# Linear Algebra in Python

June 19, 2016

```
In [1]: import numpy as np
```

---

## 1 Vectors

Vectors in Python are just NumPy arrays like we've been using in the problem sets. For example, the following code creates the vector $\mathbf{x} = \langle 0, 1, 2, 3 \rangle$:

```
In [2]: x = np.arange(4)
        x
```

```
Out[2]: array([0, 1, 2, 3])
```

The following code creates a vector $\mathbf{y} = \langle 1, 1, 1, 1 \rangle$:

```
In [3]: y = np.ones(4)
        y
```

```
Out[3]: array([ 1.,  1.,  1.,  1.])
```

Note that there is no difference in NumPy between a row vector and a column vector – they are both just 1D arrays.

### 1.1 Adding vectors

We can add vectors just by adding them. The following is equivalent to $\mathbf{x} + \mathbf{y} = \langle 0, 1, 2, 3 \rangle + \langle 1, 1, 1, 1 \rangle$:

```
In [4]: x + y
```

```
Out[4]: array([ 1.,  2.,  3.,  4.])
```

### 1.2 Inner product

To take the inner product between two vectors, we can use the `np.dot` function. The following is equivalent to $\mathbf{x}^\top \mathbf{y} = \langle 0, 1, 2, 3 \rangle \cdot \langle 1, 1, 1, 1 \rangle$:

```
In [5]: np.dot(x, y)
```

```
Out[5]: 6.0
```

### 1.3 Vector norm

Recall that the vector norm is:

$$\|\mathbf{x}\| = \sqrt{\mathbf{x}^\top \mathbf{x}}$$

We can either compute this manually with `np.sqrt` and `np.dot`:

```
In [6]: np.sqrt(np.dot(x, x))
```

```
Out[6]: 3.7416573867739413
```

Or, we can just use `np.linalg.norm`:

```
In [7]: np.linalg.norm(x)
```

```
Out[7]: 3.7416573867739413
```

---

## 2 Matrices

We can create matrices in the same way as we create vectors. For example, the following code creates a $4 \times 4$ matrix $\mathbf{A}$ of all 1's:

```
In [8]: A = np.ones((4, 4))
        A
```

```
Out[8]: array([[ 1.,  1.,  1.,  1.],
               [ 1.,  1.,  1.,  1.],
               [ 1.,  1.,  1.,  1.],
               [ 1.,  1.,  1.,  1.]])
```

### 2.1 Identity matrices

Identity matrices are a special type of matrix that is all zeros, except for ones along the diagonal. You can create them with `np.eye`, e.g. the following code creates a $4 \times 4$ indentity matrix $\mathbf{I}$:

```
In [9]: I = np.eye(4)
        I
```

```
Out[9]: array([[ 1.,  0.,  0.,  0.],
               [ 0.,  1.,  0.,  0.],
               [ 0.,  0.,  1.,  0.],
               [ 0.,  0.,  0.,  1.]])
```

### 2.2 Diagonal matrices

Diagonal matrices are a lot like identity matrices, except that they can have other values along the diagonal besides 1. To create them, we start with a vector, and use `np.diag` to turn it into a diagonal matrix. For example, the following code turns the vector $\mathbf{z} = \langle 5, 8, 2, 9 \rangle$ into a diagonal matrix $\mathbf{B}$:

```
In [10]: z = np.array([5, 8, 2, 9])
         B = np.diag(z)
         B
```

```
Out[10]: array([[5, 0, 0, 0],
                [0, 8, 0, 0],
                [0, 0, 2, 0],
                [0, 0, 0, 9]])

In [11]: np.diag(B)

Out[11]: array([5, 8, 2, 9])
```

## 2.3 Transposing matrices

To transpose a matrix, we can just use the `.T` attribute of NumPy arrays. To illustrate, let's first create a random $4 \times 4$ matrix $\mathbf{C}$:

```
In [12]: C = np.random.randint(0, 10, (4, 4))
         C

Out[12]: array([[9, 4, 5, 3],
                [2, 5, 0, 1],
                [6, 4, 1, 4],
                [0, 9, 7, 3]])
```

Then, $\mathbf{C}^\top$ is just:

```
In [13]: C.T

Out[13]: array([[9, 2, 6, 0],
                [4, 5, 4, 9],
                [5, 0, 1, 7],
                [3, 1, 4, 3]])
```

## 2.4 Matrix multiplication

To do matrix multiplication, we cannot use the `*` operator, as this will perform elementwise multiplication. **Matrix multiplication is a different thing!** Instead, we need to use the function `np.dot` (which we also used for the inner product of two vectors). First, let's just take a look at $\mathbf{B}$ and $\mathbf{C}$ as a reminder of what they are:

```
In [14]: B

Out[14]: array([[5, 0, 0, 0],
                [0, 8, 0, 0],
                [0, 0, 2, 0],
                [0, 0, 0, 9]])

In [15]: C

Out[15]: array([[9, 4, 5, 3],
                [2, 5, 0, 1],
                [6, 4, 1, 4],
                [0, 9, 7, 3]])
```

Then, to compute $\mathbf{B} \cdot \mathbf{C}$, we use `np.dot`:

```
In [16]: np.dot(B, C)
```

```
Out[16]: array([[45, 20, 25, 15],
                [16, 40,  0,  8],
                [12,  8,  2,  8],
                [ 0, 81, 63, 27]])
```

If instead we were to use the * operator instead, we would no longer get the appropriate matrix product. In this case, NumPy performs element-wise multiplication, where the element in the $(i, j)$ position of $\mathbf{B}$ is multiplied by the element in the $(i, j)$ position of $\mathbf{C}$:

```
In [17]: B * C

Out[17]: array([[45,  0,  0,  0],
                [ 0, 40,  0,  0],
                [ 0,  0,  2,  0],
                [ 0,  0,  0, 27]])
```

Remember that taking the dot product of any matrix with the identity will produce that matrix again:

```
In [18]: np.dot(C, np.eye(4))

Out[18]: array([[ 9.,  4.,  5.,  3.],
                [ 2.,  5.,  0.,  1.],
                [ 6.,  4.,  1.,  4.],
                [ 0.,  9.,  7.,  3.]])
```

Also remember that the matrix dimensions have to match: the number of columns of the left matrix have to be the same as the number of rows of the second matrix. For example, let's create two random matrices. The first, $\mathbf{M}$, will be $2 \times 4$, while the second, $\mathbf{N}$, will be $3 \times 4$:

```
In [19]: M = np.random.randint(0, 10, (2, 4))
         M

Out[19]: array([[1, 5, 2, 3],
                [1, 4, 7, 2]])

In [20]: N = np.random.randint(0, 10, (3, 4))
         N

Out[20]: array([[0, 9, 4, 4],
                [5, 2, 1, 8],
                [8, 0, 0, 3]])
```

Now, if we try to use `np.dot` on these matrices as they are, the dimensions won't match:

```
In [21]: np.dot(M, N)


         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-21-f24d244a469c> in <module>()
    ----> 1 np.dot(M, N)


         ValueError: shapes (2,4) and (3,4) not aligned: 4 (dim 1) != 3 (dim 0)
```

Instead, we need to transpose $\mathbf{N}$ so that the dimensions line up, resulting in a $2 \times 3$ matrix:

```
In [22]: np.dot(M, N.T)

Out[22]: array([[65, 41, 17],
                [72, 36, 14]])
```

---

## 3 Eigenvectors and eigenvalues

Recall that an eigenvector $\mathbf{v}$ and eigenvalue $\lambda$ of a matrix (in this case, $\mathbf{W}$) satisfies the following:

$$\mathbf{W}\mathbf{v} = \lambda\mathbf{v}$$

First, we'll construct our matrix $\mathbf{W}$ from the matrix $\mathbf{C}$ by multiplying it with itself (for those curious, this is one way that you can create a positive semidefinite matrix, which we need if we want all our eigenvalues to be real-valued):

```
In [23]: W = np.dot(C, C.T)
         W

Out[23]: array([[131,  41,  87,  80],
                [ 41,  30,  36,  48],
                [ 87,  36,  69,  55],
                [ 80,  48,  55, 139]])
```

To compute the eigenvalues of a matrix, we can use the function `np.linalg.eigvals`:

```
In [24]: np.linalg.eigvals(W)

Out[24]: array([ 286.70012541,   65.46156356,   14.29526922,    2.5430418 ])
```

If you also want the eigen vectors, then the function `np.linalg.eig` will return a tuple of both the eigenvalues and the eigenvectors:

```
In [25]: eigvals, eigvecs = np.linalg.eig(W)

In [26]: eigvals

Out[26]: array([ 286.70012541,   65.46156356,   14.29526922,    2.5430418 ])

In [27]: eigvecs

Out[27]: array([[ 0.62054223,  0.50919142, -0.48131564, -0.35211744],
                [ 0.27110025, -0.08496482,  0.72154474, -0.63139435],
                [ 0.44153078,  0.36771619,  0.46609946,  0.67274562],
                [ 0.58862769, -0.77349201, -0.17452719,  0.15737794]])
```

We can verify that these values and vectors do in fact satisfy the above equation:

```
In [28]: np.dot(W, eigvecs)

Out[28]: array([[ 177.9095362 ,   33.33246638,   -6.88053661,   -0.89544938],
                [  77.72447645,   -5.56192979,   10.31467628,   -1.60566222],
                [ 126.58693032,   24.07127678,    6.66301728,    1.71082022],
                [ 168.75963283,  -50.63399609,   -2.49491324,    0.40021869]])
```

```
In [29]: eigvals * eigvecs
```

```
Out[29]: array([[ 177.9095362 ,   33.33246638,   -6.88053661,   -0.89544938],
                [  77.72447645,   -5.56192979,   10.31467628,   -1.60566222],
                [ 126.58693032,   24.07127678,    6.66301728,    1.71082022],
                [ 168.75963283,  -50.63399609,   -2.49491324,    0.40021869]])
```

```
In [ ]:
```

```
In [ ]:
```