ENGN 2020:

Homework #1

Brown University School of Engineering

Assigned: January 28, 2018, Due: February 4, 2018

Readings: You should make sure you are familiar with the material in Chapter 7 while doing this problem set.

Problem 1

This problem will build basic working skills with arrays in numpy. It's recommended you work through the beginning of this problem in an interactive python shell. (Open a bash terminal and type `ipython3`.)

In python, if you type commands like

```
x = [1, 2, 3]
y = [4, 5, 6]
```

you get *lists* of numbers. What happens if you add `x` and `y`? (Try it: `x + y`.)

The package 'numpy' adds numeric functions to python, and the `array` command lets python behave like a matrix manipulation engine. [1]

```
import numpy as np
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
```

Now try adding `x` and `y` and note the difference in adding arrays versus lists.

The default operation for arrays is element-by-element.[2] Try some of the following commands and see what you get:

```
x + y
x - y
x * y
x / y
np.exp(y)
```

[1] Note you can also create a "matrix" object in numpy, but this is planned to be discontinued in future versions, and the developers recommend recommend that you use arrays.

[2] This is different than in Matlab, where you tend to use symbols like `.*` to mean element-by-element multiplication, for example.

```
np.log(x)
np.log10(x)
x - 3
x * 9
```

You can pull individual values from an array by giving an index or range inside square brackets. (Note that python starts counting at 0, not 1!) Try playing with commands like the following to see what happens:

```
x = np.array([8, 3, 9, 12, 21, 2, 7])
x[3]  # return the 4th value
x[-1]  # return the last value
x[-3]  # return the third-from-last value
x[0:5]  # return the first five values
x[:5]  # return the first five values (the missing 0 is implied)
x[2:-2]  # return the third value through the third to last
x[2:]  # return the third value on
```

A two-dimensional array (that is, a matrix) is made simply by embedding lists; that is,

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
```

Similarly, for a two-dimensional array you can address values by including a comma inside the square brackets. For example, `A[3,1]` would index the element at the 4th row and 2nd column of a matrix, while `A[:,1]` would return the entire 2nd column.

Part a. Write a function that returns the sum of the elements in the 2nd and 3rd columns of an arbitrary matrix A. (In python-speak, this is columns 1 and 2.)

Submission

Label: hw1_1a
Points: 4
Input variables: `A` (numpy array)
Output: a number

Part b. Recall that in matrix multiplication, each element (i, j) of the output matrix is created by taking a dot product of the ith row of the first matrix with the jth row of the second matrix. Create a function that takes in two matrices $\mathbf{A}$ and $\mathbf{B}$ as well as indices i and j and returns the corresponding element of the matrix produced by multiplying A and B.

Submission

Label: hw1_1b
Points: 4
Input variables: `A` (numpy array), `B` (numpy array), `i` (integer), `j` (integer)

Output: a number

Part c. Of course, you don't need to hard-code your own matrix multiplication algorithm, you can use the built in command `matmul` to multiply two matrices.[3] *E.g.*,

```
A = np.random.random((5, 3))  # Creates a 5x3 matrix of random numbers.
B = np.random.random((3, 4))  # Creates a 3x4 matrix of random numbers.
C = np.matmul(A, B)
```

Recall that matrix multiplication does not commute—that is, the order matters. Write a function that takes in four matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ and multiplies them in the order $\mathbf{ABCD}$.

Submission

Label: hw1_1c
Points: 2
Input variables: `A` (numpy array), `B` (numpy array), `C` (numpy array), `D` (numpy array)
Output: a numpy array

Problem 2

In this problem, we will work with row operations, focusing on the Guass–Jordan Elimination algorithm to find the inverse of a matrix. We will start by following Example 1 of Section 7.8, and perform the row-operations in python.

Start by creating the matrices $\mathbf{A}$ and $\mathbf{I}$ as separate numpy arrays in python. Hint: you can quickly create a 3×3 identity matrix $\mathbf{I}$ in numpy by typing `np.eye(3)`; you will need to type in $\mathbf{A}$ by hand. You can then merge them with the command `np.hstack` (horizontally stack) to make an augmented matrix we'll call $\mathbf{W}$.

At this stage, you should see

```
>>> W = np.hstack([A, I])
>>> W
array([[-1.,  1.,  2.,  1.,  0.,  0.],
       [ 3., -1.,  1.,  0.,  1.,  0.],
       [-1.,  3.,  4.,  0.,  0.,  1.]])
```

We'll refer to the left half of $\mathbf{W}$ as $\mathbf{A}'$ and the right half as $\mathbf{I}'$, to distinguish them from our original matrices.

We can now perform elementary row operations, as defined in Section 7.3, to put $\mathbf{A}'$ in upper-triangular form in which all the elements of $\mathbf{A}'$ below the diagonal are 0. *I.e.*, we start by replacing the middle row with a linear combination of the top and middle rows, chosen in such a manner that the first element is zero:

[3] In newer versions of python ($\geq$3.5) there is now a symbolic way to do this: `A @ B`, which is equivalent to `np.matmul(A, B)`.

```
>>> W[1,:] = W[1,:] + 3 * W[0, :]
>>> W
array([[-1.,  1.,  2.,  1.,  0.,  0.],
       [ 0.,  2.,  7.,  3.,  1.,  0.],
       [-1.,  3.,  4.,  0.,  0.,  1.]])
```

You should continue in this fashion until all the values below the $\mathbf{A}'$ diagonal are 0. Then, carry on until you have replaced $\mathbf{A}'$ with the identity matrix; at this stage the $\mathbf{I}'$ region is our inverse of $\mathbf{A}$; that is, $\mathbf{A}^{-1}$.

Part a. Write a function that takes in a square matrix $\mathbf{A}$, creates $\mathbf{W}$ as above, and uses elementary row operations to make the first entry of each row in $\mathbf{W}$ (except the top row) equal to zero, by subtracting a multiple of the top row. Your function should return the new $\mathbf{W}$. Note that $\mathbf{A}$ will not necessarily be the same size as the example; you can find the shape of $\mathbf{A}$ with `A.shape`. Hint: You should look how `for` loops and the `range` command work in python if you need help getting started.

Submission

Label: hw1_2a
Points: 3
Input variables: `A` (numpy array)
Output: a numpy array

Part b. Now, write a similar script that takes in a square matrix $\mathbf{A}$, creates $\mathbf{W}$, and makes all values below the diagonal zero, resulting in an upper-triangular form. To ensure that your answer matches the solutions, first, clear the left-most column by subtracting the top row (as in part a), then use the next row to clear the second column, etc.

Submission

Label: hw1_2b
Points: 3
Input variables: `A` (numpy array)
Output: a numpy array

Part c. Of course, you don't need to hard-code your own matrix inversion routine. You can use `np.linalg.inv` for this purpose.

Consider the system of equations:

$$\begin{aligned} x_1 - x_2 + x_3 &= 0 \\ 10x_2 + 25x_3 &= 90 \\ 20x_1 + 10x_2 &= 80 \end{aligned}$$

First, write these in the form $\mathbf{A}\ \mathbf{x} = \mathbf{b}$; that is, form $\mathbf{A}$ and $\mathbf{b}$. Hint: you can make $\mathbf{b}$ a column vector in either of two ways:

```
b = np.array([[0], [90], [80]])  # method 1
b = np.array([[0, 90, 80]]).T  # method 2
```

As discussed in class, this can be solved for $\mathbf{x}$ by premultiplying with the inverse of $\mathbf{A}$; that is,

$$\begin{aligned}
\mathbf{A}\mathbf{x} &= \mathbf{b} \\
\mathbf{A}^{-1}\mathbf{A}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\
\mathbf{I}\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\
\mathbf{x} &= \mathbf{A}^{-1}\mathbf{b}
\end{aligned}$$

Write a function that takes in $\mathbf{A}$ and $\mathbf{b}$ and returns $\mathbf{x}$ using this method. When you run it on this problem, you should find $\mathbf{x}^{\mathrm{T}} = [x_1 \; x_2 \; x_3] = [2 \; 4 \; 2]$. Submit your function.

Submission

Label: hw1_2c
Points: 4
Input variables: `A`, `b` (numpy arrays)
Output: a numpy array