Gauss-Jordan Elimination

Mahbuba Perveen

August 25, 2020
CSE-5400
Basics

Identity matrix, $I_5$

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Row Echelon form

$$\begin{pmatrix}
3 & 1 & 4 & 6 & 7 \\
0 & 4 & 1 & 4 & 7 \\
0 & 0 & 0 & 4 & 1 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Reduced Row-Echelon form

$$\begin{pmatrix}
1 & 0 & 4 & 0 & 0 \\
0 & 1 & 2 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

5x5

(Lower triangle is 0s)
Properties of G-J Elimination

● It is an algorithm to solve systems of linear equations
● The basic idea is to add or subtract linear combinations of the given equations until each equation contains only one of the unknowns
● It produces:
  ○ Solutions of the equations for one or more right-hand side vectors $\mathbf{b}$
  ○ The matrix inverse $\mathbf{A}^{-1}$
Algorithm

- Repeat N times, where N is the number of columns of the input matrix
  - Locate a pivot
  - Move the row/column containing the pivot so that the pivot is on the diagonal
  - Divide the pivot’s row by the value of the pivot
  - Subtract multiple of the pivot’s row from the rows above and below to make them zero
  - If solving a system of equations, make sure to do the same operations on the vector matrix as well
- Input matrix is replaced by its inverse and vector matrix is replaced by the solution
Elimination on Column-Augmented Matrices

Consider the linear matrix equation:

\[
\begin{pmatrix}
a_{00} & a_{01} \\
 & \\
a_{02} & a_{10} & a_{11} \\
 & \\
a_{12} & \\
 & \\
a_{20} & a_{21} \\
 & \\
a_{22}
\end{pmatrix}
\begin{pmatrix}
x_{00} \\
&
\end{pmatrix}
\begin{pmatrix}
x_{01} \\
&
\end{pmatrix}
\begin{pmatrix}
y_{00} & y_{01} & y_{02} \\
&
\end{pmatrix}
= 
\begin{pmatrix}
b_{00} \\
&
\end{pmatrix}
\begin{pmatrix}
b_{01} \\
&
\end{pmatrix}
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

Or, \([A] \cdot [x_0 \ x_1 \ Y] = [b_0 \ b_1 \ 1]\)

where \(A\) and \(Y\) are square matrices, the \(b_i\)'s and \(x_i\)'s are column vectors and \(1\) is the identity matrix.

The matrix solution of this system simultaneously solves the linear sets:

\[A \cdot x_0 = b_0 \quad A \cdot x_1 = b_1 \quad A \cdot Y = 1\]
Elementary Operations

The solution set of x’s and Y are unchanged by the following operations:

- Interchanging any two rows of A and the corresponding rows of the b’s and 1
- Replacing any row in A by a linear combination of itself and any other row along with the rows of the b’s and 1
- Interchanging any two columns of A and the corresponding rows of the x’s and of Y

Gauss-Jordan elimination uses one or more of the above operations to reduce the matrix A to the identity matrix. When this is accomplished, the r.h.s. becomes the solution set.
Gauss-Jordan elimination without pivoting

- Only the second elementary operation is used.
- We reduce $A$ to identity matrix
  - The 1’s on the diagonal are the pivot positions
  - May require division operation to make the pivot element 1
- We run into trouble if we ever encounter a zero element on the diagonal
  - This is where pivoting comes in handy!
- Gauss-Jordan elimination without pivoting is numerically unstable in the presence of any roundoff error
  - Even when a zero pivot is not encountered
Pivoting

● Pivoting is interchanging rows or columns to put a particularly desirable element in the diagonal position from which the pivot is about to be selected.
● Choosing a large pivot makes it easier to reduce the rest of the rows/columns
● The pivot is usually the largest element in an unaltered row/column
Types of Pivoting

- **Partial pivoting**
  - Interchange rows only
  - Easier than full pivoting
  - Almost as good as full pivoting

- **Full pivoting**
  - Interchange both rows and columns
  - Little more complicated than partial pivoting
    - We need to keep track of the permutation of the solution vector

- **Implicit pivoting**
  - Choosing as pivot the elements which would have been largest if the original equations had all been scaled to have their largest coefficient normalized to unity
  - A variation of partial pivoting
Simple Example

Consider the following set of linear equations:

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

Or,

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix}
= 
\begin{pmatrix}
8 \\
20
\end{pmatrix}
\]
<table>
<thead>
<tr>
<th>Matrix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial matrix</td>
</tr>
<tr>
<td>2</td>
<td>Row_one /= 4</td>
</tr>
<tr>
<td>3</td>
<td>Row_zero -= 2 * Row_one</td>
</tr>
<tr>
<td>4</td>
<td>Row_zero /= -0.5</td>
</tr>
<tr>
<td>5</td>
<td>Row_one -= 0.75 * Row_zero</td>
</tr>
</tbody>
</table>
Choosing a Pivot

// Choosing a pivot
for (i=0; i<n; i++) {
    // This is the main loop over the columns to be reduced.
    big = 0.0;
    // Search for a pivot element in each column
    for (j=0; j<n; j++) {
        // This is the outer loop of the search for a pivot element.
        // Check that the column hasn't been visited
        if (ipiv[j] != 1) {
            // Now check through each member of the column
            for (k=0; k<n; k++) {
                if (ipiv[k] == 0) {
                    if (abs(a[j][k]) >= big) {
                        // Chooses the largest element on an unvisited column and row
                        big = abs(a[j][k]);
                        irow = j;
                        icol = k;
                    }
                }
            }
        }
    }
    ++(ipiv[icol]);
// Moving to diagonal
// Interchange rows to put the pivot on the diagonal
if (irow != icol) {
    // Swaps rows so that the pivot's row number and column number are equal
    for (l=0; l<n; l++)
        swap(a[irow][l],a[icol][l]);

    for (l=0; l<m; l++)
        swap(b[irow][l],b[icol][l]);

cout << "Exchanging rows " << irow << "and" << icol;
PrintMatrix(a, n);
cout << endl;
}
Normalizing row

// Divide the row by the pivot
idxr[i] = irow;
idxc[i] = icol;

if (a[icol][icol] == 0.0)
    throw("gaussj: Singular Matrix");
pivinv = 1.0 / a[icol][icol];
a[icol][icol] = 1.0;

for (l=0; l<n; l++)
    a[icol][l] *= pivinv;

for (l=0; l<m; l++)
    b[icol][l] *= pivinv;

cout << "Dividing row " << icol << " by " << abs(1.0/pivinv);
cout << endl;
PrintMatrix(a, n);
cout << endl;
Reducing Column

// Reducing column
for (ll=0; ll<n; ll++) { //Next, we reduce the rows...
    if (ll != icol) { //...except for the pivot one, of course.
        dum = a[ll][icol];
        a[ll][icol] = 0.0;

        // Subtracts multiples of the pivot row from the rows above/below
        // to make the columns mostly 0's
        for (l=0; l<n; l++)
            a[ll][l] -= a[icol][l] * dum;

        for (l=0; l<m; l++)
            b[ll][l] -= b[icol][l] * dum;

        cout << "Row " << ll << " = " << dum << " * Row " << icol;
        cout << endl;
        PrintMatrix(a, n);
        cout << endl;
    }
}
Enter the size of matrix: 3

Enter the elements of the matrix (rowwise):
\[ a[0,0] = 1 \]
\[ a[0,1] = 1 \]
\[ a[0,2] = -1 \]
\[ a[1,0] = -1 \]
\[ a[1,1] = 0 \]
\[ a[1,2] = -1 \]
\[ a[2,0] = 0 \]
\[ a[2,1] = -1 \]
\[ a[2,2] = -1 \]

1.00 1.00 -1.00
-1.00 0.00 -1.00
0.00 -1.00 -1.00

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 \( \rightarrow \) -1.00 * Row 2
-1.50 -0.50 -0.50
0.50 0.50 -0.50
0.00 1.00 -1.00

Row 2 \( \rightarrow \) 1.00 * Row 1
-1.50 -0.50 -0.50
0.50 0.50 -0.50
-0.50 -0.50 -0.50

Enter the size of vector: 1

Enter the elements of the vector:
\[ b[0,0] = 0 \]
\[ b[1,0] = -1 \]
\[ b[2,0] = -1 \]

0.00 0.00 0.00
-1.00 0.00 0.00
-1.00 0.00 0.00

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 \( \rightarrow \) 1.00 * Row 1
-1.50 -0.50 -0.50
0.50 0.50 -0.50
-0.00 1.00 -1.00

Row 1 \( \rightarrow \) 0.50 * Row 0
-0.67 0.33 0.33
0.50 0.50 -0.50
-0.50 -0.50 -0.50

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

Row 2 \( \rightarrow \) -0.50 * Row 0
-0.67 0.33 0.33
0.33 0.33 -0.67
-0.33 -0.33 -0.33

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

The solution is:
\[ b[0,0] = 0.33 \]
\[ b[1,0] = 0.33 \]
\[ b[2,0] = 0.67 \]
Storage Requirements

- The Identity matrix does not need to exist as a separate storage
- The matrix inverse of $A$ is gradually built up in $A$ as the original $A$ is destroyed
- The solution vectors $x$ can gradually replace the right-hand side vectors $b$ and share the same storage
  - After each column in $A$ is reduced, the corresponding row entry in the $b$'s is never used again
Strength

● It is straightforward
● It is as stable as any other direct method
  ○ In unstable methods, the roundoff errors get mixed into the calculation at an early stage and gets successively magnified until it comes to swamp the true answer
● It is even more stable when full pivoting is used
● For inverting a matrix it is as efficient as any other direct method
● Can produce both the solution for a set of linear equations and the matrix inverse at the same time
Limitations

- It requires all the right hand sides to be stored and manipulated (bookkeeping) at the same time, thus requires more storage.
- When the inverse matrix is not desired, it is three times slower than the best alternative technique for solving a single linear set.
Reference

- IDE: OnlineGDB