COMS 3251 Spring 2021: Lab 1

YOUR NAME(s): Vicente Farias

YOUR UNI(s): vf2272

In [1]:
```
# RUN THIS FIRST
# Environment check, one, two...
import numpy as np      # imports the numpy package
```

Introduction to NumPy

Since this is a linear algebra course, we'll be dealing a lot with 1-dimensional vectors and 2-dimensional matrices. The obvious way to represent these in code are 1-dim and 2-dim arrays.

Let's look at representing the following vectors and matrices with simple Python lists of lists, where each list element of the "outer" list is a distinct row of the overall matrix:

$$\mathbf{u} = \begin{bmatrix} 1 & 0 & 4 & 6 \end{bmatrix}$$

$$\mathbf{v} = \mathbf{u}^T = \begin{bmatrix} 1 \\ 0 \\ 4 \\ 6 \end{bmatrix}$$

$$A = \begin{bmatrix} 4 & 6 & 2 & 0 \\ 3 & 7 & 2 & 9 \\ 0 & 1 & 4 & 2 \\ 9 & 3 & 5 & 6 \end{bmatrix}$$

In [2]:
```
u = [[1, 0, 4, 6]]
v = [[1],
     [0],
     [4],
     [6]]
A = [[4, 6, 2, 0],
     [3, 7, 2, 9],
     [0, 1, 4, 2],
     [9, 3, 5, 6]]
```

Cool! Now, calculate $A^T A$.

Just kidding. While you can do it, implementing all of the various matrix operations (transpose, multiplication, etc.) using Python lists can be pretty painful. Linear algebra is a major part of computer science, so there should be a package that can do this for us! Introducing... Numpy.

Array creation

Numpy turns lists into Numpy arrays, on which we can do various linear algebra operations. Even better, these data structures are optimized for speed and efficiency. Let's convert $\mathbf{u}$, $\mathbf{v}$, and A into Numpy arrays first.

In [6]:
```
# Example 1: Creating Numpy arrays from lists
print(A)

u_arr = np.array(u)
v_arr = np.array(v)
A_arr = np.array(A)

print(A_arr)
```

```
[[4, 6, 2, 0], [3, 7, 2, 9], [0, 1, 4, 2], [9, 3, 5, 6]]
[[4 6 2 0]
 [3 7 2 9]
 [0 1 4 2]
 [9 3 5 6]]
```

And that it's! Notice how Numpy arrays print as matrices by default. We can also create special arrays using the following functions. Feel free to print any of them to verify the results.

In [9]:
```
# Example 2: Creating special matrices
m = 4
n = 3

z = np.zeros((m,n))                       # Zeros matrix of size m by n
o = np.ones((m,n))                        # Ones matrix of size m by n
c = np.full((m,n), 3)                     # Constant matrix of size m by n
id = np.eye(n)                            # Identity matrix of size n

r = np.random.random((m,n))               # Random matrix of size m by n
ri = np.random.randint(0, 10, size=(m,n)) # Random integer matrix of size m by n
```

Array indexing

You already know how to access and modify specific elements of Python lists. Indexing Numpy arrays works much the same, except that it may be even simpler. Take a look at the following:

In [10]:
```
# Example: Array indexing

# Accessing a single element
print(A[1][3])          # for a 2D list, we access it normally
print(A_arr[1,3])       # for a 2D numpy array, we have this shortened form.

# Accessing a row
print(A[1])             # these are the same.
print(A_arr[1])
```

```
9
9
[3, 7, 2, 9]
[3 7 2 9]
```

With a list of lists, you need two separate indices; the first is for the "outer" list, and the second is for the "inner" list. With a Numpy array, indices all appear in one set of brackets, separated by a comma. The first indexes the row, and the second indexes the column.

You can also take array slices using the `:` operator. Take a look at the following examples.

In [11]:
```
# Example: Array slicing

# Accessing a column
_col = []               # accessing a column is hard for a 2D list.
for arow in A:          # we have to iterate through each row and get each
  _col.append(arow[2])  # value.
print(_col)

print(A_arr[:, 2])      # but this is very easy for a numpy array.
                        # ":" indicates we want values from all rows, while
                        # "2" specifies which column.
                        # This is familiar if you know MATLAB.
```

```
[2, 2, 4, 5]
[2 2 4 5]
```

In [12]:
```
# Advanced slicing
# The colon operator is actually very powerful. In general, you can call
# A[begin:end:interval] where indices are [begin, end)

print(A_arr)
print(A_arr[0:2])       # Just the rows from 0 to 2 (but not including 2!)

print(A_arr[:2,:2])     # Just the top left corner of matrix A_arr
                        # If the begin index is dropped, 0 is assumed

print(A_arr[::2,1:])    # Every second row, from begin to end;
                        # All columns from second to end
                        # If the end index is dropped, last is assumed

print(A_arr[-1,-1])     # -1 refers to the last index of either row or column
```

```
[[4 6 2 0]
 [3 7 2 9]
 [0 1 4 2]
 [9 3 5 6]]
[[4 6 2 0]
 [3 7 2 9]]
[[4 6]
 [3 7]]
[[6 2 0]
 [1 4 2]]
6
```

Array operations

Anything that you can do with matrices in linear algebra, you can do in Numpy. Try out the following operations!

In [14]:
```
# Example: Basic array operations

B = np.random.randint(0, 10, (4, 4))
print(B)
C = np.random.randint(0, 10, (4, 4))
print(C)

print(B.shape)  # matrix dimensions
print(B + C)    # element-wise addition
print(B - C)    # element-wise subtraction
print(B * C)    # element-wise multiplication
print(B / C)    # element-wise division
print(B + 2)    # scalar addition (subtraction, multiplication, etc.)
print(B.T)      # matrix transpose
```

```
[[6 0 1 7]
 [0 3 2 4]
 [4 7 4 1]
 [1 0 2 6]]
[[4 7 0 1]
 [9 6 1 6]
 [7 9 0 6]
 [4 5 9 0]]
(4, 4)
[[10  7  1  8]
 [ 9  9  3 10]
 [11 16  4  7]
 [ 5  5 11  6]]
[[ 2 -7  1  6]
 [-9 -3  1 -2]
 [-3 -2  4 -5]
 [-3 -5 -7  6]]
[[24  0  0  7]
 [ 0 18  2 24]
 [28 63  0  6]
 [ 4  0 18  0]]
[[1.5        0.                inf 7.        ]
 [0.         0.5       2.         0.66666667]
 [0.57142857 0.77777778        inf 0.16666667]
 [0.25       0.         0.22222222        inf]]
[[8 2 3 9]
 [2 5 4 6]
 [6 9 6 3]
 [3 2 4 8]]
[[6 0 4 1]
 [0 3 7 0]
 [1 2 4 2]
 [7 4 1 6]]
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:12: RuntimeWarning: divide by zero encountered in true_divide
  if sys.path[0] == '':
```

Two operations of interest are vector-vector dot products, and matrix-matrix and matrix-vector multiplication. In Numpy they are all implemented by the same function, depending on the arguments provided. Carefully study the following examples.

In [15]:
```
# Example: Array multiplication
# As a shortcut, you can also use x@y in place of x.dot(y)

print(B.dot(C))         # regular matrix multiplication

D = np.array([1,2,3,4]) # matrix-vector multiplication
print(B.dot(D))         # matrix-vector multiplication, returns a 1d array

Dm = np.array([[1],
               [2],
               [3],
               [4]])
print(B.dot(Dm))        # this is actually matrix-matrix multiplication
                        # returns a 2d array of a column vector
E = np.array([5,6,7,8])
print(D.dot(E))         # dot product of two 1d arrays

# print(Dm.dot(E))      # error!! size mismatch since Dm is a 2d array, not vector
```

```
[[ 59  86  63  12]
 [ 57  56  39  30]
 [111 111  16  70]
 [ 42  55  54  13]]
[37 28 34 31]
[[37]
 [28]
 [34]
 [31]]
70
```

Numpy functions

Numpy contains most of the standard math functions that you might encounter, many of them operating on arrays elementwise. Examples include functions like `np.sqrt`, `np.sin`, `np.exp`, `np.log`, `np.absolute`, and many others. Complete listings of useful functions can be found in the following links:

Math: https://numpy.org/doc/stable/reference/routines.math.html

Statistics: https://numpy.org/doc/stable/reference/routines.statistics.html

You may also sometimes find the need to use aggregating functions that look at all elements within a certain column, row, or the entire array. See examples below.

In [27]:
```
# Example: Sum, max, and min

print(A_arr)                  # remind ourselves what this looks like...
print(np.sum(A_arr))          # sum of all array elements
print(np.sum(A_arr, axis=0))  # sum all rows together: row1 + row2 + ...
print(np.sum(A_arr, axis=1))  # sum all cols together: col1 + col2 + ...

print(np.max(A_arr))          # same as above, but with max and min functions
print(np.min(A_arr))
print(np.max(A_arr, axis=1))
print(np.min(A_arr, axis=0))
```

```
[[4 6 2 0]
 [3 7 2 9]
 [0 1 4 2]
 [9 3 5 6]]
63
[16 17 13 17]
[12 21  7 23]
9
0
[6 9 4 9]
[0 1 2 0]
```

Appendix: More resources

A useful Google search pattern is `numpy (insert linear algebra function here)`

Complete NumPy documentation: https://numpy.org/doc/stable/

NumPy Quickstart Tutorial: https://numpy.org/doc/stable/user/quickstart.html

Stanford CS231n Python/NumPy Tutorial: http://cs231n.github.io/python-numpy-tutorial/

Exercises

The following exercises will help you practice some of the Numpy functionalities that we covered. Remember that you should complete these with your lab partner. Your TAs will provide more specific instructions where indicated.

PROBLEM 1: Slicing Practice (10 pts)

Given a matrix `input`, use indexing and slicing to return a Numpy array matching the specifications provided by your TAs.

In [31]:
```
def PROBLEM1(input):
  # YOUR CODE GOES HERE

  output = input[1::2, 1:8]

  return output

# DO NOT MODIFY
np.random.seed(3251)
example = np.random.randint(0, 10, (10,10))
print(example)
print(PROBLEM1(example))
```

```
[[5 3 0 3 6 9 3 2 0 6]
 [0 6 8 3 5 1 8 2 9 8]
 [5 3 3 9 0 4 8 6 3 9]
 [6 0 5 2 1 5 2 4 9 1]
 [6 7 3 6 1 0 1 4 4 9]
 [5 8 0 1 7 3 1 1 8 7]
 [0 8 7 2 5 3 8 9 6 5]
 [6 5 2 0 0 1 7 7 3 0]
 [8 0 0 3 6 5 3 8 0 8]
 [1 2 6 2 7 3 7 3 2 7]]
[[6 8 3 5 1 8 2]
 [0 5 2 1 5 2 4]
 [8 0 1 7 3 1 1]
 [5 2 0 0 1 7 7]
 [2 6 2 7 3 7 3]]
```

PROBLEM 2: Matrix Operations (20 pts)

Five matrices and vectors have been defined below. Implement the computation specified by your TAs and return the result. (You do not need to know how to do matrix operations for this part. We will only be checking that you are using the Numpy functionalities correctly.)

In [37]:
```
A = np.array([[-1, 0, 2],
              [0, 1, 4]])
B = np.array([-2, 1])
C = np.array([[3, 1],
              [0, 0],
              [-2, -1]])
D = np.array([[5, -3]])
E = np.array([[-4, 2]])

def PROBLEM2(A, B, C, D, E):
  # YOUR CODE GOES HERE

  result = (D + E) @ A @ C

  return result

# DO NOT MODIFY
print(PROBLEM2(A, B, C, D, E))
```

```
[[1 1]]
```

PROBLEM 3: Linear Equations (20 pts)

Implement the backward substitution algorithm for triangular systems of linear equations, assuming the inputs are Numpy arrays. Your implementation should return the string `"No solution!"` if no solution exists (an error should not be thrown). You should only be using Numpy operations; we will not accept implementations that convert the inputs into Python lists and then copy the code from class.

In [26]:
```
# A is a 2d numpy array of equation coefficients
# b is a 1d numpy array of scalars on the RHS of the equations
def PROBLEM3(A, b):
```

```
  # YOUR CODE GOES HERE
  x = np.zeros(b.shape)
  for i in reversed(range(len(b))):
    if A[i, i] == 0:
      return 'No solution!'

    x[i] = (b[i] - x @ A[i, :]) / A[i, i]

  return x

# DO NOT MODIFY
A = np.array([[1,0.5,-2,4],
              [0,3,3,2],
              [0,0,1,5],
              [0,0,0,2]])
b = np.array([-8,3,-4,6])
print(PROBLEM3(A, b))
```

```
[-67.  18. -19.   3.]
```

Submission

When you have completed all three exercises, submit your completed notebook to your corresponding lab section assignment on Gradescope. Only one submission per lab group is necessary. Remember that late submissions are not accepted.