Linear systems and complexity

Due: Monday, February 27, 11:59pm.

Push your assignment solutions to the appropriate GitHub Classroom repository. Submit the following for this assignment:

- A PDF file, typeset in LaTeX, with answers to questions 1(b), 2(b),(d) and 3(a),(d),(g),(h).
- Python functions defined in files called `CondNumFuncA3.py` for question 1(a), `LUPsolveA3.py` for question 2(a), `LowTriangMatvecA3.py` for question 3(b), and `GenMatvecA3.py` for question 3(c). Use the templates provided in the assignment repository.
- Python scripts called `Quest1.py` for question 1(b), `Quest2.py` for question 2(c), and `Quest3.py`, for question 3(e),(f),(h). Use the templates provided in the assignment repository.
- Two plots, 1 each for questions 2(c), and 3(f).

Question 1 — 25 marks

We saw that the Secant Method can be useful when we're trying to find the zero of a function whose derivative is hard to compute, such as, if the function itself can't be written down easily. For example, consider the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 3 & a & 1 \\ 1 & -2 & 3 \end{bmatrix} \tag{1}$$

We may want to find the value of a that will make the condition number of this matrix something small, e.g. 4.

(**a**) In the file `CondNumFuncA3.py`, write a Python function called `CondNumFunc` that inputs a value of a and returns the condition number of the matrix A given in equation (1) for the corresponding value of a. Use the starter code found in the repository.

(**b**) Write a script (Python code) called `Quest1.py` that uses this function and the function for Secant iteration found in `Secant.py` to approximate the value of a that makes the condition number of A equal to 4. Find your approximation accurate to 10^{-8}, and use starting guesses $x^{(0)} = 0$ and $x^{(1)} = 5$. Write out your approximate solution in LaTex. Hint: you can use the Secant method function without modification.

Question 2 — 30 marks

The *Vandermonde matrix* arises in interpolation problems, which we will be covering soon in class. In particular, in such problems, linear systems involving this matrix must be solved. As discussed in Lecture 7, the *Vandermonde matrix* is defined as

$$V_{ij} = x_{i-1}^{n-j+1} \quad \text{for } 1 \leq i \leq n+1 \text{ and } 1 \leq j \leq n+1$$

The *Vandermonde matrix* for $n = 4$ is:

$$V = \begin{bmatrix} x_0^4 & x_0^3 & x_0^2 & x_0 & 1 \\ x_1^4 & x_1^3 & x_1^2 & x_1 & 1 \\ x_2^4 & x_2^3 & x_2^2 & x_2 & 1 \\ x_3^4 & x_3^3 & x_3^2 & x_3 & 1 \\ x_4^4 & x_4^3 & x_4^2 & x_4 & 1 \end{bmatrix}$$

Let $x_i = -1 + i\Delta$ for $i = 0, \ldots, n$ and $\Delta = 2/n$ (gives equally spaced points between -1 and 1). In Python, this matrix, for any n can be constructed using the `numpy` function `vander`, as shown in Lecture 7 Slides, as well as in the `Accuracy_LinSys.py` code in the course GitHub repository.

It turns out that as n gets large, this matrix becomes very poorly conditioned, and therefore the solutions of linear systems involving it become inaccurate. In this question we explore this further.

(a) Create a function that solves general linear systems using LU-decomposition with partial pivoting. In particular, write a Python function `LUPsolve` (contained in the file `LUPsolveA3.py`) that inputs an $n+1 \times n+1$ numpy array A and an $n+1 \times 1$ numpy array $\mathbf{b}$, and computes the decomposition $PA = LU$ (i.e. with partial pivoting), and uses it to solve the linear system

$$A\mathbf{x} = \mathbf{b}$$

The inputs of your function should be A and $\mathbf{b}$, and the output of your function should be the solution $\mathbf{x}$, as well as L, U, and P, in that order. Hint: you can use the `scipy` function `scipy.linalg.lu` to compute the L, U, but note that this function does not provide the appropriate P.

The following parts involve solving systems of equations with the matrix V, as defined above.

Define a vector $\mathbf{c}$ equal to the second to last column of V (i.e. the nth column of V), and consider the linear system

$$V\mathbf{x} = \mathbf{c}$$

The exact solution of this system of equations is $\mathbf{x} \in \mathbb{R}^{n+1}$, where

$$\mathbf{x} = \mathbf{e}_n = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

(check, without solving the system explicitly, that this indeed must be the case).

(b) Compute the maximal relative error (according to the theoretical upper bound discussed in Lecture 7) for $n = 25$. Compute the true relative error, and compare to the maximal relative error. Use the 2-norm for all vector norms. Present your answer in Latex. Show your work; write down the formulas used, and indicate the values of each variable in your formula, and indicate which functions were used to compute them, e.g. use your `LUPsolve` function to find x^*.

(c) Write a Python script called `Quest2.py` that calls your function `LUPsolve` to solve $V\mathbf{x} = \mathbf{c}$, computes the relative error and the maximal relative error (according to the theoretical upper bound discussed in Lecture 7) for all $n = 15, \ldots, 40$, and plots them versus n together on a semilogarithmic scale (i.e. n on the horizontal axis and the vertical axis on a log-scale). Use the 2-norm for all vector norms.

(d) Up to what matrix size does the numerically obtained solution have any meaning? Explain. What is the condition number of V and what is the relative residual for this value of n? How does using the max-norm affect your answers to these questions?

Question 3 **30 marks**

A unit lower triangular matrix $L \in \mathbb{R}^{n \times n}$ can be written as

$$A = \begin{bmatrix} 1 & & & & \\ a_{2,1} & 1 & & & \\ a_{3,1} & a_{3,2} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ a_{n,1} & a_{n,2} & \ldots & a_n & 1 \end{bmatrix} \tag{2}$$

where the entries of the matrix that are not shown are 0. That is, a unit lower triangular matrix has all entries above the main diagonal equal to 0, while all the entries along the main diagonal are equal to 1. The entries below the main diagonal can be any real number. You might imagine that, due to the persist presence of the zeros and ones, we could write a function that computed the product of a unit lower triangular matrices with a vector more efficiently than one that did not take the special form of the matrix into account. In this question, we will investigate this further.

(a) Write a pseudocode for computing the matrix-vector product of a unit lower triangular matrix L with a vector $\mathbf{x}$ that avoids unnecessary computations (e.g. multiplication by 0 or 1). That is, given the two dimensional array L and a one-dimensional array $\mathbf{x}$ that define L (you can assume that you have already checked that it is indeed a unit lower triangular matrix), and the given vector $\mathbf{x} \in \mathbb{R}^n$, your algorithm should compute the matrix-vector product $\mathbf{y} = A\mathbf{x}$. Submit your answer type-set using LaTeX. Hints: To get an idea of how the algorithm should work, create a unit lower triangular matrix and vector $\mathbf{x}$ for $n = 4$ or 5, and compute the product by hand. Do not include your by-hand computation in your assignment submission. You may also want to have a look at the pseudo-code for matrix-vector multiplication in the case when the matrix doesn't have a special form, which can be found in the Lecture 8 Slides.

Your pseudocode should have the following form:

Input: an $n \times n$ array of floats L (for which $L_{i,j} = 0$ if $j > i$, and for which $L_{j,j} = 1$), and an array of n floats $\mathbf{x}$.

$\vdots$

Insert pseudocode here

$\vdots$

Output: array of n floats $\mathbf{y}$ such that $\mathbf{y} = L\mathbf{x}$

(b) Implement your pseudo-code from part (a) in a function called `LowTriangMatVec` defined in a file called `LowTriangMatVecA3.py`. Use the starter code that is provided.

(c) For comparison purposes, in a function called `GenMatVec` defined in a file called `GenMatVecA3.py`, implement the pseudo-code for matrix-vector multiplication in the case when the matrix doesn't have a special form, that is included in the Lecture 8 Slides. Use the starter code that is provided.

(d) Analyse the complexity of the algorithm from part (a). That is, determine how many flops are required to compute the product $\mathbf{y} = A\mathbf{x}$ with your algorithm. In terms of "Big-Oh" notation, what is the asymptotic behaviour of your algorithm as n increases? How does this compare with the algorithm for computing the general matrix-vector product you've implemented in part (c).

(e) Write a script called `Quest3.py` that generates a unit lower triangular test matrix L of size $n \times n$ and a test vector $\mathbf{x}$ for $n = 2^k,\ k = 7, \ldots, 12$, computes the product $L\mathbf{x}$, and measures the time your code takes to complete. Also, try for $n = 2^k,\ k = 7, \ldots, 13$, or 14. You can use the numpy random number generator `numpy.random.rand(n)` to create your L and $\mathbf{x}$, and the Python function `time.time()` to measure the time.

(f) Add to the script of part (e) a comparison with the time taken to compute the product computed using the `GenMatVec` function from part (c), and produce a log-log plot of the time taken versus n (i.e. both axes on a logarithmic scale, and with n on the horizontal axis), which compares the computation times of the two algorithms.

(g) Is the plot what is expected based on your analysis of the complexity of the algorithm? In particular, what are the slopes of the graphs on the log-log scale? Explain.

(h) Add to the script of part (e) a comparison to the time required for the Python built-in function to compute general matrix-vector products (i.e. `numpy.matmul` or using the `@` operator). In this case, instead of plotting the results, print out and compare the computation times with those computed using the codes of parts (b) and (c). Describe what you observe and give an explanation for these observations.