CS 3430: S22: Scientific Computing
Assignment 01
Gauss-Jordan Elimination, Determinants, Leibnitz's Formula, Cramer's Rule

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 15, 2022

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'll implement Gauss-Jordan elimination, determinants, Leibnitz's Formula, and Cramer's rule. This assignment will also give you more exposure to `numpy`.

You will save your coding solutions in `cs3430_s22_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 https://numpy.org/install/ and follow instructions for your OS/IDE.

You should write your own code. You may use the numpy methods in the assignment descriptions and in the sampel files. However, you may not use any third-party packages (e.g., `sympy` or `numpy.linalg`) that have linear system modules. There's no substitute for doing your own work if you want to understand how scientific computing works. Moreover, (and I'm speaking from experience now), these modules contain subtle bugs that are hard to detect if you don't understand the algorithms. You should implement the required methods from scratch to learn the ins and outs of Gauss-Jordan elimination and linear systems. Don't be a MATLAB programmer who knows the value of everything and the computation of nothing.

Problem 0

Review the slides of Lectures 01 and 02 in the Canvas modules for these lectures. I've include two Python files in the lecture moduels with code samples illustrating the points made in the lectures (e.g., how to create numpy matrices, how to swap two rows in a numpy matrix, how to compute the inverse of a matrix, etc.) When Python was not one of the official languages of the CS undergraduate curriculum, I used to dedicate the first 3 weeks of this class for an intensive Python tutorial. Since Python is now the official programming language of CS1, the Python tutorial is no longer part of this course. If you're new to Python, make sure that you run and read these code samples. Several students asked me what version of Python they should use. Anything greater than or equal to Py 3.6 should work. I use Py 3.6.7, which ships with many distributions of Linux.

When you read the text of the assignment and want to copy Python code segments into your Python IDE, make sure that all the commas and quotes paste properly. I use Linux (Ubuntu 18.04LTS) and LaTeX to typeset my documents and some of the characters may not paste properly into Windows/Mac editors. That's what I heard from some students in the past.

Problem 1: Gauss-Jordan Elimination (1 point)

Implement the function `gje(A, b)` that does Gauss-Jordan Elimination and returns the vector $\mathbf{x}$, if it exists, that solves the linear system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{x}$ is an $n \times 1$ matrix, and $\mathbf{b}$ is also an $n \times 1$ matrix. Let's consider the following linear system.

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

In this linear system, the coefficient matrix $\mathbf{A}$ is

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

and the column vector $\mathbf{b}$ of the right-hand side values of each linear equation is

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

To solve this linear system, we need to find

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

such that $\mathbf{Ax} = \mathbf{b}$ or

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

Let's define these matrices in Python with numpy. Note that in each matrix we're defining the type of the element as `dtype=float`, which is the appropriate choice when working with real numbers.

```
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)
```

Here's how your implementation of `gje` should handle this linear system. You floats should be in the same ballpark, not exactly the same. Float rounding varies from system to system.

```
>>> from cs3430_s22_hw01 import *
>>> x = gje(A, b)
>>> x
array([[0.71428571],
       [1.71428571],
       [1.42857143]])
```

Thus, $x_1 = 0.71428571$, $x_2 = 1.71428571$, and $x_3 = 1.42857143$.

We can check the correctness of this solution with numpy's functions `np.dot()` or `np.matmul()` that do matrix multiplication.

```
>>> import numpy as np
>>> np.dot(A, x)
array([[4.],
       [5.],
       [6.]])
>>> np.matmul(A, x)
array([[4.],
       [5.],
       [6.]])
```

If the linear system is inconsistent (i.e., has no solution – see slides 6 and 10 in the Lecture 02 PDF), `gje(A, b)` should return `None`. Here's an inconsistent linear system.

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

Let's convert it into Python and solve it with `gje()`.

```
A = np.array([[2, 2],
              [-2, -2]],
             dtype=float)
b = np.array([[5],
              [3]],
             dtype=float)
>>> x = gje(A, b)
>>> x == None
True
```

Let's go though another example of a consistent linear system and its solution.

$$
\begin{aligned}
x_2 + x_3 &= 6 \\
3x_1 - x_2 + x_3 &= -7 \\
x_1 + x_2 - 3x_3 &= -13
\end{aligned}
$$

```
A = np.array([[0, 1, 1],
              [3, -1, 1],
              [1, 1, -3]],
             dtype=float)
b = np.array([[6],
              [-7],
              [-13]],
             dtype=float)
>>> x = gje(A, b)
array([[-3.],
       [ 2.],
       [ 4.]])
>>> np.dot(A, x)
array([[  6.],
       [ -7.],
       [-13.]])
```

Problem 2: Determinants (1 point)

Recall that in 2D and 3D, determinants are areas and volumes. In m-dimensional spaces, determinants are critical in computing volumes of m-dimensional hyper-boxes, which lies at the very heart of integral calculus. Implement two functions `leibnitz_det()` and `gauss_det()` to compute the determinant of an $n \times n$ matrix.

The function `leibnitz_det()` (See slides 18–19 in Lecture 02 or your class notes) uses the method with the minor matrices and cofactors. The function `gauss_det()` uses the method that uses Gauss elimination and the product of pivots (see slide 20 in Lecture 02). You can use `np.linalg.det()` to check the correctness of your results.

Let's manually compute the determinant of the matrix below, test both functions on it, and compare our results with `np.linalg.det()`.

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

The determinant of $\mathbf{A}$ is

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

All values are exactly the same in Python.

```
>>> from cs3430_s22_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
>>> gauss_det(A)
21.0
>>> np.linalg.det(A)
21.0
>>> leibnitz_det(A) == gauss_det(A) == np.linalg.det(A)
True
>>> leibnitz_det(A7)
21.0
>>> gauss_det(A7)
21.0
>>> np.linalg.det(A7)
21.0
```

More examples are below. Note that even on small matrices where one can compute the determinant manually there may be slight discrepancies between `gauss_det()` and `leibnitz_det()`, on the one hand, and `np.linalg.det()`, on the other. In general, the larger the matrices, the larger the discrepancies.

```
>>> from cs3430_s22_hw01 import *
>>> A1
array([[2., 1., 0., 1.],
       [3., 2., 1., 2.],
       [4., 0., 1., 4.],
       [1., 0., 2., 1.]])
>>> leibnitz_det(A1)
-7.0
>>> gauss_det(A1)
-7.0
>>> np.linalg.det(A1)
-6.999999999999998

>>> A2
array([[ 5., -2.,  4., -1.],
       [ 0.,  1.,  5.,  2.],
       [ 1.,  2.,  0.,  1.],
       [-3.,  1., -1.,  1.]])
```

```
>>> leibnitz_det(A2)
-8.0
>>> gauss_det(A2)
-8.000000000000014
>>> np.linalg.det(A2)
-7.999999999999998

>>> A3
array([[ 3.,  2.,  0.,  1.,  3.],
       [-2.,  4.,  1.,  2.,  1.],
       [ 0., -1.,  0.,  1., -5.],
       [-1.,  2.,  0., -1.,  2.],
       [ 0.,  0.,  0.,  0.,  2.]])
>>> leibnitz_det(A3)
12.0
>>> gauss_det(A3)
12.0
>>> np.linalg.det(A3)
12.000000000000005
```

I wrote the function the function `random_mat(nr, nc, lower, upper)` in `cs3430_s22_hw01.py` that creates an `nr` $\times$ `nc` matrix of random numbers in the range `[lower, upper]`. Let's create a 10×10 matrix and compute its determinants. The call to `leibnitz_det()` will take a while.

```
>>> from cs3430_s22_hw01 import *
>>> A = random_mat(10, 10, 1, 3)
>>> A
array([[3., 2., 1., 1., 3., 1., 3., 2., 2., 3.],
       [2., 3., 2., 3., 1., 3., 1., 3., 2., 1.],
       [2., 1., 3., 1., 2., 1., 1., 1., 2., 3.],
       [1., 1., 3., 3., 3., 1., 2., 1., 1., 3.],
       [1., 2., 2., 2., 3., 1., 1., 1., 2., 2.],
       [2., 2., 1., 2., 3., 1., 3., 2., 3., 1.],
       [3., 1., 2., 2., 1., 3., 3., 3., 2., 3.],
       [3., 3., 1., 1., 2., 2., 2., 2., 1., 1.],
       [3., 2., 1., 3., 2., 1., 3., 1., 2., 3.],
       [2., 3., 2., 2., 1., 1., 1., 3., 1., 1.]])
>>> gauss_det(A)
405.00000000000006
>>> leibnitz_det(A)
405.0
>>> np.linalg.det(A)
405.0000000000009
```

They're in the same ballpark. Let's repeat this exercise with a 100x100 and a 200x200 random matrix. The eleven digits after the decimal point are identical in each determinant, then there are discrepancies b/w `gauss_det` and `np.linalg.det`. Calling `leibnitz_det()` on these matrices is a hopeless pursuit unless you have an infinite amount of time.

```
>>> from cs3430_s22_hw01 import *
>>> A = random_mat(100, 100, 1, 3)
>>> gauss_det(A)
3.977395749581346e+70
```

```
>>> np.linalg.det(A)
3.977395749584559e+70
>>> A = random_mat(100, 100, 1, 3)
>>> gauss_det(A)
3.977395749581346e+70
>>> np.linalg.det(A)
3.977395749584559e+70
>>> A = random_mat(200, 200, 1, 3)
>>> gauss_det(A)
3.373361546426203e+169
>>> np.linalg.det(A)
3.373361546436552e+169
```

Problem 3: Cramer's Rule (1 point)

Cramer's rule (see slides 27–29 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 method. Knowing Cramer's rule is useful, because it routinely shows up in advanced calculus and many areas of scientific computing. While Cramer's rule still enjoys much theoretical fame, it is not widely used to solve linear systems any more, because Gauss-Jordan elimination along with other methods discovered in the past 25 years are much more efficient.

Implement the function `cramer(A, b)` that uses Cramer's rule, as discussed in Lecture 02, to solve the linear system $\mathbf{Ax} = \mathbf{b}$. Let's solve two consistent systems with Cramer's rule and compare the solutions with those computed by `gje(A, b)`.

```
>>> from cs3430_s22_hw01 import *
A = np.array([[ 2., -1.,  3.],
           [ 3.,  0.,  2.],
           [-2.,  1.,  4.]])
b = np.array([[4.],
           [5.],
           [6.]])
>>> cramer(A, b)
array([[0.71428571],
       [1.71428571],
       [1.42857143]])
>>> gje(A, b)
array([[0.71428571],
       [1.71428571],
       [1.42857143]])
>>> np.dot(A, cramer(A, b))
array([[4.],
       [5.],
       [6.]])
>>> np.dot(A, gje(A, b))
array([[4.],
       [5.],
       [6.]])

A = np.array([[ 0.,  1., -3.],
           [ 2.,  3., -1.],
           [ 4.,  5., -2.]])
b = np.array([[-5.],
           [ 7.],
           [10.]])
>>> x = cramer(A, b)
>>> x2 = gje(A, b)
>>> x
array([[-1.],
       [ 4.],
       [ 3.]])
>>> x2
array([[-1.],
       [ 4.],
       [ 3.]])
>>> np.dot(A, cramer(A, b))
array([[-5.],
       [ 7.],
       [10.]])
>>> np.dot(A, gje(A, b))
array([[-5.],
       [ 7.],
       [10.]])
```

Testing Your Code

You should save all your code in `cs3430_s22_hw01.py` where I wrote some starter code for you. I also wrote 20 unit tests in `cs3430_s22_hw01.py` that you can use 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.

What to Submit

Save all your code in `cs3430_s22_hw01.py` and submit it via Canvas.

Happy Hacking!