Advanced Usage of Numpy

Rina Buoy

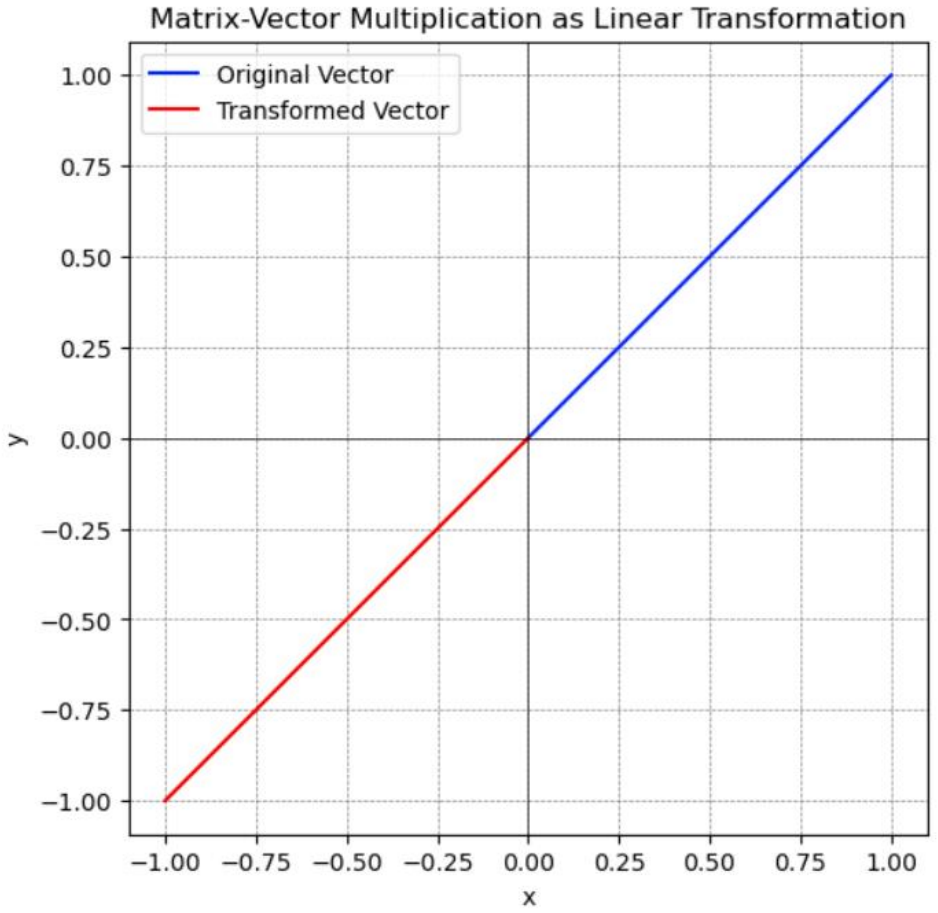
AMERICAN UNIVERSITY OF PHNOM PENH

STUDY LOCALLY. LIVE GLOBALLY.

Matrix-Vector Multiplication a Linear Transformation

Matrix-Vector Multiplication

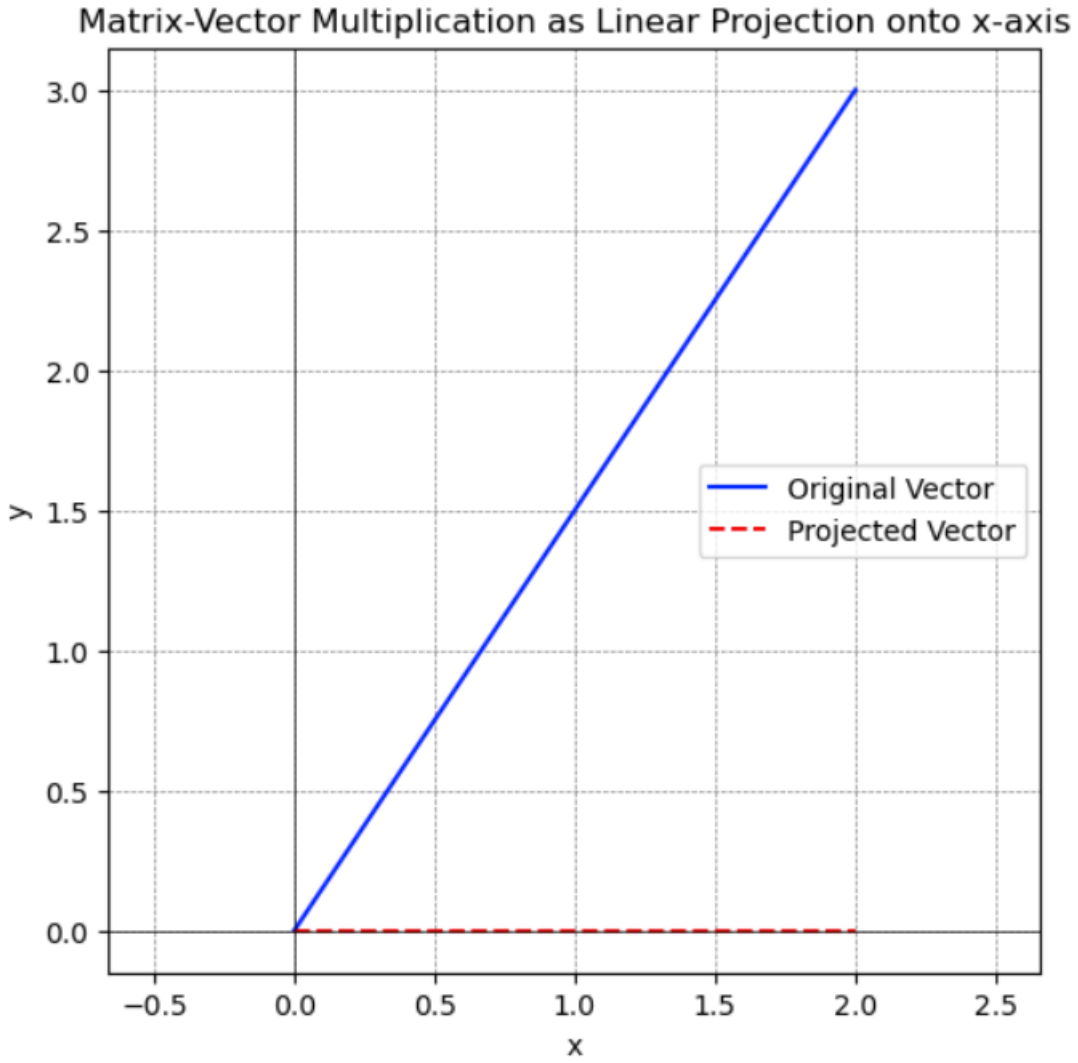
```
# Define a 2x2 matrix
A = np.array([[0, -1],
              [-1, 0]])  # (2,2)
# Define a vector
v = np.array([1, 1]) # (2,)
# Perform matrix-vector multiplication
Av = np.dot(A, v) #(2,)
```



Matrix-Vector Multiplication a Linear Projection

Matrix-Vector Multiplication

```
# Define a 2x2 matrix
A = np.array([[0, -1],
              [0, 0]])  # (2,2)
# Define a vector
v = np.array([1, 1]) # (2,)
# Perform matrix-vector multiplication
Av = np.dot(A, v) #(1,)
```



Statistical Operations

NumPy provides statistical functions such as:

- sum(), min(), max()
- average(): weighted average
- mean(), median(), std(), var(), percentile():

- naamean(), nanmedian(), nanstd(), nanvar(), nanpercentile(): ignore nan.

- corrcoef() (correlation coefficient); correlate() (cross-correlation between two 1D arrays)

```
m1 = np.array([[11, 22, 33], [44, 55, 66]])
m1.mean() # All elements, using ndarray member function

m1.mean(axis = 0)   # Over the rows
#array([27.5, 38.5, 49.5])

m1.mean(axis = 1)   # Over the columns
#array([22., 55.])
```

Linear Algebra

NumPy provides LA functions such as:

- numpy.transpose():
- numpy.trace():
- numpy.eye(dim): create an identity matrix
- numpy.dot(a1, a2): compute the dot product. For 1D, it is the inner product. For 2D, it is equivalent to matrix multiplication.
- numpy.linalg.inv(m): compute the inverse of matrix m
- numpy.linalg.eig(m): compute the eigenvalues and right eigenvectors of square matrix m.
- numpy.linalg.solve(a, b): Solving system of linear equations ax = b.

```
# Solving system of linear equations ax = b
a = np.array([[1, 3, -2], [3, 5, 6], [2, 4, 3]])
b = np.array([[5], [7], [8]])
x = np.linalg.solve(a, b)

A = np.array([[1, 0],
              [0, 0]])
Ainv = np.linalg.inv(A)
```

System of Linear Equation

2.0x + 4.0y +6.0z = 18

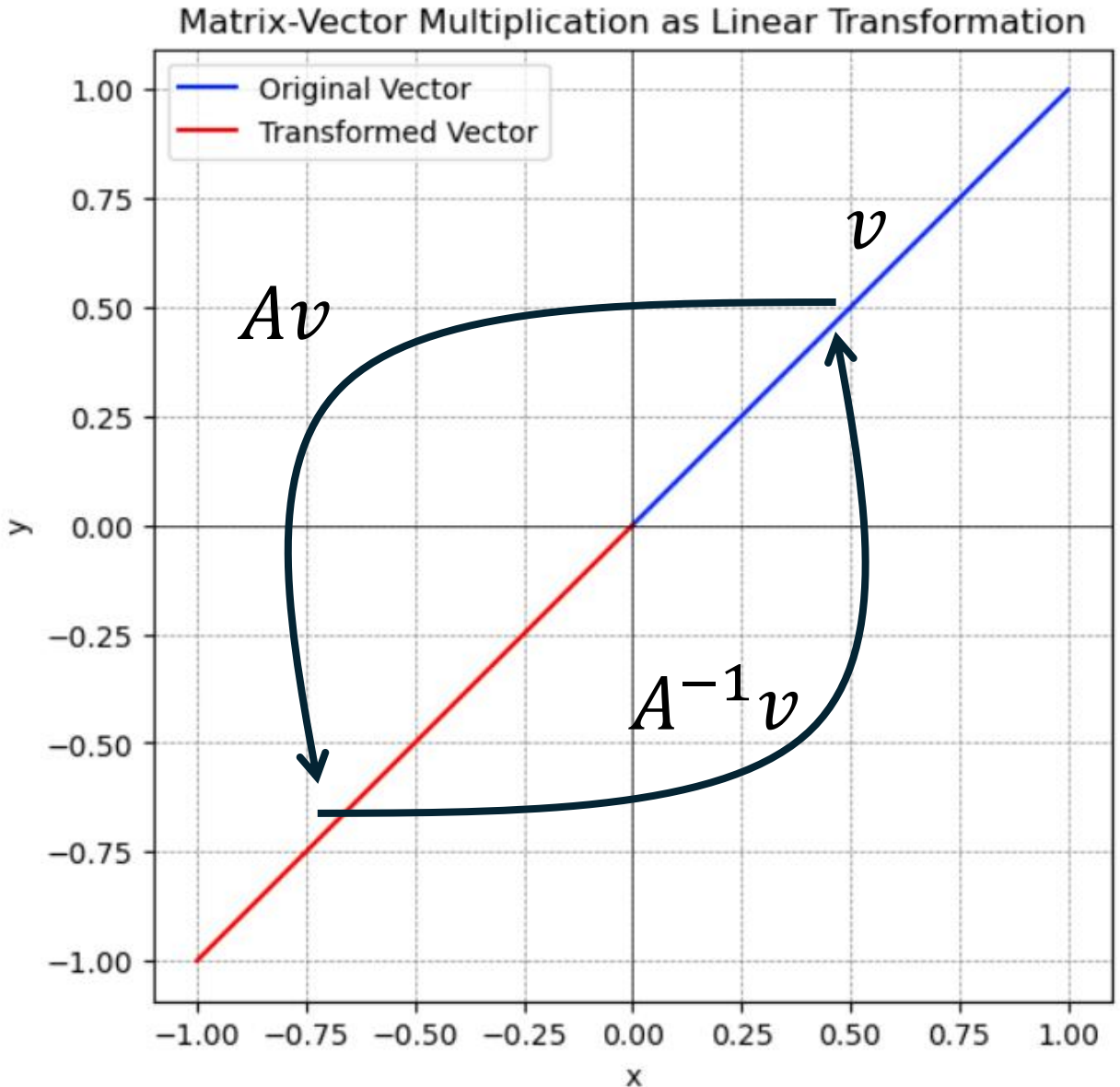
4.0x + 5.0y +6.0z = 24

3.0x + 1y - 2.0z = 4

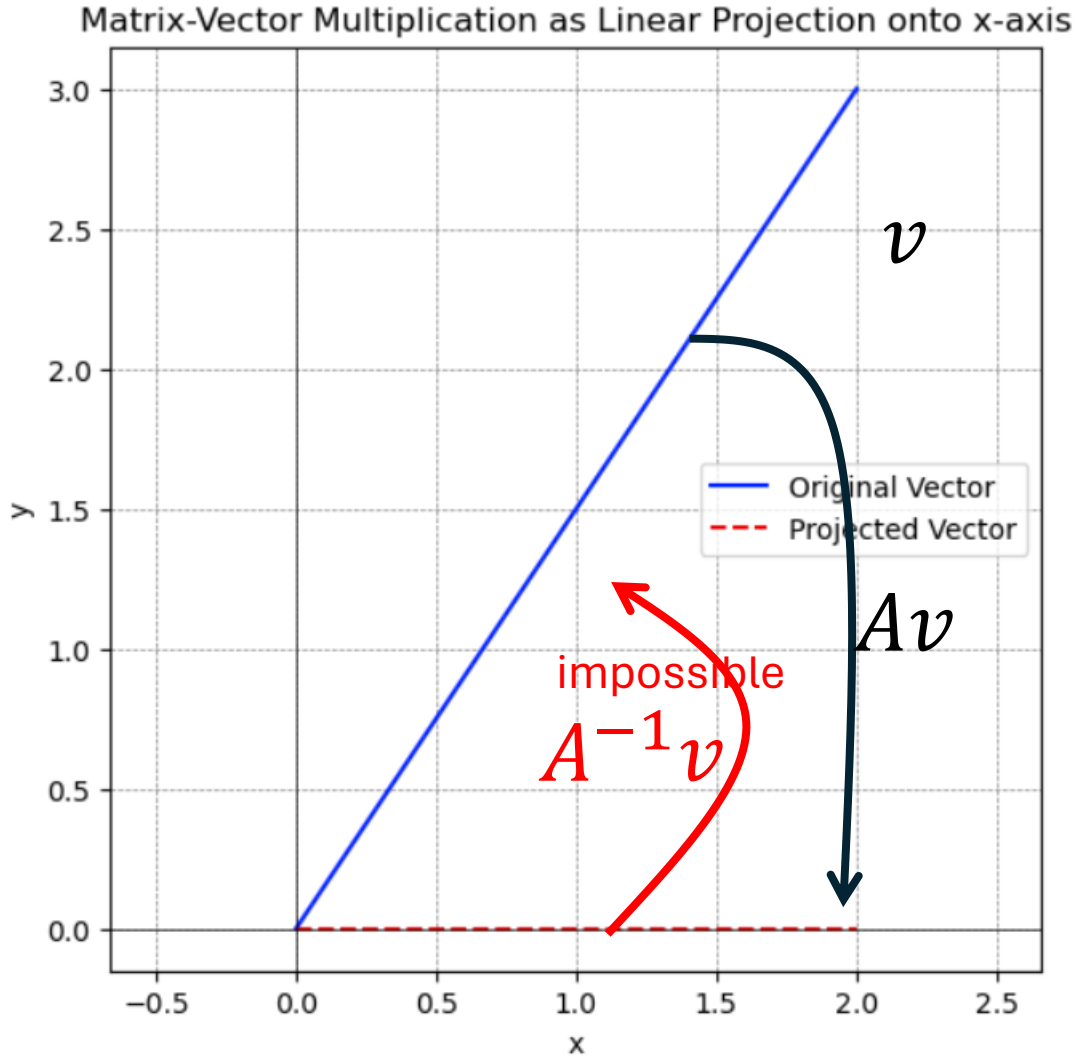
Matrix representation

$$A = \begin{bmatrix} 2.0 & 4.0 & 6.0 \\ 4.0 & 5.0 & 6.0 \\ 3.0 & 1.0 & -2.0 \end{bmatrix} \quad X = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad b = \begin{bmatrix} 18.0 \\ 24.0 \\ 4.0 \end{bmatrix}$$

Inverse Matrix



Inverse Matrix



Custom Function

Numpy.apply_along_axis(func, axis, ndarray) applies the given func along the axis for the ndarray. For example,

```
m1 = np.array([[1 , 2, 3], [4, 5, 6]])

np.apply_along_axis(np.sum, 0, m1)  # axis-0 is column-wise
# array([5, 7, 9])
np.apply_along_axis(np.sum, 1, m1)  # axis-1 is row-wise
#array([ 6, 15])

np.apply_along_axis(lambda v: v+1, 0, m1)
#OR
def my_func(v):
        return v+1
np.apply_along_axis(my_func, 0, m1)
```

Vectorization

Normal functions that work on scalar cannot be applied to list. You can vectorize the function via numpy.vectorize(func). For example,

```
# Define a scalar function
def myfunc(x):
    return x + 1
# Run the scalar function
myfunc(5) # 6

m1 = [[11, 22, 33], [44, 55, 66]]
myfunc(m1) # error

v_myfunc = np.vectorize(myfunc)
v_myfunc(m1) # (2,2) 2D array
```