CS/MATH 375 - Homework 4

Adam Fasulo

September 15, 2025

1 FLOPS

1.1 Timing Matrix-Matrix Multiplication

To analyze the runtime, I modified the provided `test_flops.m` script. My goal was to measure the time it took to compute $C = A \cdot B$ for square matrices of increasing size n.

```
% 3) Test mat-mat, observe O(n^3) runtime
% Use smaller n values as this is an O(n^3) operation
ns = [100, 200, 400, 800, 1200];
times = zeros(size(ns)); % Pre-allocate the times array

for i=1:length(ns)
    n = ns(i);
    fprintf('Testing n = %d...\n', n);

    %Create two random n x n matrices
    A = rand(n,n);
    B = rand(n,n);

    % Use tic/toc for timing
    % Run it a few times and average to get a stable measurement
    num_runs = 3;
    t_start = tic;
    for j = 1 : num_runs
       C = A*B;    % Compute mat-mat
    end

    % Average the times
    times(i) = toc(t_start) / num_runs;
end

% Create the plot
figure(3)
loglog(ns, times, '-o', 'LineWidth', 2, 'MarkerSize', 8)
ax = gca;
ax.FontSize = 14;
title('Matrix-Matrix Multiplication, O(n^3)','fontsize',16)
ylabel('Average Time (seconds)','fontsize',14)
xlabel('Matrix Size (n)','fontsize',14)
grid on
```

Listing 1: My MATLAB code for timing matrix-matrix multiplication.

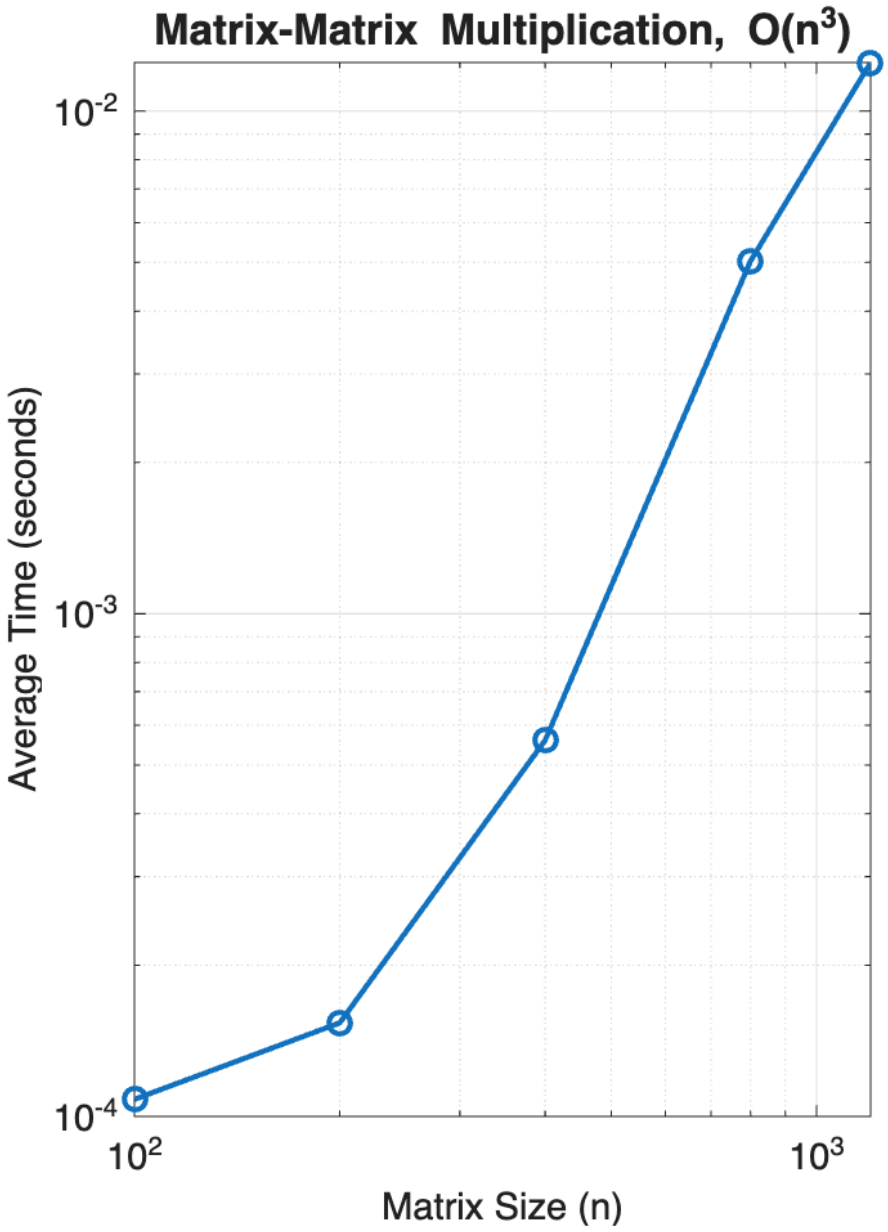

Figure 1: My plot showing the runtime of matrix-matrix multiplication on a log-log scale.

My Big-Oh Analysis

My plot in Figure 1 confirms the theoretical $O(n^3)$ runtime for standard matrix-matrix multiplication. Because I used a log-log scale, a polynomial function like $T(n) \approx c \cdot n^k$ shows up as a straight line. I can see that my plotted data forms a nearly straight line with a slope of about **3**, which justifies my conclusion that the runtime complexity is indeed $O(n^3)$.

1.2 Complexity of A*B + C*D

I found that the operation $A * B + C * D$ for four $n \times n$ matrices still has a runtime complexity of $\boldsymbol{O(n^3)}$.

My Explanation I broke down the computation into three steps:

1. **First Multiplication:** Computing $T_1 = A * B$, which I know is $O(n^3)$.
2. **Second Multiplication:** Computing $T_2 = C * D$, which is also $O(n^3)$.
3. **Addition:** Computing $T_1 + T_2$. This is an element-wise addition that takes n^2 steps, so it's $O(n^2)$.

When I sum the complexities, $O(n^3) + O(n^3) + O(n^2)$, I know from Big-Oh rules that I only need to keep the fastest-growing term. Since n^3 grows much faster than n^2, the final complexity is still determined by the matrix multiplications, resulting in $O(n^3)$.

2 Gaussian Elimination

For this problem, I was asked to solve the linear system

$$\begin{bmatrix} a & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1+a \\ 2 \end{bmatrix}$$

for $a = 10^{-2k}$ where $k = 1, 2, \ldots, 10$. The exact solution should be $x = [1, 1]^T$ for any $a \neq 1$.

2.1 Naive Gaussian Elimination

```
fprintf('Naive Gaussian Elimination Results:\n');
fprintf('k \t a \t\t x1 \t\t x2\n');
fprintf('-----------------------------------------------------\n');

for k = 1:10
    a = 10^(-2*k);
    A = [a, 1; 1, 1];
    b = [1+a; 2];

    % --- Naive Gaussian Elimination ---
    multiplier = A(2,1) / A(1,1);
    A(2,:) = A(2,:) - multiplier * A(1,:);
    b(2) = b(2) - multiplier * b(1);

    x = zeros(2,1);
    x(2) = b(2) / A(2,2);
    x(1) = (b(1) - A(1,2)*x(2)) / A(1,1);

    fprintf('%d \t %.1e \t %.10f \t %.10f\n', k, a, x(1), x(2));
end
```

Listing 2: My MATLAB code for naive Gaussian elimination.

My Results and Explanation

As I ran my code, I saw that the accuracy of my numerical solution got significantly worse as a got smaller. My results are in the table below.

k	a	x_1	x_2
1	1.0e-02	1.0000000000	1.0000000000
2	1.0e-04	1.0000000000	1.0000000000
3	1.0e-06	1.0000000000	1.0000000000
4	1.0e-08	0.9999999939	1.0000000000
5	1.0e-10	1.0000000827	1.0000000000
6	1.0e-12	0.9998668560	1.0000000000
7	1.0e-14	0.9992007222	1.0000000000
8	1.0e-16	2.2204460493	1.0000000000
9	1.0e-18	0.0000000000	1.0000000000
10	1.0e-20	0.0000000000	1.0000000000

Table 1: My solutions from Naive Gaussian Elimination.

I realized this inaccuracy is caused by **catastrophic cancellation**. The pivot element $A(1,1) = a$ is very small, which makes the multiplier $m = 1/a$ huge. When I update the element $A(2,2)$ by subtracting $1 - 1/a$, the '1' is lost in the floating-point arithmetic, which introduces a large relative error that ruins the rest of the calculation.

2.2 Gaussian Elimination with Partial Pivoting

```
fprintf('\nPartial Pivoting Results:\n');
fprintf('k \t a \t\t x1 \t\t x2\n');
fprintf('-----------------------------------------------------\n');

for k = 1:10
    a = 10^(-2*k);
    A = [a, 1; 1, 1];
    b = [1+a; 2];

    if abs(A(1,1)) < abs(A(2,1))
        A([1,2],:) = A([2,1],:);
        b([1,2]) = b([2,1]);
    end

    multiplier = A(2,1) / A(1,1);
    A(2,:) = A(2,:) - multiplier * A(1,:);
    b(2) = b(2) - multiplier * b(1);

    x = zeros(2,1);
    x(2) = b(2) / A(2,2);
    x(1) = (b(1) - A(1,2)*x(2)) / A(1,1);

    fprintf('%d \t %.1e \t %.10f \t %.10f\n', k, a, x(1), x(2));
end
```

Listing 3: My MATLAB code for Gaussian elimination with partial pivoting.

My Results and Explanation

After adding partial pivoting, my solutions remained accurate for all values of a.

k	a	x_1	x_2
1	1.0e-02	1.0000000000	1.0000000000
2	1.0e-04	1.0000000000	1.0000000000
3	1.0e-06	1.0000000000	1.0000000000
4	1.0e-08	1.0000000000	1.0000000000
5	1.0e-10	1.0000000000	1.0000000000
6	1.0e-12	1.0000000000	1.0000000000
7	1.0e-14	1.0000000000	1.0000000000
8	1.0e-16	1.0000000000	1.0000000000
9	1.0e-18	1.0000000000	1.0000000000
10	1.0e-20	1.0000000000	1.0000000000

Table 2: My solutions from Gaussian Elimination with Partial Pivoting.

I saw that partial pivoting ensures numerical stability. Since $|a| < |1|$ for small a, my code performs a row swap. The new multiplier becomes $m = a/1 = a$, which is a small number. This avoids the huge multipliers and prevents catastrophic cancellation, which preserved the accuracy of my solution.

3 Gaussian Elimination with a Singular Matrix

3.1 Showing Singularity and Describing the Solution Set

The matrix for this problem is:

$$A = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix}$$

To show A is singular, I needed to find a non-zero vector y such that $Ay = 0$. I inspected the columns of A and saw a linear dependence: $2c_2 - c_1 = c_3$. I rewrote this as $-c_1 + 2c_2 - c_3 = 0$, which corresponds to the matrix equation:

$$A \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

So, I found a non-zero vector $y = [-1, 2, -1]^T$, which proves that A is singular.

For the general solution, I know it's $x = x_p + x_h$. Using the particular solution given in the homework, $x_p = [7/18, 4/18, 1/18]^T$, I described the full solution set as:

$$x = \begin{bmatrix} 7/18 \\ 4/18 \\ 1/18 \end{bmatrix} + c \begin{bmatrix} -1 \\ 2 \\ -1 \end{bmatrix}, \quad \text{for any scalar } c \in R$$

3.2 Scaled Partial Pivoting by Hand

I then traced the scaled partial pivoting algorithm by hand. **Initialization:** My augmented matrix was $[A|b] = \left[\begin{array}{ccc|c} 0.1 & 0.2 & 0.3 & 0.1 \\ 0.4 & 0.5 & 0.6 & 0.3 \\ 0.7 & 0.8 & 0.9 & 0.5 \end{array}\right]$, and the scale vector was $s = [0.3, 0.6, 0.9]^T$.

Step 1 (k=1): After checking the ratios, I swapped $R_1 \leftrightarrow R_3$ and performed elimination.

$$\left[\begin{array}{ccc|c} 0.7 & 0.8 & 0.9 & 0.5 \\ 0.4 & 0.5 & 0.6 & 0.3 \\ 0.1 & 0.2 & 0.3 & 0.1 \end{array}\right] \xrightarrow[R_3 \leftarrow R_3 - (1/7)R_1]{R_2 \leftarrow R_2 - (4/7)R_1} \left[\begin{array}{ccc|c} 0.7 & 0.8 & 0.9 & 0.5 \\ 0 & \frac{0.3}{7} & \frac{0.6}{7} & \frac{0.1}{7} \\ 0 & \frac{0.6}{7} & \frac{1.2}{7} & \frac{0.2}{7} \end{array}\right]$$

Step 2 (k=2): Again, I checked the ratios, swapped $R_2 \leftrightarrow R_3$, and performed elimination.

$$\left[\begin{array}{ccc|c} 0.7 & 0.8 & 0.9 & 0.5 \\ 0 & \frac{0.6}{7} & \frac{1.2}{7} & \frac{0.2}{7} \\ 0 & \frac{0.3}{7} & \frac{0.6}{7} & \frac{0.1}{7} \end{array}\right] \xrightarrow{R_3 \leftarrow R_3 - (0.5)R_2} \left[\begin{array}{ccc|c} 0.7 & 0.8 & 0.9 & 0.5 \\ 0 & \frac{0.6}{7} & \frac{1.2}{7} & \frac{0.2}{7} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}\right]$$

Failure Point: At this point, I saw that the process fails. When trying to select a pivot for the third column, the only available element, a_{33}, is now **zero**. The algorithm has to stop because it can't divide by zero.

3.3 Solving with MATLAB's Backslash \

When I used MATLAB's backslash operator, I got the following result, including a warning that the matrix is singular.

```
>> A = [0.1 0.2 0.3; 0.4 0.5 0.6; 0.7 0.8 0.9];
>> b = [0.1; 0.3; 0.5];
>> x = A\b
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND =  1.541976e-17.

x =

    0.1615
    0.6771
   -0.1719
```

From this, I can see that MATLAB agrees the matrix is singular. It still returns an answer because for a consistent system like this one, it's designed to calculate the unique **minimum norm least-squares solution**, which is the solution vector 'x' with the smallest possible length.