Scientific Computing: S24

Lecture 03: Cramer's Rule, LU-Decomposition: Part 01, Numerical Instability

Vladimir Kulyukin
Department of Computer Science
Utah State University

Introduction

In this lecture, we will work with square linear systems $\mathbf{Ax} = \mathbf{b}$, (i.e., $\mathbf{A}$ is an $n \times n$ matrix (hence, square), $\mathbf{x}$ and $\mathbf{b}$ are $n \times 1$ vectors) that have a unique solution that can be found by Gauss-Jordan Elimination (GJE) and without having to swap any rows.

We will start with several useful theorems about determinants that you should know about. After these theorems, we will get back to Cramer's rule and discuss how to use this rule to solve $\mathbf{Ax} = \mathbf{b}$.

We will decompose these matrices into upper-triangular and lower-triangular matrices (these will be defined on the next slide) and then use them to solve the original linear systems $\mathbf{Ax} = \mathbf{b}$ with the two techniques: backward substitution and forward substitution.

We will conclude with some numerical instability issues.

Upper-Triangular Matrices

An $n \times n$ matrix is **upper-triangular** if it has the following form:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & a_{nn} \end{bmatrix},$$

where $a_{ij} \geq 0$, if $i \leq j$, and $a_{ij} = 0$ everywhere else. Basically, all entries below the main diagonal from top left to bottom right are 0's.

Lower-Triangular Matrices

An $n \times n$ matrix is **lower-triangular** if it has the following form:

$$A = \begin{bmatrix} a_{11} & 0 & \dots & \dots & 0 \\ a_{21} & a_{22} & \dots & \dots & 0 \\ a_{31} & a_{32} & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{n(n-1)} & a_{nn} \end{bmatrix},$$

where $a_{ij} \geq 0$, if $i \geq j$, and $a_{ij} = 0$, if $i < j$. Basically, all entries above the main diagonal from top left to bottom right are 0's.

Several Useful Theorems on Determinants

Theorem 1: If a single row of a square matrix $\mathbf{A}$ is multiplied by a scalar k, then the determinant of the resulting matrix is $k \cdot \texttt{det}(\mathbf{A})$.

Theorem 2: A square matrix $\mathbf{A}$ is inevitable if and only if $\texttt{det}(\mathbf{A}) \neq 0$.

Theorem 3: If $\mathbf{A}$ and $\mathbf{B}$ are square matrices, then $\texttt{det}(\mathbf{AB}) = \texttt{det}(\mathbf{A})\texttt{det}(\mathbf{B})$.

Theorem 4: The determinant of an upper- or lower-triangular matrix is the product of its pivots.

Example

Here's a quick example of **Theorem 3** and **Theorem 4**. Let

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 3 & 0 \\ 4 & 2 & 1 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \end{bmatrix}.$$

Then $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{A}) = (2 \cdot 3 \cdot 1)(1 \cdot 1 \cdot 2) = 6 \cdot 2 = 12$.

Review of Lecture 02

You may want to review Slides 25 – 30 in Lecture 02 to refresh your memory on the identity and invertible matrices.

In class today, 01/17/2024, we went over Cramer's rule on Slides 31 and 32 in Lecture 02.

Review of Lecture 02

You may want to review Slides 25 – 30 in Lecture 02 to refresh your memory on the identity and invertible matrices.

In class today, 01/17/2024, we went over Cramer's rule on Slides 31 and 32 in Lecture 02.

Numpy Minute: Inverting Random Matrices

The code segment below shows you how to generate and invert a random matrix when its determinant is not 0. We use `np.linalg.inv()` to to invert a random matrix, `np.allclose()` to compare AA^1 to the 3x3 identity matrix obtained with `np.eye(3)`.

```
import numpy as np
import numpy.random
if __name__== '__main__':
    n = 3
    A = np.random.rand(n, n)
    while not np.allclose(np.linalg.det(A), 0):
        A = np.random.rand(n, n)
        break
    assert np.allclose(np.dot(A, np.linalg.inv(A)),
                                np.eye(n))
    print('Assertion passed')
```

Cramer's Rule Example: Step 1

Let's solve the following linear system with Cramer's rule.

$$\begin{array}{rcrcrcl} 5x_1 & - & 2x_2 & + & x_3 & = & 1 \\ 3x_1 & + & 2x_2 & + & 0x_3 & = & 3 \\ x_1 & + & x_2 & - & x_3 & = & 0 \end{array}$$

Cramer's Rule Example: Step 2

Let's set up the linear system in numpy.

```
import numpy as np
A = np.array([[5, -2, 1],
              [3, 2, 0],
              [1, 1, -1]],
             dtype=float)
b = np.array([[1],
              [3],
              [0]],
             dtype=float)
```

Cramer's Rule Example: Step 3

Let's compute the B matrices for Cramer's rule.

```
B0, B1, B2 = A.copy(), A.copy(), A.copy()
B0[:,0] = b.reshape(3)
B1[:,1] = b.reshape(3)
B2[:,2] = b.reshape(3)
```

Cramer's Rule Example: Step 4

Let's apply Cramer's rule and check the solution. The assert below should pass.

```
detA = np.linalg.det(A)
x = np.array([[np.linalg.det(B0)/detA],
              [np.linalg.det(B1)/detA],
              [np.linalg.det(B2)/detA]])
assert np.allclose(np.dot(A, x), b)
print('Assertion passed...')
```

LU-Decomposition

In many areas of scientific computing, situations arise when it's necessary to solve many systems all having the same coefficient matrix $\mathbf{A}$ but different vectors $\mathbf{b}$. In other words, we need to solve $[\mathbf{A}|\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}]$.

One may reasonably ask why not use Gauss-Jordan Elimination? For small systems, the answer is, yes, of course, we can! And should!. For larger systems, however, where the coefficient matrix is large (e.g., 10,000 x 10,000) and the number of column vectors $\mathbf{b}$ is also large, it may not be a good idea for efficiency reasons.

There are two other factors against using Gauss-Jordan Elimination in this situation.

1. Column vectors $\mathbf{b}$ are generated over a period of time and, which is even more important, cannot be known in advance (data stream is continuous);
2. Column vector $\mathbf{b_{j+1}}$ depends on column vector $\mathbf{b_j}$.

LU-Decomposition Theorem

We'll work with square linear systems $\mathbf{Ax} = \mathbf{b}$ that have a unique solution that can be found by Gauss-Jordan Elimination and back substitution without having to interchange any rows is based on the following theorem.

Theorem: If $\mathbf{A}$ is an $n \times n$ matrix that can be reduced to row echelon form by Gauss-Jordan Elimination without any row interchanges, then $\mathbf{A}$ can be factored into a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$ such that $\mathbf{A} = \mathbf{LU}$.

To use LU-Decomposition, we need to know the backward and forward subsitution.

General Formula for Back Substitution

Assume that we have the following linear system, where the coefficient matrix is upper-triangular.

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

Then general formula for x_i is as follows.

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

General Formula for Forward Substitution

Assume that we have the following linear system where the coefficient matrix is lower-triangular.

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

Then general formulas for y_i is as follows.

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

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

Using LU-Decomposition to Solve $\mathbf{Ax} = \mathbf{b}$

The LU algorithm consists of 3 steps.

1. Decompose $\mathbf{A}$ to $\mathbf{L}$ and $\mathbf{U}$ such that $\mathbf{LU} = \mathbf{A}$ (next slide will show us how to do that). Thus, $\mathbf{Ax} = \mathbf{LUx} = \mathbf{b}$.
2. Solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$.
3. Solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$.

Thus, $\mathbf{Ax} = \mathbf{LUx} = \mathbf{Ly} = \mathbf{b}$.

Reduction of **A** to **U** and Generation of **L** from **I**

It turns out that there's a method to reduce **A** to **U** and, at the same time, to generate **L** by recording row addition operations in the identity matrix **I**. You should bear in mind that all these matrices are $n \times n$.

The method is as follows:

1. Start with $n \times n$ identity matrix **I**;
2. If during the reduction of **A** to **U**, we add $r\cdot$ row i to row k, we replace the zero in row k and column i in **I** by $-r$. In other words, we do $\mathbf{I}[\mathbf{k}, \mathbf{i}] = -\mathbf{r}$;

After this procedure ends, **A** is row-reduced to **U** and **I** has turned into **L**.

Example

Let's solve the following linear system $[\mathbf{A}|\mathbf{b}]$ with LU-Decomposition.

$$[\mathbf{A}|\mathbf{b}] = \left[\begin{array}{cccc|c} 1 & -2 & 0 & 3 & 11 \\ -2 & 3 & 1 & -6 & -21 \\ -1 & 4 & -4 & 3 & -1 \\ 5 & -8 & 4 & 0 & 23 \end{array}\right]$$

Example: Step 1

We first put the two matrices **A** and **I** side by side. Remember that we will turn **A** to **U** and **I** to **L**. Step 1 will consist of 6 substeps.

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ -2 & 3 & 1 & -6 \\ -1 & 4 & -4 & 3 \\ 5 & -8 & 4 & 0 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: Step 1

1. We add $2 \cdot$ row 1 to row 2 and record -2 in **I[2,1]**

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ -1 & 4 & -4 & 3 \\ 5 & -8 & 4 & 0 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. We add $1 \cdot$ row 1 to row 3 and record -1 in **I[3,1]**

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ 0 & 2 & -4 & 6 \\ 5 & -8 & 4 & 0 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Example: Step 1

3. We add -5 · row 1 to row 4 and record 5 in **I[4,1]**

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ 0 & 2 & -4 & 6 \\ 0 & 2 & 4 & -15 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{bmatrix}$$

4. We add 2 · row 2 to row 3 and record -2 in **I[3,2]**

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -2 & 6 \\ 0 & 2 & 4 & -15 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & -2 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{bmatrix}$$

Example: Step 1

5. We add $2 \cdot$ row 2 to row 4 and record -2 in **I[4,2]**

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 0 & 3 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -2 & 6 \\ 0 & 0 & 6 & -15 \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -1 & -2 & 1 & 0 \\ 5 & -2 & 0 & 1 \end{bmatrix}$$

6. We add $3 \cdot$ row 3 to row 4 and record -3 in **I[4,3]**

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

We are done with Step 1 of the algorithm. Note that **A** is now **U** and **I** is now **L**.

Side Note: Storing **L** and **U** in Single Matrix

We don't need two separate matrices **L** and **U**. We can store them in a single matrix.

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

We can denote this matrix **L\U**, where the values of **L** are stored below the main diagonal and the values of **U** are saved on and above the main diagonal.

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

Forward and Backward Substitution

Let A be an $n \times n$ lower-triangular matrix. Then forward substitution can be used to solve $\mathbf{Ax} = \mathbf{b}$ if $\mathbf{A}$ is lower-triangular whereas backward substitution can be used if $\mathbf{A}$ is upper-triangular.

What Does LU-Decomposition Buy Us?

Let $\mathbf{Ax} = \mathbf{b}$ be a linear system such that $\mathbf{A}$ is an $n \times n$ invertible matrix, $\mathbf{x}$ is $n \times 1$, and $\mathbf{b}$ is $n \times 1$.

We can decompose $\mathbf{A}$ to $\mathbf{L}$ and $\mathbf{U}$, solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$ with **forward substitution**, and solve $\mathbf{Ux} = \mathbf{y}$ for $\mathbf{x}$ with **back substitution**. Then $\mathbf{Ax} = \mathbf{LUx} = \mathbf{Ly} = \mathbf{b}$.

Notes on Efficiency

Assume that $\mathbf{A}$ is $n \times n$, invertible, and can be reduced to the upper-triangular form without any row interchanges. Suppose we factor $\mathbf{A}$ into $\mathbf{U}$ and $\mathbf{L}$ and save both matrices. Does it buy us anything when solving $\mathbf{Ax} = \mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_k}$?

Notes on Efficiency

1. Once we have **L** and **U**, backward and forward substitutions take $\approx n^2/2$ operations each. So, we have a total of $\approx n^2$ operations.

2. Suppose we use GJE to solve $\mathbf{Ax} = \mathbf{b_i}$, for every i. The reduction of **A** to **U** takes $\approx n^3/3$ and subsequent back substitution takes $\approx n^2/2$. So, we have a total of $\approx n^3/3 + n^2/2$ operations.

3. Suppose that we have a 1000×1000 matrix **A**. Then, using the saved **U** and **L** requires $\approx 1,000,000$ operations whereas using GJE requires $\approx 333,833,333$, i.e., `(1000**3)/3 + (1000**2)/2` operations.

If space is a concern, **U** and **L** can be stored in the same matrix.

Takeaway of LU-Decomposition

Use GJE on smaller linear systems or on systems whose coefficient matrices you intend to throw away, i.e., it's not worth it to decompose them into **L** and **U**.

Use LU-decomposition on larger linear systems whose equations you intend to keep and solve for different right-hand side vectors.

Numerical Instability in Python 3

Let's define the following 3 small, positive floats in Python.

```
>>> z1 = 5e-324
>>> z2 = 1e-323
>>> z3 = 5e-323
>>> 0 < z1 < z2 < z3
True
```

Numerical Instability in Python 3

Let's add these three numbers to non-zero positive or negative numbers.

```
>>> x = 1009
>>> x + z1 == x + z2 == x + z3
True
>>> x = -13
>>> x + z1 == x + z2 == x + z3
True
>>> import math
>>> x = math.pi
>>> x + z1 == x + z2 == x + z3
True
>>> x = -math.pi
>>> x + z1 == x + z2 == x + z3
True
```

Numerical Instability Theorem

We have constructively proved the following theorem in Python 3.

There exist positive real numbers z_1, z_2, z_3 such that $z_1 < z_2 < z_3$ and for some real numbers x such that $|x| > 0$, $x + z_1 = x + z_2 = x + z_3$.

Do not think that this theorem holds only in Python 3. I was able to find such numbers in Python 2 and many other programming languages (e.g., C, Perl, Java, Lisp, etc.).

Takeaway: We should forget about == in Python and use `np.allclose()` to compare numeric values only approximately, i.e., with some error expectation.

References

1. Online documentation at www.numpy.org.
2. J. Fraleigh, R. Beauregard. *Linear Algebra*, Ch. 09.