LAB 5 - LU DECOMPOSITIONS AND GAUSSIAN ELIMINATION

1. Elementary matrices and Gaussian elimination

Gaussian elimination (i.e. row reduction) is a basic method for solving linear systems such as:

$$\begin{aligned} x_1 + x_2 + x_3 &= 1 \\ x_1 + 4x_2 + 2x_3 &= 3 \\ 4x_1 + 7x_2 + 8x_3 &= 9. \end{aligned}$$

The basic idea involves representing a linear system as an augmented matrix

$$A = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array}\right],$$

after which we apply elementary row operations to reduce the augmented matrix to a matrix in row echelon form. Once in row echelon form, back substitution will produce the solution to the linear system, provided a solution actually exists.

There are three possible types of row operations involved in this process: swapping two rows, multiplying one row by a nonzero scalar, and adding a scalar multiple of one row to another. Although there are certainly systems of linear equations which require all three row operations to simplify, many systems can actually be reduced to row echelon form using only elementary row operations of the third type. For instance, the system introduced above can be reduced via row operations as follows:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array}\right] \xrightarrow{R2-R1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ \mathbf{0} & 3 & 1 & 2 \\ 4 & 7 & 8 & 9 \end{array}\right] \xrightarrow{R3-4R1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ \mathbf{0} & 3 & 4 & 5 \end{array}\right] \xrightarrow{R3-R2} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & \mathbf{0} & 3 & 3 \end{array}\right].$$

The final augmented matrix corresponds to the linear system

$$\begin{aligned} x_1 + x_2 + x_3 &= 1 \\ 3x_2 + x_3 &= 2 \\ 3x_3 &= 3. \end{aligned}$$

This system can then be solved via *back substitution*, by solving the last equation for x_3, then using this x_3 value to solve the second to last equation for x_2, and so on.

One way of thinking of row operations is via *elementary matrices*. An elementary matrix is obtained by performing a single row operation on the identity matrix I. For example, we can take the 3×3 identity matrix and add $-5 \times$ Row 1 to Row 2:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & 0 \\ -5 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = E.$$

This matrix E is called the elementary matrix associated to the row operation of adding $-5 \times \text{Row}\,1$ to Row 2. Now let B be any matrix with 3 rows, say

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}.$$

Multiplying B on the left by E, we obtain

$$EB = \begin{bmatrix} 1 & 0 & 0 \\ -5 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 9 & 10 & 11 & 12 \end{bmatrix}.$$

Notice that multiplying B on the left by E transforms B by adding $-5 \times \text{Row}\,1$ to Row 2, which is the same row operation we used to create the elementary matrix E! A similar procedure works for other row operations. For example, if we want to swap Row 2 and Row 3 of B, we could multiply B on the left by the elementary matrix

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

which we obtained from the identity matrix by swapping Row 2 and Row 3. Finally, if we want to multiply Row 2 of B by 3, we can instead multiply B on the left by the elementary matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

which we obtained from the identity matrix by multiplying Row 2 by 3.

Observe now that we can reinterpret Gaussian elimination using the idea of elementary matrices. Indeed, instead of thinking of each step in Gaussian elimination as being an elementary row operation, we can think of each step as involving a multiplication on the left by some elementary matrix.

For example, returning to the system at the beginning of this lab, observe that we can express the sequence of row operations we used to reduce A as a sequence of matrix products, multiplying by elementary matrices on the left. Letting the kth row operation we used above be identified with the corresponding elementary matrix E_k, the previous sequence of row operations can be expressed as $E_3E_2E_1A = U$, where U is the upper triangular coefficient matrix for the final system:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array}\right] = \left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array}\right]. \tag{1}$$

- Take careful note of the order in the matrix product above. Why do you think the elementary matrices are ordered as $E_3E_2E_1$ instead of the other way around?

One goal in this lab will be to write a function which takes as input a system of equations (expressed as an augmented matrix), and returns a solution to the system. Recall from Lab 2 that we may express a matrix in Python as an array. For example, the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

can be created with the following command:

```
A=np.array([[1,2,3],[4,5,6],[7,8,9]])
```

- If necessary, review the section of Lab 2 which describes details about Python arrays.

As a first step towards our goal, we will need to be able to create an augmented matrix $[A \mid \boldsymbol{b}]$ (as a NumPy array) from a matrix A and a vector $\boldsymbol{b}$. If we consider again the linear system at the start of this lab, we have

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 4 & 2 \\ 4 & 7 & 8 \end{bmatrix} \text{ and } \boldsymbol{b} = \begin{bmatrix} 1 \\ 3 \\ 9 \end{bmatrix}.$$

We can then use the following code to create an augmented matrix from A and $\boldsymbol{b}$.

- Recall that even though the vector $\boldsymbol{b}$ above is a column vector, when creating it in NumPy we use `b=np.array([1,3,9])`, and not `b=np.array([[1],[3],[9]])`. The same goes for any vector we create in NumPy.

```
import numpy as np

A=np.array([[1,1,1],[1,4,2],[4,7,8]])

b=np.array([1,3,9])

Ab=np.column_stack((A,b))

print(Ab)
```

```
[[1 1 1 1]
 [1 4 2 3]
 [4 7 8 9]]
```

- Can you figure out what the command `np.column_stack` is doing in the above code? Try changing to matrix `A` and vector `b` to make sure you understand how it works.

Problem 1. Define a function `augment(A,b)` that accepts as inputs a matrix `A` and a vector `b` (as NumPy arrays) with the same number of rows, and returns an augmented matrix `[A|b]`.

For example, if we input the matrix `C=np.array([[3,2,1,1],[1,-2,1,1],[5,0,1,5]])` and the vector `b=np.array([-1,3,2])`, then `augment(C,b)` should return

```
array([[ 3,  2,  1,  1, -1],
       [ 1, -2,  1,  1,  3],
       [ 5,  0,  1,  5,  2]])
```

Now that we can create augmented matrices, we can begin the work of row reduction. Recall that if we've defined an array `A` in NumPy, we can access a row of `A` using either `A[j,]`, `A[j,:]`, or `A[j]`, where `j` is the position of the row we are interested in (remember, Python starts counting at 0).

```
C=np.array([[3,2,3],[2,4,1],[5,1,1]])

print(C[0,])  # Prints the first row of C.
print(C[2,:]) # Prints the third row of C.
```

```
[3 2 3]
[5 1 1]
```

Furthermore, we can use this notation to redefine the value of a row, and in particular, do row operations. For example, if we want to subtract $(2/3) \times$ Row 1 from Row 2, we could do the following:

```
C=np.array([[3,2,3],[2,4,1],[5,1,1]])

C[1,:]=C[1,:]-2/3*C[0,:]    # Redefining the second row C[1,:].
print(C)
```

```
[[ 3  2  3]
 [ 0  2 -1]
 [ 5  1  1]]
```

Notice that we've zeroed out the first entry in the second row, which would be the first step in row reducing `A`. Furthermore, note that instead of typing in the fraction `2/3` explicitly, we could have obtained the same fraction by using `A[1,0]/A[0,0]`. In this way we can create zeros below the top left entry of `A`, without even knowing what those entries were to begin with.

- Starting with the array `A=np.array([[4,1,3],[-37,4,1],[2,1,-17],[7,7,1]])`, use row operations to make sure that every entry below the `4` in the first column is `0`. Use indexing (i.e. use `A[0,0]` instead of typing `4`) so that your code would work on any matrix of this size.

Problem 2. Define a function `first_column_zeros(A)` that accepts as input a matrix `A` (as a NumPy array) with a *nonzero entry in the upper left corner*, and returns a row equivalent matrix `B` whose first column has only one nonzero entry in the top left position. Remember, you can assume that the top left entry of `A` is nonzero, though it may not be equal to 1.

In other words, `first_column_zeros(A)` should use row operations (adding appropriate multiples of the first row to the other rows) to create a matrix `B` in which the top left entry is the only nonzero entry in the first column. For grading purposes, your function shouldn't perform any additional row operations other than the ones needed to create zeros below the top left entry.

For example, if `A=np.array([[2,1,3,1],[1,2,-1,2.5],[4,2,-1,1]])`, then the function call `first_column_zeros(A)` should return

```
array([[ 2. , 1. , 3. , 1. ],
       [ 0. , 1.5, -2.5, 2. ],
       [ 0. , 0. , -7. , -1. ]])
```

Notice that the output is an array of floats (decimal values). This is due to the original array containing the float 2.5, which forced all other entries to be floats.

Hint: You will probably need to write a `for` *loop which loops through all of the rows of* `A` *except for the first one. Recall that you can include both a starting value and ending value in the function* `range`*.*

Another helpful thing to notice is that we can use slicing to access smaller submatrices of a NumPy array. For example, if we want to take everything in the matrix `A` that is between row `i` and row `j-1`, and between column `a` and column `b-1` we could use `A[i:j,a:b]` (remember, row `i` is *included*, but row `j` is *not*, similarly with columns `a` and `b`). Remember that we can leave out any of the indices, and Python will just start counting from the beginning, or will count all the way to the end. For example by putting `A[:j,a:]`, Python will include all of the rows up to row `j-1`, and all of the columns after column `a`.

```python
D=np.array([[4,1,3,-3],[-37,8,1,1],[2,1,-17,-2],[-5,7,1,3]])

print(D[1:3,1:3]) # Print all of the entries between the second and third row
                  # and between the second and third column.

print(D[1:,:3])   # Dropping the first row (row 0) and the last column.
```

```
[[  8   1]
 [  1 -17]]
[[-37   8   1]
 [  2   1 -17]
 [ -5   7   1]]
```

- Starting with the array `A=np.array([[3,2,3],[2,4,1],[5,1,1]])`, and using indexing as described above, produce the 2×2 submatrix of `A` in the bottom right corner. In other words, drop the first row and first column of `A`.

Problem 3. Define a function `row_echelon(A,b)` that accepts as inputs a matrix `A` and vector `b`, and returns an echelon form of the augmented matrix `[A|b]`. The function should perform Gaussian elimination (row reduction) to reduce `[A|b]` to row echelon form. You do not yet need to perform the back substitution that obtains the solution to $A\boldsymbol{x} = \boldsymbol{b}$.

You may assume for now that `A` is square, invertible, and `0` never appears on the main diagonal of `A`. Thus you only need to use row operations which add a multiple of one row to another. For example, if our matrix is

```
A=np.array([[3.,1.,-2.],[1.,2.,-5.],[2.,-4.,1.]])
```

and the vector is `b=np.array([1.1,2,-3])`, then the function call `row_echelon(A,b)` should return

```
array([[ 3.        ,  1.        , -2.        ,  1.1       ],
       [ 0.        ,  1.66666667, -4.33333333,  1.63333333],
       [ 0.        ,  0.        , -9.8       ,  0.84      ]])
```

Finally, as one last helpful NumPy tool, we can use the function `np.shape()` to tell use the shape of a NumPy array. If `A` is a NumPy array, then `np.shape(A)` will return two numbers, the first will be the number of rows of `A`, while the second is the number of columns.

```python
A=np.array([[4,1],[-37,4],[2,1],[7,1]])  # A is a 4x2 matrix.

print(np.shape(A))
```

```
(4, 2)
```

2. LU Decomposition: a general approach to solving linear systems

Recall that in example (1) above, the product $E_3E_2E_1A$ is an upper triangular coefficient matrix U. Another way to view this is by writing $E_3E_2E_1 = L^{-1}$, so that $L^{-1}A = U$ is an upper triangular coefficient matrix. It turns out that L is lower triangular, and is in fact a product of elementary matrices itself. Indeed, if $E_3E_2E_1 = L^{-1}$ then $L = E_1^{-1}E_2^{-1}E_3^{-1}$. Restating this, we can arrange the equation $L^{-1}A = U$ as $A = LU$, where L is a lower triangular matrix, and U is an upper triangular matrix.

Factoring a matrix A as $A = LU$ is called an *LU decomposition* of A, and is one of the most useful algorithms in linear algebra. This is because if we want to solve $A\boldsymbol{x} = \boldsymbol{b}$ where $A = LU$, then we are actually trying to solve the equation $LU\boldsymbol{x} = \boldsymbol{b}$. We can then introduce a new variable $\boldsymbol{y}$ which we define by $\boldsymbol{y} = U\boldsymbol{x}$, and this new variable will satisfy the equation $L\boldsymbol{y} = \boldsymbol{b}$.

We can then find $\boldsymbol{x}$ by first solving the lower triangular system $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$, after which we can solve the upper triangular system $U\boldsymbol{x} = \boldsymbol{y}$ for $\boldsymbol{x}$. Both of these systems are triangular, and are very easy to solve via a process called forward and/or back substitution.

Furthermore, not only is the LU decomposition an efficient way to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}$, but once $A = LU$ is computed, the solutions of this linear system can readily be found for different values of $\boldsymbol{b}$. More precisely, we only need to find the LU decomposition once, after which if we want to solve the linear system $A\boldsymbol{x} = \boldsymbol{b}'$ for some different vector $\boldsymbol{b}'$, all we need to do is quickly solve two triangular systems.

3. Numerical Implementation

In reality it is not very efficient to compute L^{-1} (or L for that matter) via a series of multiplications of elementary matrices. Instead we keep track of the terms in the matrix L and the matrix U without resorting to any matrix multiplication. We describe the algorithm for doing this below, which we write in pseudocode.

In the pseudocode below you will see variables being defined (such as the numbers m, and n, and the arrays U and L), along with a single `for` loop. The values $L_{i,j}$ and $U_{i,j}$ represent the value in the ith row and jth columns of L and U respectively, while $U_{i,:}$ and $U_{j,:}$ represent the ith and jth rows of U respectively. Recall that the symbol $\leftarrow$ represents replacing a variable with a new value. For example, if x and y are both variables, the statement $x \leftarrow y$ indicates that we are taking the value saved in x, and replacing it with the value saved in y (while the value saved in y remains unchanged).

Algorithm 1 LU decomposition

```
LU Decomposition ($A$):
$m, n = $ shape($A$)
$U = A$
$L = I_m$ (the $m \times m$ identity matrix)
for $j = 0, \dots, n-1$ do
    for $i = j+1, \dots, m-1$ do
        $L_{i,j} \leftarrow U_{i,j}/U_{j,j}$
        $U_{i,:} \leftarrow U_{i,:} - L_{i,j}U_{j,:}$
    end for
end for
return $L$ and $U$
```

Problem 4. Define a function `LU_decomposition(A)` which accepts as input a matrix `A`, and returns the matrices `L` and `U`. You may assume that the matrix `A` is invertible and that row operations do not produce zeros on the diagonal.

For example, if `A=np.array([[3,1,-2],[1.5,2,-5],[2,-4,1]])` then the function call `LU_decomposition(A)` should return

```
(array([[ 1.        ,  0.        ,  0.        ],
        [ 0.5       ,  1.        ,  0.        ],
        [ 0.66666667, -3.11111111,  1.        ]]),
 array([[ 3.        ,   1.        ,  -2.        ],
        [  0.       ,   1.5       ,  -4.        ],
        [  0.       ,   0.        , -10.11111111]]))
```

Notice there are two matrices returned, the first is `L` while the second is `U`.

- It is worth noting here that this algorithm only accounts for the third type of row operation, and will not work if there is a zero on the main diagonal at any step in the process. Why is that, and how would you compensate for a zero on the main diagonal?

Now that we can compute the LU decomposition of a matrix, the final remaining step to solving our system is to perform the forwards and back substitutions to solve the lower and upper triangular systems respectively. We provide the pseudocode for forward substitution here. The dot $\cdot$ in the fifth row denotes the dot product of $\boldsymbol{y}$ and the row $L_{i,:}$ (thought of as a column vector). Remember from Lab 2 that NumPy has a built-in function for taking dot products.

Algorithm 2 Forward substitution

```
Forward substitution $(L, \boldsymbol{b})$:
$n$ = length($b$)
$\boldsymbol{y} = \boldsymbol{0}$ (the $n$-dimensional zero vector)
for $i = 0, \ldots, n-1$ do
    $\boldsymbol{y}_i \leftarrow (\boldsymbol{b}_i - \boldsymbol{y} \cdot L_{i,:})/L_{i,i}$
end for
return $y$
```

- Starting with a simple lower triangular system like

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 7 \\ 2 & 3 & 0 & 8 \\ 4 & 5 & 6 & 9 \end{array}\right],$$

walk through the steps of the above pseudocode on pen and paper to see why this algorithm ends up providing a solution to the system. Walking through it carefully will help you to adapt it to backwards substitution in Problem 6.

Problem 5. Using the above pseudocode, define a function `forward_substitution(L,b)` which accepts as input a square lower triangular matrix `L` and a vector `b` and returns a vector `y` which is the solution to `Ly=b`.

For example, if the matrix is `L=np.array([[1,0,0],[3,1,0],[-1.1,2,1]])`, and the vector is `b=np.array([-2.1,1,-1])` then `forward_substitution(L,b)` should return the vector

```
array([ -2.1 , 7.3 , -17.91])
```

Problem 6. Define a function `back_substitution(U,y)` which accepts as input a square upper triangular matrix `U` and a vector `y` and returns a vector `x` which is the solution to `Ux=y`.

For example, if the matrix is `U=np.array([[2,-3.1,1],[0,1,3],[0,0,4]])`, and the vector is `y=np.array([1,-2.1,3])` then `back_substitution(U,y)` should return the vector

```
array([-6.6175, -4.35 , 0.75 ])
```

Hint: The only difference between the algorithms for forward and back substitution is the order you solve for the variables in. You should be able to solve this problem by making some minor changes to your code in Problem 5.

Problem 7. Define a function `LU_solver(A,b)` which accepts as input a square matrix `A` and a vector `b` and returns the solution `x` to `Ax=b`. We assume that the matrix `A` is invertible and that row operations do not produce zeros on the diagonal. Thus `LU_solver` may call the functions `LU_decomposition`, `back_substitution`, and `forward_substitution`.

For example, if the matrix is `A=np.array([[3,1,-2],[1.5,2,-5],[2,-4,1]])`, and the vector is `b=np.array([1.1,3,-2])` then `LU_solver(A,b)` should return the vector

```
array([-0.07032967, 0.34395604, -0.48351648])
```

4. How well does this actually work?

We make one final note that the algorithms described above for Gaussian elimination and the LU decomposition in general are not completely robust, i.e. a zero on the main diagonal will 'break' this version. This isn't the only problem that can arise when we attempt to calculate the LU decomposition. In fact the current implementation we are using only works for a specific class of matrices that are very nice.

- Have your function from Problem 4 above find the LU decomposition of the following 2×2 matrix:

$$\mathtt{A} = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 1 \end{bmatrix}. \tag{2}$$

Note that the above matrix is a very slight perturbation away from the 'problematic' matrix

$$\mathtt{B} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \tag{3}$$

which has a zero on the main diagonal. In any case, we shouldn't be concerned about this because using 10^{-20} in place of 0 should eliminate the concern about dividing by zero, right?

- Perform the matrix multiplication `LU` and see how this compares to your original matrix `A`. Is it the same? What went wrong? Compute the LU decomposition of this matrix by hand, and see if you can see what happened. (You can multiply NumPy arrays using the 'at' sign @, i.e., the product of `A` and `B` is computed using `A@B`.)

Stability of an algorithm and machine precision. What happens in the above example is that LU decomposition, as developed here, is an unstable algorithm, particularly when one of the main diagonal terms is significantly larger or smaller than the others. As demonstrated by the example above, LU decomposition is clearly not stable. This is because all computations done by a computer are done approximately, that is the computer can't keep track of 20 digits for each number (roughly speaking your computer keeps track of 16 digits instead) and hence does not see any difference between $1 - 10^{20}$ and -10^{20}. That seems like a reasonable approximation to us, unless this error cascades into the solution of the problem we are interested in, which is exactly what happens for the previous example.

It turns out that there is an easy way to fix this in LU decomposition. The answer is simply to swap rows at step i so that the (i, i) entry of the matrix is larger than all entries below it. This simple fix is called partial pivoting, and allows LU decomposition (and Gaussian elimination) to be stable for almost all matrices.