An Exploration into Linear Algebra

Yash Money, Imran ”Vince McMath” Iftikar

Idk man, I just work here

1 Basic Matrix Code

To begin, we have will create code that represents matrices and performs elementary matrix operations, such as computing the inverse, multiplying, and adding.

We will utilise python to do so. Python has a data structure known as a ”list” or an ”array.” These are essentially a collection of indexed data that can be manipulated. Lists may contain sublists; it is in this way that we can represent a ”matrix” in python, for indeed, a matrix is nothing but a collection of row/column vectors, which themselves can be represented as an individual python list.

We have decided to represent a matrix as a collection of row vectors.

For example consider the following matrix, $A \in \mathbb{R}^{m\times n}$:

$$A = \begin{bmatrix} a_1 & a_2 & a_3 & ... & a_n \\ b_1 & b_2 & b_3 & ... & b_n \\ ... & ... & ... & ... & ... \\ m_1 & m_2 & m_3 & ... & m_n \end{bmatrix}$$

with m rows and n columns. We choose to represent the same matrix, A pythonically in the following way:

```
array = [
    [a1, a2, a3, ..., aN],
    [b1, b2, b3, ..., bN],
    [c1, c2, c3, ..., cN],
    ...,
    [m, m2, m3, ..., mN],
]
```

In this way, each sublist is a row of the matrix.

In the following we, will discuss how to code basic matrix operations using this data structure.

1.1 Multiplying a Matrix by a Scalar

One of the most crucial elements of matrix arithmetic is the ability to multiply a given matrix by a scalar. When doing this, the following will hold true:

$$\forall \lambda \in \mathbb{R}, \forall A \in \mathbb{R}^{m \times n}, \lambda A = \begin{bmatrix} \lambda a_1 & \lambda a_2 & \lambda a_3 & ... & \lambda a_n \\ \lambda b_1 & \lambda b_2 & \lambda b_3 & ... & \lambda b_n \\ ... & ... & ... & ... & ... \\ \lambda m_1 & \lambda m_2 & \lambda m_3 & ... & \lambda m_n \end{bmatrix},$$

This is a rather simple problem to tackle. Simply, we iterate through each row of the matrix, and then through each element (column) within that row, multiplying each entry by a given λ as we iterate. This produces a matrix that has been multiplied by lambda. The code is as follows:

```
def matrix_by_scalar(matrix1, scalar_quantity):

    '''
    TAKES:
        A matrix of the form outlined above, matrix1
        A scalar_quantity by which the matrix will be multiplied
    RETURNS:
        A matrix of the form outlined above
    '''
    try:
        if (isinstance(scalar_quantity, int)) or
            (isinstance(scalar_quantity, float)):
        # O(1) we wcheck if the scalar element is a valid real
            number; if not, we raise an error.

            return list([element * scalar_quantity for element in
                row] \
                for row in matrix1) # O(n**2) This comprehension
                    multiplies each element of each row by the
                    scalar and thus has a time complexity of n**2
        else: raise ValueError(f"Argument passed:
            '{scalar_quantity}'. Error: Expected argument of type
            'int' or 'float' ") # error raised
    except: # In Case something goes wrong; biggest error here is
        the potential for incorrect args passed
        print("something went wrong - likely the matrix argument
            was incorrect")
```

1.2 Adding Two Matrices

The addition of matrices is an elementwise one, and therefore it is easy to implemenent iteratively and makes lots of intuiative sense. What is meant by elementwise is that we can handle each element individually; in the case of matrix addition, we add each element of a given matrix to the corresponding element of a separate matrix. We can assert that the matrices must therefore be the of the same dimension to be added.

Essentially, we have the following for $\forall A, B \in \mathbb{R}^{m \times n}$, where $i = 1, ..., m$ and $j = 1, ..., n$

$$\begin{aligned} A + B &= \begin{bmatrix} A_{1,1} & A_{1,2} & ... & A_{1,j} \\ A_{2,1} & A_{2,2} & ... & A_{2,j} \\ ... & ... & ... & ... \\ A_{i,1} & A_{i,2} & ... & A_{i,j} \end{bmatrix} + \begin{bmatrix} B_{1,1} & B_{1,2} & ... & B_{1,j} \\ B_{2,1} & B_{2,2} & ... & B_{2,j} \\ ... & ... & ... & ... \\ B_{i,1} & B_{i,2} & ... & B_{i,j} \end{bmatrix} \\ &= \begin{bmatrix} A_{1,1} + B_{1,1} & A_{1,2} + B_{1,2} & ... & A_{1,j} + B_{1,j} \\ A_{2,1} + B_{2,1} & A_{2,2} + B_{2,2} & ... & A_{2,j} + B_{2,j} \\ ... & ... & ... & ... \\ A_{i,1} + B_{i,1} & A_{i,2} + B_{i,2} & ... & A_{i,j} + B_{i,j} \end{bmatrix} \end{aligned}$$

The code looks as follows:

```
def add_matrices(mat1, mat2):

    '''
    TAKES:
        two matrices of the standard form outlined previously, mat1
            and mat2
            They must be of the same dimensions
    RETURNS:
        A single matrix of the same dimensions as mat1 and mat2,
            where this matrix is the sum of mat1 and mat2
    '''
```

```
if (len(mat1) == (len(mat2)) and (len(mat1[0]) ==
    len(mat2[0]))): # check to see if possible to add the two
    lists, as their dimensions must be the same

    '''
    The following is just a long list comprehension that
        iterates through each matrix and adds corresponding
        elements. It then appends these to "l", which is what is
        ultimately what is returned
    '''
    # basically just two for loops
    l = list([mat1[row][col]+mat2[row][col] \
        for col in range(len(mat1[0]))] \
        for row in range(len(mat1)))
    return l
    # Intuitively subtraction is very similar: the "+" must be
        turned to a "-"

else: # if the matrices are not of the same size.
    raise ValueError("Args are different size and thus cannot
        be added")
```

This code will hold for subtraction as well, because of the following identity:

$$\begin{aligned} A - B &= \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,j} \\ A_{2,1} & A_{2,2} & \dots & A_{2,j} \\ \dots & \dots & \dots & \dots \\ A_{i,1} & A_{i,2} & \dots & A_{i,j} \end{bmatrix} - \begin{bmatrix} B_{1,1} & B_{1,2} & \dots & B_{1,j} \\ B_{2,1} & B_{2,2} & \dots & B_{2,j} \\ \dots & \dots & \dots & \dots \\ B_{i,1} & B_{i,2} & \dots & B_{i,j} \end{bmatrix} \\ &= \begin{bmatrix} A_{1,1} - B_{1,1} & A_{1,2} - B_{1,2} & \dots & A_{1,j} - B_{1,j} \\ A_{2,1} - B_{2,1} & A_{2,2} - B_{2,2} & \dots & A_{2,j} - B_{2,j} \\ \dots & \dots & \dots & \dots \\ A_{i,1} - B_{i,1} & A_{i,2} - B_{i,2} & \dots & A_{i,j} - B_{i,j} \end{bmatrix} \end{aligned}$$

where $\forall A, B \in \mathbb{R}^{m \times n}$, and $i = 1, ..., m$, $j = 1, ..., n$.

This implies that we must merely change the elementwise addition in our code to subtraction in order to create a function that subracts two matrices.

1.3 Transpose of a Matrix

The tranpose of a matrix is a modification to its structure such that rows become columns and vice versa. That is, the first row of a given matrix A will be the first column of it's transpose, A^T, the second row of A will be the second column of A^T, and so forth and so on.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & ... & A_{1,j} \\ A_{2,1} & A_{2,2} & ... & A_{2,j} \\ ... & ... & ... & ... \\ A_{i,1} & A_{i,2} & ... & A_{i,j} \end{bmatrix} \in \mathbb{R}^{m \times n}, i = 1, ..., \text{ and } j = 1, ..., n$$

Then,

$$A^T = \begin{bmatrix} A_{1,1} & A_{2,1} & ... & A_{i,1} \\ A_{1,2} & A_{2,2} & ... & A_{i,2} \\ ... & ... & ... & ... \\ A_{1,j} & A_{2,j} & ... & A_{i,j} \end{bmatrix} \in \mathbb{R}^{m \times n}, i = 1, ..., \text{ and } j = 1, ..., n$$

In the case of square matrices, we can state the matrix has been "flipped" over it's main diagonal, or the diagonal elements running from top-left to bottom-right.
The transpose, as with matrix addition/subtraction and multiplying by a scalar, is rather trivial to implement. We iterate through the columns of a matrix, and return them to the user as rows of a new matrix, which is the tranpose. In regards to greabbing the columns of a matrix: python provides no way to index a column as simply as one can index a row. This is because, as we have represented matrices with row vectors, the columns are comprised of elements from mutliple different rows. Therefore, we construct a function that enables us to quickly "grab" a column from a matrix given its index.

```python
def get_col(matrix_2d, _index):

    return list(row[_index] for row in matrix_2d) # O(n) this
        simply grabs the column from the specified index.

    # full time O(n)

def transpose(matrix):

    new_array = [get_col(matrix, i) for i in range(len(matrix[0]))]
        # O(n) and nested O(n), becomes O(n**2). takes a column and
        makes it a row
    return new_array
```

1.4 Row Reduction: Echelon/Upper Triangle Form

1.5 Row Reduction: Row Echelon Form (REF)

1.6 Row Reduction: Reduced Row Echelon Form (RREF)

1.7 Finding the Determinant

1.8 Inverting a Matrix

1.9 Multiplying Two Matrices

2 Vector(Sub)Spaces Pythonically

2.1 General Introduction to Vector Subspaces

A vector space is special type of a group that contains both an inner and outer operation, which are addition and multiplication by a scalar, respectively.

2.2 Issues when Representing Subspaces Pythonically

2.3 Uses Basis Vectors

3 Representing Linear Mappings Pythonically

4 More Advanced Matrix Operations

5 Projections

5.1 Solving Homogeneous Systems

5.2 Finding Eigenvalues and Eigenvectors Of A Matrix

5.3 Finding the Eiegendecomposition

5.4 Finding the Singular Value Decomposition