Gauss Elimination and Back Substitution

Thomas Moawad

April 2020

1 Introduction

This report will provide an explanation of Gauss elimination, and back substitution. Before discussing Gauss elimination it is also important to understand LU factorization. After explaining these concepts, this paper will discuss examples of how to solve systems of equations by applying LU factorization, Gauss elimination, and back substitution. Throughout this paper, example code will be provided.

1.1 Syntax

- Let M_r = The rth row of matrix M
- Let $M_{r,\,c}$= The element in matrix M at row r and column c

2 Gauss Elimination

2.1 Importance

Gauss elimination is important because it produces an upper triangular matrix. This triangular matrix will become useful for LU factorization and for Back Substitution.

2.2 General Pattern of an Upper Triangular Matrix

Before diving into Gauss elimination, we must discuss the general pattern of an upper triangular matrix of size m by m. Let's start by looking at an upper triangular matrix of size 4 by 4.

$$U = \begin{bmatrix} * & * & * & * \\ 0 & * & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{bmatrix}$$

First, we can recognize that every element underneath the diagonal is equal to zero 0. In other words, we can recognize the following pattern:

- For column 1, all elements between $U_{2,\ 1}$ to $U_{4,\ 1}$ are equal to 0
- For column 2, all elements between $U_{3,\ 2}$ to $U_{4,\ 1}$ are equal to 0.
- For column 3, $U_{4,\ 3}$ is equal to 0

This pattern can be generalized for an upper triangular matrix of any size. We can conclude this section with the following definition:

Definition of Upper Triangular Matrix

Upper Triangular Matrix: A matrix U of size m by n such that for each column_i, all elements between $U_{i\ +\ 1,\ i}$ to $U_{m,\ i}$ (inclusive) are equal to 0

2.3 Make the Value of Any Position 0

Recall from the introduction that Gauss elimination produces an upper triangular matrix. Hence, we must convert matrix of size m by n into "a matrix of size m by n such that for each column_i, all elements between $U_{i\ +\ 1,\ i}$ to $U_{m,\ i}$ (inclusive) are equal to 0."

First, for any matrix M, we must determine how to intentionally make an arbitrary element $M_{r,\ c}$ equal to 0. There are two cases that determine how to perform this task.

- **Case 1 - Row Swap:**

  If $M_{r\ +\ 1,\ i} = 0$ then we can swap row_r and $\text{row}_{r\ +\ 1}$

  Example:

  - We start with by $M_{2,\ 1} = 100$

$$\begin{bmatrix} 1 & 2 & 3 \\ 100 & 4 & 6 \\ 0 & 7 & 8 \end{bmatrix}$$

  - We notice that $M_{3,\ 1} = 0$
  - We recognize that this fits <u>case 1</u>
  - Hence, we know to swap row_2 and row_3

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 7 & 8 \\ 100 & 4 & 6 \end{bmatrix}$$

  - Now $M_{2,\ 1} = 0$

- **Case 2 - Row Replacement:**

  If $M_{r\ -\ 1,\ c} \neq 0$ then replace row_r with $row_r{}^{new}$ such that

$$row_r{}^{new} = row_r - k * row_{r\ -\ 1} \tag{1}$$

  where we find k with

$$k = \frac{M_{r,\ c}}{M_{r\ -\ 1,\ c}} \tag{2}$$

  When observing the value of $M_{r,\ c}{}^{w}$e will see that this replacement will result with

$$M_{r,\ c}{}^{new} = M_{r,\ c}\ -\ k\ *\ M_{r\ -\ 1,\ c}$$

$$= M_{r,\ c}{}^{new} = M_{r,\ c}\ -\ \frac{M_{r,\ c}}{M_{r\ -\ 1,\ c}}\ *\ M_{r\ -\ 1,\ c}$$

$$= M_{r,\ c}{}^{new} = M_{r,\ c}\ -\ M_{r,\ c}$$

$$= M_{r,\ c}{}^{new} = 0$$

  Example:

  - We start with by M[2, 1] = 100

$$\begin{bmatrix} 1 & 2 & 3 \\ 100 & 4 & 6 \\ 2 & 7 & 8 \end{bmatrix}$$

  - We notice that the M[3, 1] = 2 ≠ 0
  - We recognize that this fits <u>case 2</u>
  - Hence, we know we must find a replace row_2 with $row_2{}^{new}$ by using

$$row_2{}^{new} = row_2\ -\ k\ *\ row_1$$

  - And we find k by:

$$k = \frac{M_{2,\ 1}}{M_{1,\ 1}} = \frac{1}{100}$$

  - After this replacement we finish with

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 0.04 & 0.06 \\ 2 & 7 & 8 \end{bmatrix}$$

Make the Value of Any Position 0

If $M_{r\ -\ 1,\ c} \neq 0$ then perform a row swap (see section 2.3 case 1). Otherwise, perform a row replacement (see section 2.3 case 2)

2.4 Gauss Elimination Procedure

Let's Recap: In section 2.2 we defined an upper triangular matrix. In section 2.3 we recognized that to fulfill this definition, we must know how to set a value of a matrix to zero. We then determined how to do just that! We now have all the tools we need to systematically perform Gauss Elimination:
Start with matrix M

Gauss Elimination Procedure

> For each column_i in M, use row swapping or row operations such that all elements underneath $M_{i,\,i} = 0$

2.5 Python Implementation

In Section 2 of the Jupter notebook the following code is used to perform Gauss Elimination

```python
def gauss_elimination(matrix):
    matrix = matrix.astype(float)
    U = copy.deepcopy(matrix)

    for i in range(len(U.T)): # for each column
        U = zeros_underneath(U, i, i) #make everything 0 underneath the diagonal

    return U
```

Furthermore, in Section 2.1 of the Jupyter notebook, the following code is used to ensure zeros underneath the diagonal

```python
"""
Creates a matrix such that all elements within the same column as a selected
    index is zero

Args:
    matrix: numpy matrix of dimensions m rows by n columns
    row_index: The row index of the selected element
    col_index: The column index of the selected element.

Returns:
    A new array such that elements from matrix.getitem(row_index + 1, col_index)
        to matrix.getitem(m, col_index) = 0
"""
def zeros_underneath(matrix, row_index, col_index):
    matrix2 = copy.deepcopy(matrix)
    row_j = row_index #use the row above

    for row_i in range(row_index + 1, len(matrix), 1): # for each row underneath row_i
        if matrix2.item(row_j, col_index) == 0:
            matrix2 = row_swap(matrix2, row_i, row_j)
        else:
            k = matrix.item(row_i, col_index) / matrix2.item(row_j, col_index)
            matrix2 = row_replacement(matrix2, row_i, 1, row_j, k)
    return matrix2
```

3 Back Substitution

3.1 Importance

Recall that in Gauss elimination we produced a triangular matrix, U. In Back Substitution we manipulate this triangular matrix to solve the the general equation Ux=b

3.2 General Pattern of Back Substitution

Given upper triangular matrix U, and column b, we want to find x such that

$$\mathrm{Ux=b}$$

When expanded, this equation looks like:

$$\begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

We can quickly recognize that this provides us with the following system of equations:

1. $\mathrm{U_{1,1} \times x_1 + U_{1,2} \times x_2 + U_{1,\,3} \times x_3 = b_1}$
2. $\mathrm{U_{2,2} \times x_2 + U_{2,3} \times x_3 = b_2}$
3. $\mathrm{U_{3,3} \times x_3 = b_3}$

How convenient! There are 3 equations and 3 unknowns. What makes this even more convenient is that if we solve these equations from last to first, then each equation is of the form

$$equation_{\mathrm{r}} : constant + U_{\mathrm{r,\,r}} \times x_{\mathrm{r}} = b_{\mathrm{r}} \tag{3}$$

General Pattern of Back Substitution

Therefore, given U of r rows, if we start from the $\mathrm{equation_r}$ and work our way up to $\mathrm{equation_1}$, then we can easily solve for $\mathrm{x_r}$ as

$$\mathrm{x_r} = \frac{b_{\mathrm{r}} - constant}{U_{\mathrm{r,\,r}}}$$

3.3 Exceptions to The General Pattern of Back Substitution

In section 3.2 we concluded that we can solve each equation by starting from the last equation and work our way up to the first equation, by using:

$$\mathrm{x_r} = \frac{b_{\mathrm{r}} - constant}{U_{\mathrm{r,\,r}}}$$

However, this will not hold true if $U_{r,\,r} = 0$ because it is illegal to divide by zero. $U_{r,\,r} = 0$ then equation_r can be resolved as follows:

$$\text{constant} + U_{r,\,r} \times x_r = b_r$$

$$= \text{constant} + 0 \times x_r = b_r$$

$$= \text{constant} = b_r$$

Since constant, and b_r are both constants, we can perform a "reality check" with the following cases

- **Case 1:** If constant $= b_r$ then x_r is a free variable, and there are infinite solutions. In mathematical terms, this is because our system of equations had less pivots than rows. Hence, we have a free column which corresponds to a free variable.
- **Case 2:** If constant $\neq b_r$ the system of equations has no solution. This should make intuitive sense. If I told you that $1 = 2$, or that $30 = 25$, or that $\text{constant}_A = \text{constant}_B$ you would conclude that I made an error. In this case, the error is the system of equations which has no solution.

Exceptions to Back Substitution

If $U_{r,\,r} = 0$ then one of the two exceptions hold true.

- **Exception 1:** $A = b_r$. You can set x_r to any number you want. Hence, there are infinite solutions.
- **Exception 2:** $A = b_r$ in which case there are no solutions

3.4 Back Substitution Procedure

Let's recap: In section 3.2 we found the main pattern of back substitution. In section 3.3 we found some exceptions to this pattern. We now have all the tools to perform back substitution.

Back Substitution Procedure

Therefore, given U of r rows, if we start from the equation$_r$ and work our way up to equation$_1$, then we can easily solve for x_r as

$$x_r = \frac{b_r - constant}{U_{r,\,r}}$$

If at any point you find that $U_{r,\,r} = 0$ then one of the two exceptions hold true.

- **Exception 1:** $A = b_r$. You can set x_r to any number you want. Hence, there are infinite solutions.
- **Exception 2:** $A = b_r$ in which case there are no solutions

3.5 Python Implementation

In Section 3 of the Jupyter Notebook, we can implement back substitution with the following code.

```
"""
Solves a system of equations of the form Ux=b

Args:
    U: numpy matrix upper triangular matrix of dimensions m rows by n columns
    b: numpy matrix of dimensions n rows by 1 column

Returns:
    The x matrix such that Ux = b. If there are free variables it will be indicated by '?'
"""
#THIS ONE WORKS
def back_substitution(U, b):
    U = U.astype(float)
    b = b.astype(float)
    x = [0.0] * len(U)
    is_free_variable_found = False

    for i in range(len(x) - 1, -1, -1): # substitute from the bottom of the matrix up
        row = U[i].flatten()
        constant = expanded_dot(row, x)
        if (is_free_variable_found):
            x[i] = "( " + str(b[i].item()) + " - (" + str(constant) + ") ) " + " / " + str(row.item(i))
        else:
            if (row.item(i) == 0): #We might have found a free variable
                if (reality_check(constant, b.item(i))): # reality check
                    is_free_variable_found = True
                    x[i] = "?"
                else:
                    return "There is no solution"
            else:
                x[i] = (b[i].item() - constant) / row.item(i) # Solve for the unknown

    if is_free_variable_found:
        return (resolve_free_variables(x, 1.0), resolve_free_variables(x, -1.0))
    return x
```

4 LU Factorization

4.1 Importance

Recall that in arithmetic we learned to factor scalars such as 10 = 2 * 5. Furthermore, in algebra we learned to factor polynomials such as x^2 + 2x - 3 = (x - 1) (x + 3). The same holds for linear algebra. In this section we will look at how to factor a square matrix A such that A = LU

4.2 General Pattern of L in LU Factorization

Assume we know A. Consider the expansion of LU = A

$$\begin{bmatrix} a & 0 \\ b & c \end{bmatrix} \begin{bmatrix} d & e \\ 0 & f \end{bmatrix} = \begin{bmatrix} g & h \\ i & j \end{bmatrix}$$

Let's expand this matrix multiplication into a system of equations.

1. ad = g
2. ae = h
3. bd = i
4. be + cf = j

There are four equations but 6 unknowns (a, b, c, d, e, f). Let's set the diagonals of L to be 1. That is a = 1 and c = 1.

1. d = g
2. e = h
3. bd = i $\implies$ bg = i $\implies$ b = i/g
4. be + f = j $\implies$ $\frac{i}{g}$h + f = j $\implies$ f = j - ih/g

Suddenly we have solved all our unknowns. Thus for a 2 by 2 LU=A can be expanded to:

$$\begin{bmatrix} 1 & 0 \\ i/g & 1 \end{bmatrix} \begin{bmatrix} g & h \\ 0 & j - ih/g \end{bmatrix} = \begin{bmatrix} g & h \\ i & j \end{bmatrix}$$

Finally, recall that a row replacement is

$$\text{row}_\text{r}^{\text{new}} = \text{row}_\text{r} - \text{k} * \text{row}_{\text{r - 1}}$$

where we find k with

$$\text{k} = \frac{A_{\text{r, c}}}{A_{\text{r - 1, c}}}$$

Then in this case, if we wanted to find U we could perform a row replacement on row_2. When doing this we would find that

$$\mathrm{k} = \frac{A_{1,\,2}}{A_{1,\,1}} = \mathrm{i/g}$$

Fascinating! k=i/g is also found in L. We can substitute this to find:

$$\begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix} \begin{bmatrix} g & h \\ 0 & j - ih/g \end{bmatrix} = \begin{bmatrix} g & h \\ i & j \end{bmatrix}$$

In other words, the k that we used to eliminate eliminate $\mathrm{A}_{1,\,2}$ is the same value that belongs in $\mathrm{L}_{1,\,2}$. Without loss of generality we can expand idea this to an n by n matrix. If we performed a row swap then k = 0.

Pattern of L in LU Factorization:

For matrix A of size n by n

- The value of any element on the diagonal is 1
- The value any element $\mathrm{e}_{\mathrm{r},\,\mathrm{c}}$ underneath the diagonal is k
- where k =

$$\begin{cases} \frac{A_{\mathrm{r},\,\mathrm{c}}}{A_{\mathrm{r}\,-\,1,\,\mathrm{c}}} & A_{\mathrm{r}\,-\,1,\,\mathrm{c}} \neq 0 \\ 0 & else \end{cases}$$

A matrix of size m by n such that for each $\mathrm{column}_{\mathrm{i}}$, all elements between $\mathrm{U}_{\mathrm{i}\,+\,1,\,\mathrm{i}}$ to $\mathrm{U}_{\mathrm{m},\,\mathrm{i}}$ (inclusive) are equal to 0

4.3 LU Factorization Procedure

Let's recap. In section 2 we discussed how to find U. In section 4.2 we discussed how to find L. We now have all the tools we need for LU factorization!

LU Factorization Procedure

For a square matrix A

- Create U by performing row swaps or row operations such that all elements underneath $\mathrm{A}_{\mathrm{i},\,\mathrm{i}} = 0$
- Create L by starting with an identity matrix
  - If you performed a row swap to get $\mathrm{U}_{\mathrm{i},\,\mathrm{j}} = 0$ then $\mathrm{L}_{\mathrm{i},\,\mathrm{j}} = \mathrm{k}$
  - else $\mathrm{L}_{\mathrm{i},\,\mathrm{j}} = 0$

4.4 Python Implementation

In section 3 of the Jupyter notebook, we can find an upper triangular matrix and a lower triangular matrix at the same time.

This code block will iterate through each column of A to find the corresponding columns of L and U

```
"""
Convert a matrix into it's upper triangular form
"""
def LU_factorization(matrix):
    matrix = matrix.astype(float)
    upper = copy.deepcopy(matrix)
    lower = np.identity(len(matrix))

    for i in range(len(matrix.T)): # for each column
        result = __upper_lower_column_helper(upper, i, i, lower) #Make the column of U and the column of L
        upper = result[0]
        lower = result[1]

    return lower, upper
```

This code block will do the heavy lifting for the prior code block. This code block keeps track of the k values used to make a column of L and U.

This function **upper_lower_column_helper** simply performs the upper triangular operatin while keeping track of these k values

```
"""
Finds the upper triangular matrix while
also keeping track of the k values to use in the lower triangular matrix
"""
def __upper_lower_column_helper(matrix, row_index, col_index, lower):
    matrix2 = copy.deepcopy(matrix)
    row_j = row_index #use the row above

    k=0
    for row_i in range(row_index + 1, len(matrix), 1): # for each row underneath row_i
        if matrix2.item(row_j, col_index) == 0:
            matrix2 = row_swap(matrix2, row_i, row_j)
        else:
            k = matrix.item(row_i, col_index) / matrix2.item(row_j, col_index)
            matrix2 = row_replacement(matrix2, row_i, 1, row_j, k)
        lower[row_i][col_index] = k

    return matrix2, lower
```

5 Solving Ax=b

Let's recap. In section 2 we determined how to use Gauss elimination to produce an upper triangular matrix. In section 3 we determined how to use back substitution on an upper triangular matrix. We now have all the tools we need to solve the linear equation Ax=b

Procedure for Solving Ax=b

Given matrix A and column b, to solve the system of equations such that Ax=b

1. Create an augmented matrix, Aug such that Aug =[A—b]
2. Use Gauss elimination to convert Aug into U
3. Use Back substitution on U to solve for x
    - If Back substitution has a free variable, then there are infinite solutions. Pick any value for the free variable
    - If Back substitution fails the logic check, then there are no solutions to to this Ax=b system

5.1 Python Implementation

We already implemented Gauss elimination and back substitution in the earlier sections of this paper. Now it will only take a few lines of code to solve Ax=b. This can be seen in Section 5 of the Jupyter notebook.

```
"""
Solves the equation Ax = b

Args:
    A: numpy matrix of dimensions m rows by n columns
    b: numpy matrix of dimensions n rows by 1 column

Returns:
    The x matrix such that Ax = b
"""
def gauss_elimination_and_back_substitution(A, b):
    # First, convert to floats. This helps avoid integer rounding
    A = A.astype(float)
    b = b.astype(float)

    augmented = np.hstack((A, b)) # Convert from A, b to [A|b]
    U = gauss_elimination(augmented)
    U, b = np.hsplit(U, [np.size(U, 1) - 1]) # Convert from [U|b] to U, b
    x = back_substitution(U, b)

    return x
```

6 Examples of Gauss Elimination and Back Substitution

Throughout this section we will be using the python code that we implemented to solve a few examples. In each example we will solve for Ax=b and A=LU. This section corresponds to section 6 of the Jupyter notebook.

We will use a handy function "print solution" to print out the solution exactly how we want it.

```
def print_solution(A, b):
    print("solution = ")
    print(gauss_elimination_and_back_substitution(A, b))
    print("")
    L, U = LU_factorization(A)
    print("L = ")
    print(L)

    print("U = ")
    print(U)
```

Problem a

$$A = \begin{bmatrix} 1 & -1 & 2 & -1 \\ 2 & -2 & 3 & -3 \\ 1 & 1 & 1 & 0 \\ 1 & -1 & 4 & 3 \end{bmatrix}$$

$$b = \begin{bmatrix} -8 \\ -20 \\ -2 \\ 4 \end{bmatrix}$$

If we run this through the python code we see that:

a.

```
A = np.matrix([[1, -1, 2, -1],
               [2, -2, 3, -3],
               [1, 1, 1, 0],
               [1, -1, 4, 3]])

b = np.matrix([[-8],
               [-20],
               [-2],
               [4]])

# Solution is [-7, 3, 2, 2]
print_solution(A, b)
```

```
solution =
[-7.0, 3.0, 2.0, 2.0]

L =
[[ 1.  0.  0.  0.]
 [ 2.  1.  0.  0.]
 [ 1.  0.  1.  0.]
 [ 1.  0. -2.  1.]]
U =
[[ 1. -1.  2. -1.]
 [ 0.  2. -1.  1.]
 [ 0.  0. -1. -1.]
 [ 0.  0.  0.  2.]]
```

Therefore:

$$Ax = \begin{bmatrix} 1 & -1 & 2 & -1 \\ 2 & -2 & 3 & -3 \\ 1 & 1 & 1 & 0 \\ 1 & -1 & 4 & 3 \end{bmatrix} \begin{bmatrix} -7 \\ 3 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -8 \\ -20 \\ -2 \\ 4 \end{bmatrix} = b$$

and that:

$$A = LU = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 2 & -1 \\ 0 & 2 & -1 & 1 \\ 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Problem b

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

$$b = \begin{bmatrix} 4 \\ 6 \\ 6 \end{bmatrix}$$

If we run this through the python code we see that:

b.

```
A = np.matrix([[1, 1, 1],
               [2, 2, 1],
               [1, 1, 2]])

b = np.matrix([[4],
               [6],
               [6]])

print_solution(A, b)
```

```
solution =
([1.0, 1.0, 2.0], [3.0, -1.0, 2.0])

L =
[[1. 0. 0.]
 [2. 1. 0.]
 [1. 0. 1.]]
U =
[[ 1.  1.  1.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]]
```

Therefore, we see that there are **infinite solutions to Ax=b two of which are**

$$Ax = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 6 \end{bmatrix} = b$$

$$Ax = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 6 \end{bmatrix} = b$$

We also see the LU factorization

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix}$$

Problem c

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

$$b = \begin{bmatrix} 4 \\ 4 \\ 6 \end{bmatrix}$$

If we run this through the python code we see that:

c.

```
A = np.matrix([[1, 1, 1],
               [2, 2, 1],
               [1, 1, 2]])

b = np.matrix([[4],
               [4],
               [6]])

print_solution(A, b)
```

```
solution =
There is no solution

L =
[[1. 0. 0.]
 [2. 1. 0.]
 [1. 0. 1.]]
U =
[[ 1.  1.  1.]
 [ 0.  0.  1.]
 [ 0.  0. -1.]]
```

Therefore, we see that there are **no solutions to Ax=b** We also see the LU factorization

$$A = LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix}$$

Note that since this is the same A as problem b, this will also be the same A=LU factorization.