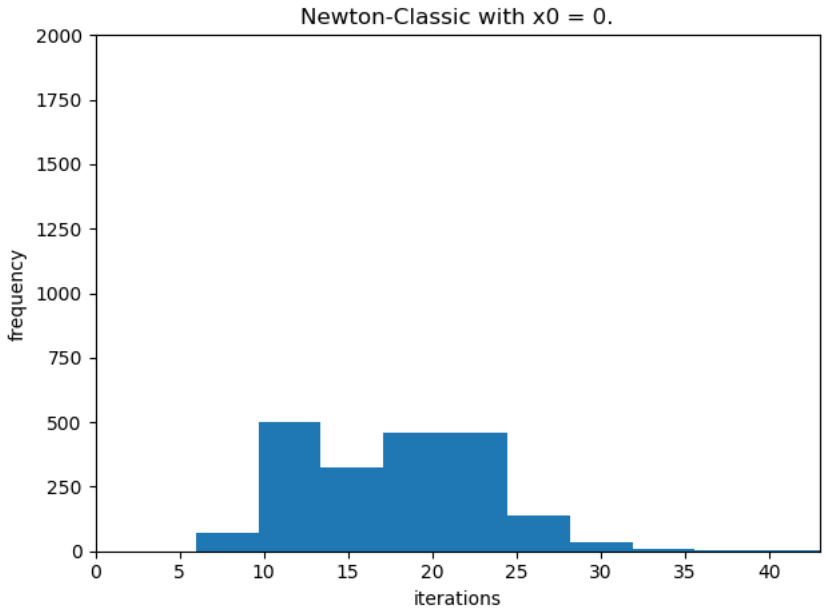
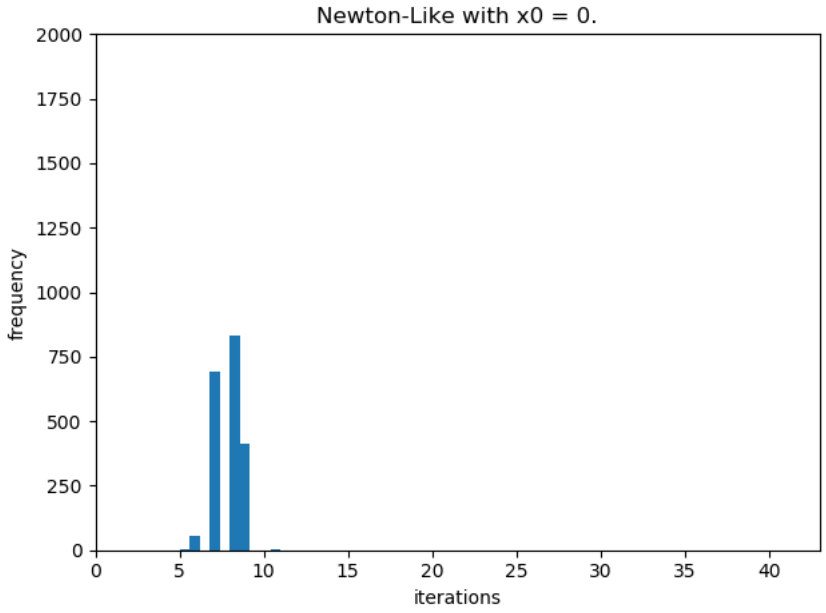


Figure 2.a.4:



Under certain conditions, the Newton-like method can converge exceptionally slowly or potentially not at all. Because Newton-like can jump to the right by

a large amount, there could be cases where the next iteration x_{k+1} causes the $f'(x_{k+1})$ to underflow a substantial portion of significant digits and a loss of gaurenteed convergence. Because of this, after ten iterations, if no convergence, Newton-classic was summoned.

Unfortunately, for the Newton-like method, even with this exception handling, it did not perform nearly as well as Newton-classic with findXzero() so it was scrapped.

Since the Newton-classic with findXzero() converges faster than Newton-classic on its own so it can be concluded that its convergences is also quadratic. Plots of the residuals vs iterations in this instance are graphically useless because the median case has two plot points nearly 90% of the time.

Using finite difference approximations of derivatives does not change the number of iterations for convergence in any method. The frequency plots and residual plots are indistinguishable, however, the computation time is noticably longer as a result of there being more calculations required to approximate each tangent line per iteration.

c) Exception handling

As mentioned in part (a), the Newton-like method was scrapped so it requires no exception handling. If the trust region becomes too small, $\delta \leq \delta_{min}$, the algorithm stops iterating as convergence may no longer be possible.

In regards to runtime exception handling, functions prone to exceptions are wrapped in try blocks. There are two exceptions that are not caught, however, no exceptions cause system crash. Trying to log-scale negative values. The other is a value error inside a few layers of python libraries and throws an exception long before the main program frame, making it difficult to catch.

d) Test variations

Create loops to vary number of equations (N), λ's, and initial conditions.[2]

The testing conditions for initial x,

$$x = x_0 = \{1_i\}_{i=1}^{N}$$

In an effort to nudge x as gently as possible with the addition of a vector x_Δ,

$$x_\Delta : X \in x_\Delta \sim \mathcal{N}(\mu = 0, \sigma^2 = \delta_{min}^2)$$

so after each inner most loop,

$$x = x + x_\Delta$$

One of the largest obstacles for convergence is the intial conditions. Optimization to find a global minimum over an infinite continuous domain could be impossible. If possible, the initial conditions could cause the conditions of

[2]See code for variation of conditions at <test_Newton-hook.py>

the algorithm to be satisfied but never converge. An example being, initial conditions at some point where $f(x)$ exponentially decays.

For improvements, as shown findXzero() seemed to accelerate things quite a bit in the file, <optimize.py>. For the other files, as discussed in class, creating a multidimensional discrete grid of f has large space and time complexities. A possible idea that wasn't explored fully is the idea that if f were a set of analytical functs. Because polynomials, and their derivatives, have "local" critical points relative to an infinite domain. Without understanding unconstrained optimization better, it seems possible that polynomial approximations of functions could potentially reduce the lattice cardinality but still gaurentee at least one lattice point per local min. This idea is not developed enough to be implemented.

The following are the results of varied conditions from the Newton-Hook method and the Newton-Raphson method which went without error and converged. Only the iterations where $x = x_0$ were the ones that converged. The red lines representing the residuals, the blue lines representing the step size, and the green lines representing the trust region.

Figure 2.d.1:

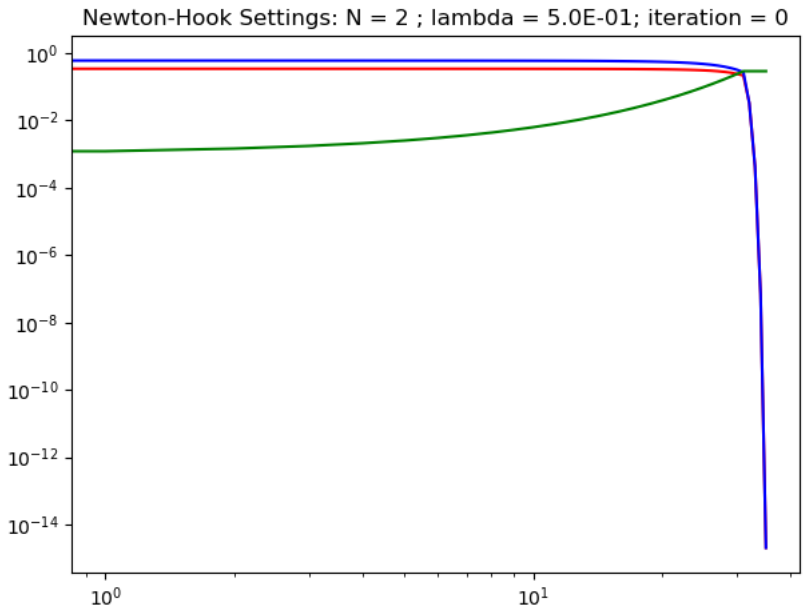


Figure 2.d.2:

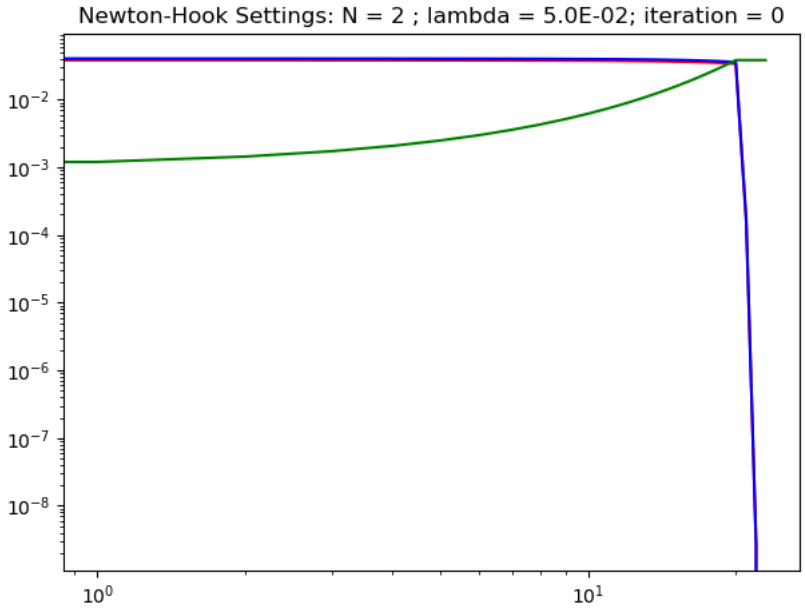


Figure 2.d.3:

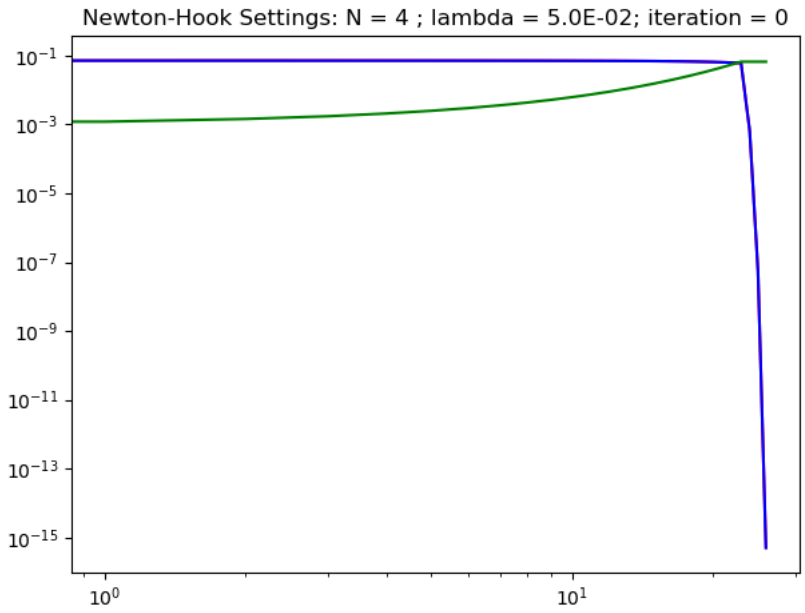

Newton-Hook Settings: N = 4 ; lambda = 5.0E-02; iteration = 0
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
10^{0}
10^{1}

Figure 2.d.4:

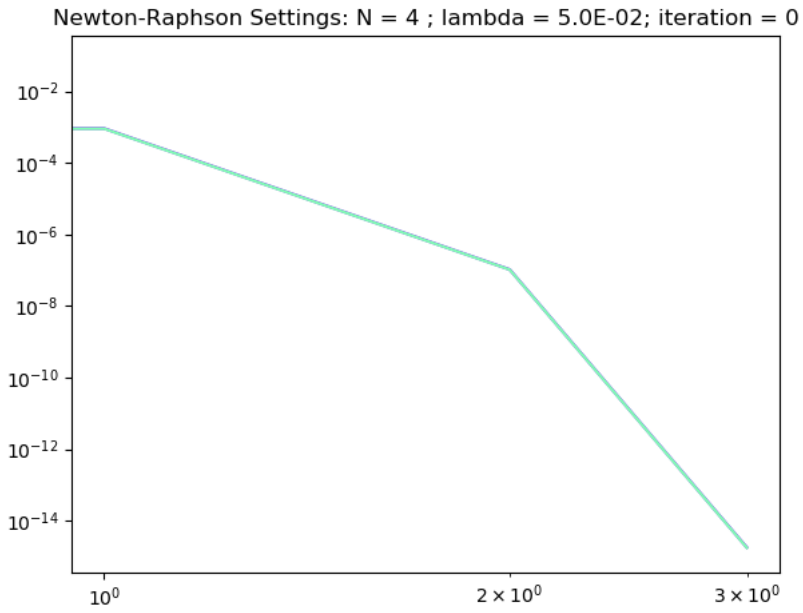


Figure 2.d.5:

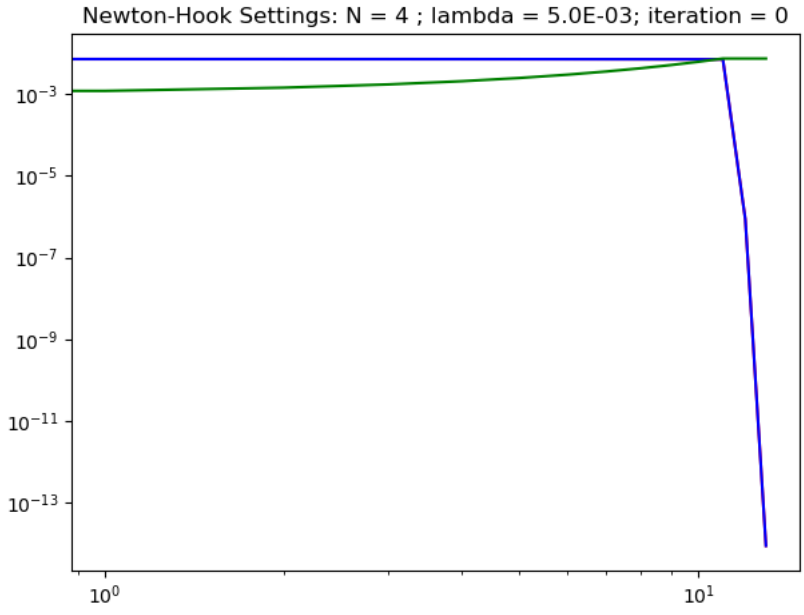
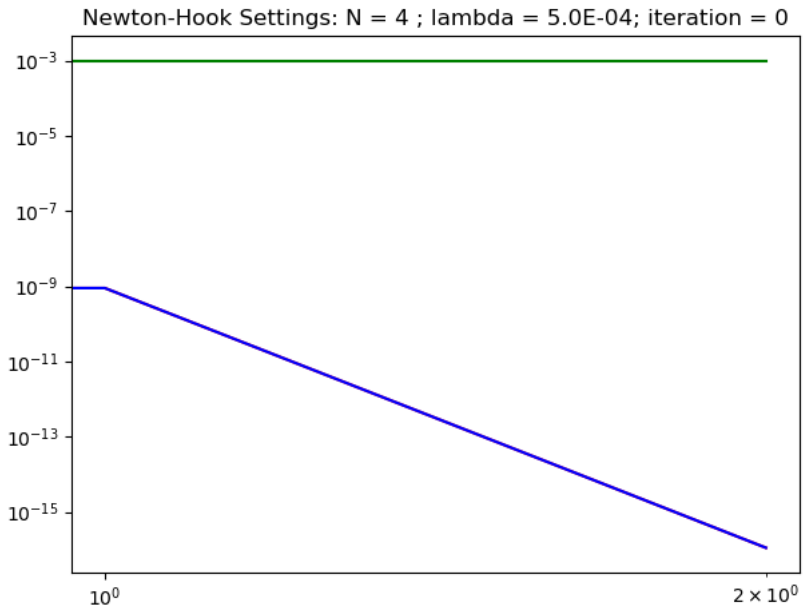


Figure 2.d.6:


Newton-Hook Settings: N = 4 ; lambda = 5.0E-04; iteration = 0
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
10^{0}
2×10^{0}

References

[1] Golub, G. H., & Loan, C. F. (2012). Matrix Computations. Baltimore, MD: JHU Press.