CS 3430: SciComp with Py
Assignment 01
Gauss-Jordan Elimination, Determinants, and Cramer's Rule

Vladimir Kulyukin
Department of Computer Science
Utah State University

January 11, 2020

Learning Objectives

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

Introduction

In this assignment, we'll implement Gauss-Jordan elimination, determinants, and Cramer's rule. This assignment will also give you more exposure to `numpy`. You will save your coding solutions in `cs3430_s20_hw01.py` included in the zip and submit it in Canvas.

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 $\mathbf{A}\mathbf{x} = \mathbf{b}$, 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,

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

and

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

To solve this linear system, we need to find

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

such that

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

Let's define these matrices in Python.

```
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.

```
>>> from cs3430_s20_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()` and `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), `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
```

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.]])
```

When you implement `gje()`, you may use all numpy functions we discussed in Lectures 01 and 02. The `numpy.linalg` package has the function `linsolve` that implements a version of Gauss-Jordan elimination. You may not use this function. You should implement `gje()` from scratch to learn the ins and outs of Gauss-Jordan elimination. Don't be a MATLAB programmer who knows the value of everything and the computation of nothing.

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 boxes, which lies at the very heart of most of integral calculus. Implement two functions `leibniz_det()` and `gauss_det()` to compute the determinant of an $n \times n$ matrix. The former should use the method with the minor matrices and cofactors. The latter – the method that uses Gauss elimination and the product of pivots. 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 the results with `np.linalg.det()`.

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

The determinant of 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_s20_hw01 import *
>>> leibniz_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 `leibniz_det()`, on the one hand, and `np.linalg.det()`, on the other, due to aggressive trunctation in `np.linalg.det()`. In general, the larger the matrices, the larger the discrepancies.

```
>>> from det import *
>>> A1
array([[2., 1., 0., 1.],
       [3., 2., 1., 2.],
       [4., 0., 1., 4.],
       [1., 0., 2., 1.]])
>>> leibniz_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.]])
>>> leibniz_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.]])
>>> leibniz_det(A3)
12.0
>>> gauss_det(A3)
12.0
>>> np.linalg.det(A3)
12.000000000000005
```

The file `cs3430_s20_hw01.py` has the function `random_mat(nr, nc, lower, upper)` that creates an nr $\times$ nc matrix of random numbers in [lower, upper]. Let's create a 10x10 matrix and compute its determinants. The call to `leibniz_det()` will take a while.

```
>>> A = random_mat(10, 10, 1, 3)
>>> gauss_det(A)
-3379.000000000002
>>> np.linalg.det(A)
-3378.9999999999964
>>> leibniz_det(A)
-3379.0
```

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 `leibniz_det()` on these matrices is a hopeless pursuit unless you have an infinite amount of time.

```
>>> 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, 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 to solve linear systems any more, because Gauss-Jordan elimination along with other methods discovered in the past 20 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)`.

```
A = array([[ 2., -1.,  3.],
           [ 3.,  0.,  2.],
           [-2.,  1.,  4.]])
b = 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 = array([[ 0.,  1., -3.],
           [ 2.,  3., -1.],
           [ 4.,  5., -2.]])
b = 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.]])
```

What to Submit

Save all your code in `cs3430_s20_hw01.py` and submit it via Canvas.

Happy Hacking!