Numerical and Symbolic Algorithms Modeling (MODEL, MU4IN901)

Sorbonne University

Implementation Project

By

Hani Abdallah - (ID: 21400302)

Houssam Eddine Jamil Nasser - (ID: 21400407)

Tan Viet Nguyen - (ID: 21400381)

Contents

1 Implementation

1.1 Matrix Multiplication

Within this section, we will be comparing two different approaches for implementing matrix multiplication.

In particular, we will be considering the naive multiplication algorithm, as well as the Strassen multiplication algorithm. We will be showcasing the algorithms implementation alongside their main functions and code organization.

1.1.1 Naive Matrix Multiplication Algorithm

The matrix multiplication naive algorithm is the straightforward of them, where it has a time complexity of $O(n^3)$ for two n x n matrices.

Input: *Matrix A and Matrix B (of size n).*

Output: *Matrix R (the resulted matrix of the matrices multiplication).*

Algorithm:

```c
void Naive_matrix_multiplication(double **A, double **B, double **R,
int rows_A, int cols_A, int cols_B)
{
    for (int i = 0; i < rows_A; i++)
    {
        for (int j = 0; j < cols_B; j++)
        {
            R[i][j] = 0.0;
            for (int k = 0; k < cols_A; k++)
            {
                R[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

1.1.2 Strassen Matrix Multiplication Algorithm

Strassen Algorithm reduces the time needed to do matrix multiplication compared to the standard method. It does this by dividing the matrices into smaller sub-matrices and using mathematical formulas to achieve a faster asymptotic time complexity of $O(n^{log_2 7})$ for two n x n matrices.

Input: *Matrix A and Matrix B (of size n).*

Output: *Matrix R (the resulted matrix of the matrices multiplication).*

Algorithm:

```
void strassen_mult(double **M, double **N, double **R, int size)
{
    if (size <= 64)
    {
        nbasecase(M, N, R, size);
        return;
    }
    int newSize = size / 2;

    // submatrices allocation
    double **a = allocate_matrix(newSize);
    double **b = allocate_matrix(newSize);
    double **c = allocate_matrix(newSize);
    double **d = allocate_matrix(newSize);
    double **x = allocate_matrix(newSize);
    double **y = allocate_matrix(newSize);
    double **z = allocate_matrix(newSize);
    double **t = allocate_matrix(newSize);
    double **q1 = allocate_matrix(newSize);
    double **q2 = allocate_matrix(newSize);
    double **q3 = allocate_matrix(newSize);
    double **q4 = allocate_matrix(newSize);
    double **q5 = allocate_matrix(newSize);
    double **q6 = allocate_matrix(newSize);
    double **q7 = allocate_matrix(newSize);
    double **r11 = allocate_matrix(newSize);
    double **r12 = allocate_matrix(newSize);
    double **r21 = allocate_matrix(newSize);
    double **r22 = allocate_matrix(newSize);
    double **temp1 = allocate_matrix(newSize);
    double **temp2 = allocate_matrix(newSize);

    // M and N submatrices (Blocks)
    for (int i = 0; i < newSize; i++)
    {
        for (int j = 0; j < newSize; j++)
        {
            a[i][j] = M[i][j];                       // M11
            b[i][j] = M[i][j + newSize];             // M12
            c[i][j] = M[i + newSize][j];             // M21
            d[i][j] = M[i + newSize][j + newSize];   // M22
            x[i][j] = N[i][j];                       // N11
            y[i][j] = N[i][j + newSize];             // N12
            z[i][j] = N[i + newSize][j];             // N21
            t[i][j] = N[i + newSize][j + newSize];   // N22
        }
    }
    // q1 = a * (x + z)
    add_matrix(x, z, temp2, newSize);
    strassen_mult(a, temp2, q1, newSize); // recursive call
    // q2 = d * (y + t)
    add_matrix(y, t, temp2, newSize);
    strassen_mult(d, temp2, q2, newSize); // recursive call
    // q3 = (d - a) * (z - y)
    subtract_matrix(d, a, temp1, newSize);
    subtract_matrix(z, y, temp2, newSize);
    strassen_mult(temp1, temp2, q3, newSize); // recursive call
    // q4 = (b - d) * (z + t)
    subtract_matrix(b, d, temp1, newSize);
    add_matrix(z, t, temp2, newSize);
    strassen_mult(temp1, temp2, q4, newSize); // recursive call
    // q5 = (b - a) * z
    subtract_matrix(b, a, temp1, newSize);
    strassen_mult(temp1, z, q5, newSize); // recursive call
    // q6 = (c - a) * (x + y)
    subtract_matrix(c, a, temp1, newSize);
    add_matrix(x, y, temp2, newSize);
    strassen_mult(temp1, temp2, q6, newSize); // recursive call
    // q7 = (c - d) * y
    subtract_matrix(c, d, temp1, newSize);
    strassen_mult(temp1, y, q7, newSize); // recursive call
    // r11 = q1 + q5
    add_matrix(q1, q5, r11, newSize);
    // r12 = q2 + q3 + q4 - q5
    add_matrix(q2, q3, temp1, newSize);
    add_matrix(temp1, q4, temp2, newSize);
    subtract_matrix(temp2, q5, r12, newSize);
    // r21 = q1 + q3 + q6 - q7
    add_matrix(q1, q3, temp1, newSize);
    add_matrix(temp1, q6, temp2, newSize);
    subtract_matrix(temp2, q7, r21, newSize);
    // r22 = q2 + q7
    add_matrix(q2, q7, r22, newSize);
    // result matrix R
    for (int i = 0; i < newSize; i++)
    {
        for (int j = 0; j < newSize; j++)
        {
            R[i][j] = r11[i][j];
            R[i][j + newSize] = r12[i][j];
            R[i + newSize][j] = r21[i][j];
            R[i + newSize][j + newSize] = r22[i][j];
        }
    }
}
```

1.2 LU Decomposition (using Gaussian elimination)

LU decomposition means Lower-Upper decomposition, a widely-used method in numerical linear algebra that expresses a matrix as a product of lower and upper triangular matrices, L and U, respectively.

The computational complexity of LU decomposition using Gaussian elimination is typically $O(n^3)$ for n x n matrix.

Input: *Matrix A (of size n).*

Output: *Resulted decomposition of Matrix L (Lower triangular) and Matrix U (Upper Triangular).*

Algorithm:

```
void LU_Decomposition(double **A, int n, double **L, double **U)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            double factor = U[j][i] / U[i][i];

            L[j][i] = factor;

            for (int k = i; k < n; k++)
            {
                U[j][k] -= factor * U[i][k];
            }
        }
    }
}
```

1.3 Matrix Inversion

In this section, we will showcasing the different approaches to implement matrix inversion algorithms alongside their main functions and code organization.

In particular, we will be considering the LU inverse based on the LU decomposition algorithm, the Strassen inversion using naive multiplication algorithm, and the Strassen inversion using the Strassen multiplication algorithm.

1.3.1 LU inversion based on LU Decomposition

The computational complexity of LU inverse based on the LU decomposition algorithm is typically $O(n^3)$ for n x n matrix.

Input: *Matrix A of size n.*

Output: *The resulted inverted Matrix A^{-1}.*

Algorithm:

```
void invertMatrix(double **A, double **A_inverse, int n) {
    double **L = allocateMatrix(n);
    double **U = allocateMatrix(n);
    LU_Decomposition(A, n, L, U);

    double *b = malloc(n * sizeof(double));
    double *y = malloc(n * sizeof(double));
    double *x = malloc(n * sizeof(double));

    for (int i = 0; i < n; i++) {
        // Set up identity matrix column
        for (int j = 0; j < n; j++) {
            b[j] = (i == j) ? 1.0 : 0.0;
        }
        // Solve Ly = b
        forwardSubstitution(L, b, y, n);
        // Solve Ux = y
        backwardSubstitution(U, y, x, n);

        // Store solution in the inverse matrix
        for (int j = 0; j < n; j++) {
            A_inverse[j][i] = x[j];
        }}
    freeMatrix(L, n);
    freeMatrix(U, n);
    free(b);
    free(y);
    free(x);
}
```

1.3.2 Strassen inversion using Strassen multiplication

The computational complexity of Strassen inversion based on the Strassen multiplication algorithm is typically $O(n^{log_2 7})$ for n x n matrix.

Input: *Matrix A of size n.*

Output: *The resulted inverted Matrix A^{-1}.*

Algorithm:

```
void strassen_inversion(double **A, double **A_inv, int size)
{
    if (size == 1)
    {
        invert_1x1(A, A_inv);
        return;
    }

    int newSize = size / 2;

    double **a = allocate_matrix(newSize);
    double **b = allocate_matrix(newSize);
    double **c = allocate_matrix(newSize);
    double **d = allocate_matrix(newSize);
    double **e = allocate_matrix(newSize);
    double **z = allocate_matrix(newSize);
    double **t = allocate_matrix(newSize);
    double **x = allocate_matrix(newSize);
    double **y = allocate_matrix(newSize);

    double **temp1 = allocate_matrix(newSize);
    double **temp2 = allocate_matrix(newSize);

    // Split A into submatrices
    for (int i = 0; i < newSize; i++)
    {
        for (int j = 0; j < newSize; j++)
        {
            a[i][j] = A[i][j];                       // A11
            b[i][j] = A[i][j + newSize];             // A12
            c[i][j] = A[i + newSize][j];             // A21
            d[i][j] = A[i + newSize][j + newSize]; // A22
        }
    }

    // e = a^-1
    strassen_inversion(a, e, newSize); // recursive call
    // z = d - c * e * b
    strassen_mult(e, b, temp1, newSize);
    strassen_mult(c, temp1, temp2, newSize);
    subtract_matrix(d, temp2, z, newSize);

    // t = z^-1
    strassen_inversion(z, t, newSize); // recursive call

    // y = -e * b * t
    strassen_mult(b, t, temp1, newSize);
    strassen_mult(e, temp1, y, newSize);
    for (int i = 0; i < newSize; i++)
    { // y = -(e * b * t)
        for (int j = 0; j < newSize; j++)
        {
            y[i][j] = -y[i][j];
        }
    }

    // z = -t * c * e
    strassen_mult(c, e, temp1, newSize);
    strassen_mult(t, temp1, z, newSize);
    for (int i = 0; i < newSize; i++)
    { // z = -(t * c * e)
        for (int j = 0; j < newSize; j++)
        {
            z[i][j] = -z[i][j];
        }
    }

    // x = e + e * b * t * c * e
    strassen_mult(b, t, temp1, newSize);      // temp1 = b * t
    strassen_mult(temp1, c, temp2, newSize); // temp2 = (b * t) * c
    strassen_mult(e, temp2, temp1, newSize); // temp1 = e * ((b * t) * c)
    strassen_mult(temp1, e, temp2, newSize); // temp2 = (e * ((b * t) * c)) * e
    add_matrix(e, temp2, x, newSize);         // x = e + (e * (b * t * c))

    // Combine x, y, z, t, into A_inv
    for (int i = 0; i < newSize; i++)
    {
        for (int j = 0; j < newSize; j++)
        {
            A_inv[i][j] = x[i][j];                          // A_inv11
            A_inv[i][j + newSize] = y[i][j];                // A_inv12
            A_inv[i + newSize][j] = z[i][j];                // A_inv21
            A_inv[i + newSize][j + newSize] = t[i][j]; // A_inv22
        }
    }
}
```

1.3.3 Strassen inversion using naive multiplication

The computational complexity of Strassen inversion based on the Strassen multiplication algorithm is typically $O(n^3)$ for n x n matrix.

Note: For this section, Structs have been employed as an alternative to the conventional method of 2D matrix allocation. This choice was made to present a unique perspective on how these algorithms can be implemented.

Input: *Matrix A of size n.*

Output: *The resulted inverted Matrix A^{-1}.*

Algorithm:

```
void inverse(matrix_t *A, matrix_t *inv_A)
{
if (A->columns == 1 && A->rows == 1)
  {
  if (A->matrix[0][0] == 0){
      printf("Cannot devide by 0!!\n");}
    else{
      inv_A->matrix[0][0] = 1 / A->matrix[0][0];
    }}
  else if (A->rows == 2 && A->columns == 2)
  {
    matrix_t A_11 = {};
    create_matrix(1, 1, &A_11);
    A_11.matrix[0][0] = A->matrix[0][0];
    matrix_t A_12 = {};
    create_matrix(1, 1, &A_12);
    A_12.matrix[0][0] = A->matrix[0][1];
    matrix_t A_21 = {};
    create_matrix(1, 1, &A_21);
    A_21.matrix[0][0] = A->matrix[1][0];
    matrix_t A_22 = {};
    create_matrix(1, 1, &A_22);
    A_22.matrix[0][0] = A->matrix[1][1];
    matrix_t A_11_inv = {};
    create_matrix(1, 1, &A_11_inv);
    inverse(&A_11, &A_11_inv);
    matrix_t A_21_A_11_inv = {};
    create_matrix(1, 1, &A_21_A_11_inv);
    mult_matrix(&A_21, &A_11_inv, &A_21_A_11_inv);
    matrix_t A_21_A_11_inv_A_12 = {};
    create_matrix(1, 1, &A_21_A_11_inv_A_12);
    mult_matrix(&A_21_A_11_inv, &A_12, &A_21_A_11_inv_A_12);
    matrix_t S_22 = {};
    create_matrix(1, 1, &S_22);
    sub_matrix(&A_22, &A_21_A_11_inv_A_12, &S_22);
    matrix_t S_22_inv = {};
    create_matrix(1, 1, &S_22_inv);
    inverse(&S_22, &S_22_inv);
    matrix_t A_12_S_22_inv = {};
    create_matrix(1, 1, &A_12_S_22_inv);
    mult_matrix(&A_12, &S_22_inv, &A_12_S_22_inv);
    matrix_t A_12_S_22_inv_A_21 = {};
    create_matrix(1, 1, &A_12_S_22_inv_A_21);
    mult_matrix(&A_12_S_22_inv, &A_21, &A_12_S_22_inv_A_21);
    matrix_t A_12_S_22_inv_A_21_A_11_inv = {};
    create_matrix(1, 1, &A_12_S_22_inv_A_21_A_11_inv);
    mult_matrix(&A_12_S_22_inv_A_21, &A_11_inv, &A_12_S_22_inv_A_21_A_11_inv);
    A_12_S_22_inv_A_21_A_11_inv.matrix[0][0] += 1;
    matrix_t B_11 = {};
    create_matrix(1, 1, &B_11);
    mult_matrix(&A_11_inv, &A_12_S_22_inv_A_21_A_11_inv, &B_11);
    matrix_t A_11_inv_A_12 = {};
    create_matrix(1, 1, &A_11_inv_A_12);
    mult_matrix(&A_11_inv, &A_12, &A_11_inv_A_12);
    matrix_t B_12 = {};
    create_matrix(1, 1, &B_12);
    mult_matrix(&A_11_inv_A_12, &S_22_inv, &B_12);
    B_12.matrix[0][0] *= -1;
    matrix_t S_22_inv_A_21 = {};
    create_matrix(1, 1, &S_22_inv_A_21);
    mult_matrix(&S_22_inv, &A_21, &S_22_inv_A_21);
    matrix_t B_21 = {};
    create_matrix(1, 1, &B_21);
    mult_matrix(&S_22_inv_A_21, &A_11_inv, &B_21);
    B_21.matrix[0][0] *= -1;
    matrix_t B_22 = {};
    create_matrix(1, 1, &B_22);
    B_22.matrix[0][0] = S_22_inv.matrix[0][0];
    merge_matrices(&B_11, &B_12, &B_21, &B_22, inv_A);
  }
  else
  {
    int split_point = A->rows / 2;
    matrix_t A_11 = {};
    create_matrix(split_point, split_point, &A_11);
    matrix_t A_12 = {};
    create_matrix(split_point, split_point, &A_12);
    matrix_t A_21 = {};
    create_matrix(split_point, split_point, &A_21);
    matrix_t A_22 = {};
    create_matrix(split_point, split_point, &A_22);
    split_matrix_into_quadrants(A, &A_11, &A_12, &A_21, &A_22);
    matrix_t id = {};
```

```
create_matrix(A_11.rows, A_11.rows, &id);
get_identity_matrix(A_11.rows, &id);
matrix_t A_11_inv = {};
create_matrix(split_point, split_point, &A_11_inv);
inverse(&A_11, &A_11_inv);
matrix_t A_21_A_11_inv = {};
create_matrix(split_point, split_point, &A_21_A_11_inv);
mult_matrix(&A_21, &A_11_inv, &A_21_A_11_inv);
matrix_t A_21_A_11_inv_A_12 = {};
create_matrix(split_point, split_point, &A_21_A_11_inv_A_12);
mult_matrix(&A_21_A_11_inv, &A_12, &A_21_A_11_inv_A_12);
matrix_t S_22 = {};
create_matrix(split_point, split_point, &S_22);
sub_matrix(&A_22, &A_21_A_11_inv_A_12, &S_22);
matrix_t S_22_inv = {};
create_matrix(split_point, split_point, &S_22_inv);
inverse(&S_22, &S_22_inv);
matrix_t A_12_S_22_inv = {};
create_matrix(split_point, split_point, &A_12_S_22_inv);
mult_matrix(&A_12, &S_22_inv, &A_12_S_22_inv);
matrix_t A_12_S_22_inv_A_21 = {};
create_matrix(split_point, split_point, &A_12_S_22_inv_A_21);
mult_matrix(&A_12_S_22_inv, &A_21, &A_12_S_22_inv_A_21);
matrix_t A_12_S_22_inv_A_21_A_11_inv = {};
create_matrix(split_point, split_point, &A_12_S_22_inv_A_21_A_11_inv);
mult_matrix(&A_12_S_22_inv_A_21, &A_11_inv, &A_12_S_22_inv_A_21_A_11_inv);
matrix_t A_12_S_22_inv_A_21_A_11_inv_id = {};
create_matrix(split_point, split_point, &A_12_S_22_inv_A_21_A_11_inv_id);
sum_matrix(&A_12_S_22_inv_A_21_A_11_inv, &id, &A_12_S_22_inv_A_21_A_11_inv_id);
matrix_t B_11 = {};
create_matrix(split_point, split_point, &B_11);
mult_matrix(&A_11_inv, &A_12_S_22_inv_A_21_A_11_inv_id, &B_11);
matrix_t A_11_inv_A_12 = {};
create_matrix(split_point, split_point, &A_11_inv_A_12);
mult_matrix(&A_11_inv, &A_12, &A_11_inv_A_12);
matrix_t B_12d = {};
create_matrix(split_point, split_point, &B_12d);
mult_matrix(&A_11_inv_A_12, &S_22_inv, &B_12d);
matrix_t B_12 = {};
create_matrix(split_point, split_point, &B_12);
mult_number(&B_12d, -1, &B_12);
matrix_t S_22_inv_A_21 = {};
create_matrix(split_point, split_point, &S_22_inv_A_21);
mult_matrix(&S_22_inv, &A_21, &S_22_inv_A_21);
matrix_t B_21d = {};
create_matrix(split_point, split_point, &B_21d);
mult_matrix(&S_22_inv_A_21, &A_11_inv, &B_21d);
matrix_t B_21 = {};
create_matrix(split_point, split_point, &B_21);
```

```
    mult_number(&B_21d, -1, &B_21);
    matrix_t B_22 = {};
    create_matrix(split_point, split_point, &B_22);
    for (int i = 0; i < split_point; i++)
    {for (int j = 0; j < split_point; j++){
        B_22.matrix[i][j] = S_22_inv.matrix[i][j];
      }}
    merge_matrices(&B_11, &B_12, &B_21, &B_22, inv_A);
  }}
```

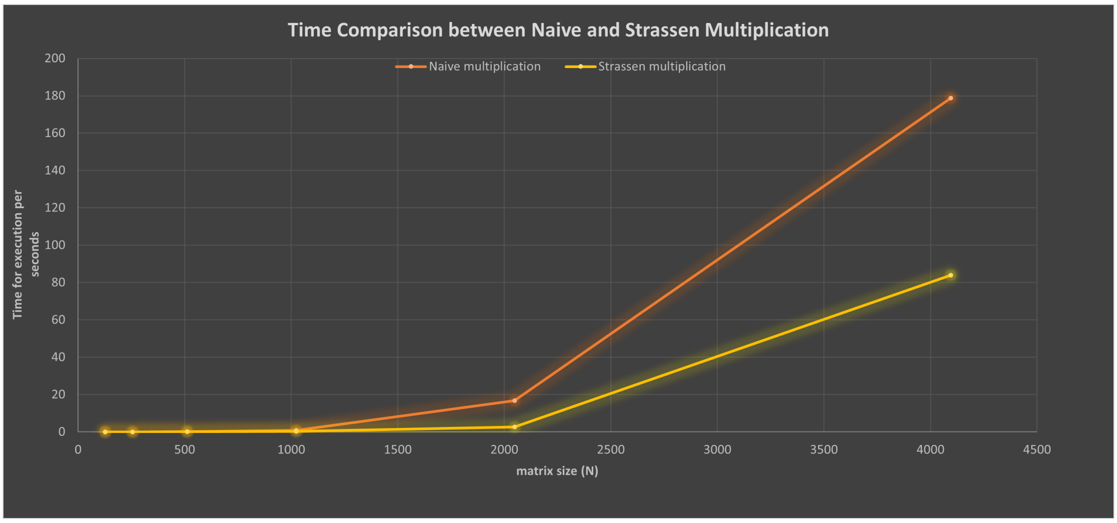
2 Performance Evaluation

In this section, we will be comparing the performance for each subsection, with Matrix Multiplication, Matrix Inversion and LU decomposition (using Gaussian elimination).

Note: The test was carried out on the following hardware :
- CPU: i7-13700H 2.40 GHz
- RAM: 48 GB
- Storage (SSD): 512 GB.

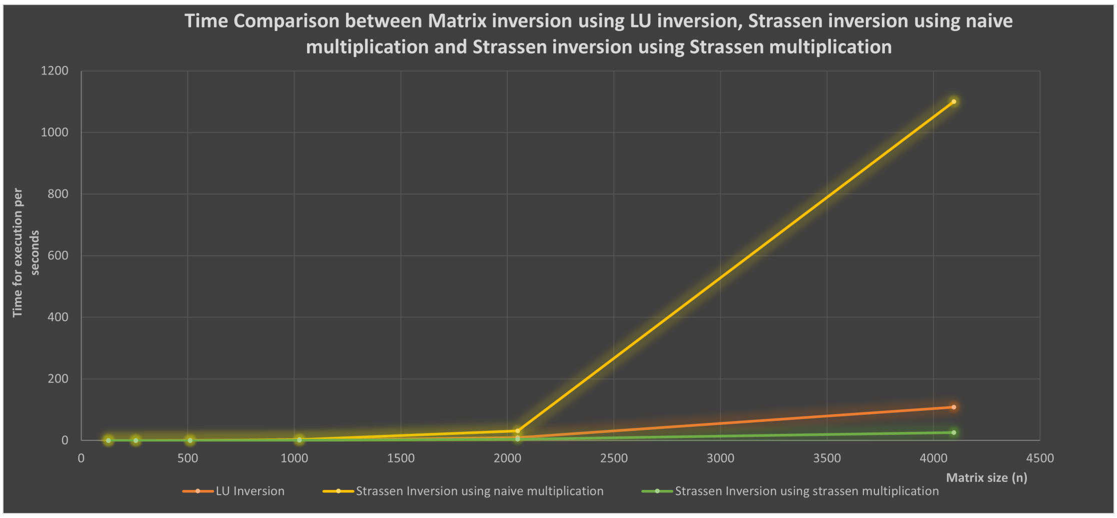
2.1 Matrix Multiplication Benchmarking

Matrix Multiplication		
Matrix Size	Naive Algorithm (per seconds)	Strassen Algorithm (per seconds)
128	0.002	0.001
256	0.013	0.01
512	0.099	0.057
1024	0.862	0.366
2048	16.743	2.573
4096	178.7	83.84



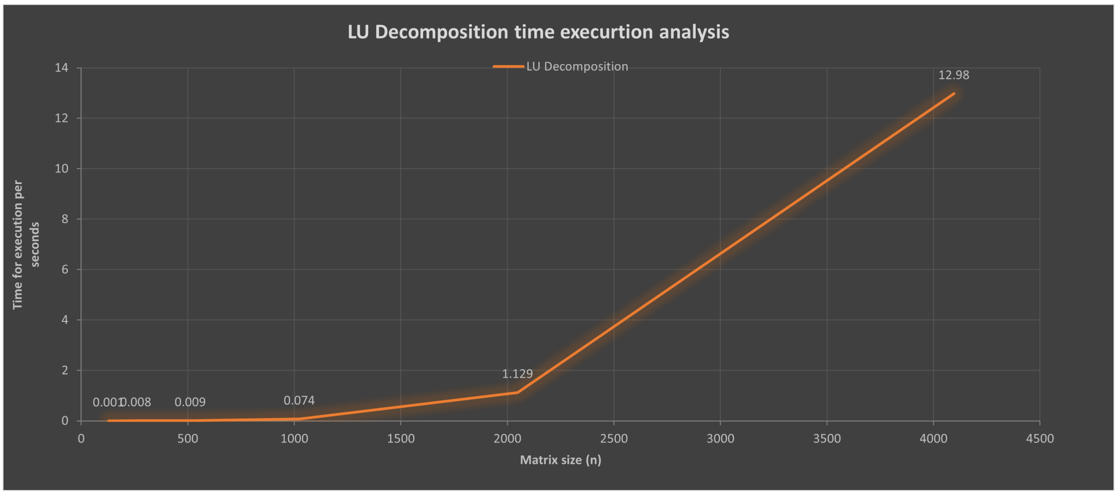
2.2 Matrix Inversion Benchmarking

Matrix Inversion			
Matrix Size	LU inverse (per seconds)	Strassen inverse using naive multiplication (per seconds)	Strassen inverse using strassen multiplication (per seconds)
128	0.003	0.004	0.003
256	0.011	0.017	0.013
512	0.88	0.116	0.078
1024	0.762	2.983	0.52
2048	9.513	31.022	3.876
4096	107.93	1100.114	26.045



2.3 LU Decomposition Benchmarking

LU Decomposition	
Matrix Size	Execution time per seconds
128	0.001
256	0.008
512	0.009
1024	0.074
2048	1.129
4096	12.98



3 Conclusion and Performance analysis

3.1 Matrix Multiplication

Naive Multiplication:

Matrix size 128 took 0.002 seconds, whereas for bigger sizes of matrices, the multiplication time grew significantly. The graph scaled rapidly for big matrix sizes, going up to 178.7 seconds for matrices of size 4096.

Strassen Multiplication:

The execution time for the Strassen multiplication algorithm outperformed the naive approach. As the size of matrices increased, the time required by the Strassen Algorithm also grew but at a lesser rate compared to that of the naive approach, reaching 83.84 seconds for matrices of size 4096.

Conclusion:

Efficiency of Strassen Algorithm: The Strassen matrix multiplication algorithm is more efficient compared to the naive multiplication approach, as depicted by the results of execution time for different matrix sizes.

Scalability: The advantage of the Strassen algorithm will be more pronounced with increasing matrix size, which makes it scalable and effective for large matrices.

Optimal Application: Considering the time complexities reduction by Strassen's algorithm, it finds its perfect application when dealing with large-sized matrix computations where performance optimization is required.

Trade-offs: The Strassen algorithm has faster multiplication for larger matrices, but at the cost of implementation complexity and other practical limitations, including the overhead introduced by recursive calls.

Therefore, according to the given data, the Strassen matrix multiplication algorithm is a better choice compared to the naive approach in matrix multiplication, especially for higher matrix sizes where its performance gains are much more noticeable.

3.2 Matrix Inversion

LU Inversion:

LU Inversion by LU Decomposition (using Gaussian elimination) showed a very consistent performance on different matrix sizes, with the execution time ranging from 0.003 seconds for size 128 to 107.93 seconds for size 4096.
Though the method of LU Inversion is reliable, for larger-sized matrices, the time complexities are usually higher, as reflected in the increasing values of time. Strassen Inversion using Naive Multiplication:

Strassen Inversion based on naive multiplication:

This algorithm exhibited slower performance. Execution times from size 128 with 0.004 seconds to the same of size 4096 at 1100 seconds.
The naive multiplication idea brought a high execution time by this Strassen inversion system with a larger access type in the time access diagram

Strassen inversion with Strassen multiplication
The execution time was always the best compared to other methods for every size of the matrix input.
For all the sizes, execution times were way below other methods, with the maximum 26.045 seconds for size 4096.

Conclusion:

Efficiency of Strassen Inversion with Strassen Multiplication: The method of Strassen Inversion using Strassen multiplication demonstrated the best performance among the compared methods and proved to be quite efficient for matrix inversion tasks.

Reliability of LU Inversion: The LU Inversion based on LU Decomposition gave a performance that was stable and reliable over different matrix sizes, even though execution times increased with increasing matrix size.

Shortcomings of Strassen Inversion with Naive Multiplication: Among all the three methods, the Strassen inversion using naive multiplication performed worse, especially for larger matrices, since the time complexity of the naive multiplication is higher.

Optimal Method Selection: Based on the performance analysis, the Strassen Inversion using Strassen multiplication has emerged as the preferred way to achieve efficient and fast matrix inversion, especially for higher sizes of matrices where the optimizations of performance matter a lot.

Among the above-mentioned algorithms, Strassen Inversion using Strassen multiplication is the best suited for matrix inversion because of its performance compared to other approaches like LU Inversion and naive multiplication-based Strassen Inversion, which gets even better when matrix size increases.

3.3 LU decomposition (using Gaussian elimination)

Based on the given data, the following inferences can be drawn about the execution time analysis done for LU Decomposition to perform matrix factorization.

Steady Performance: LU Decomposition exhibited steady performance for a range of matrix sizes, with the execution times increasing gradually as the size of the matrices increased.

Efficiency for Small to Medium Matrices: For the smaller to medium-sized matrices, LU Decomposition executed very quickly and efficiently for matrix sizes up to 1024, with execution times ranging from 0.001 seconds for size 128 to 0.074 seconds for size 1024.

Increasing Time Complexity: Once the matrix size was greater than 1024, the execution time for LU Decomposition began to increase dramatically; for matrices of size 4096, execution times reached up to 12.98 seconds.

Scalability Concerns: Increasing time complexity with larger matrices does suggest some scalability concerns of using LU Decomposition for very large matrices.

While LU Decomposition exhibits good performance for small and medium-sized matrices, its efficiency decreases as the size increases. Therefore, when very large matrices are involved, there might be a need for other methods or optimizations in order to overcome the growing time complexity of the used LU Decomposition.