Scientific Computing Spring 2024

Lecture 04: LU Decomposition
Part 02

Vladimir Kulyukin
Department of Computer Science
Utah State University

Introduction

In this lecture, we will continue to work with square linear systems $\mathbf{Ax} = \mathbf{b}$ that have a unique solution that can be found by Gauss-Jordan Elimination (GJE) and back substitution without any row interchanges.

We will also continue with our exploration of LU Decomposition that we started in Lecture 03. The focus of this lecture is on one algorithm algorithm that use LU Decomposition to solve square linear systems $\mathbf{Ax} = \mathbf{b}$.

Back Substitution

Let us recall from Lecture 3 the formula for back substitution.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ 0 & 0 & \dots & . \\ . & . & \dots & . \\ . & . & \dots & . \\ 0 & 0 & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ . \\ . \\ b_n \end{bmatrix}$$

Then general formula for x_i is as follows.

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right].$$

Forward Substitution

Let us recall from Lecture 3 the formula for forward substitution.

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ . & . & \dots & 0 \\ . & . & \dots & . \\ . & . & \dots & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ . \\ . \\ b_n \end{bmatrix}$$

Then general formulas for y_i is as follows.

$$y_1 = \frac{b_1}{a_{11}}.$$

$$y_i = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij} y_j\right], i = 2, 3, \dots, n.$$

Solving Linear Systems with LU Decomposition

Let us review the algorithm that we started discussing in Lecture 03. Given $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}$ is $n \times n$ and invertible without any row interchanges, $\mathbf{x}$ is $n \times 1$, $\mathbf{b}$ is $n \times 1$. The algorithm has the following steps.

1. Create an $n \times n$ identity matrix $\mathbf{I}$.

2. Use Gauss-Jordan Elimination (GJE) to reduce $\mathbf{A}$ to $\mathbf{U}$, i.e., do $\mathbf{A} \sim \mathbf{U}$ with row scaling and row addition.

3. As we do step 2, we save the row multipliers in $\mathbf{L}$ as they are computed according to the following method: if we add $r \cdot$ row i to row k, we replace the zero in row k and column i in $\mathbf{I}$ with $-r$; in other words, we do $\mathbf{I}[\mathbf{k}, \mathbf{i}] = -\mathbf{r}$. Note that steps 2 and 3 are done simultaneously.

4. After steps 2 and 3 are finished, we have $\mathbf{A} \sim \mathbf{U}$ and $\mathbf{I} = \mathbf{L}$. We use forward substitution to solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$.

5. We use backward substitution to solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$.

6. We return $\mathbf{x}$.

Example

Let us apply this algorithm to solve the following linear system $\mathbf{Ax} = \mathbf{b}$.

$$\begin{array}{rcrcrcrcr}
1x_1 & - & 2x_2 & + & 0x_3 & + & 3x_4 & = & 11 \\
-2x_1 & + & 3x_2 & + & 1x_3 & - & 6x_4 & = & -21 \\
-1x_1 & + & 4x_2 & - & 4x_3 & + & 3x_4 & = & -1 \\
5x_1 & - & 8x_2 & + & 4x_3 & + & 0x_4 & = & 23
\end{array}$$

The matrices $\mathbf{Ax} = \mathbf{b}$ are:

$$\begin{bmatrix} 1 & -2 & 0 & 3 \\ -2 & 3 & 1 & -6 \\ -1 & 4 & -4 & 3 \\ 5 & -8 & 4 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 11 \\ -21 \\ -1 \\ 23 \end{bmatrix}$$

Example: Algo's Steps 1, 2, 3

A on the previous slide decomposes into the following matrices. You may want to stop here and review Slides 20 – 24 in Lecture 03 PDF in Canvas or your class notes if you are attended this lecture F2F. These steps turn **A** into **U** and **I** into **L**, where **U** is an upper-triangular matrix and **L** is a lower-triangular matrix such that $\mathbf{LU} = \mathbf{A}$.

$$\mathbf{U} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -2 & 6 \\ 0 & 0 & 0 & 3 \end{bmatrix} \qquad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & -2 & 1 & 0 \\ 5 & -2 & -3 & 1 \end{bmatrix}$$

After we complete steps 1 – 3, we have decomposed **A** into **U** and **L**.

Example: Algo's Step 4

We use forward substitution to solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & -2 & 1 & 0 \\ 5 & -2 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 11 \\ -21 \\ -1 \\ 23 \end{bmatrix}$$

The solution vector $\mathbf{y}$ is as follows.

$$\mathbf{y} = \begin{bmatrix} 11 \\ 1 \\ 12 \\ 6 \end{bmatrix}$$

Example: Algo's Steps 5, 6

We use back substitution to solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$.

$$\begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -2 & 6 \\ 0 & 0 & 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 11 \\ 1 \\ 12 \\ 6 \end{bmatrix}$$

The solution vector $\mathbf{x}$ is as follows.

$$\mathbf{x} = \begin{bmatrix} 3 \\ -1 \\ 0 \\ 2 \end{bmatrix}$$

This vector $\mathbf{x}$ is returned.

Numpy Checkup

Let us check our solution with `numpy`.

```
import numpy as np
import numpy.linalg
A = np.array([[1, -2, 0, 3],
              [-2, 3, 1, -6],
              [-1, 4, -4, 3],
              [5, -8, 4, 0]],
             dtype=float)
b = np.array([[11],
              [-21],
              [-1],
              [23]],
             dtype=float)
```

Numpy Checkup

Let us check our solution with `numpy`.

```
U = np.array([[1, -2, 0, 3],
              [0, -1, 1, 0],
              [0, 0, -2, 6],
              [0, 0, 0, 3]],
             dtype=float)
L = np.array([[1, 0, 0, 0],
              [-2, 1, 0, 0],
              [-1, -2, 1, 0],
              [5, -2, -3, 1]],
             dtype=float)
```

Numpy Checkup

Let us check our solution with `numpy`.

```
y = np.array([[11],
              [1],
              [12],
              [6]],
             dtype=float)
x = np.array([[3],
              [-1],
              [0],
              [2]],
             dtype=float)
```

Numpy Checkup

Let us check our solution with `numpy`.

```
assert np.allclose(np.dot(L, U), A)
assert np.allclose(np.dot(L, y), b)
assert np.allclose(np.dot(np.dot(L, U), x), b)
assert np.allclose(np.dot(L, np.dot(U, x)), b)
assert np.allclose(np.dot(A, x), b)
assert np.allclose(x, np.linalg.solve(A, b))
```

Why Does this Algo Work?

1. Recall the LU Decomposition Theorem that states that $\mathbf{A} = \mathbf{LU}$.

2. Recall a fundamental theorem of Linear Algebra states that if $[\mathbf{A}|\mathbf{b}] \sim [\mathbf{U}|\mathbf{c}]$, then $[\mathbf{A}|\mathbf{b}]$ and $[\mathbf{U}|\mathbf{c}]$ have the same solution sets.

3. Observe that $\mathbf{Ax} = \mathbf{LUx} = \mathbf{Ly} = \mathbf{b}$.

Why do we do $\mathbf{l}[\mathbf{k}, \mathbf{i}] = -\mathbf{r}$? Because otherwise the equation $\mathbf{A} = \mathbf{LU}$ would not hold.

Solving Linear Systems with LU Decomposition: Algo 2

This slide is optional. One of you asked me in class if there is an algorithm that changes the vector $\mathbf{b}$. Yes, there is one. Here's a brief outline. Given $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A}$ is $n \times n$ and is invertible w/o any row interchanges, $\mathbf{x}$ is $n \times 1$, $\mathbf{b}$ is $n \times 1$.

1. Use LU Decomposition to decompose $\mathbf{A}$ to $\mathbf{U}$ and $\mathbf{L}$.

2. Use $\mathbf{L}$ to convert $\mathbf{b}$ to $\mathbf{c}$ as one would with Gaussian-Jordan Elimination to reduce $[\mathbf{A}|\mathbf{b}]$ to $[\mathbf{U}|\mathbf{c}]$.

3. Use back substitution to solve $\mathbf{Ux} = \mathbf{c}$.

4. Return $\mathbf{x}$.

Algo 1 vs. Algo 2

This slide is optional.

1. Both Algo 1 and Algo 2 use LU Decomposition and need to save $\mathbf{L}$ and $\mathbf{U}$.
2. Both algos use back substitution.
3. Algo 1 uses $\mathbf{L}$ to solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$ with forward substitution.
4. Algo 2 uses $\mathbf{L}$ to convert $\mathbf{b}$ to $\mathbf{c}$.

While both algos are the same asymptotically, algo 1 performs more multiplications and additions (more elementary operations) during forward substitution.

Saving L and U Matrices

Let us assume that we have implemented **lud_decomp(a, n)** that takes an $n \times n$ numpy matrix **a**, does its LU-Decomposition, and returns the **u** and **l** matrices.

Here's how we can persist **u** and **l** on the disk and read them back from the disk into memory with **pickle**.

```
import pickle

def save_lud(a, n, file_name):
    u, l = lu_decomp(a, n)
    with open(file_name, 'wb') as outf:
       pickle.dump((u, l), outf)

def load_lud(file_name):
    with open(file_name, 'rb') as inf:
      return pickle.load(inf)
```

Some Notes of Leibnitz Determinant Optimization

A few students asked me in Canvas about possible optimizations of the Leibnitz determinant computation. There are several. One is to use multiprocessing. Let us define a few functions. The first function defines a minor matrix and the second one, computes cofactors and places them into a process queue q.

```
import numpy as np
import multiprocessing as mp
from multiprocessing import Queue, Process

def minor_mat(mat, i, j):
    mc = mat.copy()
    mc = np.delete(mc, np.s_[i:i+1], axis=0)
    mc = np.delete(mc, np.s_[j:j+1], axis=1)
    return mc

def cofactor_mp(mat, c, q):
    q.put(mat[0][c]*((-1)**c)*det(minor_mat(mat, 0, c)))
```

Optimizing Leibnitz with Multiprocessing

This function mounts each minor on a separate process.

```
def det_mp(mat):
    assert mat.shape[0] == mat.shape[1]
    return det_aux_mp(mat)

def det_aux_mp(mat):
    r, c = mat.shape
    if r == c == 1:
        return mat[0][0]
    elif r == c == 2:
        return mat[0][0]*mat[1][1] - mat[0][1]*mat[1][0]
    else:
        d = 0.0
        q = Queue()
        process_pool = []
        for c in range(r):
            p = Process(target=cofactor_mp, args=(mat, c, q))
            process_pool.append(p)
        for p in process_pool:
            p.start()
        while not q.empty():
            d += q.get()
        for p in process_pool:
            p.join()
        while not q.empty():
            d += q.get()
        return d
```

Some Notes of Leibnitz Determinant Optimization

Here's the timing code of my implementation on a 7-year old Bionic Beaver Computer (Ubuntu 18.04 LTS) in Python 3.6.7.

```
st = time.time()
d = det_mp(A)
et = time.time()
print('parallel leibnitz time = {}'.format(et-st))
print('det = {}'.format(d))
st = time.time()
d = det(A)
et = time.time()
print('standard leibnitz time = {}'.format(et-st))
print('det = {}'.format(d))
st = time.time()
d = np.linalg.det(A)
et = time.time()
print('GJE  determinant  time = {}'.format(et-st))
print('det = {}'.format(d))
```

Some Notes of Leibnitz Determinant Optimization

Here's the output of the timing code on the previous slides (in seconds) on a 10 x 10 random matrix of floats. The parallel leibnitz time does multiprocessing. The standard leibnitz computes the leibnitz determinant w/o any multiprocessing. The GJE determinant uses `np.linalg.det()`.

```
parallel leibnitz time = 23.996946096420288
det = -1.3075869861689833e+25
standard leibnitz time = 52.0207622051239
det = -1.3075869861689835e+25
GJE  determinant  time = 0.0001392364501953125
det = -1.3075869861689796e+25
```

We can also use memoization, i.e., a method to store the determinants of repeatedly computed minor matrices in a hash table or some other data structures with fast lookup times.

Takeaway

Just to reiterate: Scientific Computing is about tradeoffs. It may not be an easy decision to choose b/w Gauss-Jordan Elimination (GJE) with back substitution and using Algos 1, 2 (or some others) that rely on LU Decomposition.

While GJE requires more computation, it does not require any additional storage. On the other hand, LU-Decomposition, to justify decomposing **A** into **L** and **U**, must save **L** and **U** for future use, which may, in case of large matrices, require additional disk space.

We need to decide what is more affordable/important to us: CPU/GPU time or storage.

References

1. J. Fraleigh, R. Beauregard. *Linear Algebra*, Ch. 09.
2. W. Press, S. Teukolsky, W. Vetterling, B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, Ch. 02.