HW 8 Solutions

ENGR 21, Fall 2026.

November 5, 2026

Solutions

1 Counting Calculations

1.1 Quantifying the computational cost of LU Decomposition

The LU decomposition of a matrix of size 4×4 requires us to perform six **row updates** of the kind 'row x = row x - (multiplier) times row y', as you did in the worksheet in Lecture 16.

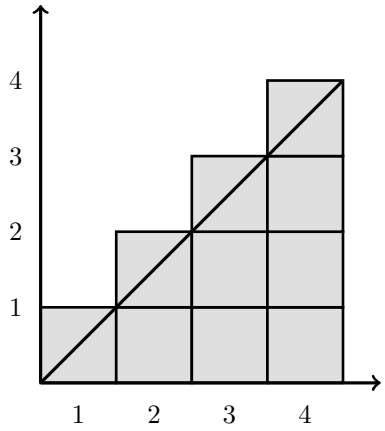
In each calculation, an entire row is updated, although some elements of that row may already be known to be zero.

1.1.1 Number of Row Updates

How many row updates are needed for the LU decomposition of a matrix of size $n \times n$? Give your answer as a simplified formula in terms of n.

> **Tip**
>
> For the first few values of n, we know that there are:
>
> | n | No. of row updates | Source |
> |---|---|---|
> | 2 | 1 | By inspection |
> | 3 | 3 | Lecture |
> | 4 | 6 | Worksheet |
> | 5 | 10 | ... |

> **Tip**
>
> The sum of all numbers from 1 to n can be calculated by finding the shaded area in the diagram below.



Use this fact to help you derive the needed formula. Then, write down how many row updates are needed for a matrix of size 5×5, 10×10, 100×100, and $10^6 \times 10^6$.

Solution

The answer is

$$1 + 2 + 3 + \cdots + n - 1 = \frac{1}{2}n(n-1)$$

n	Rows updated
5	10
10	45
100	4950
10^6	$\approx 5 \times 10^1 1$

1.1.2 Number of nonzero terms updated

Looking more closely at the LU Decomposition process from the in-class worksheet, we can see that each time a row update is carried out, the computer needs to calculate several new terms at once.

For example, in the first row update to eliminate A_{21}, we write:

Let row 2 = row 2 $-$ $(A_{21}/A_{11}) \times$ row 1.

This is one row update, which carries out several term updates. Let's write them out:

$$\Big[\underbrace{A_{21} - \tfrac{A_{21}}{A_{11}} \times A_{11}}_{\text{new value of } A_{21}=0},\quad \underbrace{A_{22} - \tfrac{A_{21}}{A_{11}} \times A_{12}}_{\text{new value of } A_{22}\neq 0},\quad \underbrace{A_{23} - \tfrac{A_{21}}{A_{11}} \times A_{13}}_{\text{new value of } A_{23}\neq 0},\quad \underbrace{A_{24} - \tfrac{A_{21}}{A_{11}} \times A_{14}}_{\text{new value of } A_{24}\neq 0}\Big]$$

We therefore see that the first row update for a 4x4 matrix requires us to update up to 3 terms, ignoring the one that will turn out to be zero by construction.

As a second example, let us look at the 4th row update. This time, we have

Let row 3 = row 3 $-$ $(A_{32}/A_{22}) \times$ row 2.

$$\left[\underbrace{\overbrace{A_{31}}^{=0} - \frac{A_{32}}{A_{22}} \times \overbrace{A_{21}}^{=0}}_{\text{new value of } A_{31}=0}, \quad \underbrace{A_{32} - \frac{A_{32}}{A_{22}} \times A_{22}}_{\text{new value of } A_{32}=0}, \quad \underbrace{A_{33} - \frac{A_{32}}{A_{22}} \times A_{23}}_{\text{new value of } A_{33}\neq 0}, \quad \underbrace{A_{34} - \frac{A_{32}}{A_{22}} \times A_{24}}_{\text{new value of } A_{34}\neq 0}\right]$$

Systematizing this procedure of counting, we can summarize the LU Decomposition / Gaussian Elimination of a matrix of size $n \times n$ as follows.

Step no.	Row	Terms Updated ($\not{x} \implies x = 0$)	No. of nonzero terms
1	Row 2	$\not{A_{21}}$, A_{22}, A_{23}, A_{24}	3
2	Row 3	$\not{A_{31}}$, A_{32}, A_{33}, A_{34}	3
3	Row 4	$\not{A_{41}}$, A_{42}, A_{43}, A_{44}	3
4	Row 3	$\not{A_{31}}$, $\not{A_{32}}$, A_{33}, A_{34}	2
5	Row 4	$\not{A_{41}}$, $\not{A_{42}}$, A_{43}, A_{44}	2
6	Row 4	$\not{A_{41}}$, $\not{A_{42}}$, $\not{A_{43}}$, A_{44}	1
Total			14

We can use this table to calculate the number of nonzero terms that are updated to Gauss-eliminate a square matrix of size 4×4:

$$\begin{aligned} 3 + 3 + 3 + \\ 2 + 2 + \\ 1 = 14 \end{aligned}$$

Using summation notation (i.e., the symbol $\sum$), determine a formula for the number of nonzero terms that are updated when Gauss-eliminating a matrix of size $n \times n$, in terms of n. Write down how many nonzero term updates are needed for a matrix of size 5×5, 10×10, 100×100, and $10^6 \times 10^6$.

Solution

The answer is

$$\sum_{i=1}^{n-1} i^2$$

It is also possible to show — but not necessary for this assignment — that this is equal to

$$\frac{1}{6} n(n-1)(2n-1)$$

n	Nonzero terms updated
5	30
10	285
100	$\approx 3.28 \times 10^5$

$$10^6 \quad \approx 3.33 \times 10^{17}$$

1.1.3 Tridiagonal Matrices and Tridiagonal solvers

When a matrix is banded, a specialized solver exploits this banded structure to dramatically reduce the number of calculations needed to perform LU decomposition on it.

1. In terms of n, how many nonzero terms need to be updated when performing LU decomposition of a tridiagonal matrix of size $n \times n$?
2. A square matrix has size n, and one engineer LU-decomposes it using an algorithm that does not take advantage of the banded structure, while a second engineer uses a specialized algorithm that only updates the necessary terms. Determine the ratio of the number of term updates carried out by the second engineer's algorithm compared to the first engineer's algorithm. **You may leave your answer in an un-simplified form**; you will compute this fraction numerically in question 5 below.
3. In terms of n, what is the maximum number of nonzero terms in a tridiagonal matrix?
4. Assume that one byte of memory is used to store one entry in a matrix. What is the percentage saving in space when only the three bands of a tridiagonal $n \times n$ matrix are stored, instead of the full matrix? Give your answer in terms of n.
5. Calculate the fraction from 2 and 4 above, for a matrix of size 5×5, 10×10, 100×100, and $10^6 \times 10^6$. Write the answers as decimal numbers using scientific notation when necessary.

Solution

1. $\boxed{n-1}$ terms need to be updated when carrying out LU decomposition of a tridiagonal matrix of size $n \times n$.
2. $\dfrac{\cancel{n-1}}{\frac{1}{6}(2n-1)n\cancel{(n-1)}} = \dfrac{6}{n(2n-1)}$. It is also acceptable to give the answer $\dfrac{n-1}{\sum_{i=1}^{n-1} i^2}$
3. $n-1+n-1+n = 3n-2$
4. $\dfrac{3n-2}{n^2}$
5. A table with the answers is shown below.

n	Fraction of Term updates	Fraction of Storage Space
5	≈ 0.133	0.52
10	≈ 0.031	0.28
100	$\approx 3.01 \times 10^{-4}$	0.0298
10^6	$\approx 3 \times 10^{-12}$	3×10^{-6}

2 Gaussian Elimination

Write a Python function named `gaussian_elimination` that takes as input a 2-dimensional `numpy` array. This array need not represent a square matrix; in practice, we can use such a function for both square matrices A, which have size $N \times N$, and 'augmented matrices' $[A|b]$, which have size $N \times (N+1)$.

Your function should return the row-echelon form of the given matrix, as a `numpy` array having the same dimensions as the input array. You do not need to account for cases where $Ax = b$ has no solution.

Solution

```
def gaussian_elimination(a):
    n = len(a)
    for k in range(0,n-1):
        for i in range(k+1,n):
            a[i,k:n] = a[i,k:n] - (a[i,k]/a[k,k])*a[k,k:n]
    return a
```

3 LU Decomposition

Write a Python function named `LUdecomp` that takes as input a square matrix `A` and outputs two square matrices in a tuple, such as `(L,U)`. The first element of the tuple should contain a lower triangular matrix, and the second element of the tuple should contain an upper triangular matrix, determined uniquely by Doolittle's Method.

Solution

```
def LUdecomp(a):
    n = len(a)

    l = np.zeros(np.shape(a)) # L, the lower triangular matrix
    for i in range(n):
        l[i,i] = 1 # L has 1 on the (1,1), (2,2) spots etc.

    for k in range(0,n-1):
        for i in range(k+1,n):
            multiplier = a[i,k]/a[k,k]
            a[i,k:n] = a[i,k:n] - multiplier*a[k,k:n]
            l[i,k] = multiplier # store the multiplier in L

    u = a # u now contains the upper triangular matrix
    return (l,u)
```

4 Solution Phase — forward and backward substitutions

4.1 Backward substitution

The idea behind *backward substitution* is that an equation of the form $Ax = c$ where A is an *upper triangular matrix* can be solved (for the vector x) by starting from the last row and working backward.

Write a function named `backward_substitution` that takes as input a matrix (i.e., 2-dimensional `numpy` array) of size $N \times (N+1)$ and outputs a 1-dimensional `numpy` array of size N. The input should be assumed to have the form

$$\left[\begin{array}{ccc|c} A_{11} & A_{12} & A_{13} & c_1 \\ 0 & A_{22} & A_{23} & c_2 \\ 0 & 0 & A_{33} & c_3 \end{array}\right]$$

and the output should be a solution to the system of equations

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + A_{13}x_3 &= c_1 \\ A_{22}x_2 + A_{23}x_3 &= c_2 \\ A_{33}x_3 &= c_3 \end{aligned}$$

Note that a formula for this has been given in lecture.

A partial annotated solution is provided below.

Solution

```
def backward_substitution(ac):
    # ac is an augmented matrix of size n x (n+1)
    c = ac[:,-1]
    a = ac[:,:-1]
    n = len(c)
    x = np.zeros(n)
    for i in range(n-1,-1,-1):
        total = 0
        for k in range(n-1,i,-1):
            total += a[i,k]*x[k]
        x[i] = (c[i]- total)/a[i,i]
    return x
```

4.2 Forward Substitution

The idea behind *forward substitution* is that an equation of the form $Ax = c$ where A is a *lower triangular matrix* can be solved (for the vector x) by starting from the first row and working forward.

Write a function named `forward_substitution` that takes as input a matrix (i.e., 2-dimensional `numpy` array) of size $N \times (N+1)$ and outputs a 1-dimensional `numpy` array of size N. The input should be assumed to have the form

$$\left[\begin{array}{ccc|c} A_{11} & 0 & 0 & c_1 \\ A_{21} & A_{22} & 0 & c_2 \\ A_{31} & A_{32} & A_{33} & c_3 \end{array}\right]$$

and the output should be a solution to the system of equations

$$\begin{aligned} A_{11}x_1 &= c_1 \\ A_{21}x_1 + A_{22}x_2 &= c_2 \\ A_{31}x_1 + A_{32}x_2 + A_{33}x_3 &= c_2 \end{aligned}$$

Solution

```python
def forward_substitution(ac):
    # ac is an augmented matrix of size n x (n+1)
    c = ac[:,-1]
    a = ac[:,:-1]
    n = len(c)
    x = np.zeros(n)
    for i in range(n):
        total = 0
        for k in range(i):
            total += a[i,k]*x[k]
        x[i] = (c[i]- total)/a[i,i]
    return x
```

5 Solving Systems of Linear Equations

5.1 LU Decomposition of a 3x3 matrix

A certain square matrix A can be decomposed using Doolittle's Decomposition into a product LU of the following matrices,

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 3/2 & 1 & 0 \\ 1/2 & 11/13 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & -3 & -1 \\ 0 & 13/2 & -7/2 \\ 0 & 0 & 32/13 \end{bmatrix}.$$

5.1.1 Find the matrix A

by multiplying the matrices LU (See page 9 of Matt Zucker's Intro to Linear Algebra if you are not sure how to multiply matrices). Show your work, even if you use a computer/calculator to help you do the computation.

Answer

The answer is

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

5.1.2 Solve $Ax = b$

1. Given $b = \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$, solve $Ax = b$ using the following procedure:

   - First, find the vector y when $Ly = b$ using *forward substitution*.
   - Then, find the vector x when $Ux = y$ using *backward substitution*.

   Show your work, even if you use a computer/calculator to help you do the computation.

2. Given $b = \begin{bmatrix} 2 \\ 0 \\ 3 \end{bmatrix}$, solve $Ax = b$ using the same procedure as above.

Hint

Use your answers to these questions to check your work in Section 2, Section 3, and Section 4.

5.2 Compuationally solving a 4 x 4 system

In this question, you will solve $Ax = b$ for the following A and b, using two different approaches.

```
A = np.array([[ 1.44,-0.36, 5.52, 0.00],
              [-0.36,10.33,-7.78, 0.00],
              [ 5.52,-7.78,28.40, 9.00],
              [ 0.00, 0.00, 9.00, 61.00]])
b = np.array([[0.04],
              [-2.15],
              [0.00],
              [0.88]])
```

1. Use the function `gaussian_elimination` followed by the function `backward_substitution`.
2. Use the function `LUdecomp`, followed by `forward_substitution` and then `backward_substitution`.

Your code should print out statements of the form

```
Answer by Gaussian elimination
[ ... ... ... ... ]
Answer by LU decomp
[ ... ... ... ... ]
```

Hint

You will need to use `hstack` in this problem. Because `numpy` makes a distinction between arrays of the form
`b1 = np.array([[0.04],[-2.15],[0.00],[0.88]])`
and those of the form
`b2 = np.array([0.04,-2.15,0.00,0.88])`
you may need to use the `reshape` function before you are able to properly `hstack` two arrays. Note that `np.reshape(b2,np.shape(b1))` will make `b2` identical to `b1`, i.e., it will turn `b2` into a column vector.
Whether or not this is necessary depends on how you write your functions, so don't worry if this doesn't apply to you!

Solution

```python
from HW8funcs import *

A = np.array([[ 1.44,-0.36, 5.52, 0.00],
              [-0.36,10.33,-7.78, 0.00],
              [ 5.52,-7.78,28.40, 9.00],
              [ 0.00, 0.00, 9.00, 61.00]])
b = np.array([[0.04],
              [-2.15],
              [0.00],
              [0.88]])

# Solve using Gaussian elimination + Backward Substitution
Ab = np.hstack([A,b])
print("Answer by Gaussian elimination")
print(backward_substitution(gaussian_elimination(Ab)))

# Solve using L and U
(l,u) = LUdecomp(A)
lb = np.hstack([l,b])
y = forward_substitution(lb)
uy = np.hstack([u,np.reshape(y,np.shape(b))])
x = backward_substitution(uy)
print("Answer by LU decomp")
print(x)
```