Lab1

January 20, 2022

1 Lab 1. Linear Algebra

1.0.1 Due date: Friday 01/14 at 10:59 pm

Welcome to lab in DSC 155/Math 182. In this first lab, you will get acquainted with the computing environment of the course and you will start to use Python and some of its libraries. You will learn:

- Some basic use of python
- How to generate vectors and matrix in Python with numpy
- How to do basic operations with vectors and matrix in Python with numpy
- How to find eigenvectors and eigenvalues in Python with the module linalg of numpy
- Singular value decomposition in Python with the module linalg

1.1 Part 1. Python Basics

References:

- https://www.w3schools.com/python/
- https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1214/readings/cs224n-python-review-code-updated.pdf

1.1.1 1.1 Variables and operations

[1]:
```
# variables don't need explicit declaration
var = 10       # int
print(var)

var = 10.0     # float
print(var)

var = [1,2,3] # pointer to list
print(var)

a = "Hello" # string
b = "Welcome to DSC 155"
print( a + ", " + b + "/Math 182")

var = True     # boolean
print(var)
```

```
var = None     # empty pointer
print(var)
```

```
10
10.0
[1, 2, 3]
Hello, Welcome to DSC 155/Math 182
True
None
```

[2]:
```
# type conversions
var = 10
print(int(var))
print(str(var))
print(float(var))
```

```
10
10
10.0
```

[3]:
```
# basic math operations
a = 10
b = 3
print("a + b =", a + b)
print("a - b =", a - b)
print("a * b =", a * b)
print("a ^ b =", a ** b)
print("int(a) / b =", a//b)    # // for int division
print("float(a) / b =", a/b)   # / for float division

# All compound assignment operators available
# including += -= *= **= /= //=
# pre/post in/decrementers not available (++ --)

# Some basic function as abs or round are built into the Python language and␣
↪they are available by default.
# The name of the function appears first, followed by expressions in␣
↪parentheses.
print("absolute value of -3 is ", abs(-3))
print("round 3.7 = ", round(3.7))
print("round 2.143 = ", round(2.143))
```

```
a + b = 13
a - b = 7
a * b = 30
a ^ b = 1000
int(a) / b = 3
```

```
float(a) / b = 3.3333333333333335
absolute value of -3 is  3
round 3.7 =  4
round 2.143 =  2
```

[4]:
```
"""
TODO: 1. compute simple numerical expression (2 + 3)/4 + 2^3 - 7 using float␣
↪division.
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
(2 + 3)/4 + 2**3 - 7
### END SOLUTION
```

[4]:
```
2.25
```

Most functions built into the Python language are stored in a collection of functions called a library. An import statement is used to provide access to a library, such as math and operator.

Note that anytime you call a function that belongs to a particular module (math), you need to import the library and to call it before the name of the function, followed by dot as math.log(var)

If you need a particular function and you don't know the name or how to use it, look for it on the internet. You will find manual references and examples.

[5]:
```
import math
import operator

print("square root of 4+5", math.sqrt(operator.add(4, 5)))
print("square root of 4+5", math.sqrt(4 + 5))

print("log_2(16) = ", math.log(16, 2)) # base 2
print("log(16) = ", math.log(16))     # base e
print("e^2 = ", math.exp(2))
print("exp(log(16)) = ", math.exp(math.log(16)))
```

```
square root of 4+5 3.0
square root of 4+5 3.0
log_2(16) =  4.0
log(16) =  2.772588722239781
e^2 =  7.38905609893065
exp(log(16)) =  15.999999999999998
```

1.1.2 1.2 if-else and loops

[6]:
```python
# if-else
var = 3
if var > 5:
    print(">")
elif var == 5:
    print("=")
else:
    print("<")
```

```
<
```

[7]:
```python
# use "if" to check null pointer or empty arrays
var = None
if var:
    print(var)
var = []
if var:
    print(var)
var = "string"
if var:
    print(var)
```

```
string
```

[8]:
```python
# while-loop
var = 7
while var > 0:
    print(var)
    var -=1
```

```
7
6
5
4
3
2
1
```

[9]:
```python
# for-loop
for i in range(5):  # prints 0 1 2 3 4
    print(i)
print("-------")

# range (start-inclusive, stop-exclusive, step)
for i in range(1, 10, 2):
    print(i)
```

```
print("-------")

for i in range(5, -5, -2):
    print(i)
```

```
0
1
2
3
4
-------
1
3
5
7
9
-------
5
3
1
-1
-3
```

[10]:
```
# control flows
# NOTE: No parentheses or curly braces
#       Indentation is used to identify code blocks
#       So never ever mix spaces with tabs
for i in range(0,3):
    for j in range(i, 3):
        print("inner loop", j)
    print("outer loop", i)
```

```
inner loop 0
inner loop 1
inner loop 2
outer loop 0
inner loop 1
inner loop 2
outer loop 1
inner loop 2
outer loop 2
```

1.1.3 1.3 function and class

In Python, you can also create your own function, using the statement 'def'. In the following, we create a function that squares a number and adds 1.

[11]:
```python
def f(x):
    return x**2+1

for i in range(3):
    print(f(i))
```

```
1
2
5
```

[12]:
```python
# use default parameters and pass values by parameter name
def rangeCheck(a, min_val = 0, max_val=10):
    return min_val < a < max_val    # syntactic sugar

print("Is 3 in (0, 10):", rangeCheck(3))
print("Is 3 in (4, 10):", rangeCheck(3, min_val = 4))
print("Is 3 in (0, 2):", rangeCheck(3, max_val = 2))
print("Is 3 in (4, 7):", rangeCheck(3, min_val = 4, max_val = 7))
```

```
Is 3 in (0, 10): True
Is 3 in (4, 10): False
Is 3 in (0, 2): False
Is 3 in (4, 7): False
```

[13]:
```python
# define class
class Car:

    # optinal constructor
    def __init__(self, color, weight):
        # first parameter "self" for instance reference.
        self.color = color
        self.weight = weight

    # instance method
    def printInfo(self): # instance reference is required for all function␣
 ↪parameters
        print("Color:", self.color)
        print("Weight:", self.weight)


car1 = Car("white", 2000)
car1.printInfo()
```

```
Color: white
Weight: 2000
```

1.2 Part 2. Numpy

Numpy is a very powerful python tool for handling matrices and higher dimensional arrays. In the following, we will use np as a shorthand for numpy.

[14]:
```
import numpy as np
```

[15]:
```
# create matrices and vectors.
M = np.array(([1,2,3],
            [1,4,6],
            [7,8,9]))

print(M)
print(M.shape) # visualise the dimension of the array
print("-------")

v = np.array([1, 2, 3])
print(v)
print(v.shape)
print("-------")

v = np.array(([1],
            [2],
            [3]))
print(v)
print(v.shape)
```

```
[[1 2 3]
 [1 4 6]
 [7 8 9]]
(3, 3)
-------
[1 2 3]
(3,)
-------
[[1]
 [2]
 [3]]
(3, 1)
```

Numpy provides also many convenient functions for creating matrices and vectors, such as

[16]:
```
print(np.zeros((3,4))) # all zero matrix
print("--------------")
print(np.ones((4,3))) # all one matrix
print("--------------")
print(np.eye(3)) # eye denotes the identity matrix
print("--------------")
print(np.full((3,3), 4)) # This will have the same output as 4*np.ones((3,3))
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
--------------
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
--------------
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
--------------
[[4 4 4]
 [4 4 4]
 [4 4 4]]
```

You can use the plus and the minus symbol to perform addition and subtraction between arrays. At the same time, you can also create a matrix from different vectors using the function vstack, hstack or concatenate.

[17]:
```python
v1 = np.array([1,2,3])
v2 = np.array([4,5,6])
v3 = np.array([7,8,9])
print(v1 + v2)
print(v1 - v3)
print("--------------")

print(np.vstack([v1,v2,v3]))
print("--------------")
print(np.hstack([v1,v2,v3]))
```

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

[18]:
```python
"""
TODO: 2. create a 3*3 zero matrix and put elements of vector v1 + v2 - v3 on␣
 ↪diagonal
HINT: np.diag
"""
# Delete raise NotImplementedError()
### BEGIN SOLUTION
```

```
np.diag(v1 + v2 - v3)
### END SOLUTION
```

[18]:
```
array([[-2,  0,  0],
       [ 0, -1,  0],
       [ 0,  0,  0]])
```

[19]:
```
# concatenating arrays
a = np.ones((4,3))
b = np.ones((4,3))
c = np.concatenate([a,b], 0)
print(c)
print(c.shape)
print("--------------")
d = np.concatenate([a,b], 1)
print(d)
print(d.shape)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
(8, 3)
--------------
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
(4, 6)
```

[20]:
```
# access array slices by index
M = np.zeros([10, 10])
M[:3] = 1
M[:, :3] = 2
M[:3, :3] = 3
rows = [4,6,7]
cols = [9,3,5]
M[rows, cols] = 4
print(M)
print("--------------")
print(M[0,2])
print("--------------")
print(M[np.array((0,2)),:])
```

```
[[3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 4.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 4. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 4. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]
 [2. 2. 2. 0. 0. 0. 0. 0. 0. 0.]]
--------------
3.0
--------------
[[3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]
 [3. 3. 3. 1. 1. 1. 1. 1. 1. 1.]]
```

[21]:
```
# reshaping arrays
a = np.arange(8)          # [8,] similar range() you use in for-loops
b = a.reshape((4,2))      # shape [4,2]
c = a.reshape((2,2,-1))   # shape [2,2,2] -- -1 for auto-fill
d = c.flatten()           # shape [8,]
e = np.expand_dims(a, 0) # [1,8]
f = np.expand_dims(a, 1) # [8,1]
g = e.squeeze()           # shape[8, ]    -- remove all unnecessary dimensions
print(a)
print(b)
print(c)
print(d)
print(e)
print(f)
print(g)
```

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

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

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

[22]:

```python
# transposition
M = np.array(([1,2,3],
              [1,4,6],
              [7,8,9]))
print(M.T)
print(M.shape)
print("--------------")

M = np.arange(24).reshape(2,3,4)
print(M)
print(M.shape)
print("--------------")

M = np.transpose(M, (2,1,0)) # swap 0th and 2nd axes
print(M)
print(M.shape)
```

```
[[1 1 7]
 [2 4 8]
 [3 6 9]]
(3, 3)
--------------
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
(2, 3, 4)
--------------
[[[ 0 12]
  [ 4 16]
  [ 8 20]]

 [[ 1 13]
  [ 5 17]
  [ 9 21]]

 [[ 2 14]
  [ 6 18]
  [10 22]]
```

```
 [[ 3 15]
  [ 7 19]
  [11 23]]]
(4, 3, 2)
```

You can use np.dot(array1,array2) to do the dot product between array1 and array2. Note that if array1 is a matrix and array2 is a vector, this is equivalent to do the usual matrix-vector multiplication. If array1 and array2 are matrices, this is equivalent to do the usual matrix-matrix multiplication.

Be careful with the dimensions of the two arrays. As you know, the second dimension of array1 needs to be the same as the first dimension of array2.

[23]:
```python
# dot product
a = np.array([1,2])
b = np.array([3,4])
print(np.dot(a,b)) # Compute a*b
print("--------------")

M = np.array(([1,2,3],
            [1,4,6],
            [7,8,9]))
v = np.array(([1],
            [2],
            [3]))
print(np.dot(M,v)) # Compute M*v
print("--------------")
print(np.dot(M.T,M)) # Compute M^T * M
print("--------------")
print(np.dot(v.T,v)) # Compute v^T * v
print("This is equivalent to || v ||^2")
print(np.linalg.norm(v)**2) # l2 norm by default
```

```
11
--------------
[[14]
 [27]
 [50]]
--------------
[[ 51  62  72]
 [ 62  84 102]
 [ 72 102 126]]
--------------
[[14]]
This is equivalent to || v ||^2
14.0
```

[24]:
```
print(M)
print(np.linalg.norm(M))
print("--------------")
# summing a matrix
print(np.sum(M))
print("--------------")
# the optional axis parameter
print(np.sum(M, axis=0)) # sum along axis 0
print(np.sum(M, axis=1)) # sum along axis 1
```

```
[[1 2 3]
 [1 4 6]
 [7 8 9]]
16.15549442140351
--------------
41
--------------
[ 9 14 18]
[ 6 11 24]
```

[25]:
```
# matrix multiplication
a = np.ones((5,4)) # 5,4
b = np.ones((4,1)) # 4,1 --> 5,1
print(a @ b)       # same as a.dot(b)
print(a.dot(b))
print(np.dot(a,b))
print("--------------")

# automatic repetition along axis
c = a @ b
d = np.array([1,2,3,4,5]).reshape(5,1)
print(c + d)

# handy for batch operation
batch = np.ones((3,32))
weight = np.ones((32,10))
bias = np.ones((1,10))
print((batch @ weight + bias).shape)
```

```
[[4.]
 [4.]
 [4.]
 [4.]
 [4.]]
[[4.]
 [4.]
 [4.]
 [4.]
```

```
 [4.]]
[[4.]
 [4.]
 [4.]
 [4.]
 [4.]]
--------------
[[5.]
 [6.]
 [7.]
 [8.]
 [9.]]
(3, 10)
```

[26]:

```python
# element-wise operations, for examples
M = np.ones((5,5))

print(np.log(M))
print(np.exp(M))
print(np.sin(M))
# operation with scalar is interpreted as element-wise
print(M * 3)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
[[2.71828183 2.71828183 2.71828183 2.71828183 2.71828183]
 [2.71828183 2.71828183 2.71828183 2.71828183 2.71828183]
 [2.71828183 2.71828183 2.71828183 2.71828183 2.71828183]
 [2.71828183 2.71828183 2.71828183 2.71828183 2.71828183]
 [2.71828183 2.71828183 2.71828183 2.71828183 2.71828183]]
[[0.84147098 0.84147098 0.84147098 0.84147098 0.84147098]
 [0.84147098 0.84147098 0.84147098 0.84147098 0.84147098]
 [0.84147098 0.84147098 0.84147098 0.84147098 0.84147098]
 [0.84147098 0.84147098 0.84147098 0.84147098 0.84147098]
 [0.84147098 0.84147098 0.84147098 0.84147098 0.84147098]]
[[3. 3. 3. 3. 3.]
 [3. 3. 3. 3. 3.]
 [3. 3. 3. 3. 3.]
 [3. 3. 3. 3. 3.]
 [3. 3. 3. 3. 3.]]
```

You can use np.multiply(array1,array2) to do elementwise multiplications.

[27]:

```python
M = np.array(([1,2,3],
            [1,4,6],
```

```
            [7,8,9]))
v = np.array(([1],
            [2],
            [3]))

print(np.multiply(M,v)) # the same of using the multiplication symbol in Python.
 ↪
print(M*v)
```

```
[[ 1  2  3]
 [ 2  8 12]
 [21 24 27]]
[[ 1  2  3]
 [ 2  8 12]
 [21 24 27]]
```

You can also use Python to find the the inverse of a matrix, if it exists. For this, you need the module linalg.inv, or pseudo inversion for stability

[28]:
```
print(np.linalg.inv(M))
# pinv is pseudo inversion for stability
print(np.linalg.pinv(M))
```

```
[[ 2.         -1.          0.        ]
 [-5.5         2.          0.5       ]
 [ 3.33333333 -1.         -0.33333333]]
[[ 2.00000000e+00 -1.00000000e+00 -5.21827944e-17]
 [-5.50000000e+00  2.00000000e+00  5.00000000e-01]
 [ 3.33333333e+00 -1.00000000e+00 -3.33333333e-01]]
```

1.3 Part 3: Eigenvalues and Eigenvectors

Reference

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html

[29]:
```
a = 10
b = 3
n = 4
M = b*np.ones([n, n])
M[:int(n/2), :int(n/2)] = a
M[int(n/2):n, int(n/2):n] = a
print(M)
```

```
[[10. 10.  3.  3.]
 [10. 10.  3.  3.]
 [ 3.  3. 10. 10.]
 [ 3.  3. 10. 10.]]
```

[30]:
```
eigvals, eigvecs = np.linalg.eigh(M)
# Eigenvalues are not necessarily ordered, nor are they necessarily real for␣
↪real matrices.
# When computing eigenvalues and eigenvectors, it is better to use np.linalg.
↪eigh rather than np.linalg.eig

print(eigvals) # The eigenvalues are actually [26, 14, 0, 0], up to some small␣
↪computational errors.
```

```
[-9.83743471e-16  9.55650516e-17  1.40000000e+01  2.60000000e+01]
```

[31]:
```
print(eigvecs) # Eigenvectors (each column is the eigenvector corresponding to␣
↪eigenvalue above.)
```

```
[[-0.21040776 -0.67507672  0.5        -0.5       ]
 [ 0.21040776  0.67507672  0.5        -0.5       ]
 [-0.67507672  0.21040776 -0.5        -0.5       ]
 [ 0.67507672 -0.21040776 -0.5        -0.5       ]]
```

[32]:
```
"""
TODO: 3. calculate dot(M, third_eigenvector).
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
np.dot(M, eigvecs[:,2])
### END SOLUTION
```

[32]:
```
array([ 7.,  7., -7., -7.])
```

[33]:
```
"""
TODO: 4. calculate third_eigenvalue * third_eigenvector.
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
eigvals[2]*eigvecs[:,2]
### END SOLUTION
```

[33]:
```
array([ 7.,  7., -7., -7.])
```

The eigenvectors returned by the function eig are the normalized ones. We can check this using the function norm of the module linalg.

[34]:
```
"""
TODO: 5. check the l2 norm of third eigenvector.
"""
```

```
# Delete raise NotImplementedError()
### BEGIN SOLUTION
np.linalg.norm(eigvecs[:,2])
### END SOLUTION
```

[34]: 1.0000000000000004

1.4 Part 4: Singular Value Decomposition

Recall that the singular value decomposition of an $m \times n$ complex matrix M is a factorization of the form USV^*, where U is an $m \times m$ complex unitary matrix, S is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and V is an $n \times n$ complex unitary matrix. If M is real, U and V can also be guaranteed to be real orthogonal matrices. In such contexts, the SVD is often denoted USV^T.

We can also use the module linalg to find the singular value decomposition of a matrix with the function svd.

Reference:

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html

[35]:
```
M = 4*np.ones((5, 3))
M[:2,] = 3
M[:,1] = 7
M[2:4,1:3] = 5
print(M)

U, S, Vh = np.linalg.svd(M)
print(U.shape, S.shape, Vh.shape)
```

```
[[3. 7. 3.]
 [3. 7. 3.]
 [4. 5. 5.]
 [4. 5. 5.]
 [4. 7. 4.]]
(5, 5) (3,) (3, 3)
```

[36]:
```
print(U)
```

```
[[-4.38900010e-01  4.58631218e-01 -3.11487058e-01 -3.85784701e-01
  -5.92596122e-01]
 [-4.38900010e-01  4.58631218e-01 -3.11487058e-01  3.85784701e-01
   5.92596122e-01]
 [-4.33271443e-01 -5.31610332e-01 -1.72239110e-01 -5.92596122e-01
   3.85784701e-01]
 [-4.33271443e-01 -5.31610332e-01 -1.72239110e-01  5.92596122e-01
  -3.85784701e-01]
```

```
 [-4.89167942e-01  1.18725399e-01  8.64071180e-01  0.00000000e+00
  -3.33066907e-16]]
```

[37]:
```
print(S)  #(singular values: diagonal of S)
```

```
[18.37853642  3.02755553  0.25160832]
```

[38]:
```
print(Vh)
```

```
[[-0.43835065 -0.75639811 -0.48550037]
 [-0.33895126  0.63939753 -0.69013248]
 [ 0.83244264 -0.13795906 -0.53666241]]
```

[39]:
```
# Reconstruction based on full SVD, 2D case:
U, S, Vh = np.linalg.svd(M, full_matrices=True)
print(U.shape, S.shape, Vh.shape)
print(np.allclose(M, np.dot(U[:, :3] * S, Vh)))

Smat = np.zeros((5, 3), dtype=complex)
Smat[:3, :3] = np.diag(S)
print(np.allclose(M, np.dot(U, np.dot(Smat, Vh))))
```

```
(5, 5) (3,) (3, 3)
True
True
```

[40]:
```
# Reconstruction based on reduced SVD, 2D case:
U, S, Vh = np.linalg.svd(M, full_matrices=False)
print(U.shape, S.shape, Vh.shape)
```

```
(5, 3) (3,) (3, 3)
```

[41]:
```
"""
TODO: 6. reconstruct M by USVh using np.dot.
HINT: you should diagonalize S first, and then use np.allclose
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
print(np.allclose(M, U@np.diag(S)@Vh))
# np.allclose(M, np.dot(np.dot(U, np.diag(S)), Vh))
# np.allclose(M, np.dot(U * S, Vh))
### END SOLUTION
```

```
True
```

You know that the singular values of the matrix M are the square roots of the eigenvalues of

$$M^\top M.$$

In the following, verify this fact.

[42]:
```python
"""
TODO: 7. compute eigenvalues of M^T * M.
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
eigvals, eigvecs = np.linalg.eigh(M.T@M)
print(eigvals)
#np.linalg.eigvalsh(np.dot(M.T,M))
### END SOLUTION
```

```
[6.33067458e-02 9.16609249e+00 3.37770601e+02]
```

[43]:
```python
"""
TODO: 8. compute square of S.
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
np.square(S)
### END SOLUTION
```

[43]:
```
array([3.37770601e+02, 9.16609249e+00, 6.33067458e-02])
```

The rows of $V^\top$ are the eigenvectors of $M^\top M$.

[44]:
```python
"""
TODO: 9. compute the eigenvectors of M^T*M.
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
eigvals, eigvecs = np.linalg.eigh(M.T@M)
print(eigvals)
### END SOLUTION
```

```
[6.33067458e-02 9.16609249e+00 3.37770601e+02]
```

[45]:
```python
print(Vh.T)
```

```
[[-0.43835065 -0.33895126  0.83244264]
 [-0.75639811  0.63939753 -0.13795906]
 [-0.48550037 -0.69013248 -0.53666241]]
```

The columns of U are the eigenvectors of $MM^\top$. Verify that this is true.

[46]:
```
"""
TODO: 10. compute the eigenvectors of M*M^T.
"""

# Delete raise NotImplementedError()
### BEGIN SOLUTION
eigvals, eigvecs = np.linalg.eigh(M@M.T)
print(eigvecs)
### END SOLUTION
```

```
[[ 1.23341393e-04  7.07106770e-01  3.11487058e-01  4.58631218e-01
  -4.38900010e-01]
 [-1.23341393e-04 -7.07106770e-01  3.11487058e-01  4.58631218e-01
  -4.38900010e-01]
 [-7.07106770e-01  1.23341393e-04  1.72239110e-01 -5.31610332e-01
  -4.33271443e-01]
 [ 7.07106770e-01 -1.23341393e-04  1.72239110e-01 -5.31610332e-01
  -4.33271443e-01]
 [ 2.31639794e-16  1.43607348e-13 -8.64071180e-01  1.18725399e-01
  -4.89167942e-01]]
```

[47]:
```
U, S, Vh = np.linalg.svd(M, full_matrices=True)
print(U)
```

```
[[-4.38900010e-01  4.58631218e-01 -3.11487058e-01 -3.85784701e-01
  -5.92596122e-01]
 [-4.38900010e-01  4.58631218e-01 -3.11487058e-01  3.85784701e-01
   5.92596122e-01]
 [-4.33271443e-01 -5.31610332e-01 -1.72239110e-01 -5.92596122e-01
   3.85784701e-01]
 [-4.33271443e-01 -5.31610332e-01 -1.72239110e-01  5.92596122e-01
  -3.85784701e-01]
 [-4.89167942e-01  1.18725399e-01  8.64071180e-01  0.00000000e+00
  -3.33066907e-16]]
```

1.5 Submission Instructions

1.5.1 Download Code Portion

- Restart the kernel and run all the cells to make sure your code works.
- Save your notebook using File > Save and Checkpoint.
- Use File > Downland as > PDF via Latex.
- Download the PDF file and confirm that none of your work is missing or cut off.
- **DO NOT** simply take pictures using your phone.

1.5.2 Submitting

- Submit the assignment to Lab1 on Gradescope.
- **Make sure to assign each page of your pdf to the correct question.**