Condition number

## Function : Reading a Matrix from a File in C

**Purpose:** This function reads a square matrix of size n x n from a file and stores it in a 2D array (matrix). It is used to load matrix data from external files for computational tasks.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Opening the File**:
   * FILE \*file = fopen(filename, "r");
   * Opens the file specified by filename in read mode.
2. **Error Handling**:
   * if (!file) { perror("Error opening file"); exit(EXIT\_FAILURE); }
   * Checks if the file was successfully opened. If not, prints an error message and exits the program.
3. **Reading Matrix Elements**:
   * The nested for loops iterate over each row (i) and column (j) of the matrix:
     + fscanf(file, "%lf", &matrix[i][j]);
     + Reads a floating-point number (%lf) from the file and stores it in the corresponding position in the matrix.
4. **Closing the File**:
   * fclose(file);
   * Closes the file after all elements are read to release system resources.

## **Function: Calculating the Matrix Norm**

**Purpose:** This function calculates the **maximum row sum norm** (also known as the infinity norm) of a square matrix of size n x n.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * double norm = 0.0;
     + Initializes norm to store the maximum row sum.
2. **Outer Loop**:
   * for (int i = 0; i < n; i++) {
     + Iterates over each row (i) of the matrix.
3. **Row Sum Calculation**:
   * double row\_sum = 0.0;
     + Initializes row\_sum to accumulate the absolute values of elements in the current row.
4. **Inner Loop**:
   * for (int j = 0; j < n; j++) {
     + Iterates over each column (j) in the current row.
5. **Absolute Value Addition**:
   * row\_sum += fabs(matrix[i][j]);
     + Adds the absolute value of each matrix element to row\_sum.
6. **Updating Norm**:
   * if (row\_sum > norm) { norm = row\_sum; }
     + Compares row\_sum with norm and updates norm if row\_sum is larger.
7. **Return Statement**:
   * return norm;
     + Returns the maximum row sum as the calculated matrix norm.

## Recursive Determinant Calculation

**Purpose:** This function calculates the determinant of a square matrix of size n×n using the cofactor expansion method.

A screenshot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

**1. Base Cases**

* The function handles two base cases for efficiency:
  + **1×1 Matrix**: The determinant is simply the single element (matrix).
  + **2×2 Matrix**: Uses the direct formula: det([[a,b],[c,d]]) = ad - bc.

**2. Cofactor Expansion**

For matrices larger than 2×2, the function uses the Laplace expansion along the first row:

**double** det = 0.0;

**for** (**int** p = 0; p < n; p++) {

}

* The variable p iterates through each column of the first row.
* The function will sum up the contributions of each element to the determinant.

**3. Submatrix Construction**

For each element in the first row, we need a submatrix that excludes the first row and the column containing that element:]

**double** \*\*submatrix = (**double** \*\*)malloc((n - 1) \* **sizeof**(**double** \*));

**for** (**int** i = 0; i < n - 1; i++) {

submatrix[i] = (**double** \*)malloc((n - 1) \* **sizeof**(**double**));

}

**for** (**int** i = 1; i < n; i++) {

**int** col\_index = 0;

**for** (**int** j = 0; j < n; j++) {

**if** (j != p) {

submatrix[i - 1][col\_index++] = matrix[i][j];

}

}

}

* Allocates memory for a submatrix of size (n-1)×(n-1).
* The outer loop starts from row 1 (skipping row 0, which is the first row we're expanding along).
* For each row, we copy elements from the original matrix, skipping the p-th column.
* col\_index tracks the position in the submatrix where we're copying elements.

**4. Recursive Determinant Calculation**

det += (p % 2 == 0 ? 1 : -1) \* matrix[0][p] \* determinant(submatrix, n - 1);

* This implements the cofactor formula: det(A) =.
* Since we're expanding along the first row (i=0), the sign factor is (-1)^(0+j) or simply (-1)^j.
* (p % 2 == 0 ? 1 : -1) computes (-1)^p, which alternates between 1 and -1.
* The function recursively calls itself to calculate the determinant of the submatrix.

**5. Memory Management**

**for** (**int** i = 0; i < n - 1; i++) {

free(submatrix[i]);

}

free(submatrix);

* Properly frees all dynamically allocated memory to prevent memory leaks.
* Each row of the submatrix is freed individually, then the array of pointers is freed.

## **Adjoint Matrix Calculation**

**Purpose:** This function calculates the adjoint of a square matrix of size n×n.

A screenshot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

**1. Base Case Handling**

**if** (n == 1) {

adj[0][0] = 1.0;

**return**;

}

* For a 1×1 matrix, the adjoint is simply 1.0
* This is a special case that aligns with the mathematical definition where the adjoint of a 1×1 matrix [a] is1

**2. Iterating Through Matrix Elements**

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < n; j++) {

}

}

* The nested loops iterate through each position (i,j) in the original matrix
* For each position, we'll calculate the corresponding cofactor

**3. Submatrix Construction**

**double** \*\*submatrix = (**double** \*\*)malloc((n - 1) \* **sizeof**(**double** \*));

**for** (**int** k = 0; k < n - 1; k++) {

submatrix[k] = (**double** \*)malloc((n - 1) \* **sizeof**(**double**));

}

**int** row\_index = 0;

**for** (**int** row = 0; row < n; row++) {

**if** (row == i) **continue**;

**int** col\_index = 0;

**for** (**int** col = 0; col < n; col++) {

**if** (col == j) **continue**;

submatrix[row\_index][col\_index++] = matrix[row][col];

}

row\_index++;

}

* Allocates memory for a submatrix of size (n-1)×(n-1)
* Creates the minor matrix M(i,j) by excluding the i-th row and j-th column from the original matrix
* The row\_index and col\_index variables track positions in the submatrix
* continue statements skip the i-th row and j-th column of the original matrix

**4. Cofactor Calculation and Assignment**

adj[j][i] = ((i + j) % 2 == 0 ? 1 : -1) \* determinant(submatrix, n - 1);

* Calculates the determinant of the submatrix using the previously defined determinant () function
* Applies the cofactor sign pattern: +1 if (i+j) is even, -1 if (i+j) is odd
* **Critical detail**: Assigns to adj[j][i] not adj[i][j] - this performs the transpose operation, as the adjoint is the transpose of the cofactor matrix

**5. Memory Cleanup**

**for** (**int** k = 0; k < n - 1; k++) {

free(submatrix[k]);

}

free(submatrix);

* Properly frees the dynamically allocated memory for each submatrix
* Prevents memory leaks by first freeing each row, then the array of pointers

## **Matrix Inversion Using the Classical Adjoint Method**

**Purpose:** This function computes the inverse of a square matrix using the classical formula: **A⁻¹ = adj(A)/det(A)**.



**Step-by-Step Explanation**

**1. Determinant Calculation and Singularity Check**

**double** det = determinant(matrix, n);

**if** (fabs(det) < 1e-9) {

fprintf(stderr, "Matrix is singular or nearly singular.\n");

exit(EXIT\_FAILURE);

}

* Calls the determinant() function to compute the determinant of the input matrix
* Checks if the absolute value of the determinant is less than 10⁻⁹ (effectively zero)
* If the determinant is near zero, the matrix is singular (non-invertible), so the function prints an error message and terminates the program
* This check is critical as divisions by zero (or very small numbers) would lead to numerical instability

**2. Memory Allocation for Adjoint Matrix**

**double** \*\*adj = (**double** \*\*)malloc(n \* **sizeof**(**double** \*));

**for** (**int** i = 0; i < n; i++) {

adj[i] = (**double** \*)malloc(n \* **sizeof**(**double**));

}

* Allocates memory for an n×n matrix to store the adjoint
* Uses a two-dimensional dynamic array structure with row pointers

**3. Adjoint Calculation**

adjoint(matrix, adj, n);

* Calls the adjoint() function to calculate the adjoint matrix
* The adjoint is the transpose of the cofactor matrix of the original matrix

**4. Inverse Calculation**

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < n; j++) {

inverse[i][j] = adj[i][j] / det;

}

}

* Calculates each element of the inverse matrix using the formula: inverse[i][j] = adj[i][j] / det
* Implements the mathematical relationship: A⁻¹ = adj(A)/det(A)
* This nested loop performs n² operations (one division per matrix element)

**5. Memory Cleanup**

**for** (**int** i = 0; i < n; i++) {

free(adj[i]);

}

free(adj);

* Frees all memory allocated for the adjoint matrix
* Prevents memory leaks by first freeing each row, then the array of row pointers

## FUNCTION : MAIN PROGRAM FOR MATRIX CONDITION NUMBER CALCULATION

**Purpose:** This function serves as the main entry point for a program that reads a matrix from a file, calculates its condition, and displays the result. It handles memory allocation, function calls, and memory cleanup.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation:**

1. **User Input**:
   * The program prompts the user to enter the size of the square matrix (n).
   * scanf("%d", &n); reads the value entered by the user.
2. **Memory Allocation**:
   * The program allocates memory for two n×n matrices: the original matrix and its inverse.
   * double \*\*matrix = (double \*\*)malloc(n \* sizeof(double \*)); allocates an array of n pointers.
   * The nested loop allocates memory for each row of both matrices.
3. **Matrix Reading**:
   * read\_matrix("inputs.txt", matrix, n); calls the previously defined function to read matrix data from a file.
4. **Matrix Norm Calculation**:
   * double norm = calculate\_norm(matrix, n); calculates the norm of the original matrix.
5. **Matrix Inversion**:
   * invert\_matrix(matrix, inverse, n); inverts the matrix and stores the result in the inverse array.
6. **Condition Number Calculation**:
   * double inverse\_norm = calculate\_norm(inverse, n); calculates the norm of the inverse matrix.
   * double condition\_number = norm \* inverse\_norm; computes the condition number as the product of the two norms.
   * The condition number is a measure of how sensitive a matrix is to errors in input data or computational operations.
7. **Result Output**:
   * The program prints the calculated condition number using printf.
8. **Memory Cleanup**:
   * The nested loop frees memory for each row of both matrices.
   * The final two free() calls release the memory for the arrays of pointers.
9. **Program Termination**:
   * return 0; indicates successful program execution.