Week 1 Coding Lecture 3: Matrices

If you are familiar with linear algebra or physics, then you already know the importance of matrices. If you have not encountered matrices before, you can think of them as several row vectors stacked on top of each other or as several column vectors side by side. For instance, the following matrix can either be seen as four row vectors, each with three entries, or as three column vectors, each with four entries. We call it a 4×3 matrix.

$$A = \begin{pmatrix} -1 & 2 & 1 \\ 3 & 0 & -1 \\ 4 & -2 & 2 \\ -2 & 1 & 3 \end{pmatrix}.$$

In general, an $m \times n$ matrix has m rows and n columns.

Constructing matrices

In python, matrices are represented as 2D arrays. We have already seen some simple examples of matrices - row and column vectors are actually special cases of matrices. A row vector with n entries is a $1 \times n$ matrix and a column vector with n entries is an $n \times 1$ matrix.

We know how to make row and column vectors in python by first making a 1D array and then using the reshape command. In principle, you could use this method to make all matrices, but it quickly becomes tedious. Instead, there is a more direct way to construct a 2D array. As an example, let's construct the matrix A.

```
import numpy as np

A = np.array([[-1, 2, 1], [3, 0, -1], [4, -2, 2], [-2, 1, 3]])
print(A)
```

```
    [[-1  2  1]
     [ 3  0 -1]
```

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

The syntax is very similar to that for 1D arrays, but with some extra brackets. One set of brackets encloses the entire matrix, and then each row is enclosed in another set of brackets. You then use commas to separate entries within a row and to separate different rows.

As practice, let's also make a 3×4 matrix B and two 3×3 matrices C and D.

```
B = np.array([[1, -2, 0, 3], [2, 1, -4, 1], [-3, 0, 1, 1]])
print(B)
```

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

```
C = np.array([[1, 0, 2], [3, -1, 1], [2, 2, 0]])
print(C)
```

```
[[ 1  0  2]
 [ 3 -1  1]
 [ 2  2  0]]
```

```
D = np.array([[1, 2, -1], [3, -2, 2], [-1, 1, 0]])
print(D)
```

```
[[ 1  2 -1]
 [ 3 -2  2]
 [-1  1  0]]
```

Notice that these arrays still have the same type as the vectors and arrays we discussed last time, but they have a different number of dimensions and shape.

```
print("The matrix A has the following type:")
print(type(A))
print("And the following number of dimensions:")
print(A.ndim)
print("And the following shape:")
print(A.shape)
```

```
    The matrix A has the following type:
    <class 'numpy.ndarray'>
    And the following number of dimensions:
    2
    And the following shape:
    (4, 3)
```

It's also worth noting that this gives us a much easier way to construct row and column vectors. For example, if we wanted to make the vectors

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

and

$$\mathbf{y} = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix},$$

we could use the code

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

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

```
y = np.array([[4, 5, 6]])
print(y)
```

```
    [[4 5 6]]
```

This is often much easier than using the reshape command.

Matrix arithmetic

Since matrices have the same type (ndarray) as the arrays we looked at last time, it should not surprise you that the basic arithmetic operations work the same with matrices as they do with vectors. In particular, you can add or subtract the same number from each entry of a matrix and multiply or divide every entry in a matrix by the same number, or even raise every entry of a matrix to the same power.

```
print(A + 5)
```

```
[[4 7 6]
 [8 5 4]
 [9 3 7]
 [3 6 8]]
```

```
print(B - 2)
```

```
[[-1 -4 -2  1]
 [ 0 -1 -6 -1]
 [-5 -2 -1 -1]]
```

```
print(C * 3)
```

```
[[ 3  0  6]
 [ 9 -3  3]
 [ 6  6  0]]
```

```
print(D / 4)
```

```
[[ 0.25  0.5  -0.25]
 [ 0.75 -0.5   0.5 ]
 [-0.25  0.25  0.  ]]
```

```
print(A ** 2)

    [[ 1  4  1]
     [ 9  0  1]
     [16  4  4]
     [ 4  1  9]]
```

You can also combine matrices with elementwise operations. For instance, we could add all the corresponding elements of C and D. (That is, add the top left entry of C to the top left entry of D, then add the top middle entry of C to the top middle entry of D, etc.)

```
print(C + D)

    [[ 2  2  1]
     [ 6 -3  3]
     [ 1  3  0]]
```

The same thing works for subtraction, multiplication, division and exponentiation. For example,

```
print(C - D)

    [[ 0 -2  3]
     [ 0  1 -1]
     [ 3  1  0]]
```

```
print(C * D)

    [[ 1  0 -2]
     [ 9  2  2]
     [-2  2  0]]
```

Side note about types: I did not show `C / D` because some one of the entries of `D` is zero and division by zero is not allowed. You should try it on your own and see what error message you get. I also did not show `C ** D`, but this is for more technical reasons. The problem is that we only used whole numbers when constructing our matrices and so python assumed that we wanted elements of type `int` (more technically, type `int64`, but don't worry about the difference now). Unfortunately, python does not allow raising integers to negative powers. This is another example of how the type of a variable determines which operations you can perform on it. There are a few ways to fix this. The simplest is to construct the matrix using decimals in the first place so that python knows the elements are supposed to be floats.

```
C_float = np.array([[1.0, 0, 2], [3, -1, 1], [2, 2, 0]])
print(C_float ** D)
```

```
[[ 1.    0.    0.5]
 [27.    1.    1. ]
 [ 0.5   2.    1. ]]
```

Notice that I replaced the first entry "1" with "1.0". If any of the entries have a decimal point, then python will make every entry a float (more technically, a float64 (or what we used to call a double back in the day), but don't worry about the difference now). I really don't understand why Python chooses to call it float64 instead of just double.

A better way is to use one of the pre-defined properties of ndarrays, the `astype` function. You can use the code

```
C_float = C.astype("float64")
print(C ** D)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-79-3d91ceeb0af9> in <module>
      1 C_float = C.astype("float64")
----> 2 print(C ** D)

ValueError: Integers to negative integer powers are not allowed.
```

SEARCH STACK OVERFLOW

This won't usually be a problem for us, because most of the arrays we create won't just have whole numbers in them, and so they will automatically use floats already. However, it is a good thing to keep in mind. If you aren't sure what type the elements of your array are, you can use the `dtype` property to check. For instance,

```
print("The elements of A have type: ")
print(A.dtype)
print("The elements of C_float have type:")
print(C_float.dtype)
```

```
The elements of A have type: 
int64
The elements of C_float have type:
float64
```

End of side note

Just like with 1D arrays, the shapes of your arrays must match. It doesn't make sense to perform elementwise operations with two matrices of different dimensions. For example, we cannot add A and B, because there is no way to match up elements of A with corresponding elements of B.

```
print(A + B)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-81-9630b0879df7> in <module>
----> 1 print(A + B)

ValueError: operands could not be broadcast together with shapes (4,3) (3,4)
```

SEARCH STACK OVERFLOW

The same is true for all of the elementwise operations: addition, subtraction, multiplication, division and exponentiation.

Accessing and Modifying Matrices

Accessing and modifying entries of a matrix works much like it did with 1D arrays. Recall that you can specify an entry of a vector by giving an integer index, so `x[2]` means "the third entry (index 2) of x". You can use similar notation with 2D arrays, but the output might look slightly unusual at first. For example, if A were a 1D array, then the code `A[2]` would give you the third entry (index 2) of the array. If we try the same thing with a matrix, we get

```
print(A[2])
```

```
    [ 4 -2  2]
```

This is the third *row*, not the third *entry*, of A. This works for any index (up to the number of rows in the matrix):

```
print(A[0])
```

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

```
print(A[1])
```

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

```
print(A[2])
```

```
    [ 4 -2  2]
```

```
print(A[3])
```

```
    [-2  1  3]
```

You can also use negative index syntax. (Remember, negative indices count backwards from the end of an array, so if A were a 1D array then `A[-1]` would be the last entry. Since A is a 2D array, this gives the last row instead.

```
print(A[-1])
```

```
    [-2  1  3]
```

If you want to access an individual element of a matrix instead of an entire row, you have to use one more set of brackets. For example, the second entry (index 1) in the third row (index 2) can be found with the code

```
print(A[2, 1])
```

```
    -2
```

The general syntax is `name_of_array[row_index, column_index]`. It is important to remember the order: row index and then column index. This mirrors the mathematical syntax. In mathematics, the entry at row i, column j of the matrix A is usually denoted by a_{ij}. The second entry in the third row (i.e., row 3, column 2) would therefore be written a_{32}. Notice that python starts counting indices at 0 and not 1, so the indices in the python code are one smaller than the row/column numbers in mathematical notation.

You can also use slice syntax to access more than one element at a time. For instance,

```
print(A[1:3, 0:3])
```

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

pulls out the second and third rows (because `1:3` means index 1 and 2, but **not** index 3) and the first through third columns (because `0:3` means index 0, index 1 and index 2, but **not** index 3).

The shortcuts for slicing that we talked about in the last lecture apply here as well. For example, if you skip the `start` index of a slice, then python starts from the beginning of the array, so

```
print(A[:3, 0:3])
```

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

prints the first through third rows and the first through third columns. Likewise, if you skip the `stop` index of a slice, then python goes all the way to the end of the array, so

```
print(A[1:3, 1:])
```

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

prints the second through third row (`1:3` means to include index 1 and index 2, but not index 3) and the second through last column (`1:` means to start at index 1 and go to the last index).

If you skip both the `start` and the `stop` indices (so you just have a `:`), then python uses *every* index. For instance,

```
print(A[:, 0:2])
```

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

prints *every* row of the first and second columns (because `:` means every index and `0:2` means index 0 and index 1, but not index 2).

It is also possible to mix slices with single indices. For example, if you wanted the entire second row of `A`, you could use the code

```
print(A[1, :])
```

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

Likewise, if you wanted the entire third column of `A`, you could use

```
print(A[:, 2])
```

```
[ 1 -1  2  3]
```

Unfortunately, python throws away some information about the shape of your matrix when you do this. As you can see, the previous two answers are both 1D arrays, not rows or columns. If the shape is important to you (and it almost always will be in this class), then you have to manually reshape these arrays afterwards. For example,

```
print(np.reshape(A[1, :], (1, -1)))
```

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

```
print(np.reshape(A[:, 2], (-1, 1)))
```

```
[[ 1]
 [-1]
 [ 2]
 [ 3]]
```

A useful trick to remember is that you can replace a single number with a slice to avoid this. For example, the slice `2:3` is exactly the same as the index 2 (because `2:3` means to start at index 2 and go up to just before index 3). Likewise, the slice `1:2` is the same as just the index 1. You could therefore get the second row of `A` or the third column of `A` with the code

```
print(A[1:2, :])
```

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

```
print(A[:, 2:3])
```

```
[[ 1]
 [-1]
 [ 2]
 [ 3]]
```

This is a little ugly, but saves us from having to reshape arrays over and over again.

Matrix Multiplication

In mathematics (particularly linear algebra), we very rarely use elementwise multiplication for matrices. Instead, another definition of multiplication is much more common. (This new version is so common that it doesn't have a special name; it is just "matrix multiplication".)

We will not really need matrix multiplication until week 3 in this class, but it will become very important later on. We are introducing it right now because it is essentially impossible to avoid in MATLAB, and so half of the class will need to know these definitions right away. If you are focusing on python, you should still read this section.

Dot Product

The definition of matrix multiplication is fairly messy. Before we jump into the full definition, let's take a moment to review the concept of the *dot product*. The dot product of two vectors of the same length is the sum of the products of their corresponding entries. For example, if

$$\mathbf{x} = \begin{pmatrix} 2 \\ -3 \\ 1 \\ 4 \end{pmatrix} \text{ and } \mathbf{y} = \begin{pmatrix} -1 \\ -2 \\ 1 \\ 3 \end{pmatrix}$$

then the dot product of $\mathbf{x}$ and $\mathbf{y}$ (which we denote $\mathbf{x} \cdot \mathbf{y}$ is

$$\mathbf{x} \cdot \mathbf{y} = (2)(-1) + (-3)(-2) + (1)(1) + (4)(3) = 17.$$

We can calculate this in python with

```
x = np.array([[2], [-3], [1], [4]])
y = np.array([[-1], [-2], [1], [3]])
x_dot_y = x[0, 0] * y[0, 0] + x[1, 0] * y[1, 0] + x[2, 0] * y[2, 0] + x[3, 0] * y[3, 0]
print("The dot product of x and y is: ")
print(x_dot_y)
```

```
The dot product of x and y is:
17
```

In general, if $\mathbf{x}$ and $\mathbf{y}$ are vectors of the same length n, then we define

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

Note that this definition only makes sense when $\mathbf{x}$ and $\mathbf{y}$ are the same length.

Multiplication

We will define matrix multiplication as follows: If A is an $m \times n$ matrix and B is an $n \times k$ matrix, then the product AB is an $m \times k$ matrix where the entry in the ith row and jth column is the dot product of the ith row of A with the jth column of B.

It is important to notice that this definition only works when the number of columns of A is the same as the number of rows of B, because that way we will only take dot products of vectors of the same length.

This definition may seem overly complicated, but it turns out to be very useful. We will go into much more detail about why when we start solving linear systems. For now, you just need to know how to perform this multiplication in python. As an example, let's multiply the following two matrices:

```
A = np.array([[-1, 2, 1], [3, 0, 1], [4, -2, 2], [-2, 1, 3]])
B = np.array([[1, -2], [2, 1], [-3, 0]])
```

A and B are 4×3 and 3×2 matrices, respectively. Since the number of columns of A is the same as the number of rows of B, we are allowed to multiply A times B. In python, we can do this with the `@` operator:

```
print(A @ B)
```

```
[[  0   4]
 [  0  -6]
 [ -6 -10]
 [ -9   5]]
```

As expected from our definition, the result is a 4×3 matrix (the same number of rows as A and the same number of columns as B).

However, since the number of columns of B is not the same as the number of rows of A, we cannot multiply B times A. If we try it, python will give the following error:

```
print(B @ A)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-102-caf68677c2af> in <module>
----> 1 print(B @ A)

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),
(k,m?)->(n?,m?) (size 4 is different from 2)
```

SEARCH STACK OVERFLOW

When you see the error "mismatch in its core dimension", that tells you that you are trying to multiply matrices when the numbers of rows and columns don't match.

In particular, notice that we can't multiply a row vector times a row vector, because it doesn't make sense to multiply an $n \times 1$ by an $m \times 1$ (the 1 and the m don't match). We also can't multiply a column vector times a column vector, because it doesn't make sense to multiply a $1 \times n$ by a $1 \times m$ (the n and the 1 don't match).

From now on, whenever we talk about matrix multiplication in this class (including on all of the homework assignments), we will mean this definition. That is, if you see an expression like AB or $A\mathbf{x}$, you should use the `@` operator in python.