Question 2 report

by tanish verma

roll NO: co24btech11023

Table of Contents

[Question 2 4](#_Toc195868661)

[Matrix and Vector 4](#_Toc195868662)

[Subquestions 4](#_Toc195868663)

[Condition Number of a matrix 5](#_Toc195868664)

[Defination 5](#_Toc195868665)

[Well-Conditioned vs. Ill-Conditioned Matrices 5](#_Toc195868666)

[Algorithm 5](#_Toc195868667)

[Result 5](#_Toc195868668)

[Estimation of Q matrix using Grahm Schmidt 6](#_Toc195868669)

[Algorithm 6](#_Toc195868670)

[For Q matrix 6](#_Toc195868671)

[For Solutions 6](#_Toc195868672)

[Result 6](#_Toc195868673)

[Householder Transformation and Solutions 7](#_Toc195868674)

[Algorithm 7](#_Toc195868675)

[For Q and R matrix 7](#_Toc195868676)

[For Solutions 9](#_Toc195868677)

[Results 9](#_Toc195868678)

[Code Reports 10](#_Toc195868679)

[Condition Number 10](#_Toc195868680)

[Function: Reading a Matrix from a File in C 10](#_Toc195868681)

[Function: Calculating the Matrix Norm 11](#_Toc195868682)

[Recursive Determinant Calculation 13](#_Toc195868683)

[Adjoint Matrix Calculation 16](#_Toc195868684)

[Matrix Inversion Using the Classical Adjoint Method 18](#_Toc195868685)

[Function: MAIN PROGRAM FOR MATRIX CONDITION NUMBER CALCULATION 21](#_Toc195868686)

[Gram schmidt orthogonalisation 23](#_Toc195868687)

[Function: GRAM-SCHMIDT ORTHOGONALIZATION 23](#_Toc195868688)

[Function: Backward Substitution for Solving Upper Triangular Systems 25](#_Toc195868689)

[Function: Matrix Transposition 26](#_Toc195868690)

[Function: Matrix Multiplication for Square Matrices 27](#_Toc195868691)

[Function: Matrix-Vector Multiplication 28](#_Toc195868692)

[Function: QR Factorization for Solving Linear Systems 29](#_Toc195868693)

[Function: Reading Matrix and Vector Input from a File 30](#_Toc195868694)

[Function: Main Program for Solving Linear Systems via QR Factorization 32](#_Toc195868695)

[HouseHolder Transformation 35](#_Toc195868696)

[Function: Euclidean Norm Calculation for Vectors 35](#_Toc195868697)

[Function: Matrix Multiplication for Square Matrices 35](#_Toc195868698)

[Function: Matrix Transposition 36](#_Toc195868699)

[Function: QR Factorization Using Householder Reflections 37](#_Toc195868700)

[Function: Solving Linear Systems Using QR Decomposition 42](#_Toc195868701)

[Function: Reading Matrix and Vector Input from a File 43](#_Toc195868702)

[Main Program for Solving Linear Systems via QR Factorization 44](#_Toc195868703)

# Question 2

## Matrix and Vector

Given Matrix A = and vector b =

## Subquestions

A white background with black text

AI-generated content may be incorrect.

# Condition Number of a matrix

## Defination

The condition number of a matrix is a fundamental concept in numerical linear algebra that quantifies how sensitive a matrix-based computation is to small changes or errors in the input data.

## Well-Conditioned vs. Ill-Conditioned Matrices

* **Well-conditioned matrix**: Has a low condition number, meaning small changes in input produce correspondingly small changes in output.
* **Ill-conditioned matrix**: Has a high condition number, indicating that small perturbations in input data can result in significantly large changes in the solution

## Algorithm

1. **Choose a matrix norm**
2. **Calculate the matrix inverse** A⁻¹ using
3. **Compute both norms**: ‖A‖ and ‖A⁻¹‖
4. **Multiply the norms**: κ(A) = ‖A‖ · ‖A⁻¹‖

## Result



Since the **Condition number** of the matrix is much greater than 1, hence the Matrix is ill – conditioned.

# Estimation of Q matrix using Grahm Schmidt

## Algorithm

### For Q matrix

Let the columns of matrix **A** be:

A=

We want to create orthonormal vectors:

Q=

Steps:

1. **Initialize**:  
   Set:
2. **Iterate for j = 2 to n**:
3. **Now use the to form the Q matrix**

### For Solutions

Steps:

1. **Calculate R (upper triangular) matrix using the relation**:
2. **Calculate b’** using the relation
3. Solve Rx = b’ using backward substitution.

## Result

A screenshot of a computer code

AI-generated content may be incorrect.

# Householder Transformation and Solutions

## Algorithm

### For Q and R matrix

Householder method of QR factorization uses a sequence of orthogonal matrices known as Householder matrices to transform the given matrix A to QR form. The idea of the method is to reduce the coefficient matrix to upper triangular form by multiplying the matrix by a series of Householder matrices.

As Householder matrices are orthogonal, the matrix HnHn−1 ...H1 is also orthogonal. The upper triangular matrix U thus obtained is R and thus

Consider a Householder matrix H1 defined as

H1HT1 = (I−2uuT )(I−2uuT )T = (IT −2(uuT )T )(I−2uuT ) = (I−2uuT )(I−2uuT ) = I−4uuT +4uuTuuT = I−4uuT +4u(uTu)uT = I−4uuT +4uuT = I

Let vector x represent the first column of the coefficient matrix. x T =

We desire that this vector be transformed, on pre-multiplication by H1, to xˆ1e1 where e1 is the unit vector defined as eT1 = [1 0 0··· 0 ]

and x1 is a non-zero number i.e. all the elements below the pivot element are zero. Thus, we should have H1x = x1e1

On pre-multiplication by (H1x)T Equation becomes (H1x)TH1x = xTHT1H1x = xTx = (x1e1)T x1e1 = x21

x1 = sqrt(xTx)

Therefore using the above relation for H1x = x1e1 we get

HT1H1x = x = HT1 x1e1 = x1HT1 e1 = x1(I−2uuT )T e1 = x1(e1 −2uuT e1)

x1 = x1(1−2u21 ); xi = x1(0−2u1ui) for i 1

A black and white image of a square root

AI-generated content may be incorrect.

hence we get for i=1

A math equation with black text

AI-generated content may be incorrect.

and for i 1 we have

Then we can calculate

A group of black and white letters

AI-generated content may be incorrect.

Multiplying A with H1  we get

A second Householder transformation is used to put zeros in all the elements in the second column below the pivot. For this consider the n −1× n −1 matrix obtained by removing the first row and column of the matrix shown on the right in Equation 2.86. We use the n−1×1 vector given by

xT = [ai2,2 ai3,2 ··· aii,2 ···ain,2 ]

We construct a Householder matrix given by H’2 = I − 2uuT where the elements of u are obtained such that the vector x is transformed to a unit vector x2e2 where e2 is n −1 ×1 unit vector. Note that I in the above is the n−1× n−1 identity matrix. It is clear that the analysis presented earlier can be used directly now to get the vector u. By adding the row and column that were removed earlier, we construct a matrix H2 = I − 2u’u’T where I is the n× n identity matrix and u’ is n×1 vector given by u’T = [0 u1 u2 ··· un−1]

Premultiplication of H1A by H2 should lead to

A group of black letters and numbers

AI-generated content may be incorrect.

The process above may be repeated with matrices obtained by removing two rows and two columns, three rows and three columns and so on till we are able to transform the coefficient matrix to upper triangle form R.

A number and numbers on a white background

AI-generated content may be incorrect.

It is also clear that Hn−1Hn−2 ...H2H1A = R and hence Q should be given by Q = (Hn−1Hn−2 ...H2H1)−1 = HT1HT2 ...HTn−1

### For Solutions

Steps:

1. **Calculate b’** using the relation
2. Solve Rx = b’ using backward substitution.

## Results

A screenshot of a computer program

AI-generated content may be incorrect.

# Code Reports

## 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)**.

A screen shot of a computer program

AI-generated content may be incorrect.

**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.

## Gram schmidt orthogonalisation

### Function: GRAM-SCHMIDT ORTHOGONALIZATION

**Purpose:** This function implements the Gram-Schmidt orthogonalization process, which transforms a set of linearly independent vectors (columns of matrix A) into an orthonormal basis (stored in matrix Q).

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation:**

1. **First Vector Initialization**:
   * The function copies the first column of matrix A directly to matrix Q.
   * Q[i] = A[i]; for all rows i from 0 to n-1.
2. **Orthogonalization Process**:
   * The outer loop for (int i = 1; i < n; i++) iterates through each column of A starting from the second column.
   * For each column i:
     + Initially copies column i from A to Q: Q[j][i] = A[j][i];
     + For each previously processed column k (0 to i-1):
       - Calculates the dot product between column k of Q and column i of A: vector\_product+=Q[l][k]\*A[l][i];
       - Calculates the squared norm of column k of Q: norm += Q[l][k] \* Q[l][k];
       - Subtracts the projection of column i onto column k: Q[j][i] -= (vector\_product\*Q[j][k]) / norm;
     + This makes column i orthogonal to all previous columns.
3. **Normalization**:
   * After all columns are orthogonalized, each column is normalized to have unit length.
   * For each column i:
     + Calculates the Euclidean norm: norm = sqrt(norm);
     + Divides each element by the norm: Q[k][i] /= norm;
     + This transforms orthogonal vectors into orthonormal vectors.
4. **Result Output**:
   * The function prints the resulting orthonormal matrix Q.
   * Each row is displayed in bracket notation with values formatted to 5 decimal places.

### Function: Backward Substitution for Solving Upper Triangular Systems

**Purpose:** This function solves a system of linear equations **Ux = b**, where **U** is an upper triangular matrix.

A computer screen shot of a program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Memory Allocation**:
   * double \*x = (double \*)malloc(n \* sizeof(double));  
     Allocates memory for the solution vector **x**.
   * Checks for allocation failure and exits if memory is unavailable.
2. **Initialize Last Element**:
   * x[n-1] = b[n-1] / A[n-1][n-1];  
     Solves for the last variable directly using the bottom row of the upper triangular matrix **U**.
3. **Backward Substitution Loop**:
   * for(int i=n-2; i>=0; i--)  
     Iterates from the second-last row to the first row.
   * double sum = 0;  
     Initializes a sum to accumulate contributions from already solved variables.
   * for(int j=i+1; j<n; j++)  
     Sums the products of coefficients (A[i][j]) and known solutions (x[j]).
   * x[i] = (b[i] - sum) / A[i][i];  
     Computes the current variable x[i] by subtracting the accumulated sum from b[i] and dividing by the diagonal element A[i][i].
4. **Return Result**:
   * Returns the solution vector **x**, which satisfies **Ux = b**.

### Function: Matrix Transposition

**Purpose:** This function transposes a **square matrix** of size n x n

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Outer Loop** (i from 0 to n-1):
   * Iterates over each row of the matrix.
2. **Inner Loop** (j from i+1 to n-1):
   * For each row i, iterates over columns starting from i+1 (avoids redundant swaps on the diagonal and below).
3. **Swap Elements**:
   * Swaps A[i][j] (element above the diagonal) with A[j][i] (element below the diagonal).
   * Example: For i=0 and j=1, A and A are swapped.
4. **Result**:
   * After all iterations, the original matrix A becomes its transpose.

### Function: Matrix Multiplication for Square Matrices

**Purpose:** This function multiplies two **square matrices** A and B of size n x n and stores the result in matrix C.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * The outer loop for (int i = 0; i < n; i++) iterates over each **row** of matrix A.
2. **Column Iteration**:
   * The middle loop for (int j = 0; j < n; j++) iterates over each **column** of matrix B.
3. **Dot Product Calculation**:
   * C[i][j] = 0; initializes the element at row i, column j of matrix C to zero.
   * The inner loop for (int k = 0; k < n; k++) computes the dot product of the i-th row of A and the j-th column of B:
     + C[i][j] += A[i][k] \* B[k][j]; accumulates the product of corresponding elements.
4. **Result Storage**:
   * After completing all iterations, matrix C contains the product of A and B.

### Function: Matrix-Vector Multiplication

**Purpose:** This function multiplies a **square matrix** A of size n x n with a **vector** B of size n and stores the result in vector C. It computes the dot product of each row of A with B to produce C.

A computer screen shot of a black screen with colorful text

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * The outer loop for (int i = 0; i < n; i++) iterates over each **row** of matrix A.
2. **Reset Result Vector**:
   * C[i] = 0; initializes the i-th element of the result vector C to zero.
3. **Dot Product Calculation**:
   * The inner loop for (int j = 0; j < n; j++) computes the dot product of the i-th row of A and the vector B:
     + C[i] += A[i][j] \* B[j]; accumulates the product of the matrix element A[i][j] and vector element B[j].
4. **Result Storage**:
   * After the inner loop completes, C[i] holds the dot product result for the i-th row.

### Function: QR Factorization for Solving Linear Systems

**Purpose:** This function solves a linear system **Ax = b** using QR factorization. It decomposes matrix **A** into **Q** (orthonormal) and **R** (upper triangular), then solves **Rx = Qᵀb** via backward substitution.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Memory Allocation**:
   * Allocates memory for matrices **Q** (orthonormal) and **R** (upper triangular).
2. **Gram-Schmidt Process**:
   * Gram\_schmidt(n, A, Q) transforms **A** into **Q** with orthonormal columns.
3. **Transpose Q**:
   * transpose(n, Q) converts **Q** from column-orthonormal to row-orthonormal (effectively computing **Qᵀ**).
4. **Compute R**:
   * matrix\_multiply\_nn(n, Q, A, R) calculates **R = Qᵀ \* A**.
5. **Transform b**:
   * matrix\_multiply\_n1(n, Q, b, b1) computes **b1 = Qᵀ \* b**.
6. **Solve for x**:
   * backward\_substitution(n, R, b1) solves **Rx = b1** to find the solution vector **x**.
7. **Memory Cleanup**:
   * Frees **Q**, **R**, and **b1** to prevent leaks.

### Function: Reading Matrix and Vector Input from a File

**Purpose:** This function reads a square matrix **A** of size n x n and a vector **b** of size n from a file named "inputs.txt". It ensures proper error handling for file operations and input parsing.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Open File**:
   * FILE \*file = fopen("inputs.txt", "r"); opens the file "inputs.txt" in read mode.
   * If the file cannot be opened (e.g., it doesn't exist), the program prints an error message ("File not found") and exits.
2. **Read Matrix Elements**:
   * The nested loop for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { ... } } iterates through each row (i) and column (j) of matrix **A**.
   * fscanf(file, "%lf", &A[i][j]) reads a floating-point number from the file and stores it in the corresponding position in **A**.
   * If the input is invalid or missing, the program prints "Invalid input" and exits.
3. **Read Vector Elements**:
   * The loop for (int i = 0; i < n; i++) { ... } iterates through each element of vector **b**.
   * fscanf(file, "%lf", &b[i]) reads a floating-point number from the file and stores it in the corresponding position in **b**.
   * Similar error handling ensures that invalid or missing input causes the program to terminate with an error message.
4. **Close File**:
   * fclose(file); closes the file after all data has been read to release system resources.

### Function: Main Program for Solving Linear Systems via QR Factorization

**Purpose:** This program solves a linear system **Ax = b** using QR factorization. It reads matrix **A** and vector **b** from a file, computes the solution **x**, prints the result, and measures the execution time of the QR factorization process.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **User Input**:
   * Prompts the user to enter the number of variables (n).
2. **Memory Allocation**:
   * Allocates memory for:
     + **A** (n x n matrix) using nested malloc calls.
     + **b** (vector of size n).
   * Checks for allocation failure in b and exits if memory is unavailable.
3. **Read Input File**:
   * Calls read\_file\_input(n, A, b) to populate **A** and **b** from "inputs.txt".
4. **QR Factorization & Timing**:
   * clock() records the start time.
   * QR\_factorization(n, A, b) computes the solution vector **x**.
   * clock() records the end time.
5. **Print Results**:
   * Displays the solution vector **x** with 6 decimal places.
   * Calculates and prints the execution time in nanoseconds.
6. **Memory Cleanup**:
   * Frees all dynamically allocated memory for **A**, **b**, and **x** to prevent leaks.

## HouseHolder Transformation

### Function: Euclidean Norm Calculation for Vectors

**Purpose:** This function calculates the **Euclidean norm** (magnitude) of a vector v of length len.

A computer screen shot of a black rectangular with colorful text

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * double sum = 0.0; initializes a variable to accumulate the sum of squared elements.
2. **Loop Through Elements**:
   * for (int i = 0; i < len; ++i) iterates over each element of the vector.
3. **Sum of Squares**:
   * sum += v[i] \* v[i]; squares each element and adds it to sum.
4. **Final Calculation**:
   * return sqrt(sum); returns the square root of the accumulated sum, giving the Euclidean norm.

### Function: Matrix Multiplication for Square Matrices

**Purpose:** This function multiplies two **square matrices** A and B of size n x n and stores the result in matrix C.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Row Iteration** (i loop):
   * Iterates over each row of matrix A (from 0 to n-1).
2. **Column Iteration** (j loop):
   * For each row i, iterates over each column of matrix B (from 0 to n-1).
3. **Element Initialization**:
   * C[i][j] = 0.0; resets the target element in C to zero to avoid garbage values.
4. **Dot Product Calculation** (k loop):
   * Computes the dot product of the i-th row of A and the j-th column of B.
   * Accumulates the sum of products: C[i][j] += A[i][k] \* B[k][j];.
5. **Result Storage**:
   * After the k loop completes, C[i][j] holds the final value for the element at row i, column j of the product matrix.

### Function: Matrix Transposition

**Purpose:** This function computes the **transpose** of a square matrix src of size n x n and stores the result in another matrix dst.

A computer screen shot of a black screen with white text

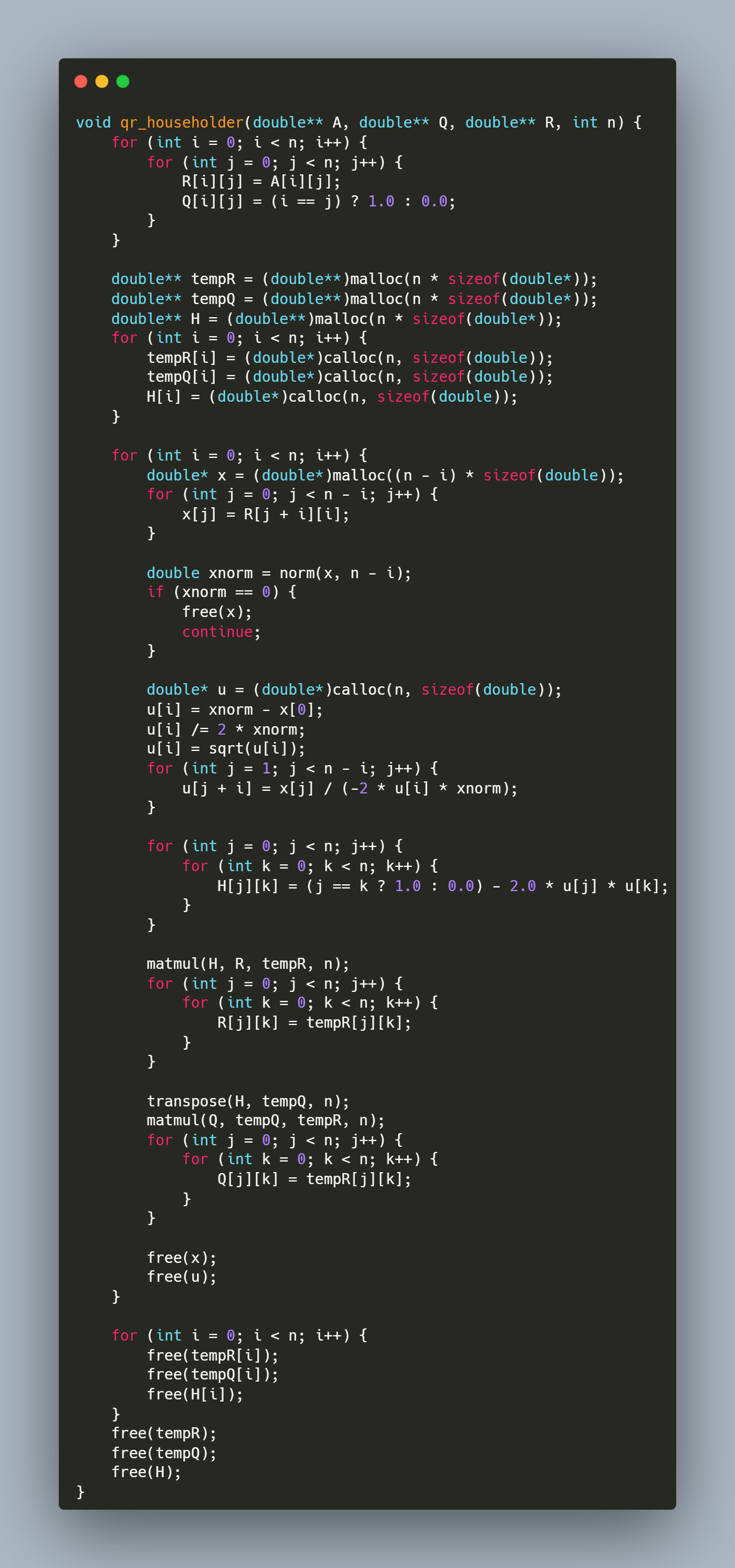
AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Outer Loop** (i loop):
   * Iterates over each row of the source matrix src (from 0 to n-1).
2. **Inner Loop** (j loop):
   * For each row i, iterates over each column of the source matrix src (from 0 to n-1).
3. **Transpose Operation**:
   * The element at position (i, j) in the source matrix is assigned to position (j, i) in the destination matrix:
     + dst[j][i] = src[i][j];.
4. **Result Storage**:
   * After all iterations, the destination matrix dst contains the transpose of the source matrix src.

### Function: QR Factorization Using Householder Reflections

**Purpose:** This function performs QR decomposition of a square matrix **A** into an orthogonal matrix **Q** and an upper triangular matrix **R** using Householder reflections.



**Step-by-Step Explanation**

**1. Initialization**

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

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

R[i][j] = A[i][j];

Q[i][j] = (i == j) ? 1.0 : 0.0;

}

}

* R is initialized as a copy of A.
* Q starts as an identity matrix, which will accumulate orthogonal transformations.

2. **Temporary Matrix Allocation**

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

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

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

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

tempR[i] = (double\*)calloc(n, sizeof(double));

tempQ[i] = (double\*)calloc(n, sizeof(double));

H[i] = (double\*)calloc(n, sizeof(double));

}

* tempR/tempQ: Store intermediate results during matrix multiplications.
* H: Stores the Householder reflection matrix for each iteration.
* calloc initializes elements to zero.

3. **Main Loop Over Columns**

The function processes each column to zero out sub-diagonal elements:

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

}

3.1 **Column Extraction**

double\* x = (double\*)malloc((n - i) \* sizeof(double));

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

x[j] = R[j + i][i];

}

* Extracts the sub-column starting at R[i][i] (elements below the diagonal).

3.2 **Norm Calculation**

double xnorm = norm(x, n - i);

if (xnorm == 0) {

free(x);

continue;

}

* Computes the Euclidean norm of the sub-column.
* Skips processing if the column is already zero (avoid division by zero).

3.3 Householder Vector (u) Construction

double\* u = (double\*)calloc(n, sizeof(double));

u[i] = xnorm - x[0];

u[i] /= 2 \* xnorm;

u[i] = sqrt(u[i]);

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

u[j + i] = x[j] / (-2 \* u[i] \* xnorm);

}

* u is computed according the Algorithm given.

3.4 **Householder Matrix (H) Construction**

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

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

H[j][k] = (j == k ? 1.0 : 0.0) - 2.0 \* u[j] \* u[k];

}

}

* H is computed as I - 2uuᵀ, where u is the Householder vector.

3.5 **Update R Matrix**

matmul(H, R, tempR, n);

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

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

R[j][k] = tempR[j][k];

}

}

* Applies the reflection to R, zeroing out sub-diagonal elements in column i.

3.6 Update Q Matrix

transpose(H, tempQ, n);

matmul(Q, tempQ, tempR, n);

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

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

Q[j][k] = tempR[j][k];

}

}

* Updates Q by accumulating the reflection: Q = Q \* Hᵀ.
* Since H is orthogonal, its transpose is its inverse.

3.7 Memory Cleanup (Per Iteration)

free(x);

free(u);

* Releases memory for temporary vectors to prevent leaks.

4. Final Memory Cleanup

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

free(tempR[i]);

free(tempQ[i]);

free(H[i]);

}

free(tempR);

free(tempQ);

free(H);

* Frees all dynamically allocated matrices.

### Function: Solving Linear Systems Using QR Decomposition

**Purpose:** This function solves the linear system **Ax = b** using the **QR factorization** of matrix **A**. It computes the solution **x** by first transforming **b** into **Qᵀb** and then performing backward substitution on the upper triangular matrix **R**.

A computer screen shot of a program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

**1. Compute Qᵀb**

* Allocates a vector Qtb to store the product of **Qᵀ** (orthogonal matrix) and **b**.
* The nested loop calculates each element of Qtb as the dot product of the **i-th column of Q** (equivalent to the **i-th row of Qᵀ**) and **b**.

**2. Backward Substitution**

* Iterates backward from the last row (i = n-1) to the first (i = 0):
  + **Initialize**: x[i] = Qtb[i] (starting value for the solution element).
  + **Subtract Known Terms**: For each column j > i, subtract R[i][j] \* x[j] to eliminate dependencies on already solved variables.
  + **Normalize**: Divide by the diagonal element R[i][i] to isolate x[i].

### Function: Reading Matrix and Vector Input from a File

**Purpose:** This function reads a square matrix **A** of size n x n and a vector **b** of size n from a file named "inputs.txt". It ensures proper error handling for file operations and input parsing.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Open File**:
   * FILE \*file = fopen("inputs.txt", "r"); opens the file "inputs.txt" in read mode.
   * If the file cannot be opened (e.g., it doesn't exist), the program prints an error message ("File not found") and exits.
2. **Read Matrix Elements**:
   * The nested loop for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { ... } } iterates through each row (i) and column (j) of matrix **A**.
   * fscanf(file, "%lf", &A[i][j]) reads a floating-point number from the file and stores it in the corresponding position in **A**.
   * If the input is invalid or missing, the program prints "Invalid input" and exits.
3. **Read Vector Elements**:
   * The loop for (int i = 0; i < n; i++) { ... } iterates through each element of vector **b**.
   * fscanf(file, "%lf", &b[i]) reads a floating-point number from the file and stores it in the corresponding position in **b**.
   * Similar error handling ensures that invalid or missing input causes the program to terminate with an error message.
4. **Close File**:

fclose(file); closes the file after all data has been read to release system resources.

### Main Program for Solving Linear Systems via QR Factorization

**Purpose:** This program solves a linear system **Ax = b** using QR decomposition. It reads matrix **A** and vector **b** from a file, computes the orthogonal matrix **Q** and upper triangular matrix **R**, solves for **x**, and prints the solution, runtime, and matrix **Q**.



**Step-by-Step Explanation**

**1. User Input**

* Prompts the user to enter the number of variables (n).

**2. Memory Allocation**

* Allocates memory for:
  + **A** (n x n matrix) using nested malloc.
  + **b** (vector of size n).
  + **Q** and **R** (n x n matrices initialized with calloc).
  + **x** (solution vector initialized with calloc).

**3. Read Input File**

* Calls read\_file\_input(n, A, b) to populate **A** and **b** from "inputs.txt".

**4. QR Factorization & Timing**

* clock() records the start time.
* qr\_householder(A, Q, R, n) decomposes **A** into **Q** (orthogonal) and **R** (upper triangular).
* clock() records the end time.

**5. Solve for x**

* solve\_using\_qr(Q, R, b, x, n) computes the solution vector **x** via backward substitution.

**6. Print Results**

* Displays the solution vector **x** with 6 decimal places.
* Prints the runtime in nanoseconds.
* Outputs the orthogonal matrix **Q** for verification.

**7. Memory Cleanup**

* Frees all dynamically allocated memory for **A**, **b**, **Q**, **R**, and **x**.

Thank You