conjugate gradient

## Reading Matrix and Vector Input from File

**Purpose:** This function reads a **square matrix A** of size n x n and a **vector b** of size n from a specified file. It ensures the file is properly formatted and handles errors during 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(filename, "r"); opens the file in read mode.
* If the file cannot be opened (e.g., it does not exist), the function prints an error message and exits with status 1.

**2. Read Matrix A**

* The nested loops iterate over each row (i) and column (j) of the matrix:
  + fscanf(file, "%lf", &A[i][j]) reads a floating-point number into A[i][j].
  + If the read operation fails (e.g., non-numeric data), the function prints an error, closes the file, and exits.

**3. Read Vector b**

* The loop iterates over each element of the vector:
  + fscanf(file, "%lf", &b[i]) reads a floating-point number into b[i].
  + If the read operation fails, the function prints an error, closes the file, and exits.

**4. Close File**

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

## 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 returns the resulting vector. It computes the dot product of each row of A with b.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Memory Allocation**:
   * Allocates memory for the result vector result of size n.
   * Exits with an error if memory allocation fails.
2. **Row Iteration**:
   * The outer loop iterates over each row i of the matrix A.
3. **Dot Product Calculation**:
   * Initializes result[i] to 0.
   * The inner loop iterates over each column j of the matrix.
   * Computes the dot product of the i-th row of A and the vector b.
4. **Return Result**:
   * Returns the computed vector result.

## function: Vector Subtraction

**Purpose**: This function computes the **element-wise subtraction** of two vectors **v1** and **v2** of size n and returns the result as a new vector

A screen shot of a computer code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Memory Allocation**:
   * Allocates memory for the result vector result of size n.
   * Exits with an error if memory allocation fails.
2. **Element-wise Subtraction**:
   * Iterates over each index i from 0 to n-1.
   * Computes result[i] = v1[i] - v2[i] for each element.
3. **Return Result**:
   * Returns the vector result containing the difference of v1 and v2.

## function: Dot Product of Two Vectors

**Purpose:** This function computes the **dot product** of two vectors **v1** and **v2** of size n

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * result is initialized to 0.0 to accumulate the sum of products.
2. **Element-wise Multiplication and Summation**:
   * Iterates over each index i from 0 to n-1.
   * Computes the product of corresponding elements v1[i] \* v2[i] and adds it to result.
3. **Return Result**:
   * Returns the scalar value result, which is the dot product of the two vectors.

## FUNCTION: **Conjugate Gradient Method**

**Purpose:** This function solves the linear system **Ax = b** using the **Conjugate Gradient (CG) method**, an iterative algorithm optimized for **symmetric positive-definite (SPD)** matrices. It minimizes the residual error iteratively until convergence.

A screen shot of a computer program

AI-generated content may be incorrect.

**Function Parameters**

* A: Symmetric positive-definite matrix (n x n).
* b: Right-hand side vector (size n).
* n: Size of the matrix/vector.
* epsilon: Tolerance threshold for stopping criteria.
* iterations: Output parameter tracking the number of iterations performed.

**Returns**: Solution vector x.

**Step-by-Step Explanation**

**1. Initialization**

* **Initial Guess**:

**double** \*x = malloc(...);

**for** (...) x[i] = 1.0; *// Initial guess x₀ = [1, 1, ..., 1]*

The algorithm starts with an initial guess of x as a vector of ones.

* **Residual Calculation**:

**double** \*Ax = multiplyMatrixVector(A, x, n);

**double** \*r = subtractVectors(b, Ax, n); *// r₀ = b - A x₀*

Compute the residual r (error between b and A x).

* **Direction Vector**:

**double** \*d = malloc(...);

**for** (...) d[i] = r[i]; *// d₀ = r₀*

Initialize the search direction d to the residual r.

* **Tolerance Check**:

**double** tol = sqrt(multiplyVectors(n, r, r)); *// ||r₀||*

Calculate the L2 norm of the residual to check convergence.

**2. Iterative Loop**

The loop continues until the residual norm tol falls below epsilon:

**while** (tol > epsilon) {

*// Step 2.1: Compute α (step size)*

**double** \*Ad = multiplyMatrixVector(A, d, n);

**double** alpha = (rᵀr) / (dᵀ A d);

*// Step 2.2: Update x and residual*

x += α \* d;

r\_new = r - α \* A d;

*// Step 2.3: Compute β (direction adjustment)*

**double** beta = (r\_newᵀ r\_new) / (rᵀ r);

*// Step 2.4: Update direction vector*

d = r\_new + β \* d;

*// Step 2.5: Update residual and tolerance*

r = r\_new;

tol = ||r\_new||;

(\*iterations)++;

}

**Key Steps in Each Iteration**:

1. **Compute Step Size (α)**:
   * α determines how far to move along the direction d.
   * Formula:  
     **α =**
2. **Update Solution (x) and Residual (r\_new)**:
   * Adjust x using the step size α and direction d.
   * Update the residual to reflect the new solution:
3. **Compute Direction Adjustment (β)**:
   * β adjusts the search direction to ensure conjugacy with previous directions.
   * Formula:  
     **β=**
4. **Update Direction Vector (d)**:
   * Combine the new residual and previous direction:  
     d=

**3. Termination and Cleanup**

* **Convergence Check**: The loop exits when tol ≤ epsilon.
* **Memory Management**:

c

free(r); free(r\_new); free(d); *// Free temporary vectors*

* **Return Result**: The solution vector x is returned.

## **Main Program for Conjugate Gradient Method**

**Purpose:** This program solves a linear system **Ax = b** using the **Conjugate Gradient (CG) method**. It reads the matrix **A** and vector **b** from a file, computes the solution **x**, and prints the results along with performance metrics.

A screen shot of a computer screen

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **User Input**:
   * Prompts the user for:
     + n: Number of variables (size of the matrix/vector).
     + epsilon: Convergence tolerance (e.g., 1e-6). Smaller values increase precision but may require more iterations.
2. **Memory Allocation**:
   * Allocates memory for:
     + **A**: n x n matrix (dynamic 2D array).
     + **b**: Vector of size n.
3. **Read Input File**:
   * Calls readInputsFromFile("inputs.txt", A, b, n) to load **A** and **b** from the file.
   * **File Format**: The file must contain n x n matrix elements followed by n vector elements.
4. **Solve with Conjugate Gradient**:
   * Conjugate\_Gradient(A, b, n, epsilon, &iterations) computes the solution vector **x**.
     + **Input**: Matrix **A**, vector **b**, size n, tolerance epsilon.
     + **Output**: Solution x and number of iterations to converge.
   * **Note**: **A** must be **symmetric positive-definite** (SPD) for the CG method to work correctly.
5. **Performance Timing**:
   * Uses clock() to measure CPU time taken by the CG method.
   * Converts time to nanoseconds for readability.
6. **Print Results**:
   * Displays the number of iterations and execution time.
   * Prints the solution vector **x** with 6 decimal places.
7. **Memory Cleanup**:
   * Frees dynamically allocated memory for **A**, **b**, and **x** to prevent leaks.