Example Solutions for Assignment 3

Question 1

(a) In this question, you are asked to find the value of a that makes the condition number of the matrix A equal to 4. Because the condition number is not defined in terms of elementary functions, we cannot find an explicit derivative, and therefore, we cannot use Newton-Raphson. However, we can use the secant method. We've defined our secant method code to find the zero of a function, and since here we want the condition number equal 4, we must define a new function f = condition number - 4, that we provide to the secant method.
The value of a which makes the condition number 4 is 3.851004086

Question 2

(b) From Lecture Notes 7, for the linear system $A\mathbf{x} = \mathbf{b}$ we have

$$\boxed{\frac{\|\mathbf{e}\|}{\|\mathbf{x}\|} \leq K(A)\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}}$$

and thus the right hand side gives the maximal relative error, where $K(A)$ is the condition number of A, $\mathbf{r} = A\mathbf{x}^* - \mathbf{b}$ is the residual vector, and $\mathbf{x}^*$ is the computed approximate solution (using the LUPsolve.py function). Here we want to find the maximal relative error when A is V the Vandermonde matrix with $n = 25$, and $\mathbf{b}$ is the next to last column of V. We use the 2-norms for all norms. As such we have:

- $K(V) = 2.131438e + 11$
- $\|\mathbf{r}\| = \|A\mathbf{x}^* - \mathbf{b}\| = 1.676859e - 16$
- $\|\mathbf{b}\| = 3.059412$

$$\rightarrow \qquad \text{Maximal Relative Error} \quad = K(V)\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|} = 1.168238e - 05$$

It is too cumbersome to print out $\mathbf{x}^*$ here, but it's worthwhile to have a look at it on your own to see how clearly it differs from the expected solution. The point is that when the condition number gets large, then, even when your residual is small (essentially as good as you can get with finite precision arithmetic), your approximation can be not so good.

(c) We'll say that the approximation loses meaning when the maximal relative error is 1, i.e. when we can no longer be sure of any of the digits (we have zero digits of accuracy). This occurs when n is close to 35, and when the condition number is close to $1.0e + 16$. Note that regardless of n, the relative residual is always very small $< 1.0e - 15$.

Using the infinity norm doesn't really change the conclusion much. The maximal relative error computed using the infinity norm seems to be about two times bigger than that computed using the 2-norm.

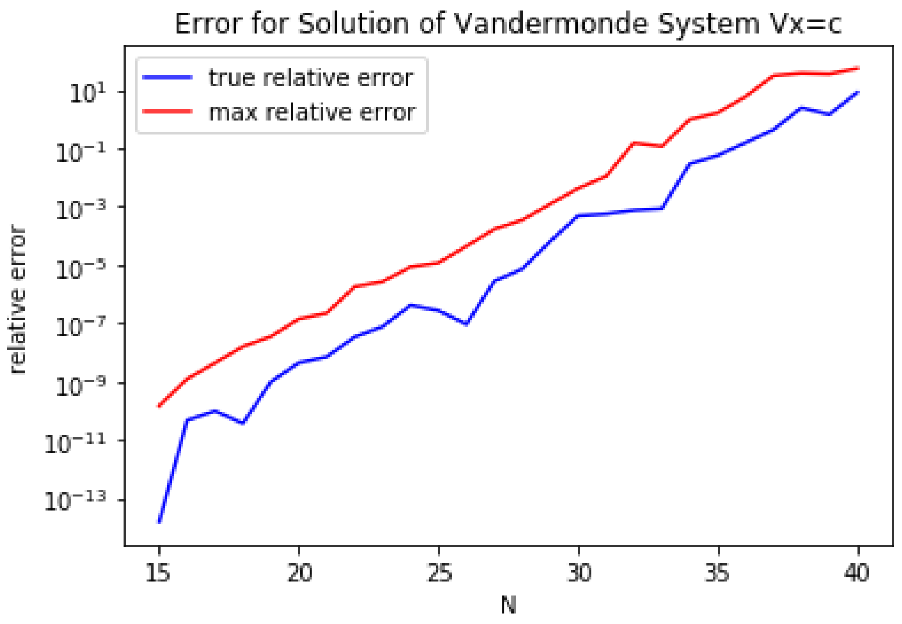

Figure 1: The maximal and actual relative errors vs size of linear system involving Vandermonde matrix.

Question 3

(a) Pseudo-code for (efficiently) computing the product matrix-vector product involving a unit lower triangular matrix

Input:$^{-}$ $n \times n$ array of floats L which represents a unit lower triangular matrix $L \in \mathbb{R}^{n \times n}$, and an $n \times 1$ array of floats $\mathbf{x}$

```
1. $c_1 \leftarrow x_1$
2. for $j = 2 : n$ do
3.     $c_j \leftarrow x_j$
4.     for $k = 1 : j - 1$ do
   (a)     $c_j \leftarrow c_j + A_{jk} \times x_k$
5.     end $k$ loop
6. end $j$ loop
```

Output: $\mathbf{c} = L\mathbf{x}$, an $n \times 1$ array of floats

(d) **Computational Complexity**: On line 4(a) of pseudo-code, there are 2 FLOPS: 1 multiply and 1 addition; Line 4(a) executes once for each $k = 1\!:\!j-1$ *and* $j = 2\!:\!n.$:

$$\begin{aligned} FLOPS &= \sum_{j=2}^{n}\sum_{k=1}^{j-1} 2 \\ &= \sum_{j=2}^{n}\left(2\sum_{k=1}^{j-1} 1\right) \\ &= \sum_{j=2}^{n} 2(j-1) \\ &= \sum_{j'=1}^{n-1} 2j' \\ &= (n-1)(n-2) = n^2 - 3n + 2 \\ &= O(n^2) \end{aligned}$$

So this algorithm is $O(n^2)$; note that the usual matrix-vector product is also $O(n^2)$, specifically it requires $2n^2 - n$ FLOPS. That is, our algorithm for unit lower triangular matrices requires approximately half as many FLOPS as the original, but still requires the same order of FLOPS.

(g) The log-log plot of computation time vs matrix size is what we would expect. The slope of the graph is approximately equal to the exponent p in $O(n^p)$ (where here $p = 2$), i.e. computation time increases as the square of the matrix size. Because the usual matrix-vector product is $O(n^2)$, we would expect this to also have the same slope, but be a little higher on the plot because it requires more computations (as is seen in the figure).

(h) The built-in function is *way* faster, regardless of whether you take into account the special form of L. It turns out that the built-in function for matrix-vector multiplication is much more efficient than the ones you just wrote. The built-in functions are written in a lower-level compiled language and are optimized for the hardware on your computer. Loops in scripting languages (e.g. Python) tend to be where you seen big differences (i.e. much slower execution) compared with the compiled languages.

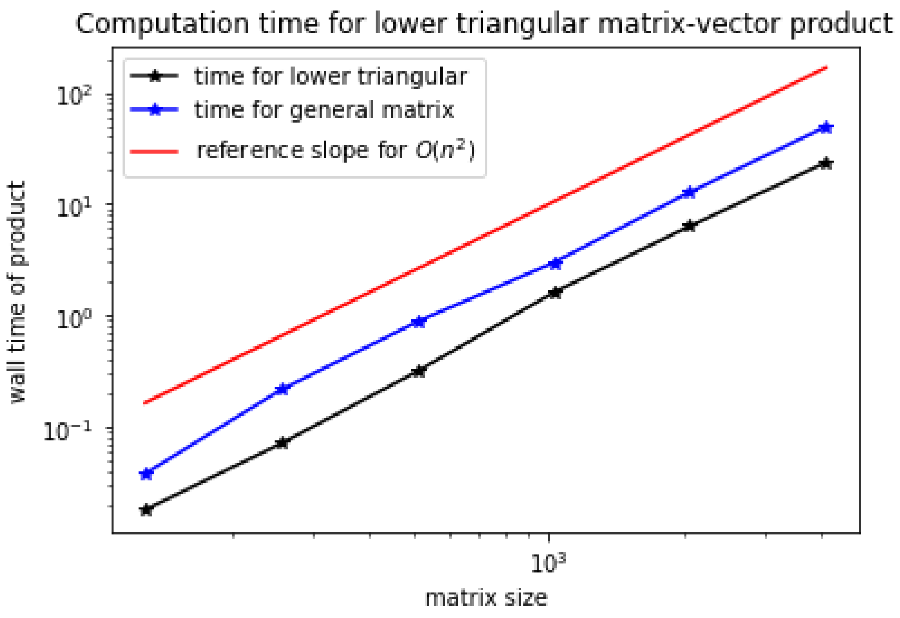

Figure 2: The computation time vs matrix size for computing matrix-vector product using the special algorithm for lower triangular matrices (black) and for a general matrix (blue). The red line has slope of 2 (on the log-log plot) for comparison. In particular, an algorithm with this slope will be $O(n^2)$.