Analisis Perbandingan Kompleksitas Waktu Metode *Inverse* Matriks dengan Metode Adjoint, Eliminasi Gauss Jordan, dan Dekomposisi LU dalam Penyelesaian SPL

Abdullah Farhan - 13523042[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523042@std.stei.itb.ac.id, [2]farhanjuneadi213@gmail.com

***Abstract*—Terdapat banyak metode dalam penyelesaian Sistem Persamaan Linier (SPL). Metode-metode tersebut meliputi: metode *inverse* matriks, metode eliminasi Gauss-Jordan, metode dekomposisi LU, dan lainnya. Setiap metode penyelesaian memiliki keunggulan, tetapi beberapa metode memiliki performa yang lebih baik dibandingkan yang lain. Perbandingan antara metode-metode tersebut dapat dilakukan dengan menganalisis kompleksitas waktu dari tiap metode. Makalah ini akan membandingkan metode *inverse* matriks, metode eliminasi Gauss-Jordan, dan metode dekomposisi LU. Makalah ini membatasi pembahasan pada SPL yang memiliki jumlah variabel dan persamaan yang sama, serta solusi yang unik, karena kaidah *inverse* matriks hanya dapat diterapkan pada SPL dengan kondisi-kondisi tersebut**

***Keywords*—Kompleksitas Algoritma, Metode *Inverse* Matriks, Metode Gauss-Jordan, Dekomposisi LU, SPL**

I. Pendahuluan

Sistem Persamaan Linear adalah suatu sistem persamaan berisi persamaan linear. Pada, persamaan linear pangkat dari variabel-variabel yang ada bersifat linear atau hanya berpangkat 1. Biasanya variabel akan diletakkan di sisi kiri persamaan sementara konstanta diletakkan di sisi kanan persamaan. Banyak variabel yang terlibat tidak terbatas melainkan tergantung kebutuhan. Akan tetapi, dalam suatu sistem persamaan linear, semua persamaan yang terlibat harus memiliki minimal 1 variabel di bagian kiri persamaan agar solusi SPL tidak bersifat parametrik.

Ada banyak cara untuk menyelesaikan SPL. Dalam tulisan ini, metode yang akan dibahas meliputi metode *inverse* matriks, metode eliminasi Gauss-Jordan, dan metode dekomposisi LU. Dalam menentukan metode yang paling optimal dari ketiga pendekatan pemrograman tersebut, perlu dilakukan pengujian efisiensi pada masing-masing algoritma.

Efisiensi algoritma dapat dihitung berdasarkan waktu dan penggunaan memori saat algoritma dijalankan tersebut. Suatu algoritma dikatakan efisien jika menggunakan waktu dan memori yang minimal dalam menjalankannya.

Kompleksitas algoritma merupakan ukuran yang digunakan dalam mengukur waktu dan penggunaan memori yang dibutuhkan suatu algoritma. Ada dua jenis kompleksitas algoritma: kompleksitas waktu yang mengukur berapa langkah yang dibutuhkan program, dan kompleksitas ruang.

Dalam tulisan ini, evaluasi berbagai metode penyelesaian SPL akan difokuskan pada perbandingan kompleksitas waktu masing-masing algoritma karena saat ini masalah memori komputer tidak terlalu kritis jika dibandingkan dengan waktu.

II. Kajian Teori

1. Sistem Persamaan Linier

Persamaan linier merupakan persamaan aljabar dengan variabel yang berpangkat satu. Persamaan linier yang memiliki variabel $x_1, x_2, \ldots, x_n$ bisa ditulis sebagai

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$$

dengan b dan koefisien $a_1, a_2, \ldots, a_n$ adalah bilangan real atau kompleks, sedangkan n adalah bilangan bulat positif.

Sistem Persamaan Linier (SPL) merupakan gabungan dari satu atau lebih persamaan linear yang mempunyai variabel sama. [2]

2. Kompleksitas Algoritma

Suatu parameter yang mengindikasikan kebutuhan waktu dan memori dalam eksekusi algoritma disebut kompleksitas algoritma, terlepas dari jenis komputer atau compiler yang digunakan.

Ada dua jenis kompleksitas: kompleksitas waktu dan kompleksitas ruang. [1]

3. Kompleksitas Waktu

Fungsi $T(n)$ yang merepresentasikan kompleksitas waktu menunjukkan banyaknya tahapan yang harus dilakukan program untuk menyelesaikan permasalahan berdasarkan ukuran input yang berukuran n. Variabel n merepresentasikan banyaknya data yang yang diproses oleh algoritma. [1]

Dengan menganalisis kompleksitas waktu, kita mampu

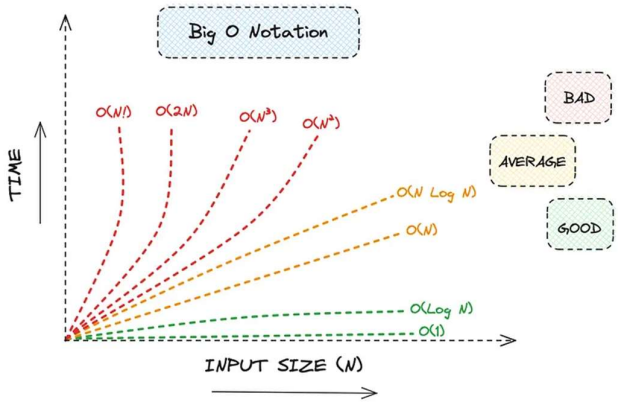
mengidentifikasi algoritma yang lebih optimal. Dalam proses perbandingan algoritma menggunakan kompleksitas waktu, algoritma yang memiliki nilai kompleksitas waktu lebih rendah dapat dianggap sebagai algoritma yang lebih efisien.

Seringkali, kompleksitas waktu suatu algoritma dinyatakan menggunakan notasi "O-Besar" atau O(n), yaitu notasi kompleksitas waktu asimptotik. Notasi "O-Besar" digunakan saat kita tidak memerlukan pengukuran kompleksitas waktu secara presisi. Notasi ini menunjukkan seberapa cepat kompleksitas waktu meningkat seiring dengan bertambahnya ukuran masukan (n). Jika kompleksitas waktu suatu algoritma ditulis sebagai T(n), dan terdapat konstanta C yang memenuhi pertidaksamaan berikut

$$T(n) \leq Cf(n)$$

untuk $n \geq n_0$, maka

$$T(n) = O(f(n)).$$



Gambar 1. Grafik *Big O Notation*

(Sumber: https://medium.com/@agustin.ignacio.rossi/coding-interview-preparation-big-o-notation-8066c3b0ce60)

4. Metode *Inverse* Matriks

Metode *inverse* matriks adalah cara yang dapat digunakan untuk mencari solusi dari suatu SPL dengan menggunakan invers dari matriks koefisien. Jika sebuah SPL bisa ditulis sebagai $Ax = b$, dengan A adalah matriks angka pengali, x adalah vektor variabel yang dicari, dan b adalah vektor konstanta, maka solusi SPL dapat ditemukan dengan mengalikan kedua sisi dengan A^{-1}, sehingga didapatkan $x = A^{-1}b$. [2]

5. Metode Eliminasi Gauss-Jordan

Metode Eliminasi Gauss-Jordan adalah metode yang dapat digunakan dalam mencari solusi suatu SPL dengan mengubah matriks *augmented* menggunakan Operasi Baris Elementer (OBE) hingga didapatkan matriks eselon baris tereduksi.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{bmatrix} \sim \text{OBE} \sim \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & * \\ 0 & 1 & 0 & \dots & 0 & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \vdots & 1 & * \end{bmatrix}$$

Gambar 2. Metode Eliminasi Gauss Jordan

(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-05-Sistem-Persamaan-Linier-2023.pdf)

Ada tiga jenis OBE yang yang bisa digunakan pada sebuah matriks *augmented*, yaitu mengalikan sebuah baris dengan konstanta bukan nol, menukar dua baris, dan menambahkan sebuah baris dengan kelipatan baris lainnya. [2]

6. Metode Dekomposisi LU

Metode dekomposisi LU adalah metode yang menguraikan matriks koefisien A menjadi perkalian dua buah matriks, yaitu matriks segitiga bawah (Lower triangular) L dan matriks segitiga atas (Upper triangular) U. Secara konsep, sebuah matriks A bisa diuraikan menjadi perkalian L dan U karena setiap operasi baris elementer yang dilakukan pada proses eliminasi Gauss dapat dinyatakan dalam bentuk perkalian matriks elementer. Matriks L merepresentasikan operasi-operasi baris yang dilakukan, sementara matriks U merupakan matriks segitiga atas hasil eliminasi.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ m_{21} & 1 & 0 & 0 \\ m_{31} & m_{32} & 1 & 0 \\ m_{41} & m_{42} & m_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

Gambar 3. Dekomposisi LU

(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-23-Dekomposisi-LU-2023.pdf)

Setelah didapatkan matriks L dan U, solusi SPL dapat ditemukan dengan menyelesaikan dua SPL, yaitu $Ly = b$ dan $Ux = y$ secara berurutan. [2]

III. Analisis Kompleksitas Waktu

A. Langkah-Langkah

1. Metode Gauss-Jordan

Langkah-langkah untuk menyelesaikan SPL dengan metode Gauss-Jordan adalah sebagai berikut:

a. Membentuk matriks augmented:
   - Gabungkan matriks koefisien A dengan vektor konstanta b
   - Untuk setiap baris i, gabungkan baris A[i] dengan b[i]
   - Hasilnya adalah matriks augmented berukuran $n \times (n+1)$

b. Melakukan proses eliminasi maju (*forward elimination*):
   - Untuk setiap baris i dari 0 hingga $n-1$:
     - Ambil elemen pivot = $augmented[i][i]$
     - Normalisasi baris pivot (baris ke-i):
       - Bagi setiap elemen pada baris tersebut dengan nilai pivot
       - Dilakukan untuk elemen dari kolom i hingga $n+1$
     - Eliminasi elemen di bawah dan di atas pivot:

- Untuk setiap baris $k \neq i$:
  - Ambil $faktor = augmented[k][i]$
  - Kurangi setiap elemen baris k dengan faktor × elemen baris i yang bersesuaian
  - Dilakukan untuk elemen dari kolom i hingga $n + 1$

c. Mengambil solusi:

- Setelah eliminasi selesai, kolom terakhir dari matriks augmented adalah solusi SPL
- Ambil elemen $augmented[i][n]$ untuk setiap i dari 0 hingga $n - 1$

2. Metode Dekomposisi LU

Langkah-langkah untuk menyelesaikan SPL dengan metode dekomposisi LU adalah sebagai berikut:

a. Inisialisasi matriks L dan U berukuran n×n dengan nilai 0

b. Untuk setiap baris i dari 0 hingga n-1:

- Untuk kolom k dari i hingga n-1 (membentuk U):
  - Hitung $sum = \sum_{j=0}^{i-1}(L[i][j] \times U[j][k])$
  - $U[i][k] = A[i][k] - sum$
- Untuk baris k dari i hingga n-1 (membentuk L):
  - Jika i = k, set L[i][i] = 1
  - Jika tidak:
    - Hitung $sum = \sum_{j=0}^{i-1}(L[k][j] \times U[j][i])$
    - $L[k][i] = (A[k][i] - sum) / U[i][i]$

c. Setelah matriks L dan U terbentuk:

- Gunakan *forward substitution* untuk menyelesaikan $Ly = b$
- Gunakan *backward substitution* untuk menyelesaikan $Ux = y$
- Vektor x adalah solusi SPL

3. Metode *Inverse* Matriks

Langkah-langkah untuk mendapatkan inverse dari matriks adalah sebagai berikut:

a. Hitung determinan matriks menggunakan ekspansi kofaktor:

- Untuk matriks 1×1, determinan adalah nilai elemen itu sendiri
- Untuk matriks 2×2, $determinan = ad - bc$ untuk matriks $[[a, b], [c, d]]$
- Untuk matriks $n \times n$ $(n > 2)$:
  - Pilih baris pertama untuk ekspansi
  - Untuk setiap elemen j pada baris tersebut:
    - Hitung minor dengan menghapus baris dan kolom elemen tersebut
    - Hitung kofaktor dengan mengalikan $(-1)^j$ dengan elemen
    - Kalikan kofaktor dengan determinan minor secara rekursif
  - Jumlahkan semua hasil perkalian

b. Hitung matriks kofaktor:

- Untuk setiap elemen (i, j) dalam matriks:
  - Hitung minor dengan menghapus baris i dan kolom j
  - Hitung determinan minor
  - Kalikan dengan $(-1)^{i+j}$

c. Transposisi matriks kofaktor untuk mendapatkan adjoin

d. Bagi setiap elemen matriks adjoin dengan determinan matriks awal

B. Implementasi

1. Metode Gauss-Jordan

```python
def gauss_jordan(A, b):
    n = len(A)
    augmented = [A[i] + [b[i]] for i in range(n)]
    for i in range(n):
        pivot = augmented[i][i]
        for j in range(i, n + 1):
            augmented[i][j] /= pivot
        for k in range(n):
            if k != i:
                factor = augmented[k][i]
                for j in range(i, n + 1):
                    augmented[k][j] -= factor * augmented[i][j]
    return [augmented[i][n] for i in range(n)]
```

Gambar 4. Fungsi gauss_jordan

(Sumber: Dokumen Penulis)

Implementasi metode Gauss-Jordan menggunakan satu fungsi utama *gauss_jordan(A, b)* yang menerima matriks A berukuran $n \times n$ dan vektor b berukuran n. Fungsi ini dimulai dengan membentuk matriks *augmented* yang menggabungkan matriks A dan vektor b menggunakan *list comprehension*. Pembentukan matriks *augmented* ini memiliki kompleksitas $O(n)$ karena harus melalui setiap baris matriks.

Proses utama dalam fungsi ini adalah *forward elimination* yang terdiri dari tiga *nested loop*. *Loop* terluar dengan variabel i berjalan sebanyak n kali untuk memproses setiap baris matriks. Untuk setiap iterasi i, pertama dilakukan pencarian pivot pada posisi $augmented[i][i]$. Kemudian dilakukan normalisasi baris pivot dengan membagi setiap elemen pada baris tersebut dengan nilai pivot melalui *loop* j yang berjalan dari i hingga $n + 1$, menghasilkan kompleksitas $O(n)$. Setelah normalisasi, dilakukan eliminasi pada semua baris lain melalui dua nested *loop*: *loop* k untuk setiap baris $(O(n))$ dan *loop* j untuk setiap kolom $(O(n))$, menghasilkan kompleksitas $O(n^2)$ untuk proses eliminasi. Total kompleksitas untuk seluruh proses *forward elimination* adalah

$$O(n \times (n + n^2)) = O(n^3).$$

Tahap final dilakukan dengan memanfaatkan *list comprehension* untuk mengekstraksi elemen-elemen yang merupakan solusi dari setiap baris hasil eliminasi, yang memiliki kompleksitas O(n). Dengan menjumlahkan

kompleksitas setiap tahap, total kompleksitas waktu untuk metode Gauss-Jordan.

$$T(n) = O(n) + O(n^3) + O(n) = O(n^3).$$

2. Metode Dekomposisi LU

a. Fungsi *lu_decomposition(A)*

```
def lu_decomposition(A):
    n = len(A)
    L = [[0.0] * n for i in range(n)]
    U = [[0.0] * n for i in range(n)]
    for i in range(n):
        for k in range(i, n): # Upper Triangular
            sum = 0
            for j in range(i):
                sum += (L[i][j] * U[j][k])
            U[i][k] = A[i][k] - sum
        for k in range(i, n): # Lower Triangular
            if i == k:
                L[i][i] = 1  # Diagonal = 1
            else:
                sum = 0
                for j in range(i):
                    sum += (L[k][j] * U[j][i])
                L[k][i] = (A[k][i] - sum) / U[i][i]
    return L, U
```

Gambar 5. Fungsi lu_decomposition

(Sumber: Dokumen Penulis)

Fungsi *lu_decomposition(A)* melakukan dekomposisi matriks A menjadi matriks segitiga bawah (L) dan segitiga atas (U). Fungsi ini diawali dengan inisialisasi matriks L dan U berukuran $n \times n$ dengan nilai 0, yang memiliki kompleksitas $O(n^2)$. Proses dekomposisi dilakukan melalui tiga nested *loop*: *loop* terluar i yang berjalan sebanyak n kali, *loop* tengah k untuk membentuk matriks U yang berjalan dari i hingga n, dan *loop* terdalam j untuk menghitung *sum* yang berjalan dari 0 hingga i. Proses yang sama diulang untuk membentuk matriks L. Total kompleksitas untuk fungsi ini adalah $O(n^3)$ karena melibatkan tiga nested *loop* yang masing-masing berjalan sebanyak $O(n)$ kali.

b. Fungsi *forward_substitution(L, b)*

```
def forward_substitution(L, b):
    """Menyelesaikan Ly = b"""
    n = len(L)
    y = [0] * n
    for i in range(n):
        sum = 0
        for j in range(i):
            sum += L[i][j] * y[j]
        y[i] = (b[i] - sum) / L[i][i]
    return y
```

Gambar 6. Fungsi *forward_substitution*

(Sumber: Dokumen Penulis)

Fungsi *forward_substitution(L, b)* menyelesaikan sistem persamaan $Ly = b$ menggunakan substitusi maju. Fungsi ini memiliki dua nested *loop*: *loop* luar i yang berjalan dari 0 hingga $n-1$, dan *loop* dalam j yang berjalan dari 0 hingga $i-1$ untuk menghitung sum. Setiap iterasi melakukan operasi perkalian dan penjumlahan yang memiliki kompleksitas $O(1)$. Total kompleksitas untuk fungsi ini adalah $O(n^2)$.

c. Fungsi *backward_substitution(U, y)*

```
def backward_substitution(U, y):
    """Menyelesaikan Ux = y"""
    n = len(U)
    x = [0] * n
    for i in range(n-1, -1, -1):
        sum = 0
        for j in range(i+1, n):
            sum += U[i][j] * x[j]
        x[i] = (y[i] - sum) / U[i][i]
    return x
```

Gambar 7. Fungsi *backward_substitution*

(Sumber: Dokumen Penulis)

Fungsi *backward_substitution*(U, y) menyelesaikan sistem persamaan $Ux = y$ menggunakan substitusi mundur. Strukturnya mirip dengan *forward_substitution* namun *loop* luar i berjalan dari $n-1$ hingga 0, dan *loop* dalam j berjalan dari $i+1$ hingga $n-1$. Kompleksitas waktu fungsi ini juga $O(n^2)$.

d. Fungsi *solve_by_lu(A, b)*

```
def solve_by_lu(A, b):
    L, U = lu_decomposition(A)
    y = forward_substitution(L, b)
    x = backward_substitution(U, y)
    return x
```

Gambar 8. Fungsi *backward_substitution*

(Sumber: Dokumen Penulis)

Fungsi utama *solve_by_lu(A, b)* mengkoordinasikan ketiga fungsi sebelumnya secara berurutan. Kompleksitas total dapat dihitung dengan menjumlahkan kompleksitas setiap fungsi:

$$T(n) = O(n^3) + O(n^2) + O(n^2) = O(n^3).$$

3. Metode Invers

a. Fungsi *get_matrix_minor(matrix, i, j)*

```
def get_matrix_minor(matrix, i, j):
    return [row[:j] + row[j+1:]
            for row in
            (matrix[:i] + matrix[i+1:])]
```

Gambar 9. Fungsi *backward_substitution*

(Sumber: Dokumen Penulis)

Fungsi *get_matrix_minor*(matrix, i, j) menghasilkan minor dari matriks dengan menghapus baris ke-i dan kolom ke-j. Fungsi ini menggunakan *list comprehension* untuk membuat matriks baru, yang memerlukan iterasi melalui semua baris dalam matriks selain baris ke-i, dan dari tiap baris itu mengambil semua angka kecuali angka di kolom ke-j. Kompleksitas waktu fungsi ini adalah $O(n^2)$ karena harus mengakses setiap elemen dalam matriks berukuran $n \times n$.

b. Fungsi *matrix_determinant(matrix)*

```
def matrix_determinant(matrix):
    if len(matrix) == 1:
        return matrix[0][0]
    if len(matrix) == 2:
        return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]

    determinant = 0
    for j in range(len(matrix)):
        minor = get_matrix_minor(matrix, 0, j)
        cofactor = ((-1) ** j) * matrix[0][j]
        determinant += cofactor * matrix_determinant(minor)
    return determinant
```

Gambar 10. Fungsi *matrix_determinant*

(Sumber: Dokumen Penulis)

Fungsi *matrix_determinant*(matrix) menghitung determinan matriks menggunakan ekspansi kofaktor. Untuk matriks 1×1 dan 2×2, perhitungan dilakukan secara langsung dengan kompleksitas $O(1)$. Untuk matriks berukuran lebih besar, fungsi menggunakan rekursi dengan ekspansi kofaktor baris pertama. Untuk setiap elemen j pada baris pertama, fungsi memanggil dirinya sendiri untuk menghitung determinan minor yang berukuran $(n-1) \times (n-1)$. Proses rekursif ini menghasilkan kompleksitas $O(n!)$ karena untuk matriks $n \times n$, diperlukan n kali pemanggilan rekursif untuk matriks $(n-1) \times (n-1)$, sehingga total operasi menjadi $n \times (n-1)! = n!$.

c. Fungsi *matrix_inverse(matrix)*

```
def matrix_inverse(matrix):
    n = len(matrix)
    determinant = matrix_determinant(matrix)
    if n == 1:
        return [[1/determinant]]
    cofactors = []
    for i in range(n):
        cofactor_row = []
        for j in range(n):
            minor = get_matrix_minor(matrix, i, j)
            cofactor_row.append(((-1) ** (i+j)) * matrix_determinant(minor))
        cofactors.append(cofactor_row)
    adjoint = list(map(list, zip(*cofactors)))
    inverse = [[adjoint[i][j]/determinant for j in range(n)] for i in range(n)
    return inverse
```

Gambar 11. Fungsi *matrix_inverse*

(Sumber: Dokumen Penulis)

Fungsi *matrix_inverse*(matrix) menghitung invers matriks menggunakan metode adjoin. Fungsi ini pertama menghitung determinan matriks ($O(n!)$), kemudian menghitung matriks kofaktor melalui dua *nested loop* dengan kompleksitas $O(n^2)$, di mana setiap iterasi memerlukan perhitungan determinan minor (O(n!)). Setelah itu, dilakukan transpose matriks kofaktor dan pembagian dengan determinan. Total kompleksitas waktu fungsi ini adalah $O(n! \times n^2)$.

d. Fungsi *solve_by_inverse(A, b)*

```
def solve_by_inverse(A, b):
    A_inv = matrix_inverse(A)
    n = len(A)
    x = []
    for i in range(n):
        x_i = sum(A_inv[i][j] * b[j] for j in range(n))
        x.append(x_i)
    return x
```

Gambar 12. Fungsi *solve_by_inverse*

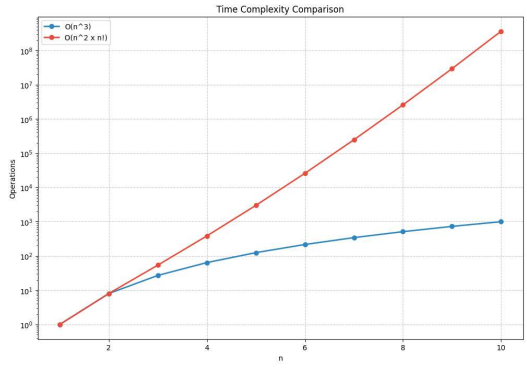
(Sumber: Dokumen Penulis)

Fungsi *solve_by_inverse*(A, b) menyelesaikan SPL dengan mengalikan invers matriks A dengan vektor b. Fungsi ini memanggil *matrix_inverse* dan melakukan perkalian matriks-vektor. Kompleksitas total fungsi ini adalah $O(n! \times n^2)$ yang didominasi oleh kompleksitas perhitungan invers matriks.

4. Perbandingan Kompleksitas Waktu

Metode	Kompleksitas Waktu	Keterangan
Inverse Matriks	$O(n^2 \times n!)$	Paling tidak efisien untuk n besar ketika menggunakan cara adjoint untuk menentukan balikan matriksnya
Eliminasi Gauss-Jordan	$O(n^3)$	Efisien untuk semua ukuran n
Dekomposisi LU	$O(n^3)$	Efisien dan memungkinkan penyelesaian *multiple* b (Jika matriks b berbeda, tetapi matriks A tetap sama)

Tabel 1. Perbandingan Kompleksitas Waktu



Gambar 13. Grafik Kompleksitas Waktu

(Sumber: Dokumen Penulis)

IV. KESIMPULAN

Berdasarkan analisis kompleksitas waktu yang telah dilakukan terhadap tiga metode penyelesaian Sistem Persamaan Linear (SPL), dapat disimpulkan bahwa metode eliminasi Gauss-Jordan dan metode dekomposisi LU memiliki kompleksitas waktu yang sama, yaitu $O(n^3)$, sementara metode inverse matriks memiliki kompleksitas waktu $O(n^2 \times n!)$. Metode inverse matriks dengan pendekatan adjoint terbukti menjadi cara yang paling tidak efisien, khususnya saat ukuran matriks (n) bertambah besar, karena kompleksitas waktunya meningkat secara faktorial. Di sisi lain, meskipun metode eliminasi Gauss-Jordan dan dekomposisi LU memiliki kompleksitas waktu yang sama, metode dekomposisi LU memiliki keunggulan tambahan, yaitu kemampuannya untuk menyelesaikan *multiple* sistem persamaan linear dengan matriks koefisien yang sama tanpa perlu mengulang proses dekomposisi. Dengan demikian, untuk penyelesaian SPL secara umum, metode eliminasi Gauss-Jordan dan dekomposisi LU lebih direkomendasikan dibandingkan dengan metode inverse matriks, dengan dekomposisi LU menjadi pilihan yang lebih baik ketika diperlukan penyelesaian multiple sistem persamaan.

V. UCAPAN TERIMA KASIH

Puji Syukur saya haturkan kepada Tuhan Yang Maha Esa atas segala berkat dan anugerah-Nya, yang memungkinkan penulis untuk menyelesaikan makalah ini dengan baik. Tak lupa Penulis juga mengucapkan terima kasih kepada

1. Bapak Dr. Ir. Rinaldi Munir, M.T. selaku dosen Aljabar Linier dan Geometri kelas 02 untuk bimbingannya, dan untuk website-nya yang memudahkan proses pencarian informasi mengenai topik yang dibahas dalam makalah ini.
2. Teman teman dan orang tua penulis yang telah memberikan dukungan selama pembuatan makalah ini.

DAFTAR PUSTAKA

[1] Munir, Rinaldi, 2023. "Kompleksitas Algoritma (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/25-Kompleksitas-Algoritma-Bagian2-2023.pdf diakses pada 28 desember pada pukul 09.00

[2] D. C. Lay, S. R. Lay, and J. Mcdonald, Linear algebra and its applications. Boston: Pearson, 2016.

[3] Munir, Rinaldi, 2023. "Dekomposisi LU". https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-23-Dekomposisi-LU-2023.pdf diakses pada 28 desember pada pukul 10.00

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 Desember 2024