# Обзор методов LU разложения для больших разреженных матриц

Akim & Denis

*Аким Каленюк*
*Денис Борисов*

# Постановка задачи

**В ходе выполнения работы нами было принято решение сосредоточиться на следующих целях:**

- **Изучить разнообразные методы LU разложения для разреженных матриц**
- **Изучить проблему fill-in, при которой разреженная матрица при LU разложении раскладывается на 2 плотные матрицы, из-за чего мы теряем преимущество при решении СЛУ.**
- **Сосредоточить свое внимание на различных методах, решающих проблему fill-in**

**P.S. В первую очередь, мы смотрим на заполнение получившихся треугольных матриц, а за тем на точность и скорость алгоритма**

# Fill-In Problem

# Fill-In Problem

**Основа Проблемы: В процессе преобразования или факторизации разреженной матрицы могут возникать новые ненулевые элементы в позициях, которые изначально были нулевыми. Это увеличивает количество ненулевых элементов в матрице, что приводит к потере разреженности матрицы.**

**Причины и Следствия: При выполнении операций, таких как метод Гаусса, элементы матрицы, которые были нулями, могут превратиться в ненулевые. Это увеличивает объем занимаемой памяти и вычислительные затраты, так как в дальнейших расчетах эти новые ненулевые элементы необходимо учитывать.**

**Пример с LU-разложением: В процессе LU-разложения разреженной матрицы элементы, которые изначально были нулями, могут стать ненулевыми в матрицах L (нижней треугольной) или U (верхней треугольной), что приводит к увеличению памяти, необходимой для хранения этих матриц, а также снижает эффективность решения линейных уравнений (было отмечено выше)**

**Методы борьбы с проблемой: Перед факторизацией матрицы её строки и столбцы могут быть переупорядочены таким образом, чтобы минимизировать количество ненулевых элементов, которые будут созданы в процессе. Например, алгоритм Reverse Cuthill-McKee.**

# Incomplete
# LU Decomposition

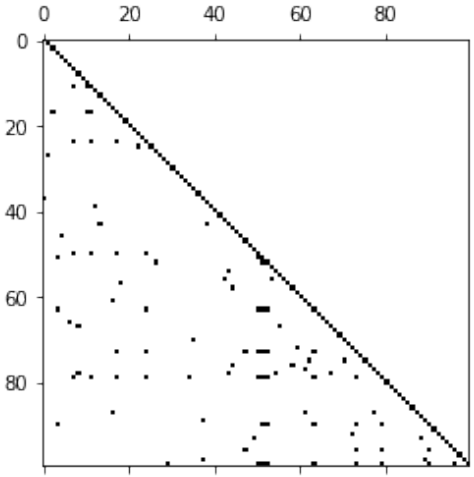
# Incomplete LU Decomposition

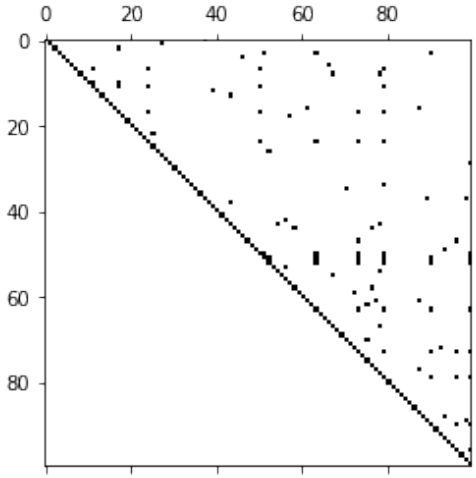
- В отличие от стандартного LU: L U ≃ A без строгого равенства
- Более эффективный по сложности и времени чем стандартный алгоритм LU. Тем не менее, значительно уступает оптимизированным алгоритмам в сложности. Интересно, что в нашем случае ошибка на ILU меньше ошибки нашей кастомной LU.
- Есть ряд реализаций в зависимости от задач и требований к точности: через запоминаний элементов исходной матрицы, через трешолд по абсолютной величине элементов и многое другое.
- Эффективно решает такие задачи как ускорение метода сопряженных градиентов, работая как предобуславливатель
- Эффективно решает проблему fill-in

```
def Incomplete_LU_Decomposition(A):

    n = number_of_rows(A)

    # Initialize L as a zero matrix of size n x n
    L = Zero_Matrix(n, n)

    # Initialize U as a zero matrix of size n x n
    U = Zero_Matrix(n, n)

    for i from 0 to n-1:
        for j from 0 to n-1:
            if i <= j:
                # Calculate U[i, j]
                sum = 0
                for k from 0 to i-1:
                    sum += L[i, k] * U[k, j]
                U[i, j] = A[i, j] - sum

            if i >= j:
                # Calculate L[i, j]
                if i == j:
                    L[i, j] = 1
                else:
                    sum = 0
                    for k from 0 to j-1:
                        sum += L[i, k] * U[k, j]
                    L[i, j] = (A[i, j] - sum) / U[j, j]

    return L, U
```

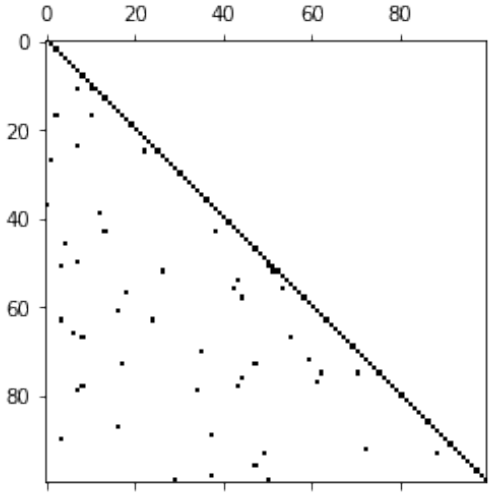
# Incomplete LU

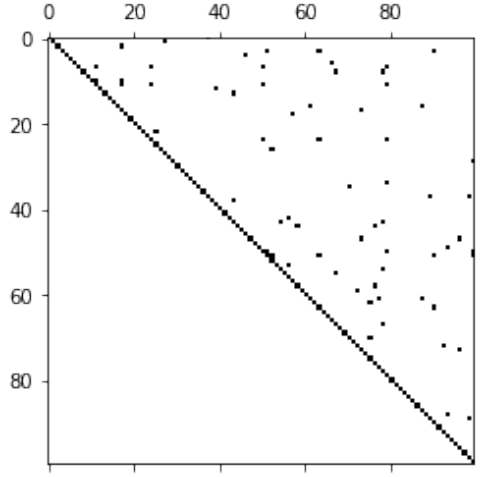
- LU



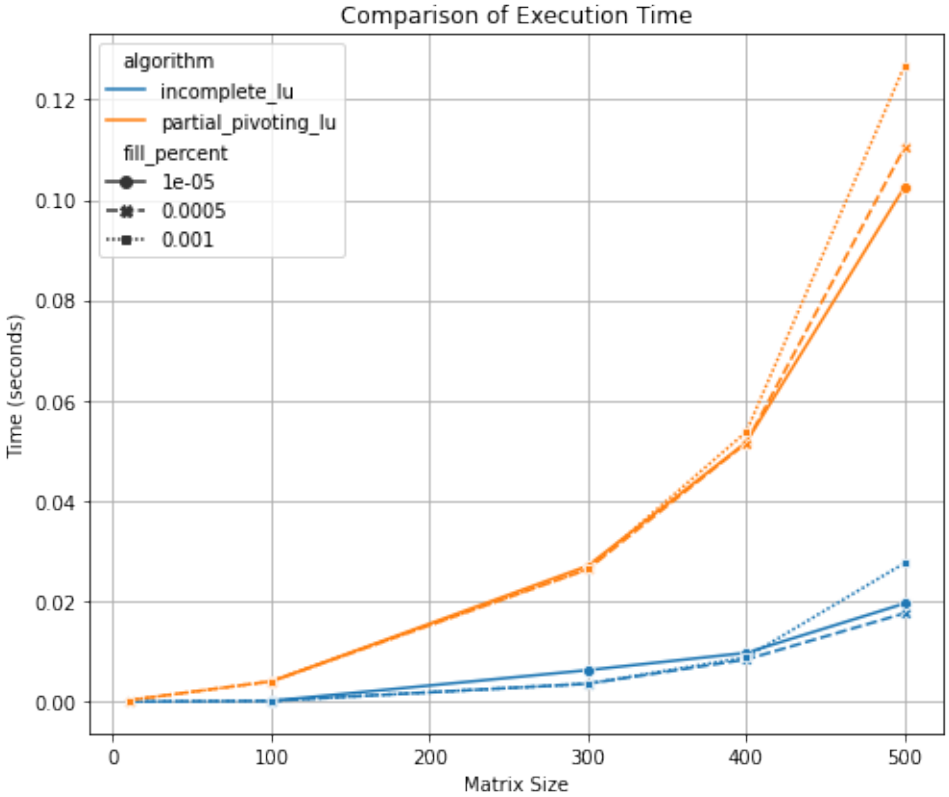



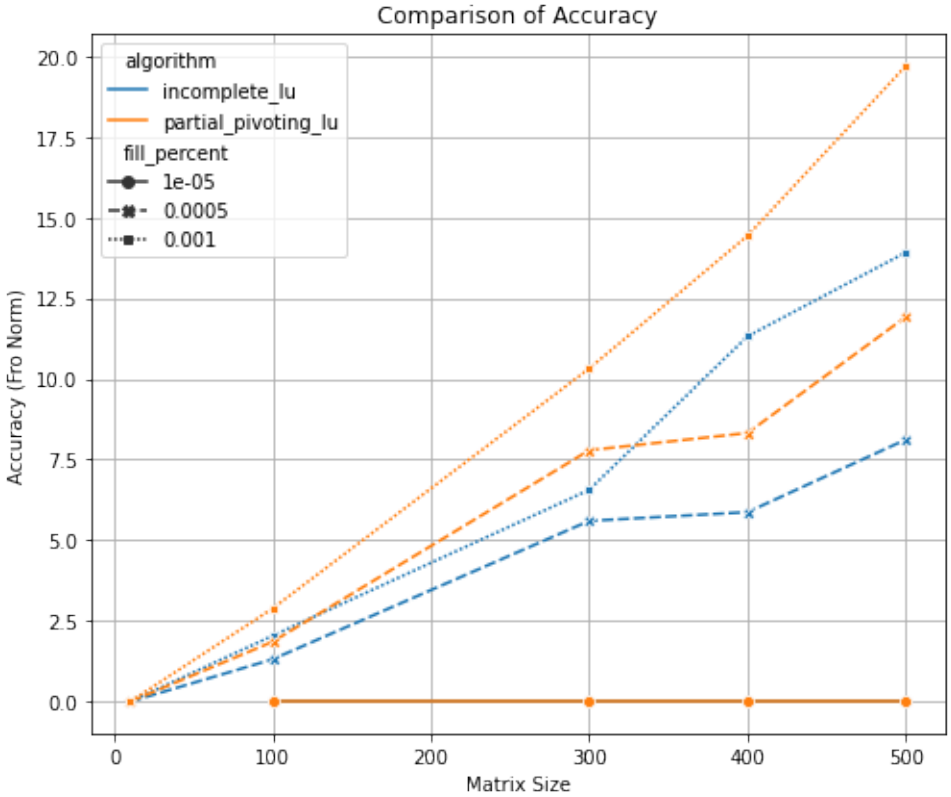

- ILU



Заметим, что ILU имеет заметный выигрыш в скорости в сравнении с нашим кастомным PLU, а также меньшие темпы роста ошибки



# Nested Dissection

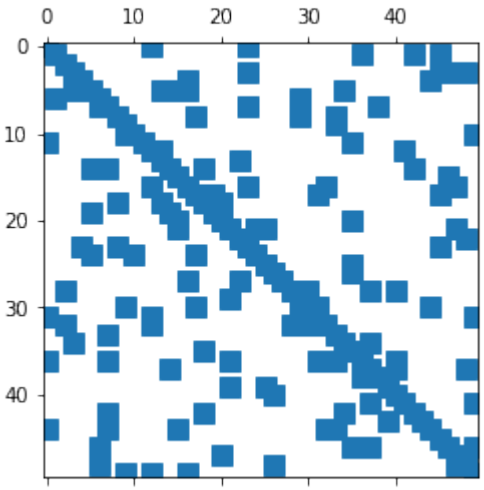
# Nested Dissection

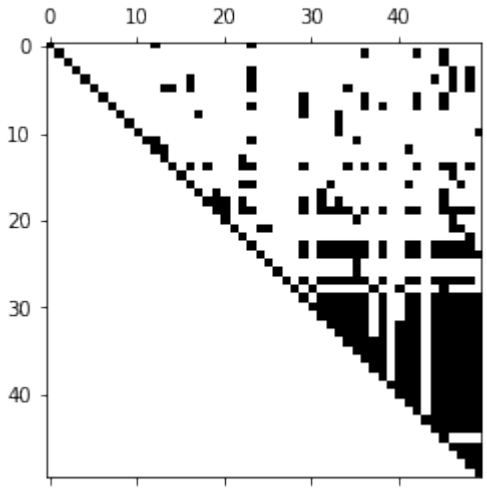
- **Основная идея метода**: LU-разложение с Nested Dissection (вложенной секционированием) - это метод разложения разреженных матриц. Основная идея состоит в разбиении графа матрицы на более мелкие части (секционирование), что позволяет эффективно выполнять LU-разложение на этих подграфах, уменьшая сложность и объем вычислений.

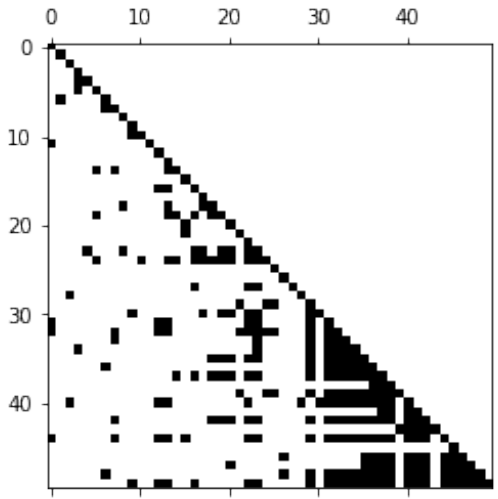
- **Ключевые особенности:** Основными особенностями являются улучшенная масштабируемость и эффективность для больших матриц. Метод уменьшает количество необходимых заполнений (fill-ins) в процессе разложения, что важно для уменьшения использования памяти и ускорения вычислений. Это делает его особенно полезным для высокопроизводительных вычислений и приложений, требующих обработки больших объемов данных.

```
def LU_Decomposition_With_Nested_Dissection(A):
    #A is a sparse matrix

    #Step 1: Decompose the graph of A using Nested Dissection
    Graph = Convert_Matrix_To_Graph(A)
    Partitioned_Graph = Nested_Dissection(Graph)

    #Step 2: Apply LU Decomposition to each partition

    for each Subgraph in Partitioned_Graph:
    #Extract submatrix corresponding to the current subgraph
        Submatrix = Extract_Submatrix(A, Subgraph)

    #Perform standard LU Decomposition on the submatrix
        L, U = Standard_LU_Decomposition(Submatrix)

    #Update the original matrix A with L and U values
        Update_Matrix(A, L, U, Subgraph)

    return A

def Nested_Dissection(Graph):
    #Recursively divide the graph into smaller parts

    if Graph is small enough:
        return [Graph]

    #Divide the graph into two parts with a separator
    Part1, Separator, Part2 = Divide_Graph(Graph)

    #Recursively apply Nested Dissection to the parts
    Subgraphs1 = Nested_Dissection(Part1)
    Subgraphs2 = Nested_Dissection(Part2)

    return Subgraphs1 + [Separator] + Subgraphs2
```

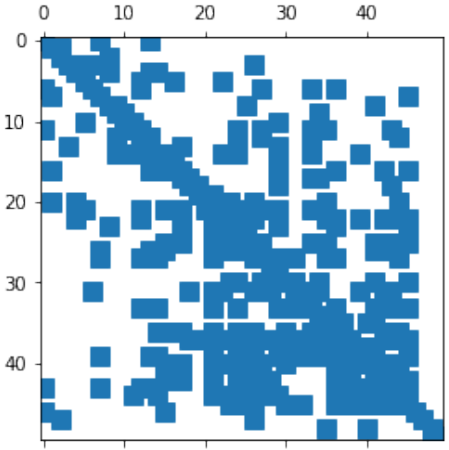
LU

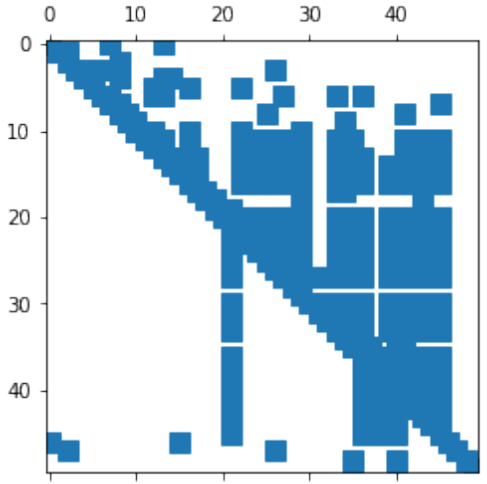

0
10
20
30
40

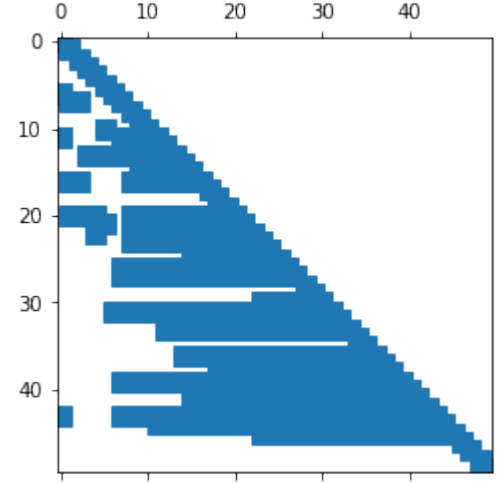

0
10
20
30
40


0
10
20
30
40

ND LU


0
10
20
30
40


0
10
20
30
40


0
10
20
30
40

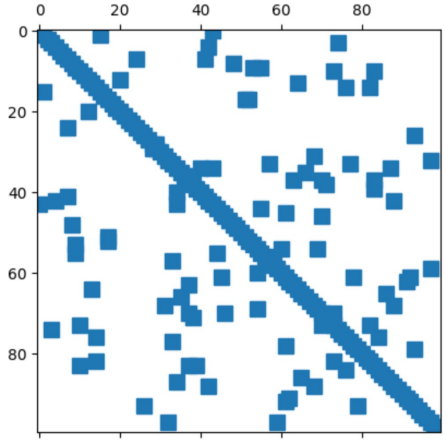
# Reverse Cuthill-McKee

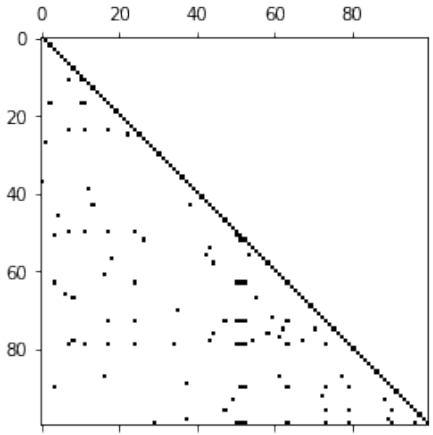
# Reverse Cuthill-McKee

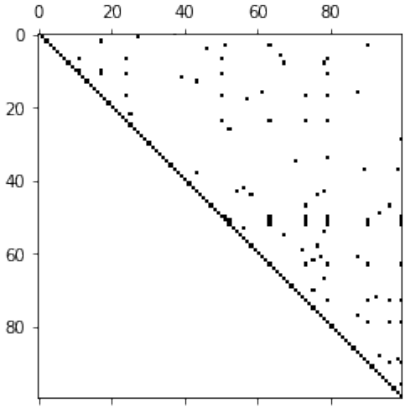
**Алгоритм Reverse Cuthill-McKee (RCM) используется для переупорядочивания разреженных матриц перед LU-разложением. Основная цель - минимизировать ширину полосы матрицы, что приводит к уменьшению числа ненулевых элементов вне диагонали. Это достигается путём переупорядочивания строк и столбцов матрицы таким образом, чтобы элементы с ненулевыми значениями располагались ближе к диагонали.**

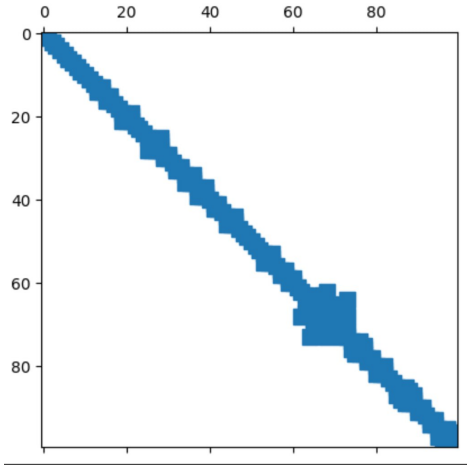
```
def LU_Decomposition_With_RCM(A):
    # A is a sparse matrix

    # Step 1: Apply RCM algorithm to reorder the matrix

    RCM_Order = Reverse_Cuthill_McKee(A)
    Reordered_A = Reorder_Matrix(A, RCM_Order)

    # Step 2: Apply standard LU to the reordered matrix

    L, U = Standard_LU_Decomposition(Reordered_A)

    return L, U

def Reverse_Cuthill_McKee(Matrix):
    # Convert the matrix to a graph representation
    Graph = Matrix_To_Graph(Matrix)

    # Compute the RCM ordering
    RCM_Order = Compute_RCM_Ordering(Graph)

    return RCM_Order
```

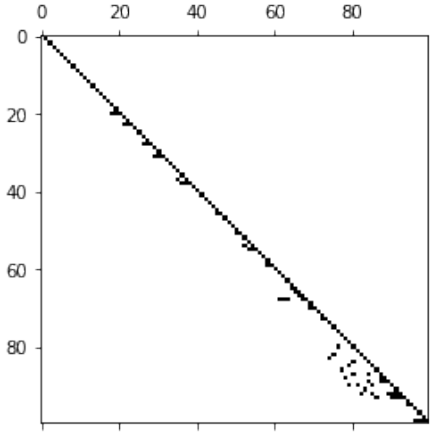
# Reverse Cuthill-McKee

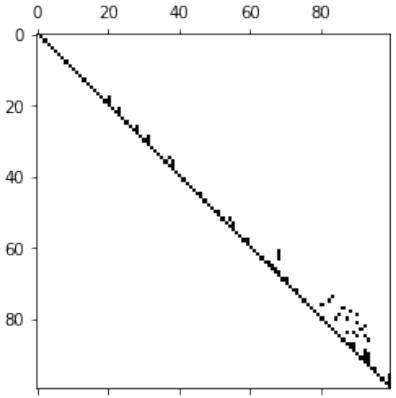


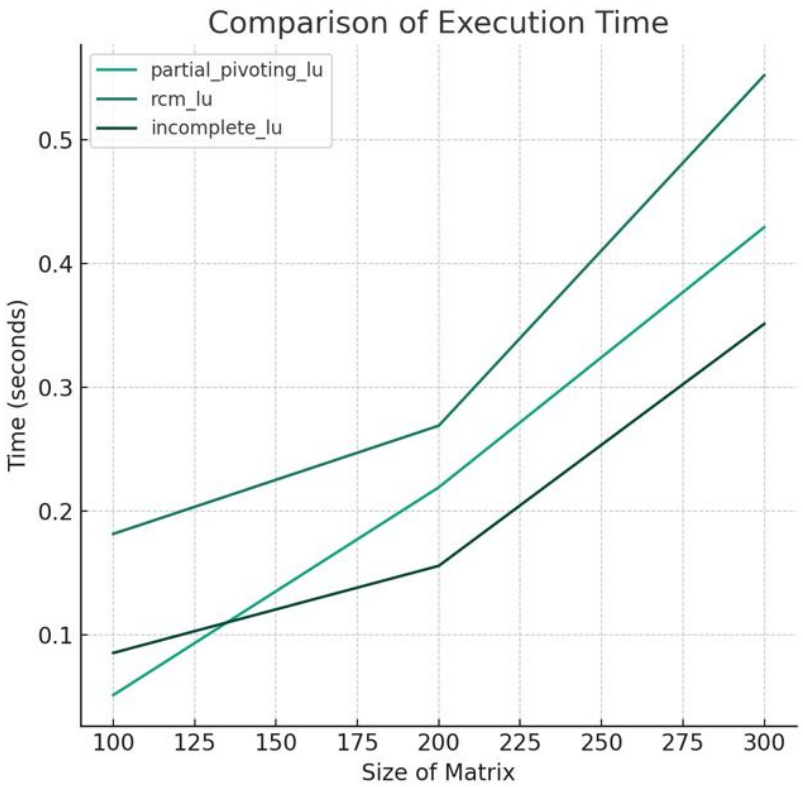



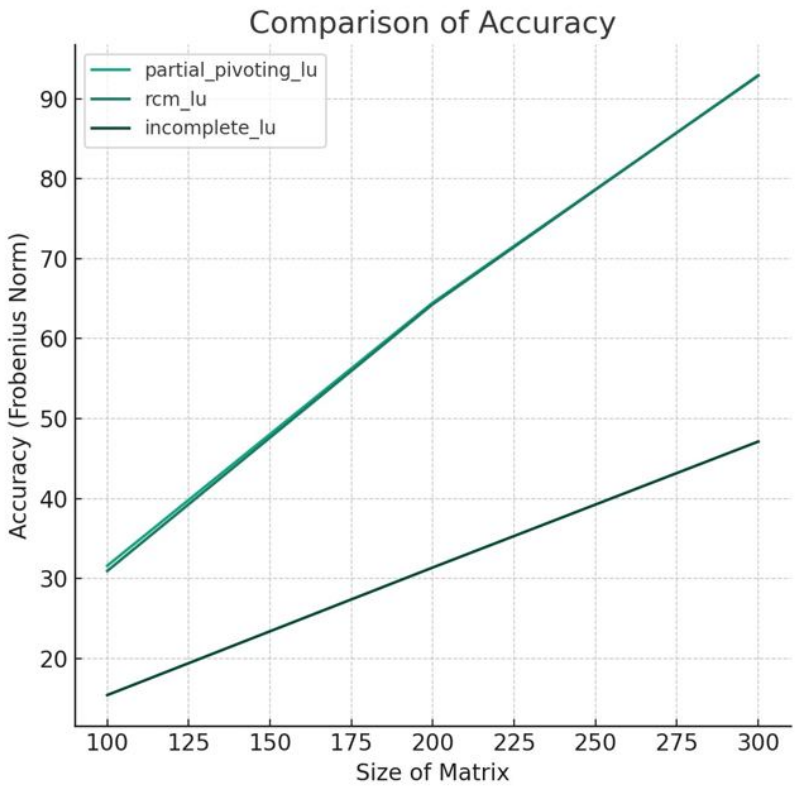



# Summary!!!


Comparison of Execution Time
partial_pivoting_lu
rcm_lu
incomplete_lu
Time (seconds)
0.1
0.2
0.3
0.4
0.5
100
125
150
175
200
225
250
275
300
Size of Matrix


Comparison of Accuracy
partial_pivoting_lu
rcm_lu
incomplete_lu
Accuracy (Frobenius Norm)
20
30
40
50
60
70
80
90
100
125
150
175
200
225
250
275
300
Size of Matrix

# Conclusions

- В рамках проекта мы исследовали различные методы LU разложений для разреженных матриц. Реализовали их с нуля и показали графически проблему fill-in и ее решение.
- Планировалось реализовать 4 алгоритма и измерить их качество и время работы
- Мы полноценно реализовали 3 алгоритма, показали проблему fill-in, однако столкнулись с рядом вычислительных сложностей в случае алгоритма Nested Dissection на разреженных матрицах

# Github

https://github.com/Akim-collab/LU-decomposition-for-large-sparse-matrices