An Exploration into Linear Algebra

Yash Money, Vince McMath

It's Too Early for This

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:

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

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.

```
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

One of the most important matrix-manipulations is the ability to perform elementary "row operations" upon a matrix. That is, multiplying a row by a scalar, swapping the position of two rows in a matrix, and adding/subtracting a scalar multiple of one row to/from another. These operations have incredibly useful properties, and are widely used to calculate determinants and inverses of matrices, and additionally are used to solved systems of linear equations using matrices.

These row operations allow one to effectively mutate a matrix into one of many various forms; for example, we can row reduce such that what we are left with is of the "upper triangle" form, in which all elements not above or in the main diagonal are 0.

This form is known as "echelon" form, and the process to find it is called "Gaussian Elimination."

We begin the algorithm by swapping rows that need to be swapped.

```
def echelon(matrix):

    # we will iterate through the columns
    for col_index in range(len(matrix[0])):

        col = get_col(matrix, col_index) #we grab the column using
            the index from the above for loop

        '''
        the following bit of code looks for places where there
            might be zeroes in the diagonal.
        if there are, and we do not handle for it, we will get a
            divinding by zero error
        We then swap the rows such that we can proceed

        thus, the following code is quite necessary.
        '''

        if col_index <= len(matrix): # O(1) we only need to look
            for zeroes in the first square - that is, if the matrix
            is longer than tall, it is uncessary to check all columns
```

```
        if all((i == 0) for i in col[col_index:]): #O(n) if the
            entire column is filled with zeroes, we call
            continue and the program returns to the initial for
            loop, and goes to the next column
            continue

        elif col[col_index] == 0: # O(1) if one of the elements
            on the diagonal is zero - this is where the dividing
            by zero error occurs so we need to handle this
            '''
            here we iterate through all of the rows below the
                diagonal.
            if we find a row that doesn't contain a zero in the
                diagonal column index,
            we will swap them
            '''
            for i in range(len(col[col_index:])): #O(n)
                if col[col_index:][i] != 0: # O(1)
                    row_idx = col_index+i # O(1)
                    break

            # the below line of code simple swaps the rows
            matrix[col_index], matrix[row_idx] =
                matrix[row_idx], matrix[col_index] # O(1)
```

Now that rows have been sufficiently swapped, we may begin subtracting scalar multiples of rows from rows below it. This enables us to achieve a zero in the desired spot. We will continue in the for loop that we currenly operate in.

```
    '''
        the following for loop is where the actual formula happens
        the algorithm works as follows:

        assume we have an array,
        we have the nth column and we want to make all elements of
            that column below
        the nth row into 0

        say n = 1 then we have:

        c1 = [          desired_c1 = [
             3                          3
```

```
      4                       4
      3                       0
      7                       0
      2                       0
        ]                       ]

we can achieve this via subtracting a scalar multiple of
    the row such that we get 0
ex: row 2. we can achieve row 1, col 2 equaling 0 by
subtracting row 1 *
    matrix[row1_idx][col2_idx]/matrix[row1_idx][row_idx]

'''
for row_index in range(len(col)): # O(n) we iterate through
    row of each colum we have grabbed earlier
    '''
    remember, we only want to turn the rows below
    the diagonal into 0. thus, we check if the row is indeed
        one we one to turn into 0
    if it is not, its idx will be less than the column idx
    if that proves to be true, we will simple pass
    '''

    if row_index <= col_index: #O(1)n checks if the row is
        one we do not want to turn to 0
        '''
        the following if statement is unnecessary, as I
            could have explicity called:
            matrix[col_index][col_index] when I called
                denominator later
            matrix[col_index] when I call
                raw_subtractant_row later
            however, will keep this code for readability, as
                I find this easier to understand.
        '''
        if row_index == col_index: #O(1)
            denominator = matrix[row_index][col_index] #O(1)
            raw_subtractant_row = matrix[row_index] #O(1)
        pass

    else:
        '''
        here we actually do the conversion to 0
```

```
            this finds teh numerator of the scalar we will
                multiply the subtractant row by
            then we will simply create the final subtractant row
            then we simply subtract the two rows, resulting in a
                0
            we then replace the old row with the new one.
            '''
            row_to_sub_from = matrix[row_index] # O(1)
            numerator = matrix[row_index][col_index] #O(1)

            subtractant = row_by_scalar(raw_subtractant_row,
                (numerator/denominator)) # O(n)
            subbed_row = subtract_row(row_to_sub_from,
                subtractant) # O(1)

            matrix[row_index] = subbed_row # O(1)

    return matrix
```

1.5 Row Reduction: Row Echelon Form (REF)

Consider a matrix that has been reduced such that it is in echelon form, where all elements below the main diagonal are 0. It is possible to reduce a matrix further. Specifically, by identifying all non-zero diagonal elements and dividing the rows in which they appear by the element itself, we can get to a form similar to echelon form, but one where all non-zero diagonal elements will be 1. This is "row-echelon" form.

In other words, we can obtain the following, given $A \in \mathbb{R}^{m \times n}$ and it is in echelon form:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & ... & A_{1,j} \\ 0 & A_{2,2} & ... & A_{2,j} \\ ... & ... & ... & ... \\ 0 & 0 & ... & A_{i,j} \end{bmatrix}$$

$$\rightsquigarrow \begin{bmatrix} 1 & A_{1,2} & ... & A_{1,j} \\ 0 & 1 & ... & A_{2,j} \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix}$$

It it is easy to turn a matrix in echelon form into one in row-echelon, as we must simply divide

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.

Early on, we realized that it was impossible to represent an infinite series of elements as a vector space in python, for example $\mathbb{R}^2$. We realized that we would have to have the user to insert a matrix representing that vector space. We created an umbrella class called "subspace". These are integral to representing linear mappings pythonically as these serve as our mapping

Our subspace class is very broad, the operations done in the class are more of a general operations as stated below

2.2 Issues when Representing Subspaces Pythonically

As I said earlier, we cannot express common vector spaces used in linear algebra and we had to settle for finite spaces. They would be in the form of:

2.3 Defining a Space with a Basis

The definition of a basis is the set of linearly independent vectors that span a space. These have many useful applications when it comes to linear mapping.

$$U = span(\begin{bmatrix}0\\1\\2\end{bmatrix}, \begin{bmatrix}1\\1\\2\end{bmatrix}, \begin{bmatrix}2\\2\\4\end{bmatrix})$$

Basis Vector

$$\begin{bmatrix}0\\1\\2\end{bmatrix}$$

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