Linear Algebra

The main Python package for linear algebra is the

import numpy as np
import numpy.linalg as la

Creating arrays, scalars, and matrices in Python

Scalars can be created easily like this:

```
In [2]:
x = .5
print(x)
0.5
```

Vectors and Lists

The numpy library (we will reference it by np) is the workhorse library for linear algebra in python. To creat a vector simply surround a python list ([1,2,3][1,2,3]) with the np.array function:

```
In [3]:
x_vector = np.array([1,2,3])
print(x_vector)
[1 2 3]
```

We could have done this by defining a python list and converting it to an array:

```
In [4]:
c_list = [1,2]
print("The list:",c_list)
print("Has length:", len(c_list))
c_vector = np.array(c_list)
print("The vector:", c_vector)
print("Has shape:",c_vector.shape)
The list: [1, 2]
Has length: 2
The vector: [1 2]
Has shape: (2,)
```

In [5]:

```
z = [5,6]
print("This is a list, not an array:",z)
print(type(z))
```

```
This is a list, not an array: [5, 6]
<class 'list'>
```

In [6]:

```
zarray = np.array(z)
print("This is an array, not a list",zarray)
print(type(zarray))
```

```
This is an array, not a list [5 6]
<class 'numpy.ndarray'>
```

NumPy Arrays

1D NumPy array as a list of numbers. We can think of a 2D NumPy array as a matrix. And we can think of a 3D array as a cube of numbers. When we select a row or column from a 2D NumPy array, the result is a 1D NumPy array (called a slice).

Array Attributes
Create a 1D (one-dimensional) NumPy array and verify its dimensions, shape and size.

```
a = np.array([1,3,-2,1])
print(a)
```

Output: [1 3 -2 1]

Verify the number of dimensions:

```
a.ndim
```

Output: 1

Verify the shape of the array:

```
a.shape
```

Output: (4,)

The shape of an array is returned as a Python tuple. The output in the cell above is a tuple of length 1. And we verify the size of the array (ie. the total number of entries in the array):

```
a.size
```

Output: 4

Create a 2D (two-dimensional) NumPy array (ie. matrix):

```
M = np.array([[1,2],[3,7],[-1,5]])
print(M)
```

Output:
```
[[ 1  2]
 [ 3  7]
 [-1  5]]
```

Verify the number of dimensions:

```
M.ndim
```

Ouput: 2

Verify the shape of the array:
M.shape
Output: (3, 2)

Finally, verify the total number of entries in the array:
M.size
Output: 6

Select a row or column from a 2D NumPy array and we get a 1D array:
col = M[:,1]
print(col)
Output: [2 7 5]

Verify the number of dimensions of the slice:
col.ndim
Output: 1

Verify the shape and size of the slice:
col.shape
Output: (3,)
col.size
Output: 3

When we select a row of column from a 2D NumPy array, the result is a 1D NumPy array. However, we may want to select a column as a 2D column vector. This requires us to use the reshape method.

<u>For example, create a 2D column vector from the 1D slice selected from the matrix M above:</u>

print(col)
Output: [2 7 5]
column = np.array([2,7,5]).reshape(3,1)
print(column)
Output: [[2]
[7]
[5]]

Verify the dimensions, shape and size of the array:

```
print('Dimensions:', column.ndim)
print('Shape:', column.shape)
print('Size:', column.size)
```

```
Output:
Dimensions: 2
Shape: (3, 1)
Size: 3
```

The variables col and column are different types of objects even though they have the “same” data.

```
print(col)
Output: [2 7 5]
print('Dimensions:',col.ndim)
print('Shape:',col.shape)
print('Size:',col.size)
```

```
Output:
Dimensions: 1
Shape: (3,)
Size: 3
```

<u>Matrix Operations and Functions</u>

Matrices

In [7]:

```
b = list(zip(z,c_vector))
print(b)
print("Note that the length of our zipped list is 2 not (2 by 2):",len(b))
```

```
[(5, 1), (6, 2)]
Note that the length of our zipped list is 2 not (2 by 2): 2
```

In [8]:

```
print( "But we can convert the list to a matrix like this:")
A = np.array(b)
print( A)
print( type(A))
print( "A has shape:",A.shape)
```

```
But we can convert the list to a matrix like this:
[[5 1]
 [6 2]]
<class 'numpy.ndarray'>
A has shape: (2, 2)
```

Matrix Addition and Subtraction

Adding or subtracting a scalar value to a matrix

To learn the basics, consider a small matrix of dimension 2×2, where 2×2 denotes the number of rows $\times$ the number of columns. Let $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$. Consider adding a scalar value (e.g. 3) to the A.

$$A + 3 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + 3 = \begin{bmatrix} a_{11} + 3 & a_{12} + 3 \\ a_{21} + 3 & a_{22} + 3 \end{bmatrix}$$

The same basic principle holds true for A-3:

$$A - 3 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - 3 = \begin{bmatrix} a_{11} - 3 & a_{12} - 3 \\ a_{21} - 3 & a_{22} - 3 \end{bmatrix}$$

Notice that we add (or subtract) the scalar value to each element in the matrix A. A can be of any dimension.

This is trivial to implement, now that we have defined our matrix A:

In [9]:

```
result = A + 3
#or
result = 3 + A
print( result)
```

```
[[8 4]
 [9 5]]
```

Adding or subtracting two matrices

Consider two small 2×2 matrices, where 2×2 denotes the # of rows $\times$ the # of columns. Let $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ and $B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$. To find the result of $A - B$, simply subtract each element of A with the corresponding element of B:

$$A - B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} - \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} - b_{11} & a_{12} - b_{12} \\ a_{21} - b_{21} & a_{22} - b_{22} \end{bmatrix}$$

Addition works exactly the same way:

$$A + B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}$$

An important point to know about matrix addition and subtraction is that it is only defined when A and B are of the same size. Here, both are 2×2. Since operations are performed element by element, these two matrices must be conformable- and for addition and subtraction that means they must have the same numbers of rows and columns. I like to be explicit about the dimensions of matrices for checking conformability as I write the equations, so write

$$A_{2\times2} + B_{2\times2} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}_{2\times2}$$

Notice that the result of a matrix addition or subtraction operation is always of the same dimension as the two operands.

Let's define another matrix, B, that is also 2×2 and add it to A:

```
B = np.random.randn(2,2)
print( B)
```

```
[[-0.9959588   1.11897568]
 [ 0.96218881 -1.10783668]]
```

In [11]:

```
result = A + B
result
```

Out[11]:

```
array([[4.0040412 , 2.11897568],
       [6.96218881, 0.89216332]])
```

Arithmetic Operations

Recall that arithmetic array operations +, -, /, * and ** are performed elementwise on NumPy arrays. Let's create a NumPy array and do some computations:

```
M = np.array([[3,4],[-1,5]])
print(M)
```

Output:

```
[[ 3  4]
 [-1  5]]
```

```
M * M
```

```
array([[ 9, 16],
       [ 1, 25]])
```

Matrix Multiplication

We use the @ operator to do matrix multiplication with NumPy arrays:

```
M @ M
```

```
array([[ 5, 32],
       [-8, 21]])
```

$$A = \begin{bmatrix} 1 & 3 \\ -1 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 2 \\ 1 & 2 \end{bmatrix}$$

and I is the identity matrix of size 2:

```
A = np.array([[1,3],[-1,7]])
print(A)
```

Output:

```
[[ 1  3]
 [-1  7]]
```

```
B = np.array([[5,2],[1,2]])
print(B)
```

Output:

```
[[5 2]
 [1 2]]
```

```
I = np.eye(2)
print(I)
```

Output:

```
[[1. 0.]
 [0. 1.]]
```

```
2*I + 3*A - A@B
```

Output:

```
array([[-3.,  1.],
       [-5., 11.]])
```

Multiplying two matricies

Now, consider the 2×1 vector $C = \begin{pmatrix} c_{11} \\ c_{21} \end{pmatrix}$

Consider multiplying matrix $A_{2\times2}$ and the vector $C_{2\times1}$. Unlike the addition and subtraction case, this product is defined. Here, conformability depends not on the row **and** column dimensions, but rather on the column dimensions of the first operand and the row dimensions of the second operand. We can write this operation as follows

$$A_{2\times2} \times C_{2\times1} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}_{2\times2} \times \begin{bmatrix} c_{11} \\ c_{21} \end{bmatrix}_{2\times1} = \begin{bmatrix} a_{11}c_{11} + a_{12}c_{21} \\ a_{21}c_{11} + a_{22}c_{21} \end{bmatrix}_{2\times1}$$

Alternatively, consider a matrix C of dimension 2×3 and a matrix A of dimension 3×2

$$A_{3\times2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}_{3\times2}, C_{2\times3} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}_{2\times3}$$

Here, A × C is

$$\begin{aligned} A_{3\times2} \times C_{2\times3} &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}_{3\times2} \times \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}_{2\times3} \\ &= \begin{bmatrix} a_{11}c_{11} + a_{12}c_{21} & a_{11}c_{12} + a_{12}c_{22} & a_{11}c_{13} + a_{12}c_{23} \\ a_{21}c_{11} + a_{22}c_{21} & a_{21}c_{12} + a_{22}c_{22} & a_{21}c_{13} + a_{22}c_{23} \\ a_{31}c_{11} + a_{32}c_{21} & a_{31}c_{12} + a_{32}c_{22} & a_{31}c_{13} + a_{32}c_{23} \end{bmatrix}_{3\times3} \end{aligned}$$

```
# Let's redefine A and C to demonstrate matrix multiplication:
```

```
A = np.arange(6).reshape((3,2))
C = np.random.randn(2,2)
print( A.shape)
print( C.shape)
```
(3, 2)
(2, 2)

We will use the numpy dot operator to perform the these multiplications. You can use it two ways to yield the same result:

```
In [14]:
print( A.dot(C))
print( np.dot(A,C))
```
```
[[-1.19691566  1.08128294]
 [-2.47040472  1.00586034]
 [-3.74389379  0.93043773]]
[[-1.19691566  1.08128294]
 [-2.47040472  1.00586034]
 [-3.74389379  0.93043773]]
```

Matrix Powers

There's no symbol for matrix powers and so we must import the function matrix_power from the subpackage numpy.linalg.

```
from numpy.linalg import matrix_power as mpow
```

```
M = np.array([[3,4],[-1,5]])
print(M)
```

Output:
```
[[ 3  4]
 [-1  5]]
```

```
mpow(M,2)
```

Output:

```
array([[ 5, 32],
       [-8, 21]])
```

mpow(M,5)

Output:

```
array([[-1525,  3236],
       [ -809,    93]])
```

Compare with the matrix multiplcation operator:

M @ M @ M @ M @ M

Output:

```
array([[-1525,  3236],
       [ -809,    93]])
```

mpow(M,3)

Output:

```
array([[-17, 180],
       [-45,  73]])
```

M @ M @ M

Output:

```
array([[-17, 180],
       [-45,  73]])
```

Transpose

We can take the transpose with .T attribute:
print(M)

Output:

```
[[ 3  4]
 [-1  5]]
```

print(M.T)

Output:

```
[[ 3 -1]
 [ 4  5]]
```

Notice that MM^T is a symmetric matrix:

```
M @ M.T
```

Output:

```
array([[25, 17],
       [17, 26]])
```

Transposing a Matrix

At times it is useful to pivot a matrix for conformability- that is in order to matrix divide or multiply, we need to switch the rows and column dimensions of matrices. Consider the matrix

$$A_{3\times 2} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}_{3\times 2}$$

The transpose of A (denoted as A') is

$$A' = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \end{bmatrix}_{2\times 3}$$

```
A = np.arange(6).reshape((3,2))
B = np.arange(8).reshape((2,4))
print( "A is")
print( A)
print( "The Transpose of A is")
print( A.T)
```

```
A is
[[0 1]
 [2 3]
 [4 5]]
The Transpose of A is
[[0 2 4]
 [1 3 5]]
```

Inverse

We can find the inverse using the function scipy.linalg.inv:

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

Output:

```
[[1 2]
 [3 4]]
```

```
la.inv(A)
```

Output:

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Trace

We can find the trace of a matrix using the function numpy.trace:

```
np.trace(A)
```

Output:
5

Determinant

We find the determinant using the function scipy.linalg.det:

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

Output:

```
[[1 2]
 [3 4]]
```

```
la.det(A)
```

Output:

-2.0

Projections

The formula to project a vector v onto a vector w is

$$\mathrm{proj}_w(v) = \frac{v \cdot w}{w \cdot w} w$$

Let's a function called proj which computes the projection v onto w.

```
def proj(v,w):
    '''Project vector v onto w.'''
    v = np.array(v)
    w = np.array(w)
    return np.sum(v * w)/np.sum(w * w) * w   # or (v @ w)/(w @ w) * w
proj([1,2,3],[1,1,1])
```

Output:

array([2., 2., 2.])

Eigenvalues and Eigenvectors

Definition

Let A be a square matrix. A non-zero vector v is an eigenvector for A with eigenvalue λ if

$$Av=\lambda v$$

Rearranging the equation, we see that v is a solution of the homogeneous system of equations

$$(A-\lambda I)v=0$$

where I is the identity matrix of size n. Non-trivial solutions exist only if the matrix $A-\lambda I$ is singular which means $\det(A-\lambda I)=0$. Therefore eigenvalues of A are roots of the characteristic polynomial

$$p(\lambda)=\det(A-\lambda I)$$

numpy.linalg.eig

The function numpy.linalg.eig computes eigenvalues and eigenvectors of a square matrix A.

Let's consider a simple example with a diagonal matrix:

```
A = np.array([[1,0],[0,-2]])
print(A)
```

Output:

```
[[ 1  0]
 [ 0 -2]]
```

The function la.eig returns a tuple (eigvals,eigvecs) where eigvals is a 1D NumPy array of complex numbers giving the eigenvalues of A, and eigvecs is a 2D NumPy array with the corresponding eigenvectors in the columns:

```
results = la.eig(A)
```

The eigenvalues of A are:

```
print(results[0])
```

Output:

```
[ 1.+0.j -2.+0.j]
```

The corresponding eigenvectors are:

```
print(results[1])
```

Output:

```
[[1. 0.]
 [0. 1.]]
```

We can unpack the tuple:

```
eigvals, eigvecs = la.eig(A)
print(eigvals)
```

Output:

```
[ 1.+0.j -2.+0.j]
```

```
print(eigvecs)
```

Output:

```
[[1. 0.]
 [0. 1.]]
```

If we know that the eigenvalues are real numbers (ie. if A is symmetric), then we can use the NumPy array method .real to convert the array of eigenvalues to real numbers:

```
eigvals = eigvals.real
print(eigvals)
```

Output:

```
[ 1. -2.]
```

Notice that the position of an eigenvalue in the array eigvals correspond to the column in eigvecs with its eigenvector:

```
lambda1 = eigvals[1]
print(lambda1)
```

Output:

```
-2.0
```

```
v1 = eigvecs[:,1].reshape(2,1)
print(v1)
```

Output:

```
[[0.]
 [1.]]
```

```
A @ v1
```

Output:

```
array([[ 0.],
       [-2.]])
```

```
lambda1 * v1
```

Output:

```
array([[-0.],
       [-2.]])
```

Symmetric Matrices

The eigenvalues of a symmetric matrix are always real and the eigenvectors are always orthogonal! Let's verify these facts with some random matrices:

```
n = 4
P = np.random.randint(0,10,(n,n))
print(P)
```

Output:

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

Create the symmetric matrix S=PPT:

```
S = P @ P.T
print(S)
```

Output:

```
[[ 89  75  22 102]
 [ 75 116  27 120]
 [ 22  27  33  62]
 [102 120  62 200]]
```

Let's unpack the eigenvalues and eigenvectors of S:

```
evals, evecs = la.eig(S)
print(evals)
```

Output:

[361.75382302+0.j 42.74593101+0.j 26.33718907+0.j   7.16305691+0.j]

The eigenvalues all have zero imaginary part and so they are indeed real numbers:

```
evals = evals.real
print(evals)
```

Output:

[361.75382302 42.74593101 26.33718907   7.16305691]

The corresponding eigenvectors of A are:

```
print(evecs)
```

Output:

```
[[-0.42552429 -0.42476765  0.76464379 -0.23199439]
 [-0.50507589 -0.54267519 -0.64193252 -0.19576676]
 [-0.20612674  0.54869183 -0.05515612 -0.80833585]
 [-0.72203822  0.4733005   0.01415338  0.50442752]]
```

Let's check that the eigenvectors are orthogonal to each other:

```
v1 = evecs[:,0] # First column is the first eigenvector
print(v1)
```

Output:

[-0.42552429 -0.50507589 -0.20612674 -0.72203822]

```
v2 = evecs[:,1] # Second column is the second eigenvector
print(v2)
```

Output:

[-0.42476765 -0.54267519 0.54869183 0.4733005]

```
v1 @ v2
-1.1102230246251565e-16
```

The dot product of eigenvectors v1 and v2 is zero (the number above is *very* close to zero and is due to rounding errors in the computations) and so they are orthogonal!

Diagonalization

A square matrix M is diagonalizable if it is similar to a diagonal matrix. In other words, M is diagonalizable if there exists an invertible matrix P such that $D=P^{-1}MP$ is a diagonal matrix. A beautiful result in linear algebra is that a square matrix M of size n is diagonalizable if and only if M has n independent eigenvectors. Furthermore, $M=PDP^{-1}$ where the columns of P are the eigenvectors of M and D has corresponding eigenvalues along the diagonal.

Let's use this to construct a matrix with given eigenvalues $\lambda_1=3,\lambda_2=1$, and eigenvectors $v_1=[1,1]^T,v_2=[1,-1]^T$.

```
P = np.array([[1,1],[1,-1]])
print(P)
```

Output:

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

```
D = np.diag((3,1))
print(D)
```

Output:

```
[[3 0]
 [0 1]]
```

```
M = P @ D @ la.inv(P)
print(M)
```

Output:

```
[[2. 1.]
```

```
 [1. 2.]]
```

Let's verify that the eigenvalues of M are 3 and 1:

```
evals, evecs = la.eig(M)
print(evals)
```

Output:

```
[3.+0.j 1.+0.j]
```

Verify the eigenvectors:

```
print(evecs)
```

Output:

```
[[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```