Gauss-Jordan Solution of "n x n" Linear Equations

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2007 as GuruGram #77
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Gaussian Elimination is the process of initially playing around with some array values ahead to time to greatly simplify the final solution to a large class of "n x n" linear equations. While a Jordan Further Processing often can greatly simplify any automated computer programming.

Presented here is a tutorial on Gauss-Jordan theory. Along with some remarkably simple and powerful JavaScript routines for your own Gauss-Jordan solutions. Applications include everything from Digital Filters to Magic Sinewaves.

Actual working code can be extracted from here.

Consider five linear equations in five unknowns...

\[
\begin{align*}
A_0v + B_0w + C_0x + D_0y + E_0z &= K_0 \\
A_1v + B_1w + C_1x + D_1y + E_1z &= K_1 \\
A_2v + B_2w + C_2x + D_2y + E_2z &= K_2 \\
A_3v + B_3w + C_3x + D_3y + E_3z &= K_3 \\
A_4v + B_4w + C_4x + D_4y + E_4z &= K_4 \\
\end{align*}
\]

While all sorts of solution methods exist, we seek one that is computationally efficient. If we dink around with some manipulations ahead of time, we can eventually end up with a solution that will be obvious by inspection!

Arrange the coefficients into a group of arrays...

\[
\begin{bmatrix}
A_0 & B_0 & C_0 & D_0 & E_0 & K_0 \\
A_1 & B_1 & C_1 & D_1 & E_1 & K_1 \\
A_2 & B_2 & C_2 & D_2 & E_2 & K_2 \\
A_3 & B_3 & C_3 & D_3 & E_3 & K_3 \\
A_4 & B_4 & C_4 & D_4 & E_4 & K_4 \\
\end{bmatrix}
\]

The rules for our "Gauss" part of rearrangement are that any row can be scaled by any constant term by term without changing the results. And that any row can be subtracted from any other row term by term and substituted. Again without changing the results.
In interests of sanity, let "~" be any coefficient that resulted from any and all previous manipulation. Scale the top row by dividing by its initial value...

\[
\begin{bmatrix}
1 & \sim & \sim & \sim & \sim & \sim \\
A1 & B1 & C1 & D1 & E1 & K1 \\
A2 & B2 & C2 & D2 & E2 & K2 \\
A3 & B3 & C3 & D3 & E3 & K3 \\
A4 & B4 & C4 & D4 & E4 & K4 \\
\end{bmatrix}
\]

Scale the top row by A1 and subtract it from the next row down and replacing...

\[
\begin{bmatrix}
1 & \sim & \sim & \sim & \sim & \sim \\
0 & \sim & \sim & \sim & \sim & \sim \\
A2 & B2 & C2 & D2 & E2 & K2 \\
A3 & B3 & C3 & D3 & E3 & K3 \\
A4 & B4 & C4 & D4 & E4 & K4 \\
\end{bmatrix}
\]

Similarly, scale the top row by A2 and subtract it from the middle row. Then scale by A3 for row 3 and A4 for row4...

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

Now, scale the second row down by its first nonzero coefficient...

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

Next, force zeros in the second column the same as we did with the first, but using the second row for subtraction and substitution...

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

Keep working your way through the array, this time scaling the third row down by its first nonzero term and then using scaled subtractions to zero out everything below in the same column.

Eventually, you should end up with…
This completes the Gauss part of the process. The lower right squiggle will be \( z \) by inspection!

Relabel the above array...

\[
\begin{bmatrix}
1 & c_{01} & c_{02} & c_{03} & c_{04} & j_{05} \\
0 & 1 & c_{12} & c_{13} & c_{14} & j_{15} \\
0 & 0 & 1 & c_{23} & c_{24} & j_{25} \\
0 & 0 & 0 & 1 & c_{34} & j_{35} \\
0 & 0 & 0 & 0 & 1 & z
\end{bmatrix}
\]

where \( c_{xx} \) is the row and column coefficient for the left side equation terms, and \( j_{xx} \) is the similar row and column coefficient for the right side equation term.

The traditional way to solve this was by **back substitution**. You can start off with \( y = j_{35} - z \cdot c_{34} \) and so on. And then work your way up a row at a time, making more complex calculations until you have \( v \) through \( z \) all solved.

The Jordan approach starts off the same way, but it **works one column at a time**, greatly simplifying computer programming. Especially when more than one \( n \times n \) equation set size is to be accommodated. The new rule is that any constant can be subtracted from one term in the left side of the equation as long as that same constant get subtracted from the right side of the equation.

Subtract \( z \cdot c_{34} \) from row 4...

\[
\begin{bmatrix}
1 & c_{01} & c_{02} & c_{03} & 0 & ~ \\
0 & 1 & c_{12} & c_{13} & 0 & ~ \\
0 & 0 & 1 & c_{23} & 0 & ~ \\
0 & 0 & 0 & 1 & 0 & y \\
0 & 0 & 0 & 0 & 1 & z
\end{bmatrix}
\]

So far, this is the same as the usual back substitution. We now can observe \( y \) by inspection. The difference with Jordan is to continue by **working columns** instead of rows. Modify the rows by subtracting \( z \cdot c_{24} \), \( z \cdot c_{14} \), and \( z \cdot c_{04} \) to get...

\[
\begin{bmatrix}
1 & c_{01} & c_{02} & c_{03} & 0 & ~ \\
0 & 1 & c_{12} & c_{13} & 0 & ~ \\
0 & 0 & 1 & c_{23} & 0 & ~ \\
0 & 0 & 0 & 1 & 0 & y \\
0 & 0 & 0 & 0 & 1 & z
\end{bmatrix}
\]
Next, modify column three by subtracting \( y \cdot c_{23} \), \( y \cdot c_{13} \), and \( y \cdot c_{03} \). And then column two by subtracting \( x \cdot c_{12} \) and \( x \cdot c_{02} \). And finally column one by subtracting \( w \cdot c_{01} \) to get...

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & v \\
0 & 1 & 0 & 0 & 0 & w \\
0 & 0 & 1 & 0 & 0 & x \\
0 & 0 & 0 & 1 & 0 & y \\
0 & 0 & 0 & 0 & 1 & z \\
\end{bmatrix}
\]

Your values \( v \) through \( z \) are now instantly readable by inspection!

Once again, the Jordan method takes just as many calculations as does a back substitution, but it greatly simplifies computation. In that loops do not have any multiple calculations or complicated cross-coefficients in them. This is especially handy when it comes to making the code \( n \) independent.

**A Code Example**

Here’s a JavaScript program that solves \( n \times n \) linear equations. It is amazingly compact, offers 64 bit arithmetic, and works for most any sane value of \( n \). But it does not trap any div0’s or handle wild coefficients. Per this main proc…

```javascript
function solveGaussJordan() {
    gjNsize = eqns.length;
    for (var iii = 0; iii <= (gjNsize-1); iii++) {
        normalize (eqns[iii], iii);
        for (var jjj = iii; jjj <= (gjNsize-2); jjj++) {
            subScaled (eqns[iii], eqns[(jjj+1)], iii);
        }
    }
    normalize (eqns[(gjNsize-1)], (gjNsize-1));
    jordanify();
}
```

It needs these three support subs...

```javascript
function normalize (bb, cc) { xx = bb[cc];
    for (var ii = 0; ii <= gjNsize; ii++) {
        bb[ii] = (bb[ii]/xx);
    }
}

function subScaled (aa, bb, cc) { xx = bb[cc];
    for (var ii = cc; ii <= gjNsize; ii++) {
        bb[ii] -= aa[ii]*xx;
    }
}
```
function jorDanify() {
    for (var i3 = (gjNsize-1); i3 >=1; i3--){
        zz = eqns[i3][gjNsize] ;
        for (var i4 = (i3-1); i4 >=0 ; i4--)
            eqns[i4][gjNsize] -= eqns[i4][i3]*zz
        eqns[i4][i3] = 0 } } ;

And here is how you would use it...

```javascript
eq0 = [4, 3, -2, 1, 22]  eq1 = [2, 1, -2, 2, 9]  
eq2 = [1, -1, 1, 5, 8]  eq3 = [3, 1, 3, 1, 22]  
eqns = [eq0, eq1, eq2, eq3] ;
solveGaussJordan () ;
```

`eq0` represents $4w + 3x - 2y + z = 22$. There is an implicit equals sign before the rightmost column.

Reals as well as integers can be used. Processing time increases sharply with increasing `n`. But is well under one second for `n = 30x30`.

Returned via Gauss-Jordan elimination is ...

```javascript
eq0 = [1, 0, 0, 0, w]  
eq1 = [0, 1, 0, 0, x]  
eq2 = [0, 0, 1, 0, y]  
eq3 = [0, 0, 0, 1, z]  
```

...and for the above example, $w=4$, $x=3$, $y=2$ and $x=1$.

**For Additional Assistance**

Similar tutorials and additional support materials are found on our PostScript, our Math Stuff, our Magic Sinewave, and our GurGram library pages.

As always, Custom Consulting is available on a cash and carry or contract basis.
As are seminars.

For details, you can email don@tinaja.com. Or call (928) 428-4073.