# Class 6 Notes: Matrix Algebra in Python

Myles D. Garvey, Ph.D

Summer, 2019

## Contents



## 1 Introduction

Matrix Algebra is incredibly important to understand without the aid of a computer. In order to effectively model certain situations in practice, as well as be able to conduct certain types of problems in Calculus, the student of Matrix Algebra needs to have a firm understanding of the underlying theory before moving onto computational efficiencies. With that stated, once the

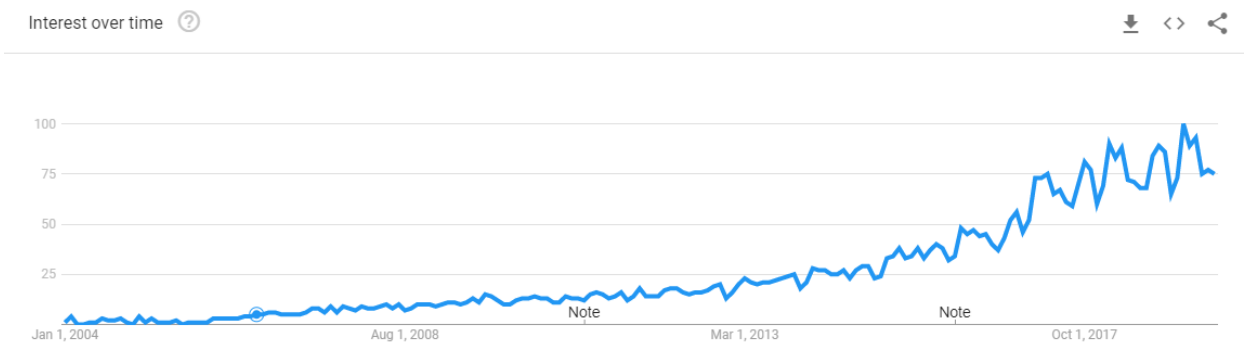


Figure 1: Google Trend for numpy over time.

student of Matrix Algebra has a fairly decent idea of the underlying concepts such as vectors, matrices, operations on these, determinants, inverses, eigenvalue and eigenvectors, powers of matrices, and systems of linear equations, they find the "compute by hand" portion of matrix algebra tiresome. As such, we often leverage software to aid in our matrix-based computations to save the time, and, reduce the chances of human error.

It should be emphasized that computational software for matrices is **not** a replacement for ascertaining a basic working knowledge of *how* to compute. This is still of immense importance for the student to understand. Computational software only helps us when we know that we have modeled our problem using the correct matrices, representations, etc. If the student has a firm knowledge of these topics, it is only then that they should seek the aid of burdensome computation. Python is one such programming language that offers us relief in having to manually compute many arithmetic operations. Specifically, the `numpy` module in Python comes to great aid. This module is of immense importance to both the data scientist and the business analyst. As we can see in Figure 1, a quick Google Trend analysis on the package name itself reveals to us that many have shown an exponentially increasing interest in the usage of the package.

Of course, there are other software that accomplish the simplification of matrix algebra computation, such as R, MatLab, Maple, etc. However, Python offers us a level of flexibility that these other packages do not offer. First, Python is itself a programming language, while the other software packages are special purpose packages. This means that if we need to embed our linear algebra computations in a larger infrastructure of software, Python allows us to easily do so. Second, the `numpy` package is based entirely on C-code. C is a programming language that first appeared over 40 years ago. It is the language that is "closest to the machine", but still easy enough to read. However, for many non-math, non-computer-science students, they need a language that is easy to understand, without getting bogged down with certain programming complexities. Python acts as a wrapper around these C libraries, which means, it makes it easy to leverage the power of C-code, without ever having to have the student write it. Python also integrates very well with Unix-Based machines running Linux, which means that your code can very easily be set up to run on the computer with either the click of a button, or, according to a pre-determined schedule.

In this lecture, we will review through some of the features of the `numpy` module that will help aid in our computations that involve matrices. We will explore the topics that we discussed the

other day in class, but from the perspective of Python. We will hand-write some of the algorithms that are used to solve equations that involve matrices, to gain an understanding of how some of the functions in `numpy` work, as well as understand how we can leverage Python to characterize vector spaces as well as eigenvalues and eigenvectors.

## 2 Installing the `numpy` Package

Many modules in Python come packaged already with Python. However, there will be occasions (like in our current situation), where you will require a hand-written module that does not come pre-packed with Python. In this instance, you have a few options. The easiest option is to use `pip`. This is Python's package manager, and it makes downloading and installing external packages easy to do. If `pip` is properly installed, then you should be able load up command prompt (or terminal if you are on a mac), and type:

```
pip install numpy
```

Listing 1: Installing `numpy` with `pip`

If `pip` is not installed on your computer, you can always try to work with Anaconda, which is an alternative package manager that will help you easily grab Python modules. You can visit the link at https://www.anaconda.com/distribution/. Once you have anaconda installed, you can run in the command prompt (or terminal)

```
conda install -c anaconda numpy
```

Listing 2: Installing `numpy` with Anaconda

More details on how to use Anaconda can be found here: https://anaconda.org/anaconda/numpy. Your last option is to install it by hand, which I never recommend doing unless you really, really know what you are doing (manual installs can be painful, laborious, and involve lots of trial and error!). I will avoid details of this, but if all else has failed, and you have some slightly higher technical ability, intructions for manual build are here:
https://docs.scipy.org/doc/numpy-1.15.0/user/building.html.

Once you have `numpy` installed, you are ready to use it! As with any module in Python, you can easily bring in the collection of functions available in the module by leveraging the `import` command in Python. We introduce a new keyword to use as well, namely, that of `as`. Sometimes, package names are too long to continually type as we code our program. Since we typically do not recommend importing all functions into a program, and instead recommend importing only the name of the module that contains the module, if the name is too long, this can slow down our productivity of coding. We hence want a way to rename the module as we are working with it. For example, if we were load in the module as we typically do, we would have the following code:

```
import numpy

x=numpy.array([3,5,2])
y=numpy.array([1,4,5])

numpy.matmul(x,y)
```

Listing 3: Loading Package `numpy`

There is nothing wrong with this code. However, notice that anytime we need to use a function from the `numpy` package, we are left with having to manually type the keyword `numpy` in front of every function from the module that we use. Instead, if we want to assign the module name a shorter name so as to help improve our productivity, we can do this when we import the module, and rename it to something else, such as:

```
import numpy as np

x=np.array([3,5,2])
y=np.array([1,4,5])

np.matmul(x,y)
```

Listing 4: Renaming `numpy` to `np`

The two code listings are the same, with the exception that in the second one, we renamed the module to `np` to help shorten our typing. You can rename it anything you want (within certain constraints of course). Standard convention just happens to be `np`.

## 3 Representing Matrices in Python

Matrices can be represented in Python as *arrays*. Suppose we would like to put the matrix $\begin{bmatrix} 3 & 7 & 4 \\ 5 & 6 & 6 \\ 3 & 8 & 9 \end{bmatrix}$ into Python in order to work with it. We can do this by leveraging the array function and providing it a list of lists. Each list in the list is a single row in the matrix. So we would have:

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

If we wanted to generate a column vector, we can also do this using the approach from above:

```
v = np.array([[3],[5],[2]])
v
```

We also have other methods of generating different types of matrices. We can generate the identity matrix by providing it the size (number of rows and columns):

```
I = np.identity(5)
I
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

If we wish to create an $mxn$ matrix filled with ones, we can do this by using the `ones` function and providing it a 2-tuple. A 2-tuple in python is specified by typing $(a, b)$, where $a$ is the first element in the tuple, and $b$ is the second element in the tuple. So if we wanted to generate, say, a 5x3 matrix of 1s, we can do so by:

```
B=np.ones((5,3))
B
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Notice the double parenthesis. This is because we are providing only one input into the function, not two. The single input is the tuple $(5,3)$. Likewise, if we want to generate a matrix of size $(m,n)$ full of zeros, we can do this in a similar manner:

```
C=np.zeros((5,3))
C
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Generally, we can fill any matrix of a given size with a specific value by using another function as well:

```
D=np.full((5,3),5)
D
array([[5, 5, 5],
       [5, 5, 5],
       [5, 5, 5],
       [5, 5, 5],
       [5, 5, 5]])
```

# 4 Working with Matrix Operations

Once we have defined our matrices in Python, we would like to operate on them. Recall that we had a few types of operations that can be conducted on matrices. We discuss how to work with these in Python in the following subsections.

## 4.1 Working with Transposes

Recall that the transpose of a vector converts a row vector to a column vector and a column vector to a row vector. Likewise, the transpose operator on a matrix turns the rows into columns. In Python, after we specify a vector, we can easily find the transpose by putting ".T" after the name of the variable that is storing the vector. We can see how this is conducted on vectors below:

```
rv=np.array([[3,4,5]])
cv=np.array([[5],[4],[3]])
rv
array([[3, 4, 5]])
cv
array([[5],
       [4],
       [3]])

rv.T
```

```
array([[3],
       [4],
       [5]])

cv.T
array([[5, 4, 3]])
```

Likewise, we do the same thing for matrices:

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

## 4.2 Addition, Subtraction, Multiplication

Other operations that we can perform on vectors and matrices in Python include addition, subtraction, and multiplication. We can easily add and subtract matrices as follows:

```
A=np.array([[4,2,6],[7,7,4],[1,1,3]])
B=np.array([[0,2,4],[74,6,6],[6,5,7]])
A+B
array([[ 4,  4, 10],
       [81, 13, 10],
       [ 7,  6, 10]])
A-B
array([[  4,   0,   2],
       [-67,   1,  -2],
       [ -5,  -4,  -4]])
```

In addition, we can perform scalar multiplication on vectors and matrices:

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

k=5
k*A
array([[20, 10, 30],
       [35, 35, 20],
       [ 5,  5, 15]])
```

Other than scalar multiplication, we have two other types of multiplication. Recall the first is the inner product between two vectors, which can accomplished in Python using:

```
cv=array([[5],
       [3],
       [6],
       [6]])
rv=array([[6, 5, 2, 4]])
rv.dot(cv)
array([[81]])
```

Notice that when we use the dot product, we need to ensure that the row vector comes first, then the column vector. If we first specify the column vector and then the row vector, then we obtain the outer product of the vectors:

```
cv=array([[5],
       [3],
       [6],
       [6]])
rv=array([[6, 5, 2, 4]])
cv.dot(rv)
array([[30, 25, 10, 20],
       [18, 15,  6, 12],
       [36, 30, 12, 24],
       [36, 30, 12, 24]])
```

However, we need to be careful not to use two column vectors or two row vectors, otherwise, and rightfully so, we obtain an error, since it is impossible to multiply two column vectors or two row vectors:

```
cv.dot(cv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (4,1) and (4,1) not aligned: 1 (dim 1) != 4 (dim 0)

rv.dot(rv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (1,4) and (1,4) not aligned: 4 (dim 1) != 1 (dim 0)
```

Last, we can multiply two matrices if they are conformable. We show below what happens when they are not conformable:

```
A=np.array([[4,2,6],[7,7,4],[1,1,3]])
B=np.array([[0,2,4,5,1],[74,6,6,5,2],[6,5,7,3,6]])

B
array([[ 0,  2,  4,  5,  1],
       [74,  6,  6,  5,  2],
       [ 6,  5,  7,  3,  6]])
A
array([[4, 2, 6],
       [7, 7, 4],
       [1, 1, 3]])

>>> np.matmul(A,B)
array([[184,  50,  70,  48,  44],
       [542,  76,  98,  82,  45],
       [ 92,  23,  31,  19,  21]])
>>> np.matmul(B,A)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with
    gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 5)
```

Here we see that when we multiply $AB$, we this is conformable, since we are multiplying a (3x3) matrix with a (3x5) matrix. But notice that when we try to multiply BA, we have non-conformable matrices, which means we are multiplying a (3x5) and a (3x3), and so, we obtain an

error message.

# 5 Characterizing Vector Spaces

## 5.1 Determining Linear Independence: Implementing Gauss-Jordan

Recall that when given a collection of vectors $\{x_1, x_2, \ldots, x_n\}$, we say that the vectors are *linearly independent* if the only values for $\alpha_1, \ldots, \alpha_n$ that satisfy the equation $\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n = 0$ are $\alpha_1 = \alpha_2 = \cdots = \alpha_n = 0$. In other words, for any $x_i$, we cannot write $x_i$ as a linear combination of the other vectors $x_j, j \neq i$. However, how can we show if a given set of vectors is linearly independent or dependent?

One way to determine this is to first rework the vectors algebraically:

$$\alpha_1 \boldsymbol{x_1} + \alpha_2 \boldsymbol{x_2} + \cdots + \alpha_n \boldsymbol{x_n} = \mathbf{0}$$

$$\alpha_1 \begin{bmatrix} x_{11} \\ x_{21} \\ \vdots \\ x_{m1} \end{bmatrix} + \alpha_2 \begin{bmatrix} x_{12} \\ x_{22} \\ \vdots \\ x_{m2} \end{bmatrix} + \cdots + \alpha_n \begin{bmatrix} x_{1n} \\ x_{2n} \\ \vdots \\ x_{mn} \end{bmatrix} = \mathbf{0}$$

$$\begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \mathbf{0}$$

We notice that if we an find a solution that is non-zero for the $\boldsymbol{\alpha}$ vector, then it means we have been able to show that one vector can be represented as a non-zero linear combination of the other vectors, which means it is linearly dependent. Notice also that:

$$\begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

So, if we can convert the matrix

$$\begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{bmatrix}$$

to the matrix

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

then we have effectively solved for the vector $\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$. We can do this conversion through a series of three fundamental matrix operations. These operations allow us to change the matrix without changing the solution:

1. Switching two rows
2. Multiplying a row by a scalar
3. Addition of a scalar multiple of one row to another row

Using a combination of all three of these operations, we can convert any matrix to the identity matrix, or a matrix the "looks like" the identity matrix, via an algorithm called *Gauss-Jordan Elimination*. The algorithm is as follows:

Suppose we have an $nxm$ matrix. Then:

1. For each $i$ in $\{1, 2, \dots, \min(m, n)\}$
   (a) Set $k = a_{ii}$.
   (b) Divide row $i$ by $k$.
   (c) For each $j$ in $\{1, 2, \dots, n\}$ such that $j \neq i$
      i. Multiply row $i$ by $-a_{ji}$
      ii. Add the result to row $j$

We will illustrate this process with the matrix $\begin{bmatrix} 4 & 1 & 5 \\ 1 & 5 & 2 \end{bmatrix}$:

$$\begin{bmatrix} 4 & 1 & 5 \\ 1 & 5 & 2 \end{bmatrix} \to \begin{bmatrix} 1 & \frac{1}{4} & \frac{5}{4} \\ 1 & 5 & 2 \end{bmatrix} \to \begin{bmatrix} -1 & -\frac{1}{4} & -\frac{5}{4} \end{bmatrix} \to \begin{bmatrix} 1 & \frac{1}{4} & \frac{5}{4} \\ 0 & \frac{19}{4} & \frac{3}{4} \end{bmatrix}$$
$$\to \begin{bmatrix} 1 & \frac{1}{4} & \frac{5}{4} \\ 0 & 1 & \frac{3}{19} \end{bmatrix} \to \begin{bmatrix} 0 & -\frac{1}{4} & -\frac{3}{(19)(4)} \end{bmatrix} \to \begin{bmatrix} 1 & 0 & \frac{92}{(19)(4)} \\ 0 & 1 & \frac{3}{19} \end{bmatrix}$$

We can see that:

$$\frac{92}{(19)(4)} \begin{bmatrix} 4 \\ 1 \end{bmatrix} + \frac{3}{19} \begin{bmatrix} 1 \\ 5 \end{bmatrix} = \begin{bmatrix} \frac{92}{(19)} \\ \frac{92}{(19)(4)} \end{bmatrix} + \begin{bmatrix} \frac{3}{19} \\ \frac{15}{19} \end{bmatrix} \begin{bmatrix} \frac{95}{19} \\ \frac{92+60}{(19)(4)} \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

We can see that the third vector can be written as a linear combination as the other two. As we can see, by definition, the vectors are therefore not linearly independent. Interestingly, we have a very useful theorem:

**Theorem 1** *Let $\{\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}\}$ by m-dimensional vectors. If $n > m$, then the vectors are linearly dependent.*

Unfortunately, `numpy` does not have a built in function in order to perform this algorithm. Luckily, the package `sympy` does have this functionality. Loading the matrix in using the `sympy` package, we are left with:

```
A=sympy.Matrix([[4,1,5],[1,5,2]])
A
Matrix([
[4, 1, 5],
[1, 5, 2]])
A.rref()
(Matrix([
[1, 0, 23/19],
[0, 1,  3/19]]), (0, 1))
```

We can also see this when the number of columns is less than the number of rows. Suppose we had the vectors $\left\{ \begin{bmatrix} 3 \\ 1 \\ 1 \\ 5 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 5 \\ 3 \end{bmatrix}, \begin{bmatrix} 10 \\ 4 \\ 12 \\ 16 \end{bmatrix}, \begin{bmatrix} 26 \\ 10 \\ 30 \\ 40 \end{bmatrix} \right\}$ and wanted to determine if they are linearly independent. We can create the matrix $\begin{bmatrix} 3 & 2 & 10 & 26 \\ 1 & 1 & 4 & 10 \\ 1 & 5 & 12 & 30 \\ 5 & 3 & 16 & 40 \end{bmatrix}$ from the columns and reduce this using the operations from before. Using Python's ability to do this, we have:

```
A=sympy.Matrix([[3,2,10,26],[1,1,4,10],[1,5,12,30],[5,3,16,40]])
A.rref()
(Matrix([
[1, 0, 2, 0],
[0, 1, 2, 0],
[0, 0, 0, 1],
[0, 0, 0, 0]]), (0, 1, 3))
```

We notice that since the reduced matrix does not "look like" the identity matrix, the vectors are dependent. In order for the vectors to be independent, a simple trick is to look at each row in the reduced matrix. If there is only a 1 and all 0s elsewhere, and there is only a 1 and zeros elsewhere in the corresponding column, then the vectors are independent. Otherwise, if there exists just a single row or column where there are more than one non-zero entries, then we can conclude the vectors are dependent. We can see in the example above that we are left with the first and second row that has more than one non-zero entry. This would imply that our original vectors are linearly dependent.

## 5.2 Characterizing Any Vector Space

Recall the vector space is a set of vectors that closed under scalar multiplication and addition. At times, these sets can comprise of an uncountably infinite number of vectors, which makes it

difficult to characterize. Hence, we need a mechanism to be able to characterize the vector space. Rather than characterize the vector space as a set, we can characterize a vector space as the set of all vectors that can be generated from only a handful of vectors. We had seen earlier that the vectors $\left\{ \begin{bmatrix} 4 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 5 \end{bmatrix}, \begin{bmatrix} 5 \\ 2 \end{bmatrix} \right\}$ are linearly dependent.

When we reduced the vectors down, we were able to reduce them to the vectors $\left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} \frac{23}{19} \\ \frac{3}{19} \end{bmatrix} \right\}$. If we notice, the first two vectors are linearly independent, and can be used to "generate" the other vector. When we are given a set of linearly independent vectors, or we can find a collection of linearly independent vectors from a give set, then we are said to have a *basis*, and the basis is how we can characterize the vector space, since all vectors in the vector space would be generated by linear combinations of the basis vectors. The number of vectors in the basis is referred to as the dimension of the vector space.

Earlier we worked on the example $\left\{ \begin{bmatrix} 3 \\ 1 \\ 1 \\ 5 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 5 \\ 3 \end{bmatrix}, \begin{bmatrix} 10 \\ 4 \\ 12 \\ 16 \end{bmatrix}, \begin{bmatrix} 26 \\ 10 \\ 30 \\ 40 \end{bmatrix} \right\}$, and found that these were linearly dependent. However, these vectors can still be used to generate a vector space. How is it then can we characterize the vector space they generate with the basis? We noticed that these vectors reduced to $\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$. We can see that the only vectors in the reduced matrix which have only one non-zero value are $\left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$. Therefore, the vector space generated by the original 4 vectors can actually be generated by the 3 vectors in the original vectors that correspond to the vectors $\left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$, and therefore, the vector space can be characterized as $\left\{ \alpha_0 \begin{bmatrix} 3 \\ 1 \\ 1 \\ 5 \end{bmatrix} + \alpha_1 \begin{bmatrix} 2 \\ 1 \\ 5 \\ 3 \end{bmatrix} + \alpha_2 \begin{bmatrix} 26 \\ 10 \\ 30 \\ 40 \end{bmatrix} \middle| \alpha_0, \alpha_1, \alpha_2 \in \mathbb{R} \right\}$, and hence, has dimension 3.

### 5.3 Finding the Row and Column Space of a Matrix

When given a matrix, it turns out that the columns of a matrix can be used to generate it's own space, called the *column space*. When we perform the Gauss-Jordan Elimination algorithm on the matrix, we can reduce the matrix to basis vectors. The basis vectors can then be used to describe the dimension and the vector space that the column vectors generate. Take for example

the matrix $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 4 \\ 2 & 5 & 1 & 17 \\ 5 & 5 & 5 & 5 \end{bmatrix}$. As we mentioned, the columns can be used to generate the vector space $\left\{ \alpha_0 \begin{bmatrix} 1 \\ 1 \\ 2 \\ 5 \end{bmatrix} + \alpha_1 \begin{bmatrix} 1 \\ 1 \\ 5 \\ 5 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 5 \end{bmatrix} + \alpha_3 \begin{bmatrix} 1 \\ 4 \\ 17 \\ 5 \end{bmatrix} \middle| \alpha_0, \alpha_1, \alpha_2, \alpha_3 \in \mathbb{R} \right\}$. We want to find the basis for this space, as well as the dimension. When we reduce the matrix, we obtain:

```
A=sympy.Matrix([[1,1,1,1],[1,1,1,4],[2,5,1,17],[5,5,5,5]])
A.rref()
(Matrix([
[1, 0,  4/3, 0],
[0, 1, -1/3, 0],
[0, 0,    0, 1],
[0, 0,    0, 0]]), (0, 1, 3))
```

We can see that the only columns that have a single non-zero value are: $\left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \right\}$, which correspond to the 1st, 2nd, and 4th columns in the original matrix, and hence, these columns in the original matrix is the basis. So, we have:

$$\left\{ \alpha_0 \begin{bmatrix} 1 \\ 1 \\ 2 \\ 5 \end{bmatrix} + \alpha_1 \begin{bmatrix} 1 \\ 1 \\ 5 \\ 5 \end{bmatrix} + \alpha_2 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 5 \end{bmatrix} + \alpha_3 \begin{bmatrix} 1 \\ 4 \\ 17 \\ 5 \end{bmatrix} \middle| \alpha_0, \alpha_1, \alpha_2, \alpha_3 \in \mathbb{R} \right\} = \left\{ \beta_0 \begin{bmatrix} 1 \\ 1 \\ 2 \\ 5 \end{bmatrix} + \beta_1 \begin{bmatrix} 1 \\ 1 \\ 5 \\ 5 \end{bmatrix} + \beta_2 \begin{bmatrix} 1 \\ 4 \\ 17 \\ 5 \end{bmatrix} \middle| \beta_0, \beta_1, \beta_2 \in \mathbb{R} \right\}$$

Notice that the dimension of this space is 3 since there are 3 vectors in the basis $\left\{ \begin{bmatrix} 1 \\ 1 \\ 2 \\ 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 5 \\ 5 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ 17 \\ 5 \end{bmatrix} \right\}$ for the vector space spanned by the column vectors. This vector space is referred to as the column space. Likewise, we can form column vectors from the rows in the matrix and conduct a similar procedure to produce new vectors generated by the transpose of the row vectors. We have the rows, transposed: $\left\{ \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 4 \end{bmatrix}, \begin{bmatrix} 2 \\ 5 \\ 1 \\ 17 \end{bmatrix}, \begin{bmatrix} 5 \\ 5 \\ 5 \\ 5 \end{bmatrix} \right\}$. which can be used to construct the vector space $\left\{ \alpha_0 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \alpha_1 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 4 \end{bmatrix} + \alpha_2 \begin{bmatrix} 2 \\ 5 \\ 1 \\ 17 \end{bmatrix} + \alpha_3 \begin{bmatrix} 5 \\ 5 \\ 5 \\ 5 \end{bmatrix} \middle| \alpha_0, \alpha_1, \alpha_2, \alpha_3 \in \mathbb{R} \right\}$. This vector space is called the row space. Following a similar procedure from before, we have:

```
A=sympy.Matrix([[1,1,2,5],[1,1,5,5],[1,1,1,5],[1,4,17,5]])
A.rref()
(Matrix([
[1, 0, 0, 5],
[0, 1, 0, 0],
```

```
[0, 0, 1, 0],
[0, 0, 0, 0]]), (0, 1, 2))
```

We notice that columns 1,2, and 3 are linearly independent, and so they form the basis for the rowspace. That is,we have:

$$\left\{ \alpha_0 \begin{bmatrix}1\\1\\1\\1\end{bmatrix} + \alpha_1 \begin{bmatrix}1\\1\\1\\4\end{bmatrix} + \alpha_2 \begin{bmatrix}2\\5\\1\\17\end{bmatrix} + \alpha_3 \begin{bmatrix}5\\5\\5\\5\end{bmatrix} \middle| \alpha_0, \alpha_1, \alpha_2, \alpha_3 \in \mathbb{R} \right\} = \left\{ \beta_0 \begin{bmatrix}1\\1\\1\\1\end{bmatrix} + \beta_1 \begin{bmatrix}1\\1\\1\\4\end{bmatrix} + \beta_2 \begin{bmatrix}2\\5\\1\\17\end{bmatrix} \middle| \beta_0, \beta_1, \beta_2 \in \mathbb{R} \right\}$$

Once again, we see that this vector space has dimension 3, since there are three vectors in the basis $\left\{ \begin{bmatrix}1\\1\\1\\1\end{bmatrix}, \begin{bmatrix}1\\1\\1\\4\end{bmatrix}, \begin{bmatrix}2\\5\\1\\17\end{bmatrix} \right\}$. Notice that the dimension of the row space is the same as the dimension of the column space. In general, this is true:

**Theorem 2** *The dimension of the rowspace of a matrix is equal to the dimension of the columnspace of a matrix.*

**Definition 1** *The dimension of the column space of a matrix is referred to as the rank of the matrix. When the rank is equal to the number of columns, the matrix is said to be full rank.*

# 6 Solving Systems of Linear Equations

## 6.1 Converting Systems to Matrix Form

While we have explored some of the operations and properties of matrices and vectors, we have somewhat lost sight of their original meaning in doing so, namely, solving *systems of linear equations*. To bring this idea back around, suppose we have the following system of linear equations:

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \ldots a_{0,n}x_n &= b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + \ldots a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{m,0}x_0 + a_{m,1}x_1 + \ldots a_{m,n}x_n &= b_m \end{aligned}$$

First, we call this a *system* since we have multiple equations. Second, we called this *linear* since all of the equations are linear equations (please refer back to the algebra review for a definition of linear equations/functions). Last, it is assumed that when a system is provided to you, the $a_{i,j}$ are numbers, $b_i$ is a number, and the $x_i$ are unknowns that we would like to solve such that **all** of the equations are true. The specific values assigned to the $x_i$s once we solve for them is called a solution to the system. Geometrically, each equation represents a *hyper-plane* in space, and the solution to the system represents a point of intersection of the system. There are three possible situations, which is easy to see in 2-dimensional space:

1. There is no solution $x_i$ that satisfies the system (Geometrically, this means that the lines in 2-D space are different and parallel, that is, they do not intersect).

2. There is only one unique solution $x_i$ that satisfies the system (Geometrically, this means that the lines in 2-D space all intersect at a single point)

3. There is an infinite number of solutions $x_i$ that satisfies the system (Geometrically, this means that the lines in 2-D space intersect at an infinite number of points, which means all the lines in the equations are the same line [this is only true in 2D space]).

As it turns out, the system can be represented in matrix form via matrix multiplication:

$$\begin{matrix} a_{0,0}x_0 + a_{0,1}x_1 + \dots a_{0,n}x_n = b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + \dots a_{1,n}x_n = b_1 \\ \vdots \\ a_{m,0}x_0 + a_{m,1}x_1 + \dots a_{m,n}x_n = b_m \end{matrix} \rightarrow \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,0} & a_{m,1} & \dots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \rightarrow \boldsymbol{Ax} = \boldsymbol{b}$$

For example, we can convert the system to matrix format as follows:

$$\begin{matrix} 3x_0 - x_1 + 5x_3 = -1 \\ 2x_1 - 3x_2 + 2x_3 = 5 \\ -6x_0 + x_1 - 4x_2 + 5x_3 = 6 \end{matrix} \rightarrow \begin{bmatrix} 3 & -1 & 0 & 5 \\ 0 & 2 & -3 & 2 \\ -6 & 1 & -4 & 5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 5 \\ 6 \end{bmatrix}$$

Furthermore, we can form a special kind of matrix called the *augmented matrix* by moving the $\boldsymbol{b}$ column into the columns of the matrix on the left:

$$\left[\begin{array}{cccc|c} 3 & -1 & 0 & 5 & -1 \\ 0 & 2 & -3 & 2 & 5 \\ -6 & 1 & -4 & 5 & 6 \end{array}\right]$$

## 6.2 Solving and Characterizing Solutions of Non-Homogenous Linear Systems

When at least one of the values on the right side of the system is non-zero, then the system is referred to as a *non-homogenous linear system*. We can solve the system by first conducting Gauss-Jordan Elimination on the augmented matrix, converting the matrix form back into equation form, and solving for the unknowns. We illustrate this process here with the system from the previous example and the augmented matrix

$$\left[\begin{array}{cccc|c} 3 & -1 & 0 & 5 & -1 \\ 0 & 2 & -3 & 2 & 5 \\ -6 & 1 & -4 & 5 & 6 \end{array}\right]$$

:

```
>>> A = sympy.Matrix([[3,-1,0,5,-1],[0,2,-3,2,5],[-6,1,-4,5,6]])
>>> A
Matrix([
[ 3, -1,  0, 5, -1],
[ 0,  2, -3, 2,  5],
[-6,  1, -4, 5,  6]])
>>> A.rref()
(Matrix([
```

```
[1, 0, 0,   6/11,  -1/11],
[0, 1, 0, -37/11,   8/11],
[0, 0, 1, -32/11, -13/11]]), (0, 1, 2))
```

So, when it is reduced, we have the augmented matrix as:

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & \frac{6}{11} & \frac{-1}{11} \\ 0 & 1 & 0 & \frac{-37}{11} & \frac{8}{11} \\ 0 & 0 & 1 & \frac{-32}{11} & \frac{-13}{11} \end{array}\right]$$

This converts back to the equations as:

$$\begin{bmatrix} 1 & 0 & 0 & \frac{6}{11} \\ 0 & 1 & 0 & \frac{-37}{11} \\ 0 & 0 & 1 & \frac{-32}{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{-1}{11} \\ \frac{8}{11} \\ \frac{-13}{11} \end{bmatrix}$$

$$\begin{aligned} (1)x_0 + (0)x_1 + (0)x_2 + (\frac{6}{11})x_3 &= \frac{-1}{11} \\ (0)x_0 + (1)x_1 + (0)x_2 + (\frac{-37}{11})x_3 &= \frac{8}{11} \\ (0)x_0 + (0)x_1 + (1)x_2 + (\frac{-32}{11})x_3 &= \frac{-13}{11} \\ \rightarrow x_0 + (\frac{6}{11})x_3 &= \frac{-1}{11} \\ x_1 + (\frac{-37}{11})x_3 &= \frac{8}{11} \\ x_2 + (\frac{-32}{11})x_3 &= \frac{-13}{11} \\ \rightarrow x_0 &= -(\frac{6}{11})x_3 + \frac{-1}{11} \\ x_1 &= -(\frac{-37}{11})x_3 + \frac{8}{11} \\ x_2 &= -(\frac{-32}{11})x_3 + \frac{-13}{11} \end{aligned}$$

Notice that we do not have an expression for $x_3$. In this instance, $x_3$ is referred to as a *free variable*, and can be set to any $s \in \mathbb{R}$ without changing the solution to the system. If we set $x_3 = s$, then we have the following solution expressed in matrix notation:

$$
\begin{aligned}
x_0 &= -(\frac{6}{11})x_3 + \frac{-1}{11} \\
x_1 &= -(\frac{-37}{11})x_3 + \frac{8}{11} \\
x_2 &= -(\frac{-32}{11})x_3 + \frac{-13}{11} \\
x_3 &= s \\
\rightarrow x_0 &= -(\frac{6}{11})s + \frac{-1}{11} \\
x_1 &= -(\frac{-37}{11})s + \frac{8}{11} \\
x_2 &= -(\frac{-32}{11})s + \frac{-13}{11} \\
x_3 &= s
\end{aligned}
$$

$$
\rightarrow \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -(\frac{6}{11})s + \frac{-1}{11} \\ -(\frac{-37}{11})s + \frac{8}{11} \\ -(\frac{-32}{11})s + \frac{-13}{11} \\ s \end{bmatrix} \rightarrow \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -(\frac{6}{11})s \\ -(\frac{-37}{11})s \\ -(\frac{-32}{11})s \\ s \end{bmatrix} + \begin{bmatrix} (\frac{-1}{11}) \\ (\frac{8}{11}) \\ (\frac{-13}{11}) \\ 0 \end{bmatrix} = s \begin{bmatrix} -(\frac{6}{11}) \\ -(\frac{-37}{11}) \\ -(\frac{-32}{11}) \\ 1 \end{bmatrix} + \begin{bmatrix} (\frac{-1}{11}) \\ (\frac{8}{11}) \\ (\frac{-13}{11}) \\ 0 \end{bmatrix}
$$

We can see that multiple solutions exist since we can choose any value for $s \in \mathbb{R}$ and it would still be a solution to the system (check this for varying values of $s$!). This means that there is an infinite number of solutions to the system.

If we take a different example, we can see what it looks like for the system to have a unique solution. Take, for example, the system

$$
\begin{aligned}
3x_0 - 2x_1 + 3x_2 &= 5 \\
x_0 - 5x_1 + 5x_3 &= 4 \\
2x_0 + 6x_1 - 7x_3 &= 10
\end{aligned}
\rightarrow \left[\begin{array}{cccc} 3 & -2 & 3 & 5 \\ 1 & -5 & 5 & 4 \\ 2 & 6 & -7 & 10 \end{array}\right]
$$

Reducing this matrix, we get:

```
A=sympy.Matrix([[3,-2,3,5],[1,-5,5,4],[2,6,-7,10]])
A
Matrix([
[3, -2,  3,  5],
[1, -5,  5,  4],
[2,  6, -7, 10]])
A.rref()
(Matrix([
[1, 0, 0,   91/29],
[0, 1, 0, -143/29],
[0, 0, 1, -138/29]]), (0, 1, 2))
```

So, the reduced form of the system looks like:

$$\left[\begin{array}{ccc|c} 3 & -2 & 3 & 5 \\ 1 & -5 & 5 & 4 \\ 2 & 6 & -7 & 10 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 0 & 0 & \frac{91}{29} \\ 0 & 1 & 0 & \frac{-143}{29} \\ 0 & 0 & 1 & \frac{-138}{29} \end{array}\right] \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{91}{29} \\ \frac{-143}{29} \\ \frac{-138}{29} \end{bmatrix}$$

$$\begin{array}{r} (1)x_0 + (0)x_1 + (0)x_2 = \frac{91}{29} \\ \rightarrow (0)x_0 + (1)x_1 + (0)x_2 = \frac{-143}{29} \\ (0)x_0 + (0)x_1 + (1)x_2 = \frac{-138}{29} \end{array} \rightarrow \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \frac{91}{29} \\ \frac{-143}{29} \\ \frac{-138}{29} \end{bmatrix}$$

Since the equation for the solution $\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$ comprises of only numbers, we therefore conclude that this is the only and unique solution to the system.

## 6.3 Solving and Characterizing Solutions of Homogenous Linear Systems

There are some applications that we will come across where we would like to solve $\boldsymbol{Ax} = \boldsymbol{0}$. That is, the $\boldsymbol{b}$ vector is set to all zeros. When we are solving for such a system, the system is often referred to as a *homogeneous system of equations*. For example, suppose we have the system:

$$\begin{array}{r} 5x_0 + 2x_1 - 3x_2 = 0 \\ x_0 + 3x_1 + 4x_2 = 0 \\ -5x_0 - 4x_1 - 3x_2 = 0 \end{array} \rightarrow \left[\begin{array}{ccc|c} 5 & 2 & -3 & 0 \\ 1 & 3 & 4 & 0 \\ -5 & -4 & -3 & 0 \end{array}\right]$$

Reducing this down gives us:

```
A=sympy.Matrix([[5,2,-3,0],[1,3,4,0],[-5,-4,-3,0]])
>>> A
Matrix([
[ 5,  2, -3, 0],
[ 1,  3,  4, 0],
[-5, -4, -3, 0]])
A.rref()
(Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0]]), (0, 1, 2))
```

We therefore see that there is only one unique solution, namely $\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$. There are situations, however, where there will be infinite solutions to such an equation. For example, suppose we have the system:

$$\begin{array}{r} 5x_0 + x_1 + 2x_2 = 0 \\ x_0 - x_1 + x_2 = 0 \\ 11x_0 + x_1 + 5x_2 = 0 \end{array} \rightarrow \begin{bmatrix} 5 & 1 & 2 \\ 1 & -1 & 1 \\ 11 & 1 & 5 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

```
>>> B=sympy.Matrix([[5,1,2],[1,-1,1],[11,1,5]])
>>> B.rref()
(Matrix([
[1, 0,  1/2],
[0, 1, -1/2],
[0, 0,    0]]), (0, 1))
```

Rearranging this, we have:

$$\begin{bmatrix}1 & 0 & \frac{1}{2}\\ 0 & 1 & -\frac{1}{2}\\ 0 & 0 & 0\end{bmatrix}\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix}=\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}\rightarrow\begin{array}{c}(1)x_0+(0)x_1+\frac{1}{2}x_2=0\\ (0)x_0+(1)x_1-\frac{1}{2}x_2=0\\ (0)x_0+(0)x_1+(0)x_2=0\end{array}\rightarrow\begin{array}{c}x_0=-\frac{1}{2}x_2\\ x_1=\frac{1}{2}x_2\\ x_2=s\end{array}$$

Just as before, we can set $x_2 = s$ since this is a free variable. Further simplifying, we can see that we can represent the solutions as follows:

$$\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix}=s\begin{bmatrix}\frac{-1}{2}\\ \frac{1}{2}\\ 1\end{bmatrix}$$

We can see this with another example. Suppose we want to solve the system:

$$\begin{array}{c}5x_0-x_1+3x_2=0\\ 10x_0-2x_1+6x_2=0\\ 15x_0-3x_1+9x_2=0\end{array}\rightarrow\begin{bmatrix}5 & -1 & 3\\ 10 & -2 & 6\\ 15 & -3 & 9\end{bmatrix}\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix}=\begin{bmatrix}0\\ 0\\ 0\end{bmatrix}$$

```
>>> B=sympy.Matrix([[5,-1,3],[10,-2,6],[15,-3,9]])
>>> B
Matrix([
[ 5, -1, 3],
[10, -2, 6],
[15, -3, 9]])
>>> B.rref()
(Matrix([
[1, -1/5, 3/5],
[0,    0,   0],
[0,    0,   0]]), (0,))
```

Here, we have two free variables since we do not have an expression for $x_1$ and $x_2$, hence, we can set any value to $x_1 = s$ and any value $x_2 = t$,thus we have:

$$\begin{array}{c}x_0=\frac{1}{5}s-\frac{3}{5}t\\ x_1=s\\ x_2=t\end{array}\rightarrow\begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix}=\begin{bmatrix}\frac{1}{5}s-\frac{3}{5}t\\ s\\ t\end{bmatrix}=\begin{bmatrix}\frac{1}{5}s\\ s\\ 0\end{bmatrix}+\begin{bmatrix}-\frac{3}{5}t\\ 0\\ t\end{bmatrix}=s\begin{bmatrix}\frac{1}{5}\\ 1\\ 0\end{bmatrix}+t\begin{bmatrix}-\frac{3}{5}\\ 0\\ 1\end{bmatrix}$$

# 7 Eigenvalues and Eigenvectors

## 7.1 Finding the Eigenvalues and Eigenvectors of a Matrix

Recall that when given a square matrix $\boldsymbol{A}$, the values $\lambda$ such that $\boldsymbol{Ax} = \lambda\boldsymbol{x}$. If we try to solve these by hand, this can be a cumbersome process. Luckily, the `numpy` module provides a method for us to easily calculate the eigenvalues and eigenvectors. For example, suppose we would like to find the eigenvalues and vectors of the matrix $\begin{bmatrix} 4 & 2 & 1 \\ 1 & 1 & 2 \\ 2 & 5 & 2 \end{bmatrix}$. This can be easily accomplished in Python as follows:

```
>>> A=np.array([[4,2,1],[1,1,2],[2,5,2]])
>>> A
array([[4, 2, 1],
       [1, 1, 2],
       [2, 5, 2]])
>>> eval,evec=np.linalg.eig(A)
>>> eval
array([ 6.38719961,  2.30838907, -1.69558869])
>>> evec
array([[ 0.6066876 ,  0.73891359,  0.07686588],
       [ 0.37319505, -0.33173812, -0.61232756],
       [ 0.70189431, -0.58647806,  0.78685869]])
```

Notice that the result of the function offers back two outputs: eigenvaues and eigenvectors. We can extract both of them by providing two different variable names, separated by commas, on the left, as we do above.

## 7.2 Representing the Matrix in Decomposed Form

Recall that if we found the eigen values and eigenvectors for a matrix, we can represent the matrix in decomposed form: $\boldsymbol{A} = \boldsymbol{C}\Lambda\boldsymbol{C}^{-1}$, where $\Lambda$ is the diagonal matrix of eigenvalues and $\boldsymbol{C}$ is the matrix formed by the eigenvectors. Using the previous example, we can check if this is the case:

```
>>> A=np.array([[4,2,1],[1,1,2],[2,5,2]])
>>> eval,evec=np.linalg.eig(A)
>>> C=evec
>>> CI=np.linalg.inv(C)
>>> C
array([[ 0.6066876 ,  0.73891359,  0.07686588],
       [ 0.37319505, -0.33173812, -0.61232756],
       [ 0.70189431, -0.58647806,  0.78685869]])
>>> CI
array([[ 0.68168913,  0.68867261,  0.46932771],
       [ 0.79523291, -0.46544506, -0.43988981],
       [-0.01536116, -0.96122559,  0.52435823]])
>>> L=np.diag(eval)
>>> L
array([[ 6.38719961,  0.        ,  0.        ],
       [ 0.        ,  2.30838907,  0.        ],
       [ 0.        ,  0.        , -1.69558869]])
>>> B=np.matmul(C,L)
```

```
>>> np.matmul(B,CI)
array([[4., 2., 1.],
       [1., 1., 2.],
       [2., 5., 2.]])
```

### 7.3 Finding Powers of Matrices

With Python, we can easily find powers of matrices by either using the Eigenvalue Decomposition or the built in packages. With the example above, let us explore both of these. First, let us look at the decomposition:

$$\begin{bmatrix} 4 & 2 & 1 \\ 1 & 1 & 2 \\ 2 & 5 & 2 \end{bmatrix} = \begin{bmatrix} 0.6066876 & 0.73891359 & 0.07686588 \\ 0.37319505 & -0.33173812 & -0.61232756 \\ 0.70189431 & -0.58647806 & 0.78685869 \end{bmatrix} \begin{bmatrix} 6.38719961 & 0. & 0. \\ 0. & 2.30838907 & 0. \\ 0. & 0. & -1.69558869 \end{bmatrix} \begin{bmatrix} 0.68168913 & 0.68867261 & 0.46932771 \\ 0.79523291 & -0.46544506 & -0.43988981 \\ -0.01536116 & -0.96122559 & 0.52435823 \end{bmatrix}$$

Recall that when we take powers of matrices, we have: $\boldsymbol{A^n} = \boldsymbol{C}\Lambda^n\boldsymbol{C}^{-1}$. Since $\Lambda$ is a diagonal matrix, then it's power to the $n$ is just the diagonal elements taken to the $n$. Seeing this in Python, we have:

```
>>> Ln=np.matmul(L,L)
>>> Ln=np.matmul(Ln,L)
>>> Ln
array([[260.5742323 ,   0.        ,   0.        ],
       [  0.        ,  12.30062077,   0.        ],
       [  0.        ,   0.        ,  -4.87485306]])
>>> B=np.matmul(C,Ln)
>>> np.matmul(B,CI)
array([[115., 105.,  70.],
       [ 63.,  66.,  49.],
       [119., 133.,  87.]])
>>> An=np.matmul(A,A)
>>> np.matmul(An,A)
array([[115, 105,  70],
       [ 63,  66,  49],
       [119, 133,  87]])
```

We can clearly see that both matrices are the same. Hence, an easy way to compute the power of the matrix is to just simply compute the power of the matrix of the eigenvalues.