# Numerical Optimization and Linear Algebra

## Assignment 1

Mouselinos Spyridon – Part Time

## Question 1: Gaussian Elimination Implementation

The implementation of both partial and full pivoting was implemented in a Python Class named LU_eliminator.

The class implements the following methods:

### 1. *perform_LU_analysis_partial:*

Takes as input a square numpy matrix A and uses partial pivoting to create the matrices L (lower triangular), U (upper triangular), P (permutation matrix) as well as the variable elapsed (the execution time in seconds). The following equation holds:

$LU = PA$, according to the LU decomposition. As a memory optimization step, the L and U parts can be saved in-place during execution in matrix A, with the method storing L as the lower triangular part of A and U as the upper triangular part. Finally, the execution time is measured after the initialization of matrices L,U and P until the end of the iterative method.

### 2. *perform_LU_analysis_full:*

Takes as input a square numpy matrix A and uses full pivoting to create the matrices L (lower triangular), U (upper triangular), P (row permutation matrix) , Q(column permutation matrix), as well as the variable elapsed (the execution time in seconds). The following equation holds:

$LU = PAQ$, according to the LU decomposition. The same rules as partial analysis apply to the memory optimization and execution time procedures.

### 3. *linear_solve_partial:*

Takes as input a square numpy matrix A and a target vector b.

First, it calls the perform_LU_analysis_partial method to obtain the L,U and P components and finally solves the linear system Ax=b, by 2-step substitution:

- solve the system $Ly = Pb$ for $y$,
- solve the system $Ux = y$ for $x$,

thus, obtaining vector x.

### 4. *linear_solve_full:*

Takes as input a square numpy matrix A and a target vector b.

First, it calls the perform_LU_analysis_full method to obtain the L,U,P and Q components and finally solves the linear system Ax=b, by 3-step substitution:

- solve the system $Lz = Pb$, for $z$,
- solve the system $Uy = z$ for $y$,
- solve the system $Q^T x = y$ for $x$,

thus, obtaining vector x.

## Question 2: On the Solution of the Toeplitz Matrix

For the purpose of this exercise we will need to create 5 linear systems of the form $Ax = b$, with A being the respective Toeplitz Matrices of size: [64, 128, 256, 512, 1024].

### *1. Creation Step*

For this reason, for each size (n) we:

- Create a random vector x of size n, using NumPy's random number method.
- Create the n_sized Toeplitz Matrix A, using create_toeplitz_matrix(n) function
- Calculate the dot product of two and store it as the target vector b of size n.

For the rest of the exercise we will act as matrix A and b were given, and we try to solve for x.

### *2. Solution Step*

Again, for each size (n) we:

- Calculate and save the condition number of matrix A.
- Create a Partial Pivoting Linear Solver of the LU_eliminator class.
- Create a Full Pivoting Linear Solver of the LU_eliminator class.
- Use the partial solver to solve the system for x, get solution px and execution time ptime.
- Use the full solver to solve the system for x, get solution fx and execution time ftime.
- Calculate the infinite norm of the backward partial solver perror = $|x - px|_{inf}$
- Calculate the infinite norm of the residual of the partial solver res_partial = $|b - A * px|_{inf}$
- Calculate the infinite norm of the backward full solver ferror = $|x - fx|_{inf}$.
- Calculate the infinite norm of the residual of the full solver res_full = $|b - A * fx|_{inf}$

*Results:*

| size | condition_number | cpu_times_partial | cpu_times_full | error_partial | error_full | res_partial | res_full |
|---|---|---|---|---|---|---|---|
| **64** | 4550790 | 4.986328 | 4.015137 | 4.80E-11 | 6.39E-12 | 4.26E-14 | 2.84E-14 |
| **128** | 70460127 | 8.975342 | 11.96729 | 3.96E-10 | 5.92E-10 | 7.11E-14 | 3.55E-14 |
| **256** | 1.11E+09 | 28.92285 | 36.90137 | 1.88E-08 | 6.42E-09 | 1.14E-13 | 5.33E-14 |
| **512** | 1.76E+10 | 342.0847 | 516.616 | 2.13E-07 | 1.09E-07 | 1.35E-13 | 7.82E-14 |
| **1024** | 2.8E+11 | 3380.04 | 5243.968 | 1.62E-06 | 5.38E-07 | 1.99E-13 | 1.14E-13 |
| **2048** | 4.47E+12 | 31302.26 | 46614.3 | 3.96E-05 | 1.98E-05 | 3.27E-13 | 2.84E-13 |

Before passing to the analysis of the results, we can easily state that the size of the matrix is highly correlated with the condition number of the matrix. Especially, for every x2 increase in size we have a x10 increase in the condition number. Furthermore, we expect the partial pivoting algorithm to perform worse in terms of error but better in terms of clock time since from theory it is the best tradeoff between accuracy and complexity. The full pivoting is expected to perform better but at a higher time complexity penalty, since it performs optimally at every step the pivoting method.

The results confirm our assumptions.

*Analysis of the infinite norm of the error*

Regarding the infinite norm of the absolute backward error: $|x-px|_{inf}$ or $|x-fx|_{inf}$ , then according to Wilkinson's equations:

$$\frac{\|\hat{x}-x\|}{\|x\|} \leq \frac{\text{cond}(A)}{1-\text{cond}(A)\varepsilon_M}(\varepsilon_M+\varepsilon_M) \approx 2\,\text{cond}(A)\varepsilon_M$$

Then for backward error we have:

Backward = $|x-x^*| \leq 2 * e_{machine} * cond(A) * |x|$

For each of the sizes 64 through 2048 the product $2 * e_{machine}$ is constant, let us name it c. Then:

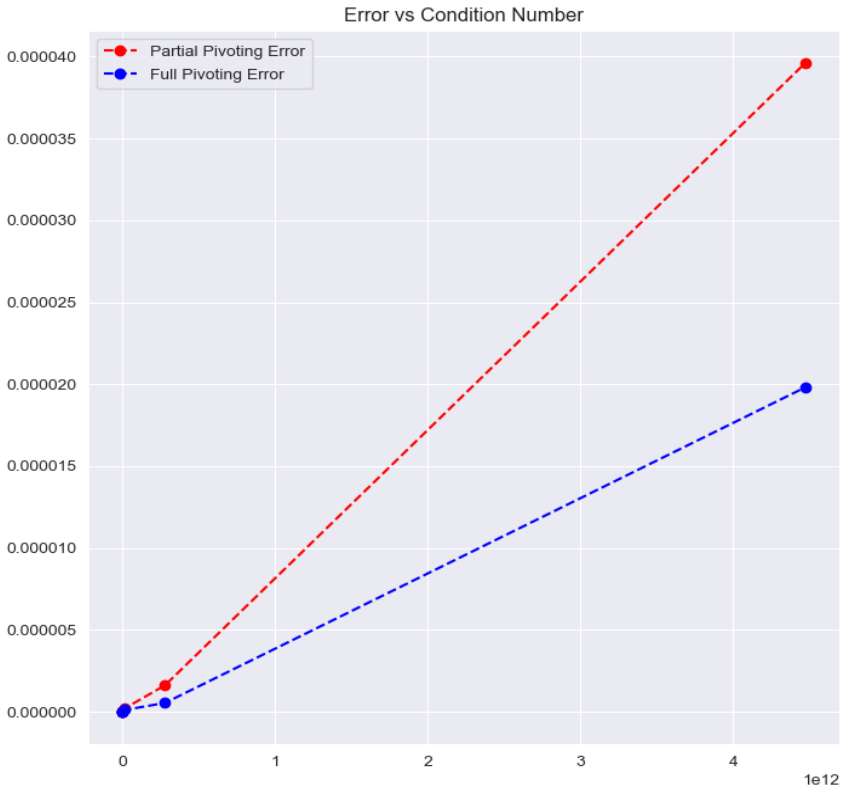
$$BackwardError = O\big(cond(A)\big) * |x|$$

Since x's are produced randomly from the same Uniform[0-1] distribution we can calculate the infinite norm of |x| for all the sizes and make it equal to 1. Without loss of generality, by keeping the same random number generation process the infinite norm of |x| could be simply bounded by d, across all examples. So, for $d = 1$ we have:

$$BackwardError = O\big(cond(A)\big)$$

We can observe this behavior in the result table where for size 64 and condition_number in the magnitude of $O(10^6)$ both partial and full algorithms give errors in the magnitude of $O(10^{-11})$ and O(n^-12) respectively, while at size 256 where a change of $O(10^3)$ is shown in the condition number we expect a same magnitude change in the errors. Indeed, we observe that both errors rise at a magnitude of exactly $O(10^3)$.

So regarding the question "what would happen when we move from size 1024 to size 2048" we can pre-calculate that based on the magnitude of change in the condition number that is approximately $O(10^1)$ the error should rise by a factor of 10 in both cases that is very close to the results obtained by running the experiments for size 2048. The respective plot is shown below:



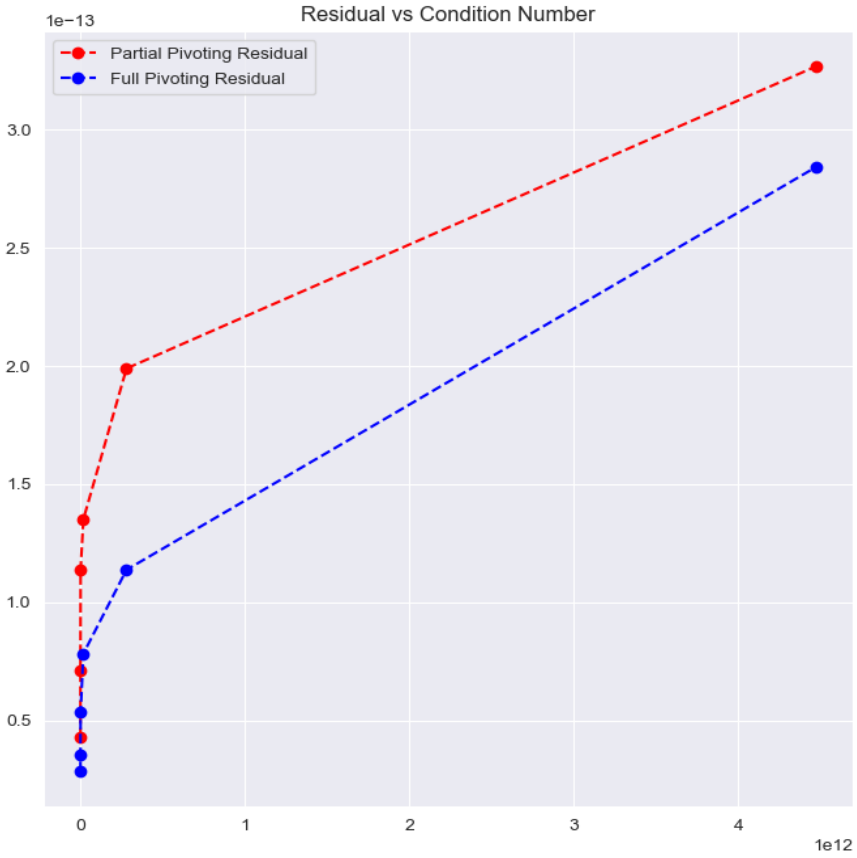
*Analysis of the infinite norm of the residual*

As far as the residual is concerned it is easy to show that $|b - Ax^*| = |A(x - x^*)| \leq |A| * |x - x^*| \leq e_{machine} * |A| * |x^*|$

However, because the infinite norm of a matrix equals to the maximum of the sum of the absolute row sums, every matrix A that we create from the formula given has a row sum that has as maximum value the diagonal entry while the rest are approaching 0 as size increases. Indeed, if we plot the infinite norm of the matrix a for sizes 64 through 2048, we can see that is stays around 96-97, since the summation comprises of quadratically decreasing terms. For this reason, we can cancel out the |A| of the equation replacing it with a constant $g = 97$. Now:

$$Residual \leq \mathrm{e}_{\mathrm{machine}} * 97 * 1 = 100O(\mathrm{e}_{\mathrm{machine}})$$

As we see in the experiments, indeed the residuals in both methods tend to stay around the same values and deviate slightly as the condition number increases. The residual of the partial pivoting is higher than the residual of the full pivoting as expected. So, regarding the question what would happen when we move from size 1024 to size 2048, we can safely say that we will see a slight change in the same order of magnitude as 1024. Indeed, the change is trivial in the actual experiment. The respective plot is shown below:



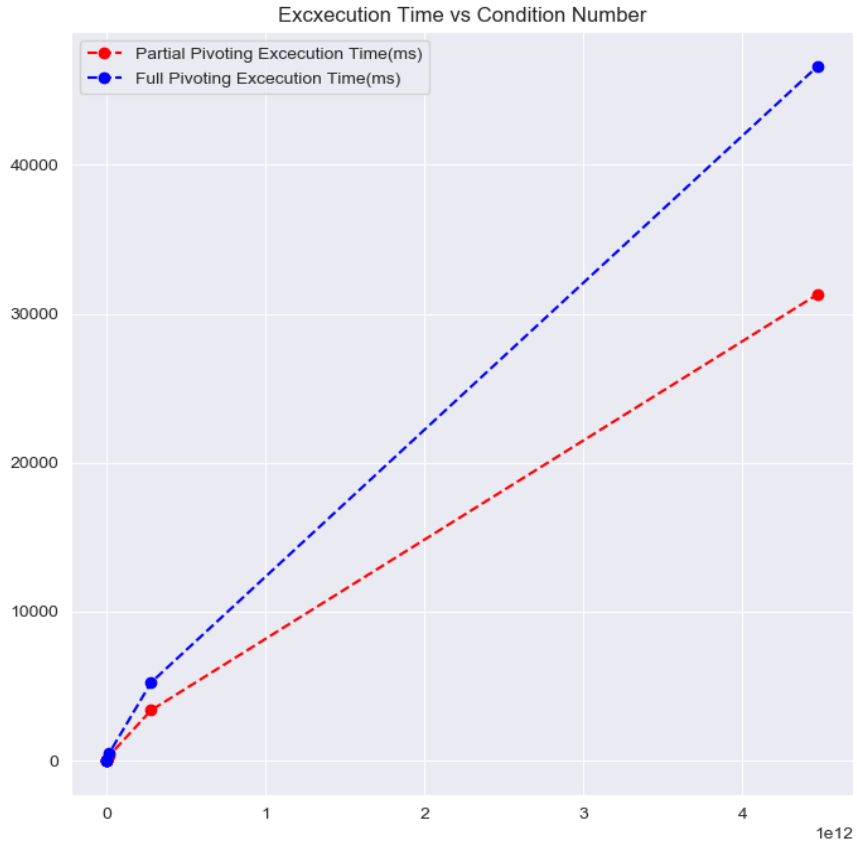
## Question 3: On the Time Complexity of the Algorithms.

The Partial Pivoting Algorithm has time asymptotic time complexity: $O(2/3 * n^3)$

The extension of this algorithm is the full pivoting algorithm that performs column pivoting as well and has asymptotic time complexity : $O(4/3 * n^3)$

As expected, for every doubling in size n the complexity rises by "almost" a factor of 8 (+ $O(n^2)$ terms) in both cases, something we can confirm especially as the size increases.

So, regarding the question what would happen when we move from size 1024 to size 2048, we could expect a similar behavior as the condition number, a change of magnitude $O(10)$, that is confirmed.



## Question 4: On the custom linear system

In this question we are required to perform all the previous steps for the matrix A, with 1's in the main diagonal as well as the last column, and -1's in the lower triangular part.

In partial pivoting, due to the form of the matrix, no pivoting is performed as the largest item in each step is the 1 on the main diagonal. Also, assuming we are in step j ,the $jth$ row is added to every row below itself in order to create 0's in the respective $jth$ column. That accumulates power of 2's in the last column where each item's value is given by $A[i,n] = 2^{i-1}, for\ i \in [1,n]$. That creates huge numerical instability during backwards substitution after the LU decomposition, leading to large errors and residuals in the process.

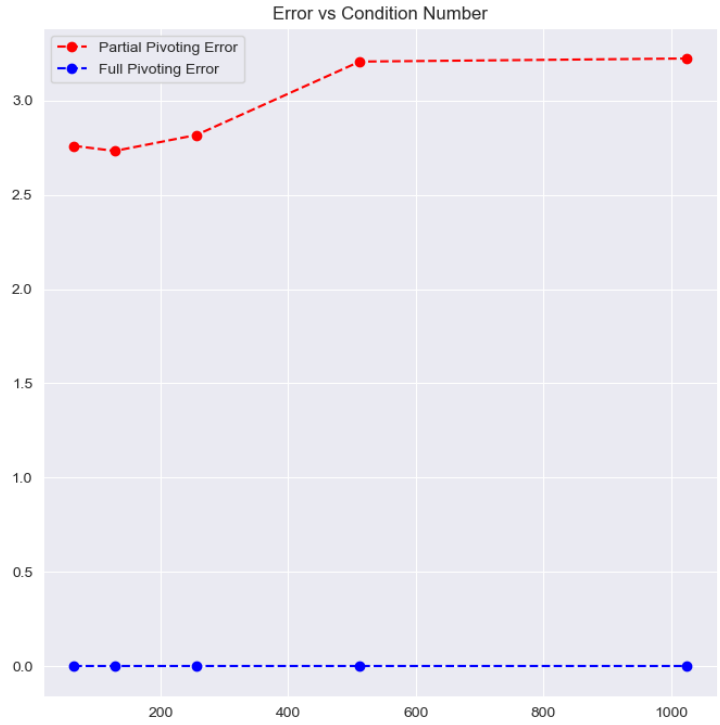
The problem is alleviated with full pivoting, where at each step the largest item chosen does not come from the same column, thus the accumulating values in the last column are no more a problem.

Here are the results:

| size | condition_number | cpu_times_partial | cpu_times_full | error_partial | error_full | res_partial | res_full |
|---|---|---|---|---|---|---|---|
| **64** | 64 | 4.984863 | 5.977539 | 2.760012 | 1.11E-15 | 9.705972 | 4.44E-15 |
| **128** | 128 | 10.97046 | 19.94653 | 2.732935 | 2.55E-15 | 17.04149 | 7.11E-15 |
| **256** | 256 | 26.92773 | 42.88477 | 2.817609 | 2.22E-15 | 13.39123 | 7.99E-15 |
| **512** | 512 | 383.9712 | 487.6943 | 3.207088 | 6.88E-15 | 16.22037 | 1.42E-14 |
| **1024** | 1024 | 3378.076 | 6499.919 | 3.223485 | 1.02E-14 | 53.00689 | 3.55E-14 |

## *Analysis of the infinite norm of the error*

As expected, due to the reasons mentioned above we see a tremendous error progression in the partial pivoting method, while full pivoting keeps a sensible magnitude in terms of error.

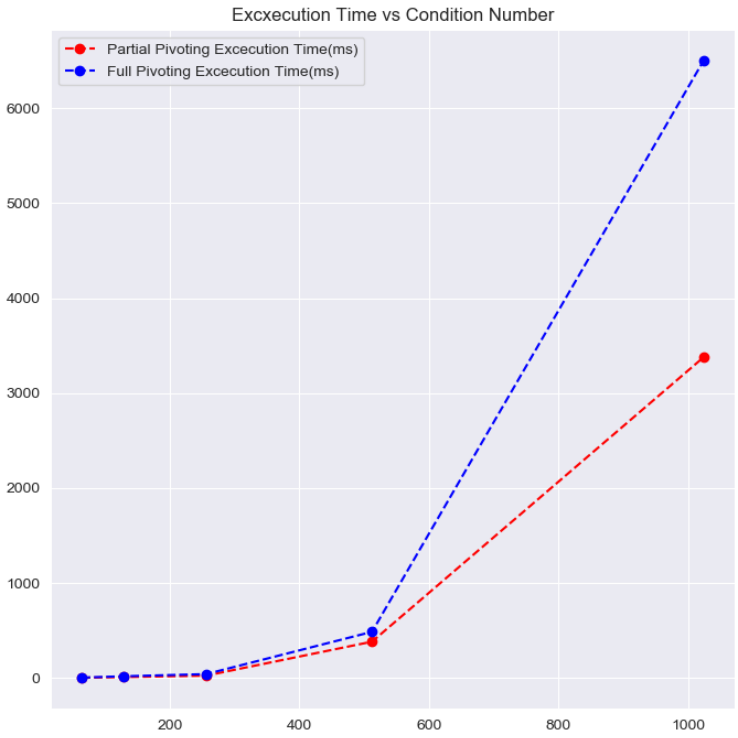


## *Analysis of the infinite norm of the residual*

Exactly the same behavior is expected and seen in the residual in both methods, as shown above.


## *Analysis of the time complexity*

The only thing unchanged is the time complexity of the methods since it is not affected at all by the special form of the matrix itself.



## Question 5: On the solution of the perturbed system.

Regarding the final question, one can see that the problem is a rank-1 update / perturbation of the linear system $Ax = b$.

Since we already assume knowledge of the LU decomposition of matrix A, we can implement the Sherman-Morrison Formula and get a numerically cheap way to calculate the result without having to perform the LU decomposition of the A matrix again.

The formula is:

$$x_{new} = (A - uv^T)^{-1}b = A^{-1}b + A^{-1}u(1 - u^T A^{-1}u)^{-1}v^T A^{-1}b$$

We can solve in this system in 3 steps:

*Step 1:*

We can solve the Az = u system, for z by one forward (using pre-calculated L) and one backward (using pre-calculated U) substitution = O(n^2)

*Step 2:*

We can solve the Ay = b system, for b by one forward (using pre-calculated L) and one backward (using pre-calculated U) substitution = O(n^2)

*Step 3:*

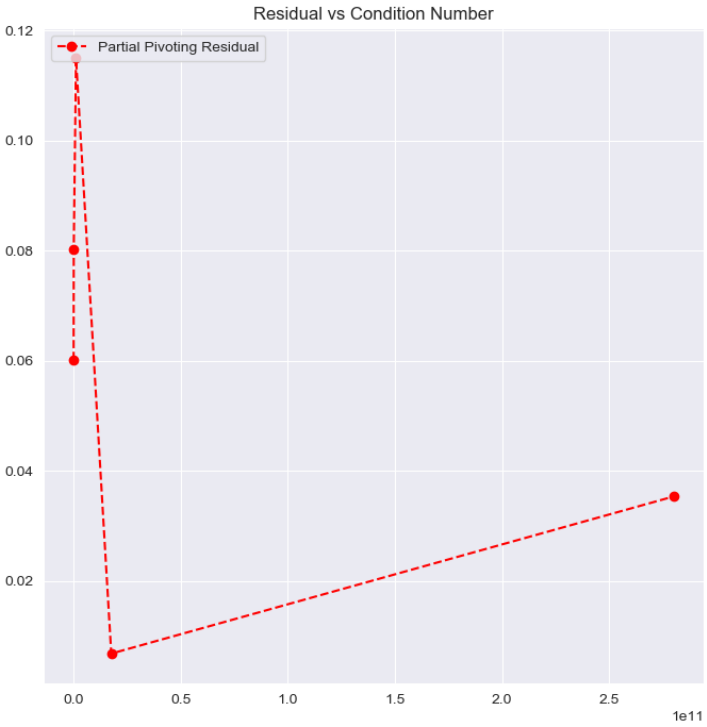
Compute final solution as $x = y + \left(v^T y/(1 - v^T z)\right)z$

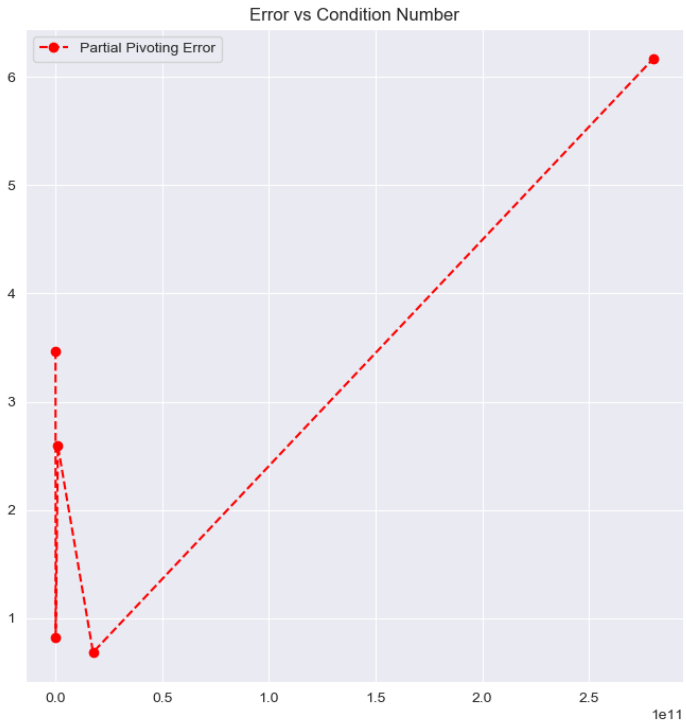
We can see the final solution comprises only from triangular solutions and inner products; thus, the final computation time is $O(n\text{^}2)$.

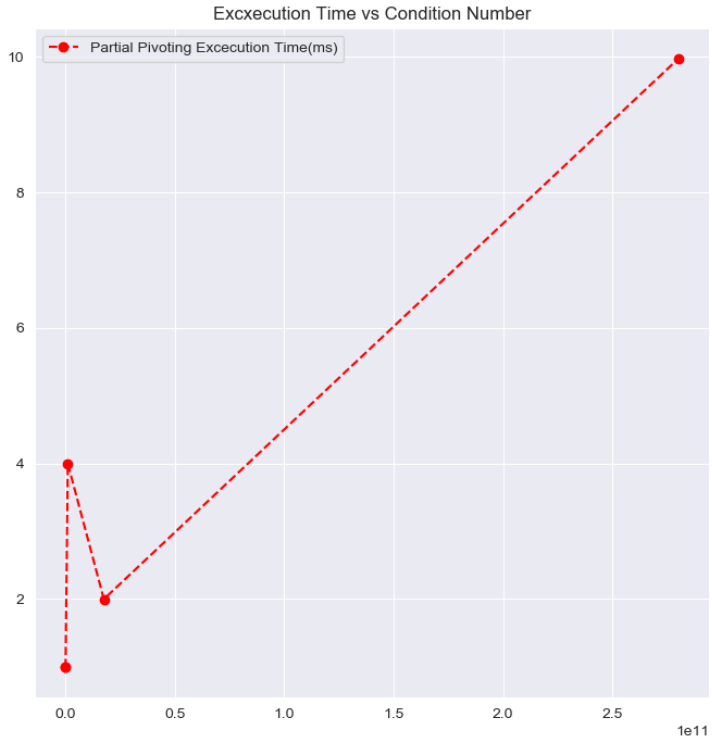
Here are the results:

| size | condition number | error partial | cpu_times_partial | res_partial |
|---|---|---|---|---|
| **64** | 4550790 | 3.469039 | 0.995874 | 0.060174 |
| **128** | 70460127 | 0.81825 | 0.994921 | 0.080247 |
| **256** | 1.11E+09 | 2.592441 | 3.997087 | 0.115021 |
| **512** | 1.76E+10 | 0.684107 | 1.996756 | 0.006861 |
| **1024** | 2.8E+11 | 6.169257 | 9.97448 | 0.035388 |

As we can observe the CPU time scales far less that the solution in the former exercises with the total execution time around the same magnitude at all experiments. The error and the residual seem to also have the same behavior with the Sherman-Morrison being an excellent ad-hoc solution to the problem.


Residual vs Condition Number
Partial Pivoting Residual
0.12
0.10
0.08
0.06
0.04
0.02
0.0
0.5
1.0
1.5
2.0
2.5
1e11


Error vs Condition Number
Partial Pivoting Error
6
5
4
3
2
1
0.0
0.5
1.0
1.5
2.0
2.5
1e11


Excxecution Time vs Condition Number
Partial Pivoting Excecution Time(ms)
10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
1e11