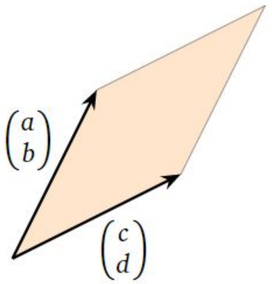
Calculating the Determinant using LU Decomposition

Team Members: Angel Badillo and Chad Callender

Determinant Applications

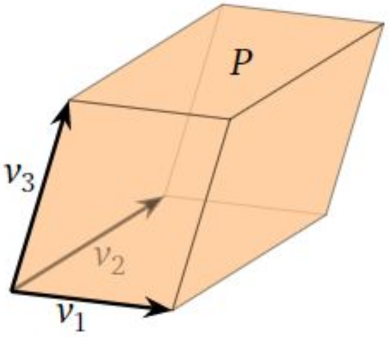
- Singularity and Invertibility
- Cardinality of a solution set in a system of linear equations
- Used to solve for the inverse of a non-singular matrix.
- Volume of regions in R^n

Area of Parallelogram



$$\text{area} = \left|\det\begin{pmatrix} a & b \\ c & d \end{pmatrix}\right| = |ad - bc|$$

Volume of Parallelepipeds



$$|\det(A)| = \mathrm{vol}(P).$$

About the Determinant

- A scalar value from the real valued function det on the set of square matrices
- Determinant rules:
  - **det(A) = 0 means not invertible.**
  - **det(A*B) = det(A)*det(B).**
- Many formulas for calculating the determinant:
  - **Leibniz formula**
  - **Cofactor Expansion**
  - **Gaussian Elimination**
- Determinant of a triangular matrix is the product of the diagonal entries.

Leibniz Formula

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{\sigma(i),i}$$

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh.$$

Cofactor Expansion

$$\det(A) = \sum_{j=1}^{n} a_{ij} C_{ij} = a_{i1} C_{i1} + a_{i2} C_{i2} + \cdots + a_{in} C_{in}.$$

$$\det \mathrm{A} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - d \begin{vmatrix} b & c \\ h & i \end{vmatrix} + g \begin{vmatrix} b & c \\ e & f \end{vmatrix}$$

Triangular Matrices

upper-triangular

$$\begin{pmatrix} \star & \star & \star & \star \\ 0 & \star & \star & \star \\ 0 & 0 & \star & \star \\ 0 & 0 & 0 & \star \end{pmatrix}$$

lower-triangular

$$\begin{pmatrix} \star & 0 & 0 & 0 \\ \star & \star & 0 & 0 \\ \star & \star & \star & 0 \\ \star & \star & \star & \star \end{pmatrix}$$

diagonal

$$\begin{pmatrix} \star & 0 & 0 & 0 \\ 0 & \star & 0 & 0 \\ 0 & 0 & \star & 0 \\ 0 & 0 & 0 & \star \end{pmatrix}$$

About LU Decomposition

- **Lower-Upper Decomposition**
- Any square matrix can be decomposed into:
  - **Lower triangular matrix**
  - **Upper triangular matrix**
- Can be done using:
  - **Gaussian Elimination**
  - **Doolittle Algorithm**
- In MATLAB, LU decomposition is used for the computation of determinants, inverse matrices, and the symbolic matrix division operators.

$$[A] = [L]\,[U]$$

$[L] = \text{Lower triangular matrix}$

$[U] = \text{Upper triangular matrix}$

$$[A] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

Doolittle's Algorithm

Suppose A ∈ $F^{n,n}$. Then A can be decomposed into matrices L, U ∈ $F^{n,n}$ where A = LU.

$$\mathbf{LU} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \mathbf{A}$$

If we multiply the two matrices on the left together, we have

$$\begin{bmatrix} l_{11}u_{11} & l_{11}u_{12} & l_{11}u_{13} \\ l_{21}u_{11} & l_{21}u_{12}+l_{22}u_{22} & l_{21}u_{13}+l_{22}u_{23} \\ l_{31}u_{11} & l_{31}u_{12}+l_{32}u_{22} & l_{31}u_{13}+l_{32}u_{23}+l_{33}u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Now if we take the same approach as before where the l_{ii}'s are the 1's we can solve the first row of equations trivially, namely

$$u_{11} = a_{11}, \quad u_{12} = a_{12}, \quad u_{13} = a_{13},$$

then we have enough information to solve the rest of the first column,

$$l_{21} = a_{21}/a_{11}, \quad l_{31} = a_{31}/a_{11},$$

and the rest of the second row,

$$u_{22} = (a_{22} - a_{21}^2/a_{11}), \quad u_{23} = (a_{23} - a_{21}a_{23}/a_{11}),$$

etc.

Sequential Approach

Steps	
1.	For $k = 1, 2, \ldots, n$ do Steps 2-3, 5
2.	Set $l_{kk} = 1$
3.	For $j = k, k+1, \ldots, n$ do Step 4
4.	$u_{kj} = a_{kj} - \sum_{m=1}^{k-1} l_{km} u_{mj}$
5.	For $i = k+1, k+2, \ldots, n$ do Step 6
6.	$l_{ik} = \left(a_{ik} - \sum_{m=1}^{k-1} l_{im} u_{mk} \right) \Big/ u_{kk}$

Doolittle's method of LU factorization

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

By matrix-matrix multiplication

$$a_{1j} = u_{1j}, \qquad j = 1,2,\ldots,n$$

$$a_{ij} = \begin{cases} \sum_{t=1}^{j} l_{it} u_{tj}, & \text{when } j < i \\ \sum_{t=1}^{i-1} l_{it} u_{tj} + u_{ij} & \text{when } j \geq i \end{cases}$$

Therefore

$$u_{1j} = a_{1j}, \qquad j = 1,2,\ldots,n \text{ (1st row of } U)$$

$$l_{j1} = a_{j1}/u_{11}, \qquad j = 1,2,\ldots,n \text{ (1st column of } L)$$

For $i = 2,3,\ldots,n-1$ **do**

$$u_{ii} = a_{ii} - \sum_{t=1}^{i-1} l_{it} u_{tj}$$

$$u_{ij} = a_{ij} - \sum_{t=1}^{i-1} l_{it} u_{tj} \qquad \text{for } j = i+1,\ldots,n \text{ (ith row of } U)$$

$$l_{ji} = \frac{a_{ji} - \sum_{t=1}^{i-1} l_{jt} u_{ti}}{u_{ii}} \qquad \text{for } j = i+1,\ldots,n \text{ (ith column of } L)$$

End

$$u_{nn} = a_{nn} - \sum_{t=1}^{n-1} l_{nt} u_{tn}$$

Serial C++ Code

```cpp
#include <iostream>
#include <cstddef>
#include <vector>
#include <random>
#include <limits>
#include <ctime>
#include <chrono>
using namespace std;

// Represents a 2-D vector of integers, i.e a 2-D dynamic array that can
// change in size.
using Matrix_t = vector<vector<double>>;
// Represents a row vector of integers, i.e a 1-D array that can change
// in size.
using Row_t = vector<double>;

// Size N for NxN square matrix.
const size_t matrix_size = 512;

/// @brief Performs LU Decomposition using the Doolittle Algorithm.
/// @param matrix Matrix of size NxN to decompose.
/// @param lower_matrix Lower triangular matrix.
/// @param upper_matrix Upper triangular matrix.
void lu_decomposition(const Matrix_t &matrix, Matrix_t &lower_matrix, Matrix_t &upper_matrix);

/// @brief Computes the determinant of a triangular matrix.
/// @param matrix Triangular matrix.
/// @return The determinant, a double.
double determinant_triangular(const Matrix_t &matrix);
```

Serial C++ Code

```cpp
#include <iostream>
#include <cstddef>
#include <vector>
#include <random>
#include <limits>
#include <ctime>
#include <chrono>
using namespace

// Represents a                      mic array that can
// change in si
using Matrix_t
// Represents a                      y that can change
// in size.
using Row_t = v

// Size N for NxN square matrix.
const size_t matrix_size = 512;

/// @brief Performs LU Decomposition using the Doolittle Algorithm.
/// @param matrix Matrix of size NxN to decompose.
/// @param lower_matrix Lower triangular matrix.
/// @param upper_matrix Upper triangular matrix.
void lu_decomposition(const Matrix_t &matrix, Matrix_t &lower_matrix, Matrix_t &upper_matrix);

/// @brief Computes the determinant of a triangular matrix.
/// @param matrix Triangular matrix.
/// @return The determinant, a double.
double determinant_triangular(const Matrix_t &matrix);
```

Serial C++ Code

```cpp
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_matrix(matrix_size, -1.0, 1.0);

    // Lower and upper triangular matrices
    Matrix_t lower(matrix_size, Row_t(matrix_size, 0));
    Matrix_t upper(matrix_size, Row_t(matrix_size, 0));

    // Begin timing
    auto start_time = std::chrono::steady_clock::now();

    // Perform LU decomposition
    lu_decomposition(matrix, lower, upper);

    // Comput determinant
    l_det = determinant_triangular(lower);
    u_det = determinant_triangular(upper);

    // det(A) = det(U) since the main diagonal of L is all 1's
    // and therefore does not contribute to the product
    determinant = u_det;

    // Stop timing
    auto end_time = std::chrono::steady_clock::now();

    // Calculate elapsed time
    auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time).count();
```

Serial C++ Code

```cpp
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_ma

    // Lower and upper triangul
    Matrix_t lower(matrix_size,
    Matrix_t upper(matrix_size,

    // Begin timing
    auto start_time = std::chro

    // Perform LU decompositior
    lu_decomposition(matrix, lo

    // Comput determinant
    l_det = determinant_triangu
    u_det = determinant_triangu

    // det(A) = det(U) since th
    // and therefore does not c
    determinant = u_det;

    // Stop timing
    auto end_time = std::chrono::steady_clock::now();

    // Calculate elapsed time
    auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time).count();
```

```cpp
void lu_decomposition(const Matrix_t &matrix, Matrix_t &lower_matrix, Matrix_t &upper_matrix)
{
    size_t size = matrix.size();

    for (size_t i = 0; i < size; i++)
    {
        // Upper triangular
        for (size_t k = i; k < size; k++)
        {
            // Summation of L(i, j) * U(j, k)
            double sum = 0;
            for (size_t j = 0; j < i; j++)
            {
                sum += (lower_matrix[i][j] * upper_matrix[j][k]);
            }
            // Evaluating U(i, k)
            upper_matrix[i][k] = matrix[i][k] - sum;
        }
```

Serial C++ Code

```cpp
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_m

    // Lower and upper triangu
    Matrix_t lower(matrix_size
    Matrix_t upper(matrix_size

    // Begin timing
    auto start_time = std::chr

    // Perform LU decompositio
    lu_decomposition(matrix, l

    // Comput determinant
    l_det = determinant_triang
    u_det = determinant_triang

    // det(A) = det(U) since t
    // and therefore does not
    determinant = u_det;

    // Stop timing
    auto end_time = std::chron

    // Calculate elapsed time
    auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time).count();
```

```cpp
            // Lower triangular
            for (size_t k = i; k < size; k++)
            {
                // Diagonal as 1
                if (i == k)
                {
                    lower_matrix[i][i] = 1;
                }
                else
                {
                    // Summation of L(k, j) * U(j, i)
                    double sum = 0;
                    for (size_t j = 0; j < i; j++)
                    {
                        sum += (lower_matrix[k][j] * upper_matrix[j][i]);
                    }

                    // Evaluating L(k, i)
                    lower_matrix[k][i] = (matrix[k][i] - sum) / upper_matrix[i][i];
                }
            }
        }
    }
```

Serial C++ Code

```cpp
int main()
{
    // Determinants for origina
    double determinant, l_det,

    // Matrix to decompose
    Matrix_t matrix = random_ma

    // Lower and upper triangul
    Matrix_t lower(matrix_size,
    Matrix_t upper(matrix_size,

    // Begin timing
    auto start_time = std::chro

    // Perform LU decomposition
    lu_decomposition(matrix, lo

    // Comput determinant
    l_det = determinant_triangu
    u_det = determinant_triangu

    // det(A) = det(U) since th
    // and therefore does not c
    determinant = u_det;

    // Stop timing
    auto end_time = std::chrono

    // Calculate elapsed time
    auto elapsed = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time).count();
```

```cpp
double determinant_triangular(const Matrix_t &matrix)
{
    size_t size = matrix.size();

    // 0x0 matrix has determinant of 1
    // or a ln(det) = 0
    double determinant = 0;

    // Compute determinant of triangular matrix
    // Here, we compute the natural log of the absolute
    // value of the determinant because of data type
    // range constraints
    for (size_t i = 0; i < size; i++)
    {
        determinant += log(abs(matrix[i][i]));
    }

    return determinant;
}
```

Parallel (Multi-threaded) Approach Explanation

- Uses C++ 11 STL <thread>
    - std::thread
        - Cross-platform
        - Language integration
        - Object-Oriented
- Based on the Doolittle Algorithm.
- Every thread shares matrix **A, L, and U**.
- Each thread performs a portion of the outermost for-loop.
- Each thread computes **n / t** rows of **U**, and columns of **L** where **n** is the order of a square matrix of **n x n** size, and **t** is the number of threads.

*In code, row / column indices begin at 0.

$$[A] = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

$$= \begin{array}{c} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix} \\ \begin{array}{cccc} \text{t0} & \text{t1} & & \text{t(n-1)} \end{array} \end{array} \begin{array}{cc} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} & \begin{array}{c} \text{t0} \\ \text{t1} \\ \\ \text{t(n-1)} \end{array} \end{array}$$

Multi-threaded C++ Code

```
#include <iostream>
#include <cstddef>
#include <vector>
#include <thread>
#include <utility>
#include <functional>
#include <random>
#include <limits>
#include <ctime>
#include <chrono>
using namespace std;

// Represents a 2-D vector of integers, i.e a 2-D dynamic array
// change in size.
using Matrix_t = vector<vector<double>>;
// Represents a row vector of integers, i.e a 1-D array that ca
// in size.
using Row_t = vector<double>;

// Size N for NxN square matrix.
const size_t matrix_size = 512;
// Number of threads
const size_t num_threads = 32;

```

Multi-threaded C++ Code

```cpp
#include <iostream>
#include <cstddef>
#include <vector>
#include <thread>
#include <utility
#include <functio
#include <random>
#include <limits>
#include <ctime>
#include <chrono>
using namespace std;

// Represents a 2-D vector of integers, i.e a 2-D dynamic array
// change in size.
using Matrix_t = vector<vector<double>>;
// Represents a row vector of integers, i.e a 1-D array that ca
// in size.
using Row_t = vector<double>;

// Size N for NxN square matrix.
const size_t matrix_size = 512;
// Number of threads
const size_t num_threads = 32;
```

#include <thread>

Multi-threaded C++ Code

```cpp
#include <iostream>
#include <cstddef>
#include <vector>
#include <thread>
#include <utility>
#include <functional>
#include <random>
#include <limits>
#include <ctime>
#include <chrono>
using namespace std;

// Represents a 2-D vect
// change in size.
using Matrix_t = vector<
// Represents a row vect
// in size.
using Row_t = vector<dou

// Size N for NxN square
const size_t matrix_size = 512;
// Number of threads
const size_t num_threads = 32;
```

```cpp
// Size N for NxN square matrix.
const size_t matrix_size = 512;
// Number of threads
const size_t num_threads = 32;
```

Multi-threaded C++ Code

```cpp
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_matrix(matrix_size, -1.0, 1.0);

    // Lower and upper triangular matrices
    Matrix_t lower(matrix_size, Row_t(matrix_size, 0));
    Matrix_t upper(matrix_size, Row_t(matrix_size, 0));

    // Vector to store threads
    vector<thread> threads;

    // Begin timing
    auto start_time = std::chrono::steady_clock::now();

    // Creating threads and storing them in the vector of threads
    for (size_t i = 0; i < num_threads; i++)
    {
        // Create thread
        thread temp_thread = thread(lu_decomposition, i, cref(matrix), ref(lower), ref(upper));
        // Move contents of temp_thread into threads vector,
        // instead of copying.
        threads.push_back(move(temp_thread));
    }

    // Join threads for synchronization
    for (size_t i = 0; i < num_threads; i++)
    {
        threads[i].join();
    }

    // Compute determinant
    l_det = determinant_triangular(lower);
    u_det = determinant_triangular(upper);
```

Multi-threaded C++ Code

```cpp
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_matrix(matrix_size, -1.0, 1.0);

    // Lower and upper triangular matrices
    Matrix_t lower(matrix_s
    Matrix_t upper(matrix_s

    // Vector to store thre
    vector<thread> threads;

    // Begin timing
    auto start_time = std::chrono::steady_clock::now();

    // Creating threads and storing them in the vector of threads
    for (size_t i = 0; i < num_threads; i++)
    {
        // Create thread
        thread temp_thread = thread(lu_decomposition, i, cref(matrix), ref(lower), ref(upper));
        // Move contents of temp_thread into threads vector,
        // instead of copying.
        threads.push_back(move(temp_thread));
    }

    // Join threads for synchronization
    for (size_t i = 0; i < num_threads; i++)
    {
        threads[i].join();
    }

    // Compute determinant
    l_det = determinant_triangular(lower);
    u_det = determinant_triangular(upper);
```

```cpp
// Vector to store threads
vector<thread> threads;
```

Multi-threaded C++ Code

```cpp
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_m

    // Lower and upper triangu
    Matrix_t lower(matrix_size
    Matrix_t upper(matrix_size

    // Vector to store threads
    vector<thread> threads;

    // Begin timing
    auto start_time = std::chr

    // Creating threads and st
    for (size_t i = 0; i < num
    {
        // Create thread
        thread temp_thread = thread(lu_decomposition, i, cref(matrix), ref(lower), ref(upper));
        // Move contents of temp_thread into threads vector,
        // instead of copying.
        threads.push_back(move(temp_thread));
    }

    // Join threads for synchronization
    for (size_t i = 0; i < num_threads; i++)
    {
        threads[i].join();
    }

    // Compute determinant
    l_det = determinant_triangular(lower);
    u_det = determinant_triangular(upper);
```

```cpp
// Creating threads and storing them in the vector of threads
for (size_t i = 0; i < num_threads; i++)
{
    // Create thread
    thread temp_thread = thread(lu_decomposition, i, cref(matrix), ref(lower), ref(upper));
    // Move contents of temp_thread into threads vector,
    // instead of copying.
    threads.push_back(move(temp_thread));
}
```

Multi-threaded C++ Code

```
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_matrix(matrix_size, -1.0, 1.0);

    // Lower and upper triangular matrices
    Matrix_t lower(matrix_size, Row_t(matrix_size, 0));
    Matrix_t upper(matrix_size, Row_t(matrix_size, 0));

    // Vector to store threads
    vector<thread> threads;

    // Begin timing
    auto start_time = std::chrono::steady_clock::now();

    // Creating threads and storing them in the vector of threads
    for (size_t i = 0; i < num_threads; i++)
    {
        // Create thread
        thread temp_thread = thread(lu_decomposition, i, cref(matrix), ref(lower), ref(upper));
        // Move contents of temp_thread into threads vector,
        // instead of copying.
        threads.push_back(move(temp_thread));
    }

    // Join threads for synchronization
    for (size_t i = 0; i < num_threads; i++)
    {
        threads[i].join();
    }

    // Compute determinant
    l_det = determinant_triangular(lower);
    u_det = determinant_triangular(upper);
```

Multi-threaded C++ Code

```cpp
void lu_decomposition(const size_t threadId, const Matrix_t &matrix, Matrix_t
{
    // N from NxN sized matrix
    size_t size = matrix.size();

    // Starting index for the outer for loop
    size_t start = threadId * size / num_threads;
    // One over the last index for the outer for loop
    size_t end = (threadId + 1) * size / num_threads;

    for (size_t i = start; i < end; i++)
    {
        // Upper triangular
        for (size_t k = i; k < size; k++)
        {
            // Summation of L(i, j) * U(j, k)
            double sum = 0;
            for (size_t j = 0; j < i; j++)
            {
                sum += (lower_matrix[i][j] * upper_matrix[j][k]);
            }
            // Evaluating U(i, k)
            upper_matrix[i][k] = matrix[i][k] - sum;
        }

        // Lower triangular
        for (size_t k = i; k < size; k++)
        {
            // Diagonal as 1
            if (i == k)
            {
                lower_matrix[i][i] = 1;
            }
            else
            {
                // Summation of L(k, j) * U(j, i)
                double sum = 0;
                for (size_t j = 0; j < i; j++)
                {
                    sum += (lower_matrix[k][j] * upper_matrix[j][i]);
                }

                // Evaluating L(k, i)
                lower_matrix[k][i] = (matrix[k][i] - sum) / upper_matrix[i][i];
            }
        }
    }
}
```

Multi-threaded C++ Code

```
void lu_decomposition(const size_t threadId, const Matrix_t &matrix, Matrix_t
{
    // N from NxN sized matrix
    size_t size = matrix.size();

    // Starting index for the outer for loop
    size_t start = threadId * size / num_threads;
    // One over the last index for the outer for loop
    size_t end = (threadId + 1) * si

    for (size_t i = start; i < end; i
    {
        // Upper triangular
        for (size_t k = i; k < size;
        {
            // Summation of L(i, j)
            double sum = 0;
            for (size_t j = 0; j < i
            {
                sum += (lower_matrix
            }
            // Evaluating U(i, k)
            upper_matrix[i][k] = matrix[i][k] - sum;
        }
```

```
        // Lower triangular
        for (size_t k = i; k < size; k++)
        {
            // Diagonal as 1
            if (i == k)
            {
                lower_matrix[i][i] = 1;
...
per_matrix[j][i]);
...
- sum) / upper_matrix[i][i];
        }
    }
}
```

```
// Starting index for the outer for loop
size_t start = threadId * size / num_threads;
// One over the last index for the outer for loop
size_t end = (threadId + 1) * size / num_threads;

for (size_t i = start; i < end; i++)
```

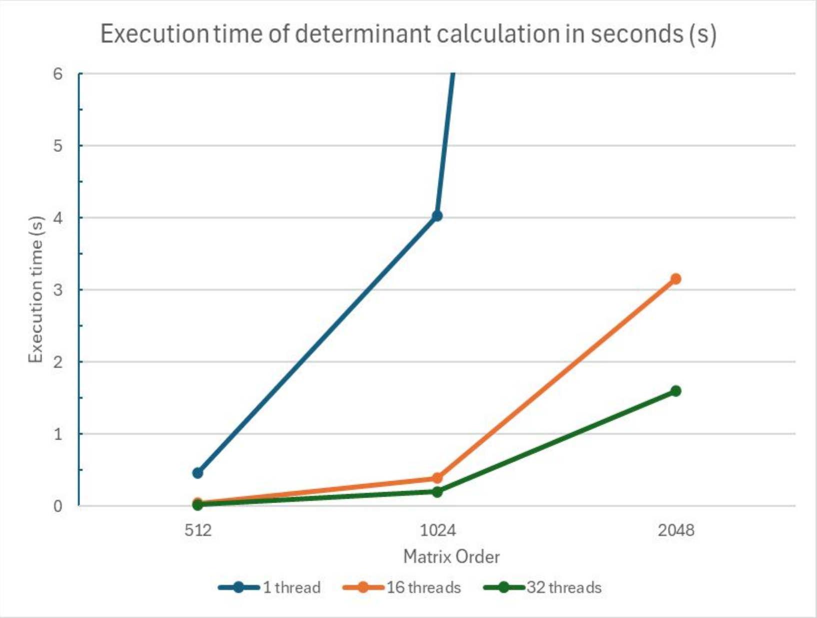
Multi-threaded C++ Code

```
int main()
{
    // Determinants for original matrix, lower triangular matrix, and upper triangular matrix
    double determinant, l_det, u_det;

    // Matrix to decompose
    Matrix_t matrix = random_matrix(matrix_size, -1.0, 1.0);

    // Lower and upper triangular matrices
    Matrix_t lower(matrix_size, Row_t(matrix_size, 0));
    Matrix_t upper(matrix_size, Row_t(matrix_size, 0));

    // Vector to store threads
    vector<thread> threads;

    // Begin timing
    auto start_time = std::chrono::steady_clock::now();

    // Creating threads and storing th
    for (size_t i = 0; i < num_threads
    {
        // Create thread
        thread temp_thread = thread(lu
        // Move contents of temp_threa
        // instead of copying.
        threads.push_back(move(temp_th
    }

    // Join threads for synchronizatio
    for (size_t i = 0; i < num_threads
    {
        threads[i].join();
    }

    // Compute determinant
    l_det = determinant_triangular(lower);
    u_det = determinant_triangular(upper);
```


```
// Join threads for synchronization
for (size_t i = 0; i < num_threads; i++)
{
    threads[i].join();
}
```

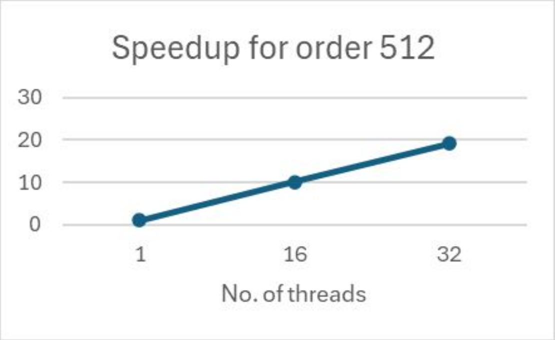
Execution Time

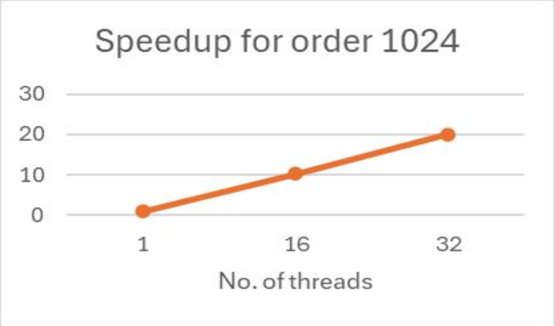
Execution Time in Seconds			
	Order of Matrix		
Number of Threads	512	1024	2048
1	0.463626449	4.026737	33.30727551
16	0.045968253	0.390925782	3.153950131
32	0.02433264	0.201477472	1.600055011

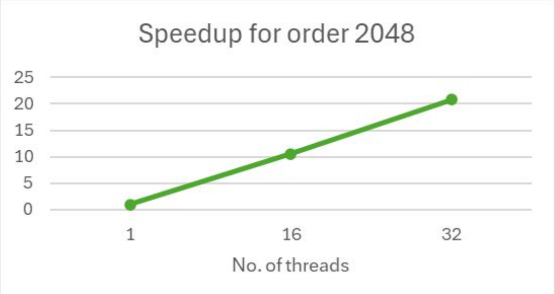


Speedup

Speedup			
	Order of Matrix		
Number of Threads	512	1024	2048
1	1.0	1.0	1.0
16	10.08579652	10.30051448	10.56049529
32	19.05368491	19.98603931	20.81633149







References:

https://www.geeksforgeeks.org/doolittle-algorithm-lu-decomposition/#

https://math.libretexts.org/Bookshelves/Linear_Algebra/Introduction_to_Matrix_Algebra_(Kaw)/01%3A_Chapters/1.07%3A_LU_Decomposition_Method_for_Solving_Simultaneous_Linear_Equations

https://math.libretexts.org/Bookshelves/Linear_Algebra/A_First_Course_in_Linear_Algebra_(Kuttler)/03%3A_Determinants/3.02%3A_Properties_of_Determinants#thm:addingmultipleofrow

https://www.engr.colostate.edu/~thompson/hPage/CourseMat/Tutorials/CompMethods/doolittle.pdf

https://math.libretexts.org/Bookshelves/Linear_Algebra/Interactive_Linear_Algebra_(Margalit_and_Rabinoff)/04%3A_Determinants/4.03%3A_Determinants_and_Volumes

References:

https://www.mathworks.com/help/matlab/ref/lu.html#d126e954784

https://www3.nd.edu/~zxu2/acms40390F11/Alg-LU-Crout.pdf

https://www.geeksforgeeks.org/thread-in-operating-system/

https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_LU.html

Thank you!

Any questions?