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:

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

It it is easy to turn a matrix in echelon form into one in row-echelon, as we must simply divide by the reciprocal of the diagonal element.

1.6 Row-Reduced Echelon Form

This is an extension of the REF form, where the intended result is 1 along the diagonal and zeros above and below the diagonal with constants on the rightmost column. We do this by performing elementary row operations on a matrix from the row below to the row above to eliminate the constant terms with the exception of the diagonal. This method of elementary row operations is called Gaussian Elimination.

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

```
def rref(matrix):

# takes matrix of form outlined in flowerbox
# returns matrix of form outlined in flowerbox, but in rref form

# builds off of the ref function, which produces an upper
   triangle such that the leading elements of each row is 1
# rref returns a matrix such that all diagonals are 1 (which is
   true in ref) but also as many other elements as possible
   are 0

matrix = ref(matrix) # we apply ref to the matrix to make sure
   it is in the appropriate form
mindim = min(len(matrix), len(matrix[0])) # the minimum
   dimension of the matrix

for idx in reversed(range(mindim)):
    # for above: we wantt to work up from the bottom of the
       matrix, and will only use values that could have a pivot
    '''in other words, if
    a = [
    [1,3,8],
    [12,3,4],
    [16,7,1],
    [2,1,6],
        [12,4,5],
        [7,2,5]
        ]
        then only the top 3 rows will have good pivots

        O(n)
        '''
        if matrix[idx][idx] != 0: # we know that where pivots
            should be will only be 0s or 1s becuase of calling ref
                                  # if it is not a one and is a 0,
                                      then we will not evaluate and
                                      will just skip in

            for idx2 in reversed(range(idx)): # we then iterate
                through all above rows O(n)

                # here we find what we shoudl multiply matrix[idx]
                    by to subtract it from matrix[idx2]
                # and then we simply run through that subtraction

                scalar = matrix[idx2][idx] # O(1)
                subtractant = row_by_scalar(matrix[idx], scalar)
                    #O(n)
                row_to_sub_from = matrix[idx2] #(1)
                subbed_row = subtract_row(row_to_sub_from,
                    subtractant) # O(n)

                matrix[idx2] = subbed_row # O(1)

    return matrix

    # Full time O(n**3)
```

1.7 Finding Pivot Columns

As we discussed earlier in the RREF (Reduced Row Echelon Form) section, our result from that operation is a matrix taken from upper triangle form to one that has ones along its diagonal. In some cases, the matrix has linearly dependent columns, in that case, we want to find the index of those columns for all types of matrices, no matter the size. We iterate through each row to find the first appearance of a 1 and check all prior elements to make sure they are 0's to account for non-square matrices.

```
def identify_pivots(mat, rounded=True):

# takes matrix of form outlined in flowerbox
# returns a list of all indexes in which there is a pivot
    column upon row reducing the input matrix

# identifies the pivot columns of the matrix by row reducing

matrix = mat.copy() # we make a copy because otherwise the
    original matrix is modified and we get fucked

matrix = ref(matrix) # we run ref to row reduce it. we do not
    care about full row reduction because that it is trivial in
    this case

if rounded: # this combats error where 0.99999999999 is not
    read as one, and therefore not read as a pivot
    matrix = mround(matrix, 12)

pivot_col_idx_list = [] # we initialize a blank list will
    contain indexes of pivot cols

for row in matrix: # we iterate through each row
    try: # we add a try in case there are no ones in the row
        first_one = row.index(1) # we find the first appearance
            of a 1 within the list
        for idx2 in range(first_one): # we then check if all
            prior elelements are 0
            if row[idx2] != 0: # if one is not a 0, we contiknue
                the loops
                continue

        # however, if all elements prior to the first 1 are 0,
            we will append the index of that 1 to the pivcollist
        pivot_col_idx_list.append(first_one)

    except: # if there are no ones, it can be infered that
        there is no pivot column in that row, so we continue
        continue

return pivot_col_idx_list
```

```
    # O(n**2) time
```

1.8 Finding the Determinant

The determinant is an extremely important scalar quality that provides information about a given matrix. It essentially gives the signed area of the parallelepiped that is represented by the columns or rows of a matrix. If the determinant is 0, it tells us that the area is 0 and therfore the matrix is not of full rank.

There are many methods to find the determinant. First, we can use Laplace Expansion to recursively break down a matrix into small dimensions and calculate the determinants of such smaller matrices.

However, Laplace expansion is very slow; it is much faster to use an alternative method to solve for the determinant. We can leverage two facts to land on a faster method. First, if we subtract a multiple of a row from another row, we do not change the value of the determinant. Second, the determinant of a matrix in triangle-form is the product of its diagonal elements.

These two understandings let us know that by converting a matrix to Echelon form, we can easily find it's determinant by multiplying the diagonal elements.

```
def matrix_det(matrix):

    matrix = echelon(matrix)

    det = 1
    for idx in len(matrix):
        det = det * matrix[idx][idx] # multiply all of the diagonal
            elements

    return det
```

1.9 Inverting a Matrix

Another incredibly important computation is the ability to find the inverse of a given matrix. The inverse is a matrix that, when multiplied by the original, will produce the identity matrix. In this way, it is useful because we can somewhat simulate division with it.

The easiest way to find an inverse is to augment the original matrix with an identity matrix such that the original is on the left and the identity is on the right. The, we will row reduce the matrix such that the left hand side is in rref form, or in other words is an identity matrix. What is left on the right hand side is the inverse.

It should be noted that only square matrices have inverses.

```
def inverse(matrix):


    identity = make_identity(len(matrix))
    dim_to_take = len(matrix[0])

    matrix = append_mat_right(matrix, identity)
    matrix = rref(matrix)

    inverse = list()

    for row in matrix:
        inverse.append(row[-dim_to_take:])

    return inverse
```

1.10 Multiplying Two Matrices

Multiplying two matrices is an elementary operation and the method to computing it is by multiplying the 1st row vector of the first Matrix by the 1st column vector of the second matrix. Therefore, if the dimensionality of the column of the first matrix does not match the row of the second matrix, they cannot be multiplied. The resulting dimensionality is the row of the first matrix by the column of the second matrix.

$$\begin{aligned} A &= \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,j} \\ A_{2,1} & A_{2,2} & \dots & A_{2,j} \\ \dots & \dots & \dots & \dots \\ A_{m,1} & A_{m,2} & \dots & A_{m,j} \end{bmatrix} \\ &* \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,m} \\ A_{2,1} & A_{2,2} & \dots & A_{2,m} \\ \dots & \dots & \dots & \dots \\ A_{j,1} & A_{j,2} & \dots & A_{j,m} \end{bmatrix} \end{aligned}$$

```python
def multiply_matrix(matrix1, matrix2):

# the first one goes second

'''this function multiplies two matrices together
it takes each row and multiplies it by the column in the next
    row'''

# matrix1 row and column number. len() is O(1), constant time
matrix2_rows = len(matrix2)
matrix2_cols = len(matrix2[0]) # just taking the first. I
    assume that each argument is a completely filled list

# matrix2 row and column number. still constant time
matrix1_rows = len(matrix1)
matrix1_cols = len(matrix1[0])

# makes sure that we can indeed mutliply these two matrices -
    that is, their dimensions must be compatible
if (matrix2_rows == matrix1_cols) or (matrix2_cols ==
    matrix1_rows):

    # if mat1 rows and mat2 cols are equal we switch them so
        the following code works
    if (matrix2_rows == matrix1_cols):
        matrix2, matrix1 = matrix1, matrix2

    new_matrix = [] # making a new matrix to hold values

    # iterates through each row of the first matrix
    for m2_row in matrix2: # this line alone is O(n)
        new_matrix_row = [] # O(1) creating a new list for just
            the row we are doing
        for _index in range(len(matrix1[0])): # this line is O(n)

            '''teh following, as a whole, is O(n) we will
                mutliply the the two rows,
            except that one of these these "rows" should be a
                column
            so we will use get_col() to grab it and pass it as
                an argument.
            this si '''
```

```
                new_matrix_row.append(multip_1row( # O(n)
                    m2_row,
                    get_col(matrix1, _index) # O(n)
                ))

            new_matrix.append(new_matrix_row) # addif the new row to
                a the full matrix

        return new_matrix # we will return

    else: # raises the error
        raise ValueError("The dimensions of the arrays passed as \
        arguments are incompatible and cannot be multiplied")

    # full time O(n**3)
```

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. Subspaces are, by proxy, spaces that retain the same properties and are a part of the original vector space

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.

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:

$$V \quad = \quad \begin{bmatrix} A_{1,1} & A_{1,2} & ... & A_{1,j} \\ A_{2,1} & A_{2,2} & ... & A_{2,j} \\ ... & ... & ... & ... \\ A_{j,1} & A_{j,2} & ... & A_{j,j} \end{bmatrix} \in \mathbb{R}^{j \times j}$$

j represents the general dimensionality of the vector space.

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.

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

Basis Vector

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

The code to do this involved taking the indexes of the pivot columns from earlier and then mapping those to the original matrix. In the example above, the pivot columns would be a list with index [0] since the original matrix's RREF form comes out to be

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

```
def create_basis(self):

    # we find the basis of generatign set

    tempgset = self.gset # we create a temporary gset to not
        modify the original

    if self.columned == False: # if the input generating
        vectors are row vecs
```

```
            tempgset = mp.transpose(tempgset) # we will make then
               col vecs

        holder = mp.identify_pivots(tempgset) # now we identify
            pivots of the generating set, O(n**2)

        return [self.gset[i] for i in holder] # then we will return
            all of the vectors
```

Then any valid subspace is just the span of the linearly independent basis vectors.

2.4 Orthonormality of Basis Vectors

Orthonormality of basis vectors is when basis vectors are the unit vectors (the distance of the vector is 1) and if it is orthogonal (purpendicular). We can find this by taking the dot product, a type of inner product defined as $\begin{bmatrix} x & y \end{bmatrix} \cdot \begin{bmatrix} a & b \end{bmatrix} = xa + yb$. The following needs to be satisfied:

$\langle x \, , \, y \rangle = 0$
$\langle x \, , \, x \rangle = 1$

Here is our dot product code:

```
    def dot(v1, v2):
    # find the dot product of 2 vectors
    if not(isinstance(v1[0], list)) and not(isinstance(v2[0],
        list)) and (len(v1)==len(v2)):
        # if we find that the vectors are not 1 dimensional or that
            that they are different sizes
        total = 0
        for idx in range(len(v1)):
        # multiply elementwise
            total += v1[idx]*v2[idx]

        return total
```

These are very important to rotations of vectors. Here is our implemenentation of orthonormality:

```
    def isorthonormal(self):

        if self.columned == False: # if the input basis vecs are
    row vecs, a
    basis = mp.transpose(self.basis) # we will make them col
        vecs, but will use a new var to not modify self.basis
else:
    basis = self.basis # we don't want to fuck with
        self.basis

if len(basis) >= 2: # orthonormal must have at least two
    basis vectors
    for idx, vec1 in enumerate(basis): # now we run through
        each basis vector O(n)
        for vec2 in basis[idx+1:]: # we run through each
            other basis vecor
            if (dot(vec1, vec2) != 0) or (dot(vec1, vec1) !=
                1): # if any of them do not pass the test
                (<v1,v2> = 0 or <v1,v1> =1) (dot defined
                lower)
                return False # then we will return False
                    because all must pass the test

    return True # otherwise, if False not returned yet, we
        know it to be true

else: return False # if only one basis vec, nothing to be
    orthonormal to
```

3 Representing Linear Mappings Pythonically

As stated earlier, we would use the user inputted matrix representations of a vector space (generating set) and get a collection of basis vectors. A linear mapping is a function that maps between 2 vector spaces and also prserves the following rules:

$T : x \to y$

$T(x + y) = T(x) + T(y)$
$T(ax) = aT(x)$

where T is a function linearly mapping vector space x to y. In the case of a code application, these would be finite vector spaces. To map a vector from one space to another, we need to find the transition matrix from the basis vectors we are converting with, which could even be 2 bases for each vector space. We will cover that code in the next section. Assuming we have the transition matrix, you can multiply that to the vector to recieve a vector that has been mapped to a new vector space. We can map a given vector with this code:

```
    def map(self, vector):
        return self.apply(vector)

  def apply_mapping(self, vector):

    # expects a vector in the domain, not codomain
    # expects a vector in the domain, not codomain, of the form
        vector = [element1, ...., element 2]
    # will then apply the linear mapping (self.mapping) to the
        vector
    # will return the result
    if len(vector) != self.domain_dim:
      return "hold up! wait a minute! sumn aint right"

    else:
      return mp.multiply_matrix(self.mapping,
          mp.transpose([vector]))

  def is_subspace(self, subspace):
      # check if subspace is a part of vector_space_1
      for vector in subspace:
          if vector not in self.matrix:
              return False
      return True
```

We can also check if a given vector space is a subspace by iterating through the original vector space and verifying if each element is in the vector space.

3.1 Identifying The Type of Linear Mapping

There are a handful of different ways to classify a mapping. These can tell us useful properties about a mapping:
$\phi : J \to K$

Injective if $\forall x, y \in J : \phi(x) = \phi(y)$
Surjective if $\phi(J) = K$

Bijective if both hold

Homomorphism if its a linear mapping
Isomorphism if $\phi : J \to K$ linear and bijective
Endomorphism if $\phi : J \to J$ linear
Automorphism if $\phi : J \to J$ linear and bijective

Our code to do this is listed below:

```
    class LinearMapping:

  def __init__(self, A, B, mapping=None):
    self.domain_basis = [A[i] for i in
        mp.identify_pivots(mp.transpose(A))]
    self.domain_dim = len(self.domain_basis)
    self.codomain_basis = [B[i] for i in
        mp.identify_pivots(mp.transpose(B))]
    self.codomain_dim = len(self.codomain_basis)
    self.domain = [row[:] for row in A]
    self.codomain = [row[:] for row in B]
    self.mapping = mapping # needs to be a matrix or None
    self.injective = True if mp.rank(self.mapping) ==
        len(self.mapping[0]) else False
    self.surjective = True if mp.rank(self.mapping) ==
        len(self.mapping) else False
    self.bijective = True if (self.injective == True and
        self.surjective == True) else False
    self.homomorphism = True # (by definition, duh)
    self.isomorphism = True if (self.bijective == True) else False
        # homomorphism already satisfied
    self.endomorphism = True if len(A) == len(B) and len(A[0]) ==
        len(B[0]) else False # True if dim(A) == dim(B) and
        len(A[0]) == len(B[0]) else False
    self.automorphism = True if self.endomorphism and
        self.bijective else False
```

3.2 More advanced Basis Change

To find the transformation matrix between basis vectors requires the RREF (Row-Reduced Echelon Form) of the new basis with the older basis vectors, we place the original basis vectors on the left and the desired basis on the right. We then take the right side of the augmented matrix; this can be extended to 2 pairs of bases as shown below:

```
def find_transition(og_base, new_base, columned=False):

'''
TAKES:
    original basis of subspace, og_base
    new basis of subspace, new_base

    it is important to pay mind to the shape of these basis
        vectors.
    eg, input [[1,3,4], [1,4,7]] where each sublist is a basis
        vector will be treated as 3 col vectors,
    [1,1], [3,4], and [4,7]. we added the columned flag to help
        with this
'''

if columned==False:
    og_base, new_base = transpose(og_base), transpose(new_base)

# we find a transition matrix that changes the coordinates
    expressed from one base to another base

# going from base 1 to base 2
if len(og_base) == len(new_base) and len(og_base[0]) ==
    len(new_base[0]):

    dim_to_take = len(og_base[0]) # we find what the dimensions
        of the transmat will be
    total = append_mat_right(new_base, og_base) # we create an
        augmented matrix with new basis on left and old basis on
        right
    reduced = rref(total) # row reduce, O(n**3)

    transition_matrix = list() # init our blank transition
        matrix

    for row in reduced: # we go through the reduced augmat to
        find what should be added to the transition matrix
        transition_matrix.append(row[-dim_to_take:])

    return transition_matrix

# Full time O(n**3)
```

$A'_\phi = T^{-1} A_\phi S$

Where $\phi : J \to K$ and ordered bases

$B = (b_1, ..., b_n)B' = (b'_1, ..., b'_n)$

$C = (c_1, ..., c_n)C' = (c'_1, ..., c'_n)$

B is a tuple of the basis vectors in that given space and S represents the mapping of coordinate from B to B'. T represents the mapping of coordinates from C to C' and A_ϕ is the Transformation matrix from B to C.

```python
def change_transformation_basis(ogtransfmat, ogb1, ogb2,
    tildab1, tildab2, columned=False):

'''
TAKES:
    the transformation matrix/linmapping, ogtransfmat, of
        standard form outlined in flowerbox
    the original basis of domain, ogb1
    the original basis of codomain, ogb2
    the new basis of domain, tildab1
    the new basis of codomain, tildab2
    AS WITH find_transition, the base inputs can be finicky,
        the columned flag should help. check that for more
        indepth docs

'''

if columned==False: # we make sure the basis inputs are the
    correct form ie they are column vectors
    ogb1, ogb2, tildab1, tildab2 =
        transpose(ogb1),transpose(ogb2),transpose(tildab1),transpose(tildab2)

transition1 = find_transition(tildab1, ogb1) # we find
    transitino matrix from the new domain basis to the old
    domain basis, O(n**3)
transition2 = inverse(find_transition(tildab2, ogb2)) # we find
    inverse of transition matrix from new codomain basis to old
    codomain basis, O(n**3)

return multiply_matrix(multiply_matrix(transition2,
    ogtransfmat), transition1) # we multiply the matrices
    according to formula, O(n**3)

# full time O(n**3)
```

4 More Advanced Matrix Operations

4.1 Solving Homogeneous System

Some matrices contain linearly dependent columns or rows making certain variables ”free”. We learned about the -1 trick for solving such matrices.

```
def solve_homogeneous(coef_matrix):
# takes matrix of form outlined in flowerbox
# returns a matrix using the minus 1 trick to solve homogeneous
    linear systems

coef_matrix = rref(coef_matrix) # takes the rref of the
    coefficient matrix (system of linear eq) O(n**3)

pivotcols = identify_pivots(coef_matrix) # finds the pivot
    columns of the rref matrix O(n**2)
notpiv_cols = list() # since this function is called because of
    different size dimension matricies, there are going to be
    necessary added piv-columns
for i in range(len(coef_matrix[0])): # iterates through the
    columns O(n)
    if i not in pivotcols: # Iterates through the list of pivot
        columns
        notpiv_cols.append(i) # appends the non pivot column to
            the non-pivot col list

if notpiv_cols == []: # if the non-pivot columns are empty (in
    other words had total rank)
    return [0 for i in range(len(coef_matrix[0]))]

if len(coef_matrix) == len(coef_matrix[0]):

    for idx in notpiv_cols:
        coef_matrix[idx][idx] = -1
    coef_matrix = transpose(coef_matrix)
    final = [coef_matrix[idx] for idx in notpiv_cols]
    return final


for idx in notpiv_cols: #Iterate through the non pivot columns
    coef_matrix.insert(idx, [0 if i != idx else -1 for i in
        range(len(coef_matrix[0]))]) # by the minus 1 trick, we
        insert a row with minus 1 in the nth index to preserve
        diagonality
coef_matrix = transpose(coef_matrix) # we transpose the matrix
final = [coef_matrix[idx] for idx in notpiv_cols] # we return
    the solution as a square matrix
return final

# O(n**3)
```

4.2 QR Decompositions

One of the most important ways to manipulate a matrix is to decompose it into other matrices. Doing this can help us understand properties of the original matrix and can let us draw conclusions about the data presented within that matrix.

Additionally, we can use matrix decompositions to store a matrix of data in smaller matrices - for example, we can greatly reduce the data we need to store by using rank-k approximations.

One such decomposition is the QR decomposition, which will ultimately enable us to find eigenvalues of a matrix.

In this decomposition, we find a matrix Q that is orthonormal and a matrix R that is in upper-triangle form such that QR is equivalent to the original matrix, A.

The algorithm to solve for the matrix Q involves the Graham-Schmidt method. (Consult wikipedia for the full algorithm)

It should be noted that some issues arise when we pass a rank deficient matrix, as finding the QR of such a matrix requires what is known as a "reduced QR Decomp," which we did not code due to time constraints.

```
def qrdecomp(matrix):

    if mp.rank(matrix) != len(matrix):
        raise ValueError("Matrix is rank deficient and cannot be
            fully QR Decomposed")

    matrix = mp.transpose(matrix)

    q = list()
    r = list()
    uactive = None
    ulist = list()
    for idx, column in enumerate(matrix):
        if idx == 0:
            uactive = column
        else:
            uactive = column
            for oldu in ulist:

                inprodscalar = ((dot(oldu, column)) /
                    (euclidean_norm(oldu))**2)
                projected = mp.row_by_scalar(oldu, inprodscalar)
                uactive = mp.subtract_row(uactive, projected)



        norm = 1/(euclidean_norm(uactive))
        e = mp.row_by_scalar(uactive, norm)

        ulist.append(uactive)
        q.append(e)

    q = mp.transpose(q)
    qinv = mp.inverse(q.copy())
    r = mp.multiply_matrix(qinv,mp.transpose(matrix))
    return q,r
```

4.3 Finding Eigenvalues and Eigenvectors Of A Matrix

We can use the QR decomposition to determine the eigenvalues of a matrix. This is much more efficient than attempting to solve the characteristic polynomial when it comes to a large matrices.

We are able to use the QR decomposition by exploiting the fact that similar matrices will have the same eigenvalues (consult Google for more information).

Essentially, by finding the QR decomposition, finding a new matrix of $R\dot{Q}$, and further reducing that into QR form. Eventually, what is given by $R\dot{Q}$ will converge to an upper triangle matrix. It is a fact that the eigenvalues of an upper triangle matrix are its diagonals. Therefore, the eigenvalues of the original matrix will be the eigenvalues of the upper triangle matrix we converge to by iterating through QR decomps.

It should be noted that for matrices with complex solutions, we will not end up converging to an upper triangle matrix, but rather to an upper triangle matrix with a few seemingly random entires on the bottom half. We can use the entries to determine what the complex eigenvalues are.

```
def qrdecomp(matrix):

if mp.rank(matrix) != len(matrix):
    raise ValueError("Matrix is rank deficient and cannot be
        fully QR Decomposed")

matrix = mp.transpose(matrix)

q = list()
r = list()
uactive = None
ulist = list()
for idx, column in enumerate(matrix):
    if idx == 0:
        uactive = column
    else:
        uactive = column
        for oldu in ulist:

            inprodscalar = ((dot(oldu, column)) /
                (euclidean_norm(oldu))**2)
            projected = mp.row_by_scalar(oldu, inprodscalar)
            uactive = mp.subtract_row(uactive, projected)



    norm = 1/(euclidean_norm(uactive))
    e = mp.row_by_scalar(uactive, norm)

    ulist.append(uactive)
    q.append(e)

q = mp.transpose(q)
qinv = mp.inverse(q.copy())
r = mp.multiply_matrix(qinv,mp.transpose(matrix))
return q,r
```

Next, we will find the eigenvectors by using the eigenvalues and solving the homogeneous system of equations, $(A - \lambda I)x = 0$ where x is the eigenvector associated with λ.

We simply iterate through the eigenvalues.

```
def eigvals(matrix, iterations = 1000, tolerance=6):

activemat = copy.deepcopy(matrix)

istriangle = False

for i in range(iterations):
# while not istriangle:
    q, r = qrdecomp(activemat)
    activemat = mp.multiply_matrix(r,q)

        # istriangle = isuppertriangle(activemat, tolerance)


# g = isuppertriangle(activemat, tolerance)
# return activemat

if isuppertriangle(activemat, tolerance):

    return mp.diags(activemat)


else:

    eigvals = mp.diags(activemat)

    for idx in range(len(activemat)-1):

        if round(activemat[idx+1][idx], 8) != 0:

            a = activemat[idx][idx]
            b = activemat[idx][idx+1]
            c = activemat[idx+1][idx]
            d = activemat[idx+1][idx+1]

            term1 = -(-a-d)
            term2 = ((term1**2) - 4*(a*d - b*c))
            positiveroot = (term1 + term2**(1/2))/2
            negativeroot = (term1 - term2**(1/2))/2


            del eigvals[idx:idx+2]
            eigvals.insert(idx,positiveroot)
            eigvals.insert(idx+1,negativeroot)

            idx+=1

    return eigvals

def eigvecs(matrix, eigvalitr = 1000, eigvaltol = 6, vec_tol =
    4):

    eigval_list = eigvals(matrix, eigvalitr, eigvaltol)
    matrix_size = len(matrix)
    eigvec_dict = {}

    for eigval in eigval_list:

        if isinstance(eigval, complex):
            eigval = (round(eigval.real, eigvaltol) +
                round(eigval.imag, eigvaltol)*1j)
        else:
            eigval = round(eigval, eigvaltol)
        idt = mp.make_identity(matrix_size)
        idt = mp.matrix_by_scalar(idt, eigval)
        subtracted_mat = mp.subtract_matrices(matrix, idt)
        # subtracted_mat = mp.mround(subtracted_mat, vec_tol)
        subtracted_mat = mp.echelon(subtracted_mat)
        subtracted_mat = mp.mround(subtracted_mat, vec_tol)

        sln = mp.solve_homogeneous(subtracted_mat)

        eigvec_dict[eigval] = sln


    return eigvec_dict
```

4.4 Finding the Eiegendecomposition

To find the Eiegendecomposition, we can use the eiegenvectors function we have from earlier to compute the decomposition. Then create the basis matrix with eiegenvectors (P). Then we employ the forumla where A is the orginal matrix:
$P^{-1}AP$

This will be integral to the Singular Value Decomposition

```
def eigendecomp(matrix, eigvalitr = 1000, eigvaltol = 6,
    vec_tol = 4, normalize=True):

eigvec_dict = eigvecs(matrix, eigvalitr, eigvaltol, vec_tol)

p = list()
d = list()

i = 0
for eigval, eigvec in eigvec_dict.items():

    if normalize == True:
        scalar = euclidean_norm(eigvec[0])
        eigvec = mp.matrix_by_scalar(eigvec, (1/scalar))

    d.append([0 if idx != i else eigval for idx in
        range(len(matrix))])
    p.append(eigvec[0])
    i += 1

p = mp.transpose(p)
pinv = mp.inverse(p)

return p, d, pinv
```

4.5 Finding the Singular Value Decomposition

To find the SVD, we can take the transpose of the original matrix and multiply them. A^TA Then you take the eiegenvalues of that and put it on the diagonals to get the matrix (D) and make all the columns the unit vector to get matrix (P) PDP^T Then you create the singular value matrix which is just the square root of the eiegenvalues placed along the diagonal depending on the rank of the original matrix and must be the same size as the original. $PV\sum = A$

```
def svd(matrix, eigvalitr = 1000, eigvaltol = 6, vec_tol = 4,
    normalize=True):

v, d, useless =
    eigendecomp((mp.multiply_matrix(mp.transpose(matrix),
    matrix)), normalize)
# u = eigendecomp((mp.multiply_matrix(a, mp.transpose(a))),
    normalize)[0]

v = mp.matrix_by_scalar(v, -1)
u = list()
sig = list()

for idx in range(len(matrix)):
    sig.append([0 if idx != i else ((d[idx][idx])**(1/2)) for i
        in range(len(d[idx]))])

for idx in range(len(sig)):
    u_idx = mp.multiply_matrix(matrix,
        mp.transpose([mp.get_col(v, idx)]))
    u_idx = mp.matrix_by_scalar(u_idx, 1/sig[idx][idx])
    u.append(mp.transpose(u_idx)[0])
u = mp.transpose(u)

return u, sig, mp.transpose(v)
```