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

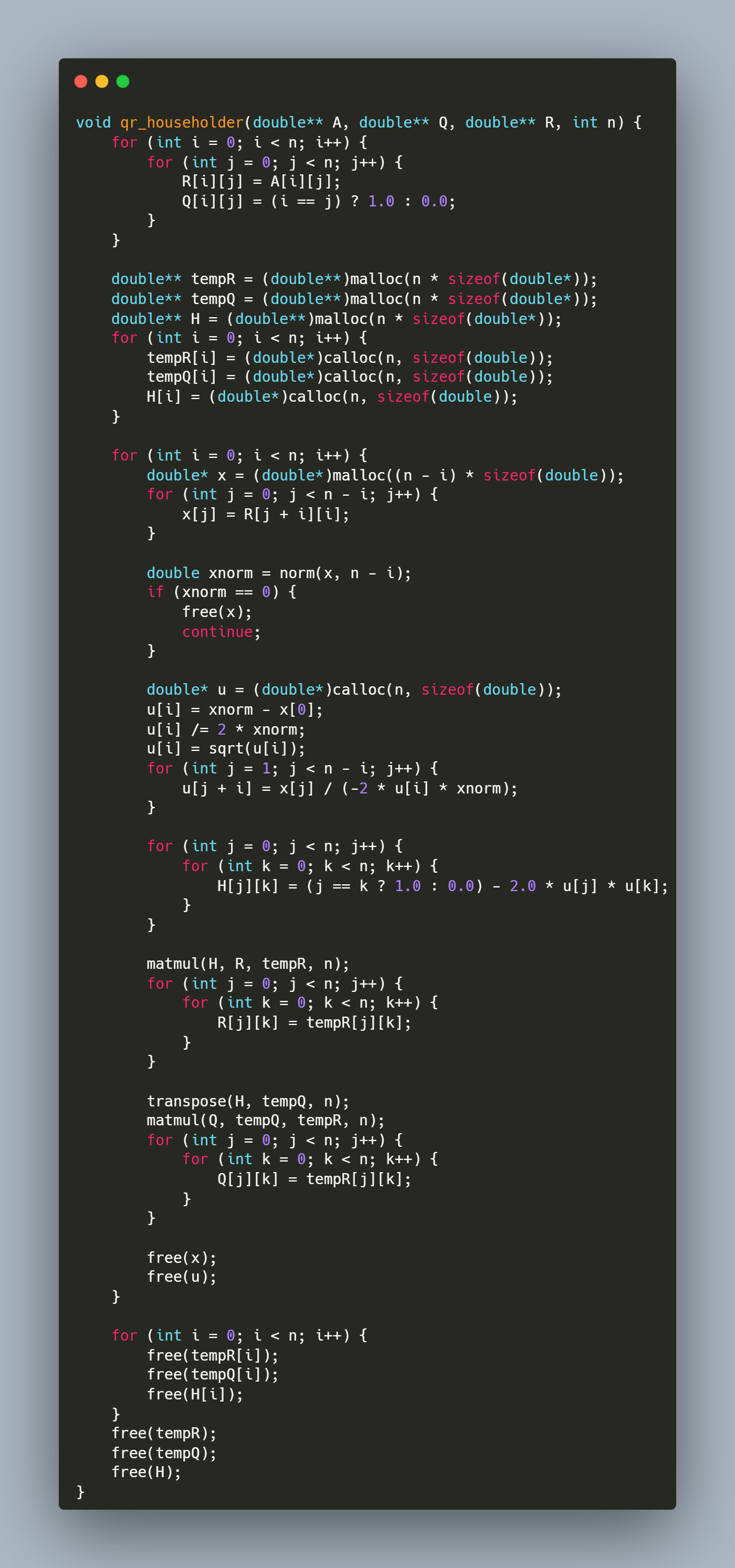
AI-generated content may be incorrect.

**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**.