Programming Gaussian Elimination

Carlos M. Torres

April 4, 2014

1 Introduction

Gaussian Elimination is a general algorithm applied to matrices in order to reduce them to row echelon form. This method is particularly useful when we want to solve systems of linear equations that can be written on the form $A\vec{x} = \vec{b}$. This particular paper will present this method when a applied to a system $A\vec{x} = \vec{b}$, where A is a $n\,x\,n$ matrix and $\vec{b} \in \mathbb{R}^n$. The pseudo-code that will be presented will be applied to the augmented matrix of this system, which will have dimension $[n, (n+1)]$.

2 Gaussian Elimination

2.1 Classification of a Row Echelon From Matrix

First, it is imperative to define what it means for a matrix to be in row echelon form. There are three conditions that must be satisfied: [1]

1. All nonzero rows are above any toes of al zeros.
2. Each leading entry of a row is in a column to the right of the leading entry above it.
3. All entries in a column below a leading entry are zeros.

   If a matrix satisfies the following two conditions, in addition to the ones stated above, then it is said to be in reduced row echelon form.
4. The leading entry in each nonzero row is 1.
5. Each leading 1 is the only nonzero entry in its column.

To illustrate these conditions, below are two matrices that are in row echelon form.

$$\begin{bmatrix} \blacksquare & * & * & * \\ 0 & \blacksquare & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & \blacksquare & * & * & * & * & * & * & * & * \\ 0 & 0 & 0 & \blacksquare & * & * & * & * & * & * \\ 0 & 0 & 0 & 0 & \blacksquare & * & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & \blacksquare & * & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \blacksquare & * \end{bmatrix}$$

For the previous examples the black square represents any real number. If these two matrices were instead in reduced row echelon form then they would look like this.

$$\begin{bmatrix} 1 & 0 & * & * \\ 0 & 1 & * & * \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 & * & 0 & 0 & 0 & * & * & 0 & * \\ 0 & 0 & 0 & 1 & 0 & 0 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 1 & 0 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & 1 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & * \end{bmatrix}$$

2.2 The Algorithm (Reduce to Upper Diagonal Matrix)

Here is when we thing of the algorithm while trying to assimilate as closely as possible the perspective of a computer scientist. We will work with the augmented matrix of the system $A\vec{x} = \vec{b}$, which has the following form:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & a_{1,n+1} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & a_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & a_{n,n+1} \end{bmatrix} \tag{2.1}$$

Where the elements of the last column represent our vector $\vec{b}$. The first step of this algorithm should be to reduce the first element of the second row to zero by means of row operations. To achieve this, I propose the following operation:

$$Row2 = Row2 - Multiplier * Row1$$

Now lets apply this operation to the first element of our second row, which is also our first element below our first pivot position.

$$a_{2,1}^{'} = a_{2,1} - Multiplier * a_{1,1}$$

Notice that in order to make our $a_{2,1}^{'}$ zero our multiplier must be equal to $\frac{a_{2,1}}{a_{1,1}}$. So our new element below the pivot position is:

$$a_{2,1}^{'} = a_{2,1} - \left(\frac{a_{2,1}}{a_{1,1}}\right) a_{1,1} = 0 \tag{2.2}$$

This operation, of course, must be carried out to all the elements of our second row. We can, therefore, express our new second row in the following form:

$$a_{2,i}^{'} = a_{2,i} - \left(\frac{a_{2,1}}{a_{1,1}}\right) a_{1,i} \;\; i = 1, 2, \cdots, n+1 \tag{2.3}$$

This specific operation will only make the first element below the pivot zero, but we must perform the same operation on all the elements below the pivot. Furthermore, our pivots

change as we move though our diagonal. While this may seem complicated to visualize initially, it gets easier to grasp as more though is enacted on the problem.

In programming lingo, the algorithm will result in a triple loop. The first loop will move though the diagonals, the second loop will move through the rows, and the third will move through all the columns. To visualize this lets continue the logic that was implemented previously. Now suppose we want to make the first element in our third row zero by means of the same login in equation (2.2). Then our first element of our third row would become:

$$a_{3,1}^{'} = a_{3,1} - \left(\frac{a_{3,1}}{a_{1,1}}\right) a_{1,1} = 0 \tag{2.4}$$

When applying this to all the elements of the third row we end up with an expression very similar to equation (2.3)

$$a_{3,i}^{'} = a_{3,i} - \left(\frac{a_{3,1}}{a_{1,1}}\right) a_{1,i} \;\; i = 1, 2, \cdots, n+1 \tag{2.5}$$

I want to expand this concept to all the elements below the pivot position, not just the second and third row. Notice that the only thing that changes between equation 2.3 and equation (2.5) is the index corresponding to the row number. As a programmer this is analogous to another looping opportunity. Therefore, I can make all the elements below the first pivot position zero with the following formula.

$$a_{k,i}^{'} = a_{k,i} - \left(\frac{a_{k,1}}{a_{1,1}}\right) a_{1,i} \quad i = 1, 2, \cdots, n+1; \quad k = 2, 3, \cdots, n \tag{2.6}$$

The k loop will move through our rows, and the i loop will move across the columns. Yet our formula is still not perfect. Equation (2.7) will only work for the first pivot position, meaning, it will only work for the first diagonal element in our matrix. We must create a formula that will work for all the pivot positions. This is where the third loop comes in. This third loop encompasses the other two and determines the starting point for our k and i loop. Why? Because if we find ourselves in any arbitrary pivot position in our matrix we only want to modify the elements below that pivot position, not any element above it. The derivation of the following equation is left as an exercise to the reader, but it follows the same logic as all the previous equations. Essentially, write a few formulas for any arbitrary elements and then identify patterns, which in turn correspond to looping opportunities. Our equation for our Gaussian Reduction Algorithm is:

$$a_{k,i}^{'} = a_{k,i} - \left(\frac{a_{k,j}}{a_{j,j}}\right) a_{j,i} \qquad j = 1, 2, \cdots, n-1; \quad k = j+1, j+2, \cdots, n; \quad i = j, j+1, \cdots, n+1 \tag{2.7}$$

Try to look at the formula and understand it. Notice that in order to determine k and i a value of j must be selected first, which is what determines the current diagonal element (or pivot position). If the previous algorithm is implemented successfully then you should end with an upper diagonal matrix of the following form, with all the elements below the diagonal zero:

$$\begin{bmatrix} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} & a'_{1,n+1} \\ 0 & a'_{2,2} & \cdots & a'_{2,n} & a'_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{n,n} & a'_{n,n+1} \end{bmatrix} \tag{2.8}$$

2.3 The Algorithm (Find the Solution)

The objective of this section is to introduce the algorithm that will reduce our upper diagonal matrix to row reduced echelon form, as described by all five conditions given in section 2.1. Now lets assume we start with a matrix of the form:

$$\begin{bmatrix} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} & a'_{1,n+1} \\ 0 & a'_{2,2} & \cdots & a'_{2,n} & a'_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{n,n} & a'_{n,n+1} \end{bmatrix}$$

Which is essentially the matrix obtained after we implemented the algorithm described in section 2.2. When we have a matrix of this form we have two options for obtaining our solution: to perform backward substitution, or to continue to reduce our matrix to row reduced echelon form. In this subsection only the latter is discussed. We are going to reduce the matrix in equation 2.8 to the following:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & b_1 \\ 0 & 1 & \cdots & 0 & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & b_3 \end{bmatrix} \tag{2.9}$$

The though process that will be used to develop this algorithm will be very similar to the one followed in section 2.2. To obtain the matrix described previously we must again employ a triple loop. The first loop will start at the last element of the diagonal. The second, will travel through the rows to make the elements above the diagonal zero, and the third, will move across the columns to modify our solution vector according to our multiplier. There are a few ways of initiating this algorithm, but I will start by scaling the last row to obtain:

$$\begin{bmatrix} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} & a'_{1,n+1} \\ 0 & a'_{2,2} & \cdots & a'_{2,n} & a'_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \frac{a'_{n,n+1}}{a'_{n,n}} \end{bmatrix}$$

Now, assume we find ourselves in the last element of the diagonal. Then in order to make the elements above the diagonal zero we must employ:

$$a''_{n-1,n} = a'_{n-1,n} - (a'_{n-1,n})(1) = 0$$

$$a''_{n-2,n} = a'_{n-2,n} - (a'_{n-2,n})(1) = 0$$

$$a''_{l,n} = a'_{l,n} - (a'_{l,n})(1); \quad l = (n-1), (n-2), \cdots, 1$$

Notice that in each of these cases the multiplier is simply the element above the diagonal that we are trying to make zero (only due to scaling first). This is the same multiplier that will be used to move across the columns to modify the rest of the elements in that row. For example, $a''_{l,n+1} = a'_{l,n+1} - a'_{l,n}(a'_{n,n+1})$, which in this case would represent the last column of our matrix since we started in the last element of our diagonal. Note that all the elements of our matrix are being constantly rewritten by the program, as a result, on our formulas it's much more important to keep track of the position of our elements rather than the exact value of the elements themselves.

If for any given row element we continue to the last column our elements would be

$$a''_{l,n+1} = a'_{l,n+1} - a'_{l,n}(a'_{n,n+1})$$

$$a''_{l,i} = a'_{l,i} - a'_{l,n}(a'_{n,i}); \quad l = (n-1), (n-2) \cdots, 1; \quad i = n, (n+1) \tag{2.10}$$

Now lets suppose we move to the second to last element of our diagonal, then our formulas would become:

$$a''_{n-2,n-1} = a'_{n-2,n-1} - (a'_{n-2,n-1})(1) = 0$$

$$a''_{n-3,n-1} = a'_{n-3,n-1} - (a'_{n-3,n-1})(1) = 0$$

$$a''_{l,n-1} = a'_{l,n-1} - (a'_{l,n-1})(1); \quad l = (n-2), (n-3) \cdots, 1 \tag{2.11}$$

If for any given row element we continue through the column our elements would be

$$a''_{l,n} = a'_{l,n} - a'_{l,n-1}(a'_{n-1,n})$$

$$a''_{l,n+1} = a'_{l,n+1} - a'_{l,n-1}(a'_{n-1,n+1})$$

$$a''_{l,i} = a'_{l,i} - a'_{l,n-1}(a'_{l,i}); \quad l = (n-2), (n-3) \cdots, 1; \quad i = (n-1), n, (n+1) \tag{2.12}$$

For all these formulas we fixed ourselves on a particular diagonal element, but if we introduce one more looping variable that allows us to move through the diagonal, then we will be able

to derive the formula that describes the algorithm. Notice also that the ranges of the l loop and the i loop also depend on which diagonal element we find ourselves in, therefore, the diagonal element loop, which I will denote by k, must be the first loop in the series. By looking at equations 2.10 and 2.12 we can notice that the only index dependent only on the diagonal element is found on the multiplier. So our general formula is:

$$a''_{l,i} = a'_{l,i} - a'_{l,k}(a'_{k,i}); \quad k = n, n-1, \cdots, 1 \quad ; l = (k-1), (k-2) \cdots, 1; \quad i = k, \cdots, (n+1) \tag{2.13}$$

If you employ this formula successfully then you will end up with matrix 2.9.

3 The Pseudo-code for Gaussian Elimination

The pseudo-code in this section will closely follow all the formulas derived in our previous sections. Comments on the pseudo code will be enclosed in the following brackets: /** **/ (Like in C).

==========Begin Pseudo-Code==========

/** Once the data has been read we initialize the Gaussian Elimination algorithm on our matrix A. For this algorithm partial pivoting will be used in order to minimize round-off errors in our calculations. **/

```
DO LOOP (j=1 TO n-1) /** This loop moves along the diagonal from left to right **/
    /** Initiate partial pivoting **/
    IF (A(j,j) == 0)
        A(j,j)=0
        max=0
        Index=j
        /** Find the maximum value in the column **/
        DO LOOP (i=j+1 TO n)
            IF (abs(A(i,j) ¿ max)
                max=abs(A(i,j))
                Index=i
            END IF
        END DO LOOP
        /** Switch the rows corresponding to the maximum value only if the index changed **/
        IF (Index != j)
            DO LOOP (i=1 TO n+1)
                temp(i)=A(j,i)
                A(j,i)=A(Index,i)
                A(Index,i)=temp(i)
            END DO LOOP
        END IF
```

```
        END IF
        /** End Partial Pivoting **/

        /** Continue after partial pivoting. The following loop will guarantee that all the
rows are covered below the pivot point. **/
        DO LOOP (k=j+1 TO n)
            /** Calculate the Multiplier to make the elements below the pivot 0 **/
            Multiplier=A(k,j)/A(j,i)
            /** Continue with all the elements in the row **/
            DO LOOP (i=j TO n+1)
                A(k,i)=A(k,i)-Multiplier*A(j,i)
            END DO LOOP
            /** Elements below the pivot are always zero. This is done in order to minimize
roundoff error **/
            A(k,j)=0
        END DO LOOP
    END DO LOOP

    /** Now that our matrix is in echelon form we will continue to reduce it to reduced row
echelon form, as described in section 2.3. After implementing this algorithm we will be able
to read of our solution. **/

    /** Start with the furthest diagonal element and move towards the first (k-loop) **/
    DO LOOP (k=n to 1)
        /** Make the pivot position 1 by scaling. Perform this scaling only if the pivot
position has a non-zero element **/
        IF (A(k,k) != 0)
            Divisor=A(k,k)
            DO LOOP (i=k TO n+1)
                A(k,i)=A(k,i)/Divisor
            END DO LOOP
        END IF
        /** Make all the elements above the pivot position 0 by performing the appropriate
row operations **/
        IF ( (A(k,k) != 0) AND (k != 1) )
            /** Make sure all the rows will be covered **/
            DO LOOP (l=k-1 TO 1)
                Multiplier=A(l,k) /** Calculate the multiplier **/
                /** Make sure all the columns are covered **/
                DO LOOP (i=k TO n+1)
                    A(l,i)=A(l,i)-Multiplier*A(k,i)
                END DO LOOP
```

```
        END DO LOOP
    END IF
END DO LOOP

====================End Pseudo-Code====================
```

If you implemented this pseudo-code correctly then your matrix A should contain an readable solution as given by equation 2.9. To implement this code it's not necessary to understand each individual line. If written with correct syntax on your programming language of choice it should correctly determine the answer.

4 Conclusion

There were a few assumptions when developing this code and it has been proven to not work for special cases. If your system of equations has two or more repeated equations, then this algorithm might not work, but it has been proven to be effective if your system is linearly independent, and if you have only one free variable in your system. If you have any questions about the pseudo-code, or if you want a sample code written in FORTRAN, you can contact the author of this paper at the email: carlos.torres.rivera@gmail.com.

References

[1] David C. Lay. *Linear algebra and its applications*. Pearson/Addison-Wesley.