ESc 101: Fundamentals of Computing

Lecture 25

Mar 8, 2010

Outline

Basic Matrix Operations

- Addition, subtraction
- Multiplication
- Inversion
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction
- Multiplication
- Inversion
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction
- Multiplication
- Inversion
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction
- Multiplication
- Inversion
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction: Simple
- Multiplication
- Inversion
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction
- Multiplication: Done
- Inversion
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction
- Multiplication
- Inversion: Will develop an algorithm
- Computing Determinant

Basic Matrix Operations

- Addition, subtraction
- Multiplication
- Inversion
- Computing Determinant: By definition, by Gaussian elimination

Determinant

Let $A = [a_{i,j}]$ be an $n \times n$ matrix. Its determinant is:

$$\sum_{\pi} sgn(\pi) \cdot \prod_{i=0}^{n-1} a_{i,\pi(i)},$$

where

- π runs over all permutations of $\{0, 1, 2, \ldots, n-1\}$, and
- $sgn(\pi) \in \{1, -1\}$ is the sign of permutation π.

Computing Determinant

- Computing determinant using the above formula will be very time consuming: as there are $n!$ permutations of $\{0, 1, 2, \ldots, n-1\}$, and the formula sums over all of these.
- There is a faster way known for computing determinant: Gaussian elimination.

Computing Determinant

- Computing determinant using the above formula will be very time consuming: as there are $n!$ permutations of $\{0, 1, 2, \ldots, n-1\}$, and the formula sums over all of these.
- There is a faster way known for computing determinant: Gaussian elimination.

Gaussian Elimination

Let $A_0 = [a_{i,j}]$ be an $n \times n$ matrix:

$$A_0 = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}.$$

First Step: Check if $a_{0,0} \neq 0$. If it is, add to it the first row whose first element is non-zero. If no such row exists, then the determinant is zero.

Gaussian Elimination

Let $A_0 = [a_{i,j}]$ be an $n \times n$ matrix:

$$A_0 = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}.$$

First Step: Check if $a_{0,0} \neq 0$. If it is, add to it the first row whose first element is non-zero. If no such row exists, then the determinant is zero.

Gaussian Elimination

Second Step: For every $i > 0$, subtract $\frac{a_{i,0}}{a_{0,0}}$ times the first row from the ith row. This makes $a_{i,0} = 0$ for all $i > 0$.

After the first two steps, the matrix looks like:

$$A_0 = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ 0 & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix},$$

where the values of many elements has been modified from their original value.

Gaussian Elimination

Second Step: For every $i > 0$, subtract $\frac{a_{i,0}}{a_{0,0}}$ times the first row from the ith row. This makes $a_{i,0} = 0$ for all $i > 0$.

After the first two steps, the matrix looks like:

$$A_0 = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ 0 & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix},$$

where the values of many elements has been modified from their original value.

Gaussian Elimination

Let matrix A_1 be:

$$A_1 = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}.$$

Next Steps: Repeat the first two steps for A_1 and all the submatrices A_2, …, A_{n-1} that arise.

Gaussian Elimination

Let matrix A_1 be:

$$A_1 = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}.$$

Next Steps: Repeat the first two steps for A_1 and all the submatrices $A_2, \ldots, A_{n-1}$ that arise.

Gaussian Elimination

Next Step: Let matrix B be defined by taking the first row of A_0, second row of A_1, ..., last row of A_{n-1}.

Matrix B looks like:

$$B = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,n-1} \\ 0 & a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} \\ 0 & 0 & a_{2,2} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1,n-1} \end{bmatrix}.$$

Last Step: The determinant of the matrix A equals the product of diagonals of B, i.e., $\prod_{i=0}^{n-1} a_{i,i}$.

Gaussian Elimination

Next Step: Let matrix B be defined by taking the first row of A_0, second row of A_1, ..., last row of A_{n-1}.

Matrix B looks like:

$$B = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,n-1} \\ 0 & a_{1,1} & a_{1,2} & \cdots & a_{1,n-1} \\ 0 & 0 & a_{2,2} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-1,n-1} \end{bmatrix}.$$

Last Step: The determinant of the matrix A equals the product of diagonals of B, i.e., $\prod_{i=0}^{n-1} a_{i,i}$.

Gaussian Elimination

Next Step: Let matrix B be defined by taking the first row of A_0, second row of A_1, …, last row of A_{n-1}.

Matrix B looks like:

$$B = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \dots & a_{0,n-1} \\ 0 & a_{1,1} & a_{1,2} & \dots & a_{1,n-1} \\ 0 & 0 & a_{2,2} & \dots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & a_{n-1,n-1} \end{bmatrix}.$$

Last Step: The determinant of the matrix A equals the product of diagonals of B, i.e., $\prod_{i=0}^{n-1} a_{i,i}$.

Why Does it Work?

Theorem

The determinant of a matrix does not change by adding or subtracting a row to another row.

The Gaussian Elimination algorithm only adds or subtracts rows.

Why Does it Work?

Theorem

The determinant of a matrix does not change by adding or subtracting a row to another row.

The Gaussian Elimination algorithm only adds or subtracts rows.

Rewriting the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Rewriting the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Rewriting the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Rewriting the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
        Find the smallest i such that A[i][0] != 0;
        If there is no such i then // determinant is zero
            return 0;
        Add row A[i] to row A[0];
   }
3. For every i > 0:
        Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Rewriting the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Converting to a C Program

```
/* Computes the determinant of size n matrix
 * stored in array A.
 */
float determinant(float A[][N], int n)
{
    float B[N][N]; // stores a submatrix of A
    int m; // the size of B
    float det = 1.0; // determinant value
    int i;

    copy_matrix(B, A, 0, n); // copy A to B
```

Converting to a C Program

```
/* Do the Gaussian elimination for the first row,
 * multiply the first diagonal element to det, and drop
 * the first row and column from B.
*/
for (m = n; m > 0; m--) {
    if (B[0][0] == 0) {
        i = find_nonzero_row(B, m);
        if (i >= m) // no non-zero row
            return 0.0; // determinant is 0
        add_row(B[0], B[i], 1, m); // add row i to row 0
    }
```

Converting to a C Program

```
        // Make first column of B zero except the first row
        for (int t = 1; t < m; t++)
            add_row(B[t], B[0], - B[t][0]/B[0][0], m);

        det = det * B[0][0]; // update determinant value

        // drop the first row and column of B
        copy_matrix(B, B, 1, m-1);
    }
    return det;
}
```

Converting to a C Program

```
/* Copies matrix A to B after dropping first i rows and
 * columns of A. The size of matrix B is m.
 */
void copy_matrix(float B[][N], float A[][N], int i, int m)
{
    for (int k = 0; k < m; k++)
        for (int j = 0; j < m; j++)
            B[k][j] = A[k+i][j+i];
}
```

Recall the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Recall the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Recall the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Recall the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Recall the Algorithm

```
Input: matrix A, and its size n.

1. If (n == 1) go to step 5;
2. If (A[0][0] = = 0) {
       Find the smallest i such that A[i][0] != 0;
       If there is no such i then // determinant is zero
           return 0;
       Add row A[i] to row A[0];
   }
3. For every i > 0:
       Replace row A[i] by A[i] - (A[i][0]/A[0][0]) * A[0];
4. Drop first row and first column of A and go back to 1;
5. Return the product of diagonal elements;
```

Recall Algorithm

- Observe that after Step 4, we get a matrix of size $n - 1$ and we need to compute its determinant.
- This is the same problem as the original one, except that the size is one less.
- So we can use the same algorithm to solve it.
- That is why, the execution goes back to Step 1.
- We can implement this algorithm in C in another way: using recursion.

Recall Algorithm

- Observe that after Step 4, we get a matrix of size $n - 1$ and we need to compute its determinant.
- This is the same problem as the original one, except that the size is one less.
- So we can use the same algorithm to solve it.
- That is why, the execution goes back to Step 1.
- We can implement this algorithm in C in another way: using recursion.

Recall Algorithm

- Observe that after Step 4, we get a matrix of size $n-1$ and we need to compute its determinant.
- This is the same problem as the original one, except that the size is one less.
- So we can use the same algorithm to solve it.
- That is why, the execution goes back to Step 1.
- We can implement this algorithm in C in another way: using recursion.

Recall Algorithm

- Observe that after Step 4, we get a matrix of size $n - 1$ and we need to compute its determinant.
- This is the same problem as the original one, except that the size is one less.
- So we can use the same algorithm to solve it.
- That is why, the execution goes back to Step 1.
- We can implement this algorithm in C in another way: using recursion.

Program Using Recursion

```
/* Computes the determinant of size n matrix
 * stored in array A.
 */
float determinant(float A[][N], int n)
{
    float B[N][N]; // stores a submatrix of A
    int i;

    if (n == 1) // Step 1: 1 x 1 matrix
        return A[0][0];
```

Program Using Recursion

```
// Step 2: Make A[0][0] non-zero
if (A[0][0] == 0) {
    i = find_nonzero_row(A, n);
    if (i >= n) // no non-zero row
        return 0.0; // determinant is 0
    add_row(A[0], A[i], 1, n); // add row i to row 0
}
```

Program Using Recursion

```
    // Step 3: Make first column of A zero except first row
    for (int t = 1; t < n; t++)
        add_row(A[t], A[0], - A[t][0]/A[0][0], n);

    // Step 4: drop the first row and column of A
    copy_matrix(B, A, 1, n-1);

    return A[0][0] * determinant(B, n-1); // recursive call!
}
```

Program Using Recursion

```
    // Step 3: Make first column of A zero except first row
    for (int t = 1; t < n; t++)
        add_row(A[t], A[0], - A[t][0]/A[0][0], n);

    // Step 4: drop the first row and column of A
    copy_matrix(B, A, 1, n-1);

    return A[0][0] * determinant(B, n-1); // recursive call!
}
```

Recursion

- A function is recursive if it is called inside its own definition.
- Such a definition is a substitute for loop, as in the example above.
- The execution jumps to the beginning of the function at the recursive call.
- To avoid infinite repetitions, it is necessary that:
  - in every successive call, some parameter value reduces,
  - and for small enough value of that parameter, there is no recursive call in the function.

Recursion

- A function is recursive if it is called inside its own definition.
- Such a definition is a substitute for loop, as in the example above.
- The execution jumps to the beginning of the function at the recursive call.
- To avoid infinite repetitions, it is necessary that:
  - in every successive call, some parameter value reduces,
  - and for small enough value of that parameter, there is no recursive call in the function.

Recursion

- A function is recursive if it is called inside its own definition.
- Such a definition is a substitute for loop, as in the example above.
- The execution jumps to the beginning of the function at the recursive call.
- To avoid infinite repetitions, it is necessary that:
  - in every successive call, some parameter value reduces,
  - and for small enough value of that parameter, there is no recursive call in the function.

Recursion

- A function is recursive if it is called inside its own definition.
- Such a definition is a substitute for loop, as in the example above.
- The execution jumps to the beginning of the function at the recursive call.
- To avoid infinite repetitions, it is necessary that:
  - in every successive call, some parameter value reduces,
  - and for small enough value of that parameter, there is no recursive call in the function.

Recursion

- A function is recursive if it is called inside its own definition.
- Such a definition is a substitute for loop, as in the example above.
- The execution jumps to the beginning of the function at the recursive call.
- To avoid infinite repetitions, it is necessary that:
  - in every successive call, some parameter value reduces,
  - and for small enough value of that parameter, there is no recursive call in the function.

Recursion

- A function is recursive if it is called inside its own definition.
- Such a definition is a substitute for loop, as in the example above.
- The execution jumps to the beginning of the function at the recursive call.
- To avoid infinite repetitions, it is necessary that:
  - in every successive call, some parameter value reduces,
  - and for small enough value of that parameter, there is no recursive call in the function.