CS 3430: SciComp with Py
Assignment 5
Solving Linear Systems

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 10, 2018

1 Learning Objectives

1. Numpy
2. Linear Algebra
3. Matrices
4. Gauss-Jordan Algorithm
5. Solving Linear Systems

2 Introduction

In this assignment, you will implement the Gauss-Jordan algorithm for solving linear systems. This assignment will also give exposure to the `numpy` package. This package is fundamental to many areas of scientific computing in Python, especially for linear algebra and Fourier transform. Another advantage of `numpy` is the availability of tools for integrating C/C++ and Fortran code into Python, which is critical for speed. For those of you who are interested in software licensing issues, `numpy` is lincensed under the BSD license (`http://www.numpy.org/license.html#license`).

3 Matrix Reduction

Recall that we can use the three elementary row operations (row interchange, row scaling, row addition) to reduce $[A|b]$ to $[H|c]$ where H is in row echelon form. In this case, the augmented matrix $[A|b]$ is row equivalent to $[H|c]$ or $[A|b] \sim [H|c]$.

The Gauss-Jordan algorithm starts by reducing a matrix to row echelon form. Recall that a matrix is in row echelon form if 1) all rows containing only 0's appear below the rows containing nonzero entries and 2) the first nonzero entry in any row appears in a column to the right of the first nonzero entry in any preceding row.

There are three logical steps to reducing a matrix:

1. If the first column of A contains only zero entries, cross it off. Continue crossing off columns until the left column of the remaining matrix has a nonzero entry or until the columns are crossed off.

2. If necessary, use row interchange to obtain a nonzero entry (pivot) p in the top row of the first column of the remaining matrix. For each row below that has a nonzero entry r in this column, add $-r/p$ times the top row to create a zero in this column. In other words, create zeros below p in the entire column. If the pivot's row is not the top row of the original matrix, create the 0's in the this column above the pivot's row.

3. Cross off this column and the pivot's row to obtain a smaller matrix and go back to step 1 to repeat this process on the smaller matrix. Keep doing it until no rows or columns remain.

Let's work out an example. Consider this matrix and convert to row echelon form.

$$\begin{bmatrix} 2 & -4 & 2 & -2 \\ 2 & -4 & 3 & -4 \\ 4 & -8 & 3 & -2 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

Step 1: Multiply row 1 by 1/2 to produce a pivot of 1 in column 1 and to obtain the following matrix.

$$\begin{bmatrix} 1 & -2 & 1 & -1 \\ 2 & -4 & 3 & -4 \\ 4 & -8 & 3 & -2 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

Step 2: Add -2 times row 1 to row 2 and then add -4 times row 1 to row 3 to obtain the following matrix.

$$\begin{bmatrix} 1 & -2 & 1 & -1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & -1 & 2 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

Step 3: Add row 2 to rows 3 and 4 to obtain the following matrix.

$$\begin{bmatrix} 1 & -2 & 1 & -1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Step 4: Add -1 row 2 to create 0 on top of the pivot of 1 in column 3 to obtain the final matrix.

$$\begin{bmatrix} 1 & -2 & 0 & 1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Implement the function `reduceToREF(m)` that takes a 2D numpy array and reduces it to row echelon form. A couple of test runs.

```
>>> import numpy as np
>>> m = np.array([
    [2, -4, 2, -2],
    [2, -4, 3, -4],
```

```
    [4, -8, 3, -2],
    [0, 0, -1, 2]
    ],
    dtype=float)
>>> reduceToREF(m)
[[ 2. -4.  2. -2.]
 [ 2. -4.  3. -4.]
 [ 4. -8.  3. -2.]
 [ 0.  0. -1.  2.]]
>>> m
array([[ 1., -2.,  0.,  1.],
       [ 0.,  0.,  1., -2.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> m2 = np.array([
        [1,  6, 3, 4],
        [1,  2, 1, 1],
        [-1, 2, 1, 2],
        ],
        dtype=float)
>>> reduceToREF(m2)
>>> m2
array([[ 1.  ,  0.  ,  0.  , -0.5 ],
       [-0.  ,  1.  ,  0.5 ,  0.75],
       [ 0.  ,  0.  ,  0.  ,  0.  ]])
```

Solving Linear Systems

Implement the function `solveLinSys(m)` that takes an augmented matrix m of the form $[A|b]$ and uses the Gauss-Jordan algorithm to reduce it to row echelon form to solve it. This function should return `False` if the system is inconsistent and `True`. Use this function to solve the following linear system.

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

Let's convert this lineary system into an augmented matrix and then call `solveLinSys(m)` on it.

```
>>> m = np.array([
        [2, -1, 3, 4],
        [3,  0, 2, 5],
        [-2, 1, 4, 6]
        ],
        dtype=float)
>>> solveLinSys(m)
[[ 2. -1.  3.  4.]
 [ 3.  0.  2.  5.]
 [-2.  1.  4.  6.]]
True
```

```
>>> m
array([[ 1.        ,  0.        ,  0.        ,  0.71428571],
       [ 0.        ,  1.        ,  0.        ,  1.71428571],
       [ 0.        ,  0.        ,  1.        ,  1.42857143]])
```

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

We can check the correctness of this solution by plugging them into each of the three equations in the original linear system.

```
>>> 2*0.71428571 + -1*1.71428571 + 3*1.42857143
4.0
>>> 3*0.71428571 + 0*1.71428571 + 2*1.42857143
4.99999999
>>> -2*0.71428571 + 1*1.71428571 + 4*1.42857143
6.000000010000001
```

Of course, this is a bit tedious. We can use matrix algebra to do the trick for us. Let's define the augmented matrix one more time.

```
>>> m = np.array([
        [2, -1, 3, 4],
        [3,  0, 2, 5],
        [-2, 1, 4, 6]
        ],
        dtype=float)
```

Let's get the matrix of coefficients by deleting the 3rd column from `m` and the right hand side coefficients of all equations from the 3rd column of `m`.

```
>>> A = np.delete(m, np.s_[3:4], axis=1)
>>> b = m[:,3]
>>> A
array([[ 2., -1.,  3.],
       [ 3.,  0.,  2.],
       [-2.,  1.,  4.]])
>>> b
array([ 4.,  5.,  6.])
```

Note that the notation `np.s_[3:4]` define a slice of columns and the keyword argument `axis=1` tells `numpy` to delete columns. Accidentally, if you want to delete rows, use `axis=0`. Also, note that `np.delete` does not destroy the original matrix but creates a copy with the appropriate columns or rows removed. Let's verify this.

```
>>> m
array([[ 2., -1.,  3.,  4.],
       [ 3.,  0.,  2.,  5.],
       [-2.,  1.,  4.,  6.]])
>>> A
array([[ 2., -1.,  3.],
       [ 3.,  0.,  2.],
       [-2.,  1.,  4.]])
```

Now we can run `solveLinSys` on `m` and take the last column to obtain the solution vector.

```
>>> solveLinSys(m)
>>> x = m[:,3]
>>> x
array([ 0.71428571,  1.71428571,  1.42857143])
```

Finally, all we need to do is to multiply `A` by `x` to get `b`.

```
>>> np.dot(A, x)
array([ 4.,  5.,  6.])
>>> b
array([ 4.,  5.,  6.])
```

In other words, we just computationally verified that `Ax=b`.

Let's solve this linear system.

$$\begin{aligned} x_1 + 2x_2 - 3x_3 &= -1 \\ 3x_1 - x_2 + 2x_3 &= 7 \\ 5x_1 + 3x_2 - 4x_3 &= 2 \end{aligned}$$

We construct the augmented matrix and apply `solveLinSys` to it.

```
>>> solveLinSys(m)
False
```

So we got `False`, which means that the system is inconsistent. Let's find out why by displaying the reduced `m`.

```
>>> m
array([[ 1.        ,  0.        ,  0.14285714,  0.        ],
       [-0.        ,  1.        , -1.57142857,  0.        ],
       [-0.        , -0.        , -0.        ,  1.        ]])
```

The last row of the reduced matrix is the reason. The equation $-0x_1 + -0x_2 + -0x_3 = 1.0$ has no solution.

Inverting Matrices

One of the ways to compute the inverse of an $n \times n$ matrix is to augment it with the $n \times n$ identity matrix on the right, the reduce the augmented matrix to row echelon form, and return the $n \times n$ matrix on the right if and only if the $n \times n$ matrix on the left is the $n \times n$ identity matrix. In other words, the row reduction must result in the identity matrix moving from the right to the left. If that is the case, the inverse of the original matrix is the $n \times n$ matrix on the right.

Let's consider an example. Suppose we have this matrix.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 7 \end{bmatrix}$$

We augment A with the 2×2 identity matrix on the right and row reduce the augmented matrix to attempt to get the 2×2 identity matrix on the left. If we succeed, the 2×2 matrix on the right is the inverse.

$$\left[\begin{array}{cc|cc} 1 & 2 & 1 & 0 \\ 3 & 7 & 0 & 1 \end{array}\right] \sim \left[\begin{array}{cc|cc} 1 & 0 & 7 & -2 \\ 0 & 1 & -3 & 1 \end{array}\right]$$

Thus, the inverse of A or A^{-1} is

$$A^{-1} = \begin{bmatrix} 7 & -2 \\ -3 & 1 \end{bmatrix}$$

Implement the function `invertMat(m)` that takes a numpy matrix and returns its inverse if `m` is invertible and `None` if it is not. Here are a few test runs.

```
>>> m = np.array([
[1, 2],
[3, 7],
],
dtype=float)
>>> invertMat(m)
array([[ 7., -2.],
       [-3.,  1.]])
>>> np.dot(m, invertMat(m))
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> np.dot(invertMat(m), m)
array([[ 1.,  0.],
       [ 0.,  1.]])
>>> m1 = np.array([
        [1, 2, -1],
        [2, 5, 4],
        [3, 7, 4],
        ],
        dtype=float)
>>> invertMat(m1)
array([[ -8., -15.,  13.],
       [  4.,   7.,  -6.],
       [ -1.,  -1.,   1.]])
>>> np.dot(m1, invertMat(m1))
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> np.dot(invertMat(m1), m1)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> m2 = np.array([
        [1, 2, -3],
        [1, -2, 1],
        [5, -2, -3],
        ],
```

```
        dtype=float)
>>> im2 = invertMat(m2)
>>> im2 == None
True
```

Let me show you a few numpy tricks that you may find handy as you work on `invertMat(m)`. You can use `np.column_stack` to create augmented matrices. Here is how you can augment one matrix with another.

```
>>> import numpy as np
>>> A = np.array([
[1, 2],
[4, 5],
],
dtype=float)
>>> B = np.array([
[10, 20],
[40, 50],
],
dtype=float)
>>> np.column_stack((A, B))
array([[  1.,   2.,  10.,  20.],
       [  4.,   5.,  40.,  50.]])
```

Another trick is using the numpy slices to retrieve slices of columns from a matrix. Let's define a 4×4 matrix.

```
>>> A = np.array([
[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12],
[13, 14, 15, 16]
],
dtype=float)
>>> A
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,   7.,   8.],
       [  9.,  10.,  11.,  12.],
       [ 13.,  14.,  15.,  16.]])
```

We can use the numpy slices, i.e., `np.s_[x:y]`, to get various column slices. For example, here is how we can get the submatrix of `A` that consists of column 0 and column 1.

```
>>> A[:,np.s_[0:2]]
array([[  1.,   2.],
       [  5.,   6.],
       [  9.,  10.],
       [ 13.,  14.]])
```

Here is how we can get the submatrix of `A` that consists of columns 2 and 3.

```
>>> A[:,np.s_[2:4]]
array([[  3.,   4.],
       [  7.,   8.],
       [ 11.,  12.],
       [ 15.,  16.]])
```

What to Submit

Place your code in `cs3430_s18_hw05.py` and submit it via Canvas. Note that you don't have to code this assignment both in Py2 and Py3. If you do this on the pi, do it in Py2, because on the pi the Py2 `numpy` version is already installed and configured. If you don't do it on the pi, you should install the Py2 `numpy` version on your machine. Remember, as I said in class, from assignment 5 on, all your submissions will be graded on the pi to avoid any library/OS compatibility issues. If you have Linux or Mac OS on your laptop/desktop, I don't anticipate any compatibility issues so long as you use Py2 and install the Py2 compatible versions of the third party libraries I announced on Canvas. I cannot make any assurances for Windows. In other words, if you code on Windows and your code doesn't run on the pi or doesn't pass unit tests due to library incompatibilities, Astha will make deductions.

Happy Hacking!