Matrix Operations

1 Lesson 3: Matrix Operations

Similarly to how we could use 1D NumPy arrays to represent a vector, we can extend this to two dimensions to store matrices.

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

2 Creating 2D Arrays

You can specify each entry using a list of lists:

[2]:
```
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
A
```

The shape of this array is specified as (rows, columns):

[3]:
```
A.shape
```

There are also some functions to create specific matrices

[4]:
```
I = np.eye(5)
I
```

To specify a two dimensional array we can give the shape as a tuple

[5]:
```
ones = np.ones((4, 4))
ones
```

[6]:
```
zeros = np.zeros((3, 5))
zeros
```

[7]:
```
d = np.array([1, 2, 3, 4])
D = np.diag(d)
D
```

2.1 Addition and Multiplication

Just like for vectors we can add/subtract matrices and multiply by a scalar

[8]:
```
I * 5
```

[9]:
```
np.ones((3, 3)) - np.eye(3)
```

2.1.1 Matrix-matrix and Matrix-vector multiplication

Numpy gives us built-in functions to multiply different matrices together.

[10]:
```
A = np.array([[1, 2, 3], [4, 5, 6]])
A
```

[11]:
```
B = np.array([[1, 2], [3, 4], [5, 6]])
B
```

We can multiply a 2×3 matrix by a 3×2 matrix to get a 2×2 matrix in return.

[12]:
```
np.dot(A, B)
```

What happens when we try to multiply two matrices whose shape doesn't match? Lets try multiplying a 2×3 by a 2×3 matrix.

[13]:
```
np.dot(A, A)
```

So we will have to be careful that the shapes of our arrays align.

We can also use the `@` symbol as shorthand for `np.dot()`

[14]:
```
A @ B
```

`np.dot()` also allows us to multiply a matrix by a vector:

[15]:
```
x = np.array([-1, 0, 1])
A @ x
```

2.2 Other matrix functions

Matrix transposition:

[16]:
```
A = np.array([[1, 2], [3,4]])
A
```

[17]:
```
print(A.T)
print(np.transpose(A))
```

2.3 Solving a system

We will need to import some linear algebra functions from NumPy:

[18]:
```
import numpy.linalg as la
```

Given a matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$, we can solve $\boldsymbol{Ax} = \boldsymbol{b}$ for some vector $\boldsymbol{x}$.

[19]:
```
A = np.array([[1, 2], [3, 4]])
b = np.array([1, 2])
```

[20]:
```
la.solve(A, b)
```

[21]:
```
A_singular = np.ones((2, 2))
A_singular
```

[22]:
```
la.solve(A_singular, b)
```

So we also need to be careful of trying to solve inconsistent systems of equations.

3 Images as Matrices

A grayscale image of size $m \times n$ can be stored as a NumPy array of the same size, with the value at each "pixel" representing the intensity at that point, some value between 0 and 1. A value of `0` represents total black, while `1` represents full white.

[23]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

[24]:
```
lincoln = np.loadtxt("lincoln.txt")
```

Inspect the size of the matrix that contains the image:

[25]:
```
lincoln.shape
```

You can check the min and max values of the array:

[32]:
```
print(lincoln.max())
```

[33]:
```
print(lincoln.min())
```

[34]:
```
plt.figure()
plt.imshow(lincoln, cmap="gray", vmin=0, vmax=1)
```

Try this!

The `vmin` and `vmax` tell Matplotlib what our range of values should be, but what does the `cmap` argument do? Try setting it to `plasma` or `twilight` (without using the `vmin` and `vmax` attributes).

[36]:
```

```

Try this!

What happens when you plot the transposition of the image?

[29]:

```

```

Try this!

How can we invert the colors of the image?

Recall that our image has values between 0 and 1. If we want black "pixels" to become white, and white to become black, we can accomplish that by computing `1 - pixel`.

But first... take a look at this:

`lincoln` is a 2d array with the image. `1` is a scalar. Before you try to plot, check what happens when you try to subract an array from a scalar. This is something new Python users will find very surprising!

[37]:

```

```

[38]:

```

```

3.0.1 Changing Image Values

Currently, the values in our image range from $[0, 1]$, but what if we were to scale up or scale down these values? We can plot the image scaled by `2`, but still using vmin=0 and vmax=1

[40]:

```python
bright = lincoln * 2
plt.imshow(bright, cmap="gray", vmin=0, vmax=1)
```

The maximum image values are now blown out past the range $[0, 1]$ (that is why you are observing more white regions). Take a look at the **mean** for this new image:

[41]:

```python
np.mean(bright)
```

Check your answers!

Let's try adjusting the values of the image `bright` so that the mean (average) value is now exactly in the middle of $[0, 1]$, i.e. `0.5`.

Hint: What operation changes the mean but does not affect the distance between values? You want to effectively "shift" each value by the mean.

Store your new image with 0.5 mean as `image_centered`.

[42]:

```python
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

You can plot the `image_centered` to see the adjusted image using `plt.imshow`.

[43]:

```python
plt.imshow(image_centered, cmap="gray", vmin=0, vmax=1)
```

What we have done here is adjust the contrast of the image so that light values are lighter, and dark values are darker. We can do this in the opposite direction as well to reduce the contrast values.

Check your answers!

First multiply the image `lincoln` by `0.5` and then adjust the mean to be exactly `0.5`. Store the adjusted array in `lincoln_adjusted`.

[48]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

You can plot the `lincoln_adjusted` to see the adjusted image using `plt.imshow`.

[49]:
```
plt.imshow(lincoln_adjusted, cmap="gray", vmin=0, vmax=1)
```

4 Transformations of Points

One way we can think of matrices is as a linear function $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$, where $f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}$ for some matrix $\boldsymbol{A}$. These can be used to represent various operations in 2D.

[50]:
```
x = np.array([1, 2])
```

We will start with a point at $(1, 2)$

[51]:
```
ax = plt.gca()
ax.set_aspect('equal')



# Set the axis limits
plt.xlim(-5, 5)
plt.ylim(-5, 5)

# Create lines for the x and y axes
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')

plt.plot(x[0], x[1], 'o')
```

We can create a transformation to scale the point by two along each axis. This scale transformation is given by the identity matrix multiplied by the scale factor.

[52]:
```
scale = np.eye(2) * 2
scale
```

[53]:
```
x_transformed = scale @ x
```

[54]:
```
ax = plt.gca()
ax.set_aspect('equal')

# Set the axis limits
plt.xlim(-5,5)
plt.ylim(-5,5)

# Create lines for the x and y axes
```

```
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')

# Plot our points
plt.plot(x[0], x[1], 'o')
plt.plot(x_transformed[0], x_transformed[1], 'o')
```

4.0.1 Transformations on Many Points

While it is certainly interesting to apply a transformation to a single point, we can apply these transformations to multiple points to rotate entire shapes and figures. Here you are given a 2D version of the Stanford bunny, a 3D model commonly used for testing computer graphics techniques.

[55]:
```
bunny = np.loadtxt("bunny.txt")
```

This figure is given as a $2 \times N$ array of points, with the first "row" corresponding to the x-value of each point, and the second "row" with the y-value. So we can plot the figure by separating the x and y coordinates and giving them to `matplotlib`'s `plot()` function.

[56]:
```
ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny[0], bunny[1], 'o-')
```

So how can we transform the entire figure? Perhaps the "obvious" solution here is to loop over each point individually and multiply it by our transformation.

Here we create a new copy of the bunny figure scaled by two along the x axis and unchanged along the y axis, using a transformation matrix as above. You can access a single column of an array using the notation `array[:,i]`.

[57]:
```
stretch = np.diag([2, 1])
print(stretch)
bunny_squash = np.zeros_like(bunny)

# Loop over each point to apply the transformation
for i in range(len(bunny[1])):
    bunny_squash[:,i] = stretch @ bunny[:,i]

ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_squash[0], bunny_squash[1], 'o-')
```

That seems like an awful lot of work for such a simple operation. If our transformation is a 2×2 matrix and our points are stored in an array of shape $2 \times N$, is there an easier way to have Numpy transform the entire figure at once?

Check your answers!

First, use your knowledge from linear transformations to write the 2×2 transformation matrix that streches a image in the x-direction by a factor of two. Store it as `stretch`.

[58]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

Now we can use matrix-matrix multiplications to get our new transformed image. Look how we have accomplished the same thing, but much simpler computation!

[59]:
```
bunny_squash = stretch @ bunny

ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_squash[0], bunny_squash[1], 'o-')
```

Try this!

Now we will try out some other transformations. Create a matrix operator to mirror points around the x-axis, meaning the y coordinate should be flipped and the x coordinate should remain unchanged.

Store your image as `bunny_mirror`.

Plot the transformed figure to make sure your transformation works correctly.

[61]:
```
# Apply the mirror transformation on bunny
bunny_mirror = ...



ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_mirror[0], bunny_mirror[1], 'o-')
```

We can also represent the rotation operation using matrices as well. Given a point (x, y) the rotated point around the origin is given by (you will later learn how to derive these equations):

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

Check your answers!

Using the equations above, create and plot a rotation 90 degrees counter-clockwise around the origin. **Hint: how can you turn the above into a matrix? Can you rewrite this as** $\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{A}\begin{bmatrix} x \\ y \end{bmatrix}$**?**

First store the rotation matrix in `rotation` (note: use radians with `np.pi`):

[63]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

Then store the rotated bunny in `bunny_rot`:

[64]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

What does the rotation matrix look like? Is the structure similar to the identity matrix and mirror transformation from above? Plot the rotated bunny below:

[65]:
```
ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_rot[0], bunny_rot[1], 'o-')
```

By multiplying these matrices together, we can combine multiple operations into one matrix:

$$\boldsymbol{T}_{\text{rot}}\boldsymbol{T}_{\text{scale}}\boldsymbol{x} = (\boldsymbol{T}_{\text{rot}}\boldsymbol{T}_{\text{scale}})\,\boldsymbol{x} = \boldsymbol{T}_{\text{both}}\boldsymbol{x}$$

Note that the first transformation we apply is first on the left multiplying our vector — matrix multiplication in general is not commutative so this distinction is important.

Check your answers!

Try creating a transformation matrix that first scales points by two in each direction, then rotates 90 degrees around the origin. Store the transformed bunny in `bunny_trans`.

[66]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

Now plot the transformed bunny:

[67]:
```
ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_trans[0], bunny_trans[1], 'o-')
```

Note that in this case it actually didn't matter if we rotated first then scaled, or scaled first then rotated. **But** this is not always the case; lets explore the commutativity of the rotation matrix from above and the matrix that stretches points along the x axis.

Try this!

Plot bunny transformed first by the stretch operator, then rotated 90 degrees CCW around the origin.

[69]:
```
# Define the bunny_rot_stretch by applying rotation and stretch to the bunny

ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_rot_stretch[0], bunny_rot_stretch[1], 'o-')
```

Try this!

Now plot the bunny with the transformations in the opposite order: first rotated then stretched.

[ ]:
```
# Define the bunny_stretch_rot by applying stretch and rotation to the bunny

ax = plt.gca()
ax.set_aspect('equal')
ax.plot(bunny_stretch_rot[0], bunny_stretch_rot[1], 'o-')
```

5 Transforming Vectors

Recall from last week that we can use a vector to store function values in one dimension. Here we will build on that, creating matrices that will transform these sets of y-values.

5.0.1 Shifting a sine wave

Here is a vector of length `20` containing a sine wave and its associated plot.

[70]:
```
sine = np.sin(np.linspace(0, np.pi * 4, 21))[:20] # chop off the ending, so we␣
 ↪don't get duplicate points!
plt.plot(sine, 'o-')
```

Check your answers!

One operation we can do on this is to "shift" each value to the left, transforming it into the plot of a cosine wave. We can achieve this by creating a 20×20 matrix to multiply the above vector, define this matrix below that shifts each point two values to the left, and plot it to verify your solution.

Hint: think of the identity matrix. For each point (row) it returns $1\times$the original point and $0\times$ everything else, so when multiplying any vector it gives back the original. What if it were to multiply $1\times$some other point?
Also, You may want to look at `np.roll`.

Store the transformation matrix in `shift` and the shifted sine array in `sine_shifted`.

[71]:
```
#grade (enter your code in this cell - DO NOT DELETE THIS LINE)
```

Now plot the shifted sine wave:

[72]:
```
plt.plot(sine_shifted, 'o-')
```

To help understand the shape and structure of your operator, print out the matrix. You can use Python's built-in `print()` function, and/or Matplotlib's `plt.imshow()` for a more visual depiction.

[74]:
```
print(shift)
plt.imshow(shift)
```

5.0.2 Extra example 1: Smoothing

(these are more challenging examples, but very interesting in case you have a chance to try them out!)

Here we have a "noisy" set of values that we would like to smooth.

[75]:
```
noisy = sine + np.random.random(20)
plt.plot(noisy, 'o-')
```

Create a matrix that when multiplying the above vector, will smooth each point by taking the average of it and its left and right neighbours. For points at the edges (`0` and `19`), you should only take the average of that point and the neighbour that actually exists. Plot your smoothened vector.

[76]:
```
smooth = np.identity(20)

for i in range(20):
    if i == 0:
        smooth[0,0] = 0.5
        smooth[0,1] = 0.5
    elif i == 19:
        smooth[19,19] = 0.5
        smooth[19,18] = 0.5
    else:
        smooth[i,i] = 1.0/3.0
        smooth[i,i-1] = 1.0/3.0
        smooth[i,i+1] = 1.0/3.0

smooth_vec = smooth @ noisy
plt.plot(smooth_vec, 'o-')
```

Again, print the matrix to understand the structure of your operator.

[77]:
```
print(smooth)
plt.imshow(smooth)
```

Random thought experiment: what happens if you apply the smoothing 100 times? 1000 times?

[78]:
```
smooth_vec = noisy
for i in range(10):
    smooth_vec = smooth @ smooth_vec

plt.plot(smooth_vec, 'o-')
```

5.0.3 Extra example 2: Edge Detection

A common problem that comes up in signal processing is to find when a signal changes from a high to a low value, or low to a high value. This is called *edge detection*, and to find these signal changes we can combine some of the above operators.

Here you are given a small step function with discernable "edges", so that we can easily determine what our output should be.

[79]:
```
square = np.zeros(20)
square[8:12] = 1
square[16:] = 1
plt.plot(square)
```

Try this!

Use the `smooth` operator defined in the previous section and apply it to the `square` function above. Store your result as `smoothened_version`.

Take a look at what happens to your `square` function:

[83]:
```
smoothened_version = ...



plt.figure()
plt.plot(square)
plt.plot(smoothened_version)
```

As you can see, the smoothened vector has dampened the extreme values somewhat. If we subtract the original array minus the smoothened one, we can actually isolate these extreme values and determine when they occur.

Try this!

Define the new array `diff` as $\boldsymbol{x} - \boldsymbol{x}_{\text{smooth}}$

[84]:
```
diff = ...



plt.plot(square)
plt.plot(diff)
```

Each of the “peaks” you see occur when the value of the wave changes. Lets combine this into one operator:

$$\boldsymbol{x} - \boldsymbol{T}_{\text{smooth}}\boldsymbol{x} = (\boldsymbol{I} - \boldsymbol{T}_{\text{smooth}})\,\boldsymbol{x}$$

Note that the $\boldsymbol{I}$ here refers to the identity matrix of same shape as $\boldsymbol{T}$.

Create and plot this transformation below, and confirm that it is identical to the plot above.

[85]:
```
edge = np.eye(20) - smooth

plt.plot(square)
plt.plot(edge @ square)
```

Again, print the matrix to understand the structure of your operator.

[86]:
```
print(edge)
plt.imshow(edge)
```

[ ]:
```

```