Gauss Jordan Elimination

Kim Day
Overview

- Background/Description
- Algorithm
- Code snippets
- Examples
- Analysis
Background/Description
Background

• Named for Carl Friedrich Gauss and Wilhelm Jordan

• Started out as “Gaussian elimination” although Gauss didn’t create it

• Jordan improved it in 1887 because he needed a more stable algorithm for his surveying calculations

Carl Gauss mathematician/scientist 1777-1855

Wilhelm Jordan geodesist 1842-1899

(geodesy involves taking measurements of the Earth)
Some Terms

- Matrix - 2D array
- Identity matrix - Matrix with all 0s except for 1s on the diagonal
- Determinant - Representative number that can be calculated from a matrix
- Matrix inverse - The matrix version of n^-1

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

A 4x4 identity matrix
Elementary Operations

- Steps that can be performed on matrices without changing their overall meaning
- Multiplying by a scalar - Replace a row/column by itself times a factor
- Linear combinations - Replace a row/column by a combination of itself and another row/column
- Pivoting - Interchanging two rows/columns
  - Don’t need pivoting but it really helps
Gaussian Elimination

• First seen used in the Chinese text “The Nine Chapters on the Mathematical Art” and in Isaac Newton’s notes

• Puts a matrix into row echelon form, and then uses back substitution to solve

• Determinant is product of diagonals

\[
\begin{bmatrix}
2 & 4 & 1 & 4 & 7 \\
0 & 6 & 3 & 5 & 3 \\
0 & 0 & 2 & 6 & 9 \\
0 & 0 & 0 & 3 & 1 \\
0 & 0 & 0 & 0 & 5 \\
\end{bmatrix}
\]

Row echelon form: Lower triangle is 0s
Gauss-Jordan Elimination

- Gauss-Jordan elimination is a faster way to solve matrices and find a matrix inverse
- Puts the matrix into row-reduced echelon form

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Reduced row echelon form: Non-diagonals are 0s
### Comparison

<table>
<thead>
<tr>
<th></th>
<th>Solves system</th>
<th>Finds determinant</th>
<th>Finds inverse</th>
<th>Form used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss Elim.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Row Echelon</td>
</tr>
<tr>
<td>Gauss-Jordan Elim.</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>Reduced Row Echelon</td>
</tr>
</tbody>
</table>

• Can produce both the solution for a set of linear equations and the matrix inverse

• As efficient as most methods when it comes to finding a matrix inverse

• Solving the system of equations doesn’t take up that much more time than finding the inverse

• Fairly stable

- Requires more storage (bookkeeping and right hand elements)

- Takes three times as long than most methods when solving for a single set
Algorithm
Algorithm

- Repeat n times, where n is the number of columns
  - Locate a pivot
  - Move the row containing the pivot so that the pivot is on a diagonal
  - Divide the pivot’s row by the value of the pivot
  - Subtract multiples of the pivot’s row from the rows above and below to make them 0
  - If solving a system of equations, make sure to do the same operations on the vector matrix as well
- Input matrix is replaced by inverse and vector matrix is replaced by solutions
What is a Pivot?

• A “special” element of a matrix, chosen to become part of the final diagonal

• The pivot is usually the largest element in an unaltered row/column

• Choose a large pivot because that makes it easier to reduce the rest of the row/column
Code Snippets
Choosing a pivot

for (int i = 0; i < n; i++) {
    double big = 0.0;
    int icol = 0;
    int irow = 0;
    // Search for a pivot element in each column
    for (int j = 0; j < n; j++) {
        // Check that the column hasn't been visited
        if (ipiv[j] != 1) {
            // Now check through each member of the column
            for (int k = 0; k < n; k++) {
                if (ipiv[k] == 0) {
                    if (fabs(a.get(j, k)) >= big) {
                        big = fabs(a.get(j, k));
                        irow = j;
                        icol = k;
                    }
                }
            }
        }
    }
}

Essentially chooses the largest (absolute value) element on an unvisited column and row
Moving To Diagonal

// Interchange rows to put the pivot on the diagonal
if (irow != icol) {
    a.exchange_rows(irow, icol);
    b.exchange_rows(irow, icol);
    if (verbose) {
        printf("Exchanging rows %d and %d\n", irow, icol);
        a.print();
    }
}
Normalizing row

// Divide the row by the pivot
double pivot_inverse = 1.0 / a.get(icol, icol);
a.set(icol, icol, 1.0);
a.mult_row(icol, pivot_inverse);
b.mult_row(icol, pivot_inverse);
if (verbose) {
    printf("Dividing row %d by %.2f\n", icol, 1.0 / pivot_inverse);
    a.print();
}

Divides the pivot's row by the value of the pivot.

Will explain this in a bit
Reducing column

// Reduce the rows (except for the pivot row)
for (int ll = 0; ll < n; ll++) {
    if (ll != icol) {
        double dummy = a.get(ll, icol);
        a.set(ll, icol, 0.0);
        a.add_rows(1.0, ll, -dummy, icol);
        b.add_rows(1.0, ll, -dummy, icol);
        if (verbose) {
            printf("Row %d -= %.2f * Row %d\n", ll, dummy, icol);
            a.print();
        }
    }
}

Subtracts multiples of the pivot row from the rows above/below to make the column mostly 0s
Note: Storage

- The code in the textbook “saves space” by not storing the identity matrix as a separate matrix. Instead, it coexists with the input matrix.

- This can be done because we know that the input matrix will eventually become the identity matrix.

- That’s why the code changes the input matrix to the identity matrix right before doing any replacements.
Simple Example

\[ x + 2y = 8 \]
\[ 3x + 4y = 20 \]

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
= 
\begin{bmatrix}
8 \\
20
\end{bmatrix}
\]
<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2</td>
<td>8 3 4</td>
</tr>
<tr>
<td>2</td>
<td>1 2</td>
<td>8 .75 1</td>
</tr>
<tr>
<td>3</td>
<td>-.5 0</td>
<td>-2 .75 1</td>
</tr>
<tr>
<td>4</td>
<td>1 0</td>
<td>4 .75 1</td>
</tr>
<tr>
<td>5</td>
<td>1 0</td>
<td>4 0 1</td>
</tr>
</tbody>
</table>
Real-World Example: Ray-Triangle Intersection

- From Shirley’s “Fundamentals of Computer Graphics”
- Goal: Find the point where a ray intersects a plane defined by a triangle
- Basic form of ray tracing

\[ p(a, \beta, \gamma) = a + \beta(b - a) + \gamma(c - a) \]

The ray marked by ED intersects the plane defined by triangle ABC at point P.

Wednesday, January 16, 13
Ray-Triangle Intersection

Point must lie on both the vector, represented by:

\[ \vec{p} = \vec{e} + t\vec{d} \]

and the plane of the triangle, represented by:

\[ \vec{p} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a}) \]

so the resulting equation to solve is:

\[ \vec{e} + t\vec{d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a}) \]

for some \( t, \beta, \) and \( \gamma \).
Ray-Triangle Intersection

\[ \vec{e} + t \vec{d} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a}) \]

In xyz coordinates, this becomes:

\[
\begin{align*}
x_e + tx_d &= x_a + \beta(x_b - x_a) + \gamma(x_c - x_a) \\
y_e + ty_d &= y_a + \beta(y_b - y_a) + \gamma(y_c - y_a) \\
z_e + tz_d &= z_a + \beta(z_b - z_a) + \gamma(z_c - z_a)
\end{align*}
\]

which can also be written as

\[
\begin{bmatrix}
x_a - x_b & x_a - x_c & x_d \\
y_a - y_b & y_a - y_c & y_d \\
z_a - z_b & z_a - z_c & z_d
\end{bmatrix}
\begin{bmatrix}
\beta \\
\gamma \\
t
\end{bmatrix}
= 
\begin{bmatrix}
x_a - x_e \\
y_a - y_e \\
z_a - z_e
\end{bmatrix}
\]

which can now be solved by Gauss-Jordan elimination!
Ray-Triangle Intersection

- Find where the ray $(1, 1, 1) + t(-1, -1, -1)$ hits a triangle with vertices $(1, 0, 0), (0, 1, 0), \text{ and } (0, 0, 1)$
Ray-Triangle Intersection

Using the previous formula

\[
\begin{bmatrix}
    x_a - x_b & x_a - x_c & x_d \\
    y_a - y_b & y_a - y_c & y_d \\
    z_a - z_b & z_a - z_c & z_d
\end{bmatrix}
\begin{bmatrix}
    \beta \\
    \gamma \\
    t
\end{bmatrix}
= \begin{bmatrix}
    x_a - x_e \\
    y_a - y_e \\
    z_a - z_e
\end{bmatrix}
\]

\[
\begin{bmatrix}
    1 & 1 & -1 \\
    0 & 0 & -1 \\
    0 & -1 & -1
\end{bmatrix}
\begin{bmatrix}
    \beta \\
    \gamma \\
    t
\end{bmatrix}
= \begin{bmatrix}
    1 & 1 \\
    0 & 0 \\
    0 & -1
\end{bmatrix}
\]

End result is

\[t = \frac{2}{3}\]
\[\beta = \frac{1}{3}\]
\[\gamma = \frac{1}{3}\]

or \[p = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)\]
Ray-Triangle Intersection

input matrix:
1.00  1.00  -1.00
-1.00  0.00  -1.00
0.00  -1.00  -1.00

vector matrix:
0.00
-1.00
-1.00

Starting Gauss-Jordan algorithm...

Dividing row 2 by -1.00
1.00  1.00  -1.00
-1.00  0.00  -1.00
-0.00  1.00  -1.00

Row 0 ← -1.00 * Row 2
1.00  2.00  -1.00
-1.00  0.00  -1.00
-0.00  1.00  -1.00

Row 1 ← -1.00 * Row 2
1.00  2.00  -1.00
-1.00  1.00  -1.00
-0.00  1.00  -1.00

Exchanging rows 0 and 1
-1.00  1.00  -1.00
1.00  2.00  -1.00
-0.00  1.00  -1.00

Dividing row 1 by 2.00
-1.00  1.00  -1.00
0.50  0.50  -0.50
-0.00  1.00  -1.00

Row 0 ← 1.00 * Row 1
-1.50 -0.50 -0.50
0.50  0.50 -0.50
-0.00  1.00 -1.00

Row 2 ← 1.00 * Row 1
-1.50 -0.50 -0.50
0.50  0.50 -0.50
-0.50 -0.50 -0.50

Dividing row 0 by -1.50
-0.67  0.33  0.33
0.50  0.50 -0.50
-0.50 -0.50 -0.50

Row 1 ← 0.50 * Row 0
-0.67  0.33  0.33
0.33  0.33 -0.67
-0.50 -0.50 -0.50

Row 2 ← -0.50 * Row 0
-0.67  0.33  0.33
0.33  0.33 -0.67
-0.33 -0.33 -0.33

Exchanging columns 0 and 1
0.33 -0.67  0.33
0.33  0.33 -0.67
-0.33 -0.33 -0.33

Done!

inverse:
0.33 -0.67  0.33
0.33  0.33 -0.67
-0.33 -0.33 -0.33

solution:
0.33
0.33
0.67

Code output
Ray-Triangle Intersection

Substitute 2/3 for $t$ to find $p$

$$\vec{p} = \vec{e} + t\vec{d}$$

$$\vec{p} = (1, 1, 1) + \frac{2}{3}(-1, -1, -1) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$$

Get the same result with $\beta = \gamma = 1/3$

$$\vec{p} = \vec{a} + \beta(\vec{b} - \vec{a}) + \gamma(\vec{c} - \vec{a})$$

$$\vec{p} = (1, 0, 0) + \frac{1}{3}(-1, 1, 0) + \frac{1}{3}(-1, 0, 1) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$$
Analysis
• Used the clock() function to time how long it took to calculate the inverse of matrices of size n

• Matrices were randomly generated

• Took an unusual amount of time to calculate for n=1

• Follows an $O(n^3)$ pattern
Efficiency (1,20)

Gauss Jordan Runtime

\[ y = 0.0618x^3 + 0.5013x^2 - 4.4488x + 15.627 \]

\[ R^2 = 0.9977 \]
Efficiency (1, 150)

Gauss Jordan Runtime

\[ y = 0.0886x^3 - 0.9067x^2 + 44.224x - 458.45 \]

\[ R^2 = 0.9998 \]
Efficiency (1, 900)

Gauss Jordan Runtime

\[ y = 0.0934x^3 - 1.3472x^2 + 205.08x - 10779 \]

\[ R^2 = 0.9999 \]
Areas for future analysis

- Investigate the impact of pivoting on processing time
- Compare against other methods for calculating inverses/solving systems of equations
Sources

- Wikipedia
- Numerical Recipes (Press)
- Fundamentals of Computer Graphics (Shirley)