CS 3430 S24: Scientific Computing
Assignment 01
Gauss-Jordan Elimination, Determinants, Leibnitz's Formula, Cramer's Rule

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 13, 2024

Learning Objectives

1. Linear Systems
2. Gauss-Jordan Elimination
3. Determinants
4. Leibnitz's Formula
5. Cramer's Rule
6. Numpy

Introduction

In this assignment, we will us the Gauss-Jordan elimination (GJE), determinants, Leibnitz's Formula, and Cramer's rule to solve linear systems. This assignment will also give you more exposure to `numpy`. You will save your coding solutions in `cs3430_s24_hw01.py` included in the zip and submit it in Canvas.

There's no universal set of instructions on how to install `numpy`. It all depends on your OS, your IDE, and frequently a combination thereof. The best strategy is to go www.numpy.org/install/ and follow instructions for your OS/IDE. Do not be shy to share your installation experiences, recommendations, or shortcuts with the class. Help each other. This is a university course, not a rat race.

Our assignments for this course, like this one, will include three types of activites: 1) optional problems, 2) coding labs, and 3) problems. Optional problems may include some reading suggestions, additional examples, or problems that you may want to do to improve your understanding of the material. You do not have to turn anything for these problems. You may view them as study tips. Coding labs are Python coding examples that you may find useful in doing the actual problems. Problems with the actual points next to them are what you are responsible for and what you should turn in your Python source files per the instructions in the section **What to Submit** at the end of each assignment.

When you read the text of each assignment and want to copy Python code segments into your Python IDE/interpreter, make sure that all the commas and quotes paste properly. I use Linux (Ubuntu 18.04LTS and Ubuntu 22.02LTS) and LaTeX to typeset my documents and some characters may not paste properly into your Python editors. Be cognizant of this source of potential error. I use the symbol `>>>` in code segments to mean that a Python expression to the right of this symbol can be evaluated in a Python interpreter.

Optional Problem 0

Review the CS 3430 S24 syllabus and your notes and/or the PDFs of Lectures 01 and 02 in Canvas.

Coding Lab 1: Gauss-Jordan Elimination

Recall that in Lectures 01 and 02, we discussed linear systems. A linear system can be expressed as $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ *coefficient matrix*, $\mathbf{x}$ is an $n \times 1$ matrix (aka *column vector*), and $\mathbf{b}$ is also an $n \times 1$ matrix. The vector $\mathbf{x}$ is sometimes called the *variable vector*, because it consists of the variables whose values we want to find. The vector $\mathbf{b}$ is sometimes called the **value vector**, because it consists of the right-hand side values of the linear equations. Our goal is to calculate $\mathbf{x}$.

Let us consider the following linear system with which we will work in the coding lab.

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 4 \\ 3x_1 + 2x_3 &= 5 \\ -2x_1 + x_2 + 4x_3 &= 6 \end{aligned}$$

If we make the zero coefficients explicit, the system will look as follows.

$$\begin{aligned} 2x_1 - x_2 + 3x_3 &= 4 \\ 3x_1 + 0x_2 + 2x_3 &= 5 \\ -2x_1 + x_2 + 4x_3 &= 6 \end{aligned}$$

In this system, the coefficient matrix is

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

and the value vector $\mathbf{b}$ is

$$\mathbf{b} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}.$$

To solve this system, we need to find the components of the variable vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

such that $\mathbf{Ax} = \mathbf{b}$ or, if we express the linear system with matrices as we did in the first two lectures,

$$\begin{bmatrix} 2 & -1 & 3 \\ 3 & 0 & 2 \\ -2 & 1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}.$$

We are ready to set us this system with `numpy`. Setting up a linear system in `numpy` means defining the coefficient and value matrices in `numpy`. Here's how.

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

You may wonder why we defined both matrices in terms of floats, i.e., used `dtype=float`. The main reason is that many linear systems do not have integer solutions, and our system is no exception. The vector `b` can also be defined with the function `np.resize()` as follows.

```
>>> b = (np.array([4, 5, 5], dtype=float))
>>> b
array([4., 5., 5.])
>>> b = np.resize(b, (3, 1))
>>> b
array([[4.],
       [5.],
       [5.]])
```

In the first line above, we define `b` as an array of 3 floats and then resize it to be a column vector, i.e., a 3 x 1 matrix. We are ready to solve the system using the GJE method. The `numpy.linalg` package has the function `numpy.linalg.solve()` that allows us to do it.

```
import numpy as np
import numpy.linalg

>>> A
array([[ 2., -1.,  3.],
       [ 3.,  0.,  2.],
       [-2.,  1.,  4.]])
>>> b
array([[4.],
       [5.],
       [5.]])
>>> x = np.linalg.solve(A, b)
>>> x
array([[0.80952381],
       [1.47619048],
       [1.28571429]])
```

The mathematical interpretation of the above output is that the solution of this linear system (i.e,, the individual components of the vector **x**) is $x_1 = 0.80952381$, $x_2 = 1.47619048$, $x_3 = 1.28571429$.

How do we know that this solution is correct? One way to check is to plug these values into each equation of the original linear system and see if it gives us the corresponding right-hand side value in **b**. Here's how.

```
>>> x1, x2, x3 = x[0][0], x[1][0], x[2][0]
>>> x1
0.8095238095238094
>>> x2
1.4761904761904763
>>> x3
1.2857142857142858
>>> 2*x1 - x2 + 3*x3
4.0
>>> 3*x1 + 0*x2 + 2*x3
5.0
>>> -2*x1 + x2 + 4*x3
5.000000000000001
```

It looks like our solution is fine. but checking its correctness sure requires a lot of typing. If we have, say, a 100 x 100 coefficient matrix **A**, this is not a method to use. A much better shortcut is to use the numpy matrix multiplication function `np.dot()` to multiply **A** and **x** and see if we get **b** back. And, in this case, we do!

```
>>> np.dot(A, x)
array([[4.],
       [5.],
       [5.]])
```

There is one more important point that I would like to make about checking numerical solutions to linear systems (and to many other scientific computing methods) – the ever constant presence of error. Rounding and precision error is invetable in scientific computing, especially when we are using very small or very large numbers. Why is this the case? Because there are only so many numbers that can be represented in a finite amount of computer memory. We can pretend that memory is infinite, but it is just that – pretense. And rounding/precision error is the price we pay for the illusion of infinite memory.

To manage error, we should avoid using the numerical equality, i.e., `==`, and instead expect that our results with be equal with some small degree of tolerance. The numpy function `np.allclose()` can be used for this. The function has the following signature

```
np.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
```

and returns `True` if two arrays, `a` and `b`, are elementwise sufficiently close, i.e., equal within a numerical tolerance. The tolerance values, `rtol` and `atol`, are very small positive numbers. The relative difference, `rtol`, and the absolute difference, `atol`, are added together to compare against the absolute difference between the corresponding elements of `a` and `b`. We can use the default values given in the signature or change them to less strict values for specific problems. Here's how we can use this function to check if our solution is tolerable.

```
>>> np.allclose(np.dot(A, x), b)
True
```

Does every linear system have a solution? Unfortunately, no. If a linear system that does not have a solution is *inconsistent*. Here's an example.

$$\begin{aligned} 2x_1 + 2x_2 &= 5 \\ -2x_1 - 2x_2 &= 3 \end{aligned}$$

Let us attempt to solve it.

```
>>> A = np.array([[2, 2], [-2, -2]], dtype=float)
>>> b = np.resize(np.array([5, 3], dtype=float), (2, 1))
>>> A
array([[ 2.,  2.],
       [-2., -2.]])
>>> b
array([[5.],
       [3.]])
>>> np.linalg.solve(A, b)
...
numpy.linalg.LinAlgError: Singular matrix
```

In the output above I skipped some of the output. What's important is the `np.linalg.solve()` raised an exception of the type `numpy.linalg.LinAlgError` stating that the matrix is singular. A singular matrix is simply a matrix for which we cannot compute the inverse matrix (cf. slides 27, 28 in Lecture 02). This exception is always raised by `np.linalg.solve()` when the system is not solvable. A lesson from this example is to always enclose the calls to `np.linalg.solve()` inside the `try ... except` block.

```
try:
   np.linalg.solve(A, b)
except np.linalg.LinAlgError as e:
   print(e)
```

Problem 1 (1 point)

Set up and solve the following linear systems.

1. $$\begin{array}{ccccc} x_1 & + & x_2 & = & 3 \\ x_1 & + & 4x_2 & = & 10 \end{array}$$

2. $$\begin{array}{ccccccc} & & x_2 & - & 3x_3 & = & -5 \\ 2x_1 & + & 3x_2 & - & x_3 & = & 7 \\ 4x_1 & + & 5x_2 & - & 2x_3 & = & 10 \end{array}$$

3.

$$\begin{array}{rcrcrcl} 2x_1 & - & x_2 & + & 3x_3 & = & 4 \\ 3x_1 & & & + & 2x_3 & = & 5 \\ -2x_1 & + & x_2 & + & 4x_3 & = & 6 \end{array}$$

4.

$$\begin{array}{rcrcrcrcl} -x_1 & & & + & x_3 & - & x_4 & = & 3 \\ 2x_1 & + & 2x_2 & - & x_3 & - & 7x_4 & = & 1 \\ 4x_1 & - & x_2 & - & 9x_3 & - & 5x_4 & = & 10 \\ 3x_1 & - & x_2 & - & 8x_3 & - & 6x_4 & = & 1 \end{array}$$

Problem 2: Leibnitz's Determinant (2 points)

Recall the absolute value of determinants can be interpreted as volumes of hyper boxes in n-dimensional hyper spaces. For example, in 2D, determinants are areas and in 3D – regular volumes. In n-dimensional spaces, $n > 3$, determinants are critical in computing volumes of n-dimensional hyper boxes, which lies at the very heart of integral calculus. Implement the function `leibnitz_det()` to compute the determinant of an $n \times n$ matrix (cf. Slides 17 – 22 in Lecture 02). The function `leibnitz_det()` uses the method with the minor matrices and cofactors. You can use `np.linalg.det()` to check the correctness of your results.

Let us start by manually computing the determinant of the matrix below, test `leibnitz_det()` on it, and compare our results to the result returned by `np.linalg.det()`. We will work with this matrix.

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

The determinant of $\mathbf{A}$ is

$$det(A) = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 1 & 2 \\ 1 & 2 & -3 \end{bmatrix} = 2 \cdot \begin{vmatrix} 1 & 2 \\ 2 & -3 \end{vmatrix} - 1 \cdot \begin{vmatrix} 4 & 2 \\ 1 & -3 \end{vmatrix} + 3 \cdot \begin{vmatrix} 4 & 1 \\ 1 & 2 \end{vmatrix} = 2 \cdot (-7) - (-14) + 3 \cdot (7) = 21.$$

Here's a sample run of my implementation of `leibnitz_det()`.

```
>>> from cs3430_s24_hw01 import *
>>> A = np.array([[2, 1, 3],
                  [4, 1, 2],
                  [1, 2, -3]],
                  dtype=float)
>>> A
array([[ 2.,  1.,  3.],
       [ 4.,  1.,  2.],
       [ 1.,  2., -3.]])
>>> leibnitz_det(A)
21.0
>>> np.linalg.det(A)
21.0
>>> np.allclose(leibnitz_det(A), np.linalg.det(A))
True
```

In implementing this function, you may want to create minor matrices by making a copy of the major matrix and then deleting specific rows and columns in it. You can do these deletions efficiently with the numpy function `np.delete()` and the numpy slices `np.s_`. Here's an example.

```
import numpy as np

### let’s create a matrix
>>> A = np.array([[2, 1, 3],
                  [4, 1, 2],
                  [1, 2, -3]],
                  dtype=float)
>>> A
array([[ 2.,  1.,  3.],
       [ 4.,  1.,  2.],
       [ 1.,  2., -3.]])

### let’s create a copy of A; this is critical, because
### np.delete() is destructive.
>>> copyOfA = A.copy()
>>> copyOfA
array([[ 2.,  1.,  3.],
       [ 4.,  1.,  2.],
       [ 1.,  2., -3.]])
### let’s use np.s_ to delte row 1; remember that
### when you delete a row, axis=0
>>> np.delete(copyOfA, np.s_[1:2], axis=0)
array([[ 2.,  1.,  3.],
       [ 1.,  2., -3.]])
### note that row 1, i.e., [4, 1, 2], is gone in copyOfA,
### A is unchanged.
>>> A
array([[ 2.,  1.,  3.],
       [ 4.,  1.,  2.],
       [ 1.,  2., -3.]])
### let’s make another copy.
>>> copyOfA = A.copy()
### let’s delete column 2 from copyOfA; remember that
### when you delete a column, axis=1.
>>> np.delete(copyOfA, np.s_[2:3], axis=1)
array([[2., 1.],
       [4., 1.],
       [1., 2.]])
>>> A
array([[ 2.,  1.,  3.],
       [ 4.,  1.,  2.],
       [ 1.,  2., -3.]])
```

Nothing super complicated. But, remember to create a copy of the major matrix and delete from the copy! When you run the unit test, you may notice that the larger the matrix becomes, the slower this method runs. We will talk about making it more efficient with multiprocessing in Lecture 03 on 01/17/2024.

Problem 3: Cramer’s Rule (2 points)

Cramer’s rule (see slides 31–33 in Lecture 02), named after the Swiss mathematician Gabriel Cramer (1704 - 1752), is a beautiful method of solving square linear systems. Learning Cramer’s rule will add another method to your repertoire of solving linear systems in addition to the Gauss-Jordan Elimination method. Knowing Cramer's rule is useful, because it routinely shows up in advanced calculus and many other areas of scientific computing. While Cramer's rule still enjoys much theoretical fame and utility, it is not widely used to solve linear systems any more, because Gauss-Jordan elimination along with other methods discovered in the past 30 years are more efficient. For example, you want to take a look at Slide 24 in Lecture 02, which outlines a more efficient way to compute the determinant.

Use your implementation of `leibnitz_det()` to implement the function `cramers_rule(A, b)` that uses Cramer's rule to solve the linear system $\mathbf{Ax} = \mathbf{b}$. Let us solve two consistent systems with Cramer's rule and compare the solutions with those computed by `gje(A, b)`. Here's a sample run of my implementation.

```
>>> from cs3430_s24_hw01 import *
A = np.array([[ 2., -1.,  3.],
              [ 3.,  0.,  2.],
              [-2.,  1.,  4.]])
b = np.array([[4.],
           [5.],
           [6.]])
>>> cramers_rule(A, b)
array([[0.71428571],
       [1.71428571],
       [1.42857143]])
>>> np.dot(A, cramer(A, b))
array([[4.],
       [5.],
       [6.]])
>>> np.dot(A,
array([[4.],
       [5.],
       [6.]])
       [ 3.]])
```

Testing Your Code

Save all your code in `cs3430_s24_hw01.py` where I wrote some starter code for you. Do not change the function names. They are used in `cs3430_s24_hw01_uts.py`, which is the unit test file where I wrote 4 unit tests for Problem 1, 5 unit tests for Problem 2, and 5 unit tests for Problem 3. You can use these to test your code for each problem. I recommend that you comment them all out at first and uncomment and run them one by one as you work on each function and problem. When you run it, your output should look as follows.

```
>>> python cs3430_s24_hw01_uts.py
CS 3430: S24: HW01: Problem 1.1: Unit Test...
CS 3430: S24: HW01: Problem 1.1: Unit Test: pass
.
CS 3430: S24: HW01: Problem 1.2: Unit Test...
CS 3430: S24: HW01: Problem 1.2: Unit Test: pass
.
CS 3430: S24: HW01: Problem 1.3: Unit Test...
CS 3430: S24: HW01: Problem 1.3: Unit Test: pass
.
```

```
CS 3430: S24: HW01: Problem 1.4: Unit Test...
CS 3430: S24: HW01: Problem 1.4: Unit Test: pass
...
```

What to Submit

Save all your code in `cs3430_s24_hw01.py` and submit it in Canvas.

Happy Hacking!