Daniel Mevs

Matrix Theory

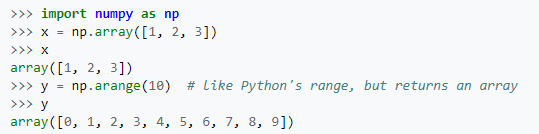
Dr. Winkowska-Nowak

Extra-Credit

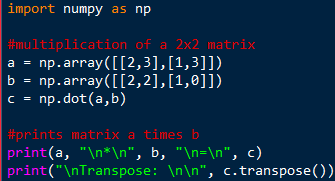
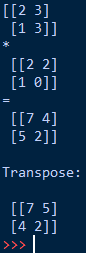
Fun with Matrices in Python

Intro to Numpy

Numpy is a library, or a collection of pre-defined operations, which exists and is executed within the Python programming language. This library is mathematically-oriented and contains multi-dimensional arrays, functions and operators to help the programmer have a conceptual frame-work for matrices. If you would like to follow along with the examples in this paper, Python can be downloaded at <https://www.python.org/downloads/>.



The above code is a snippet from Python’s IDLE tool which is Python’s Integrated Development environment. Here, Python is gaining access to the numpy library through the statement import numpy. As np is basically creating an object which you can invoke methods through such as np.array(). X is a variable that is being initialized to an array. Array() takes as arguments a list of numbers(can be integer or floating-point) and creates a row which can correspond to a row of a matrix. Arange(n) inserts values 0 to n-1 into the array.

 output: 

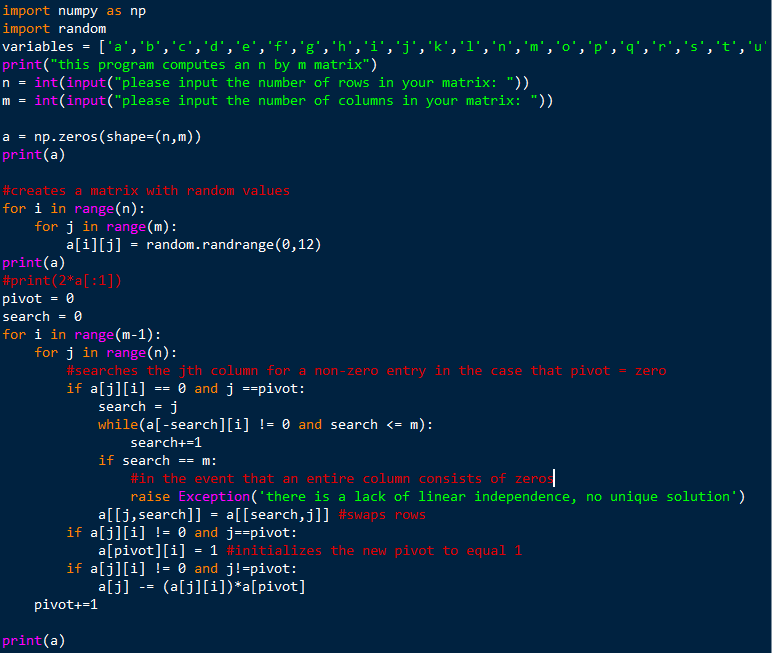
The above line of code initializes a pair of two-dimensional array, a and b. The variable c is initialized to the product of matrices a and b through the method np.dot(a,b). The transpose method returns the transpose of a given matrix. Notice, here, that the list argument being passed a list of two lists. Dot returns the dot product of two matrices.

Intro to linalg

Linalg is a python module that is more suited for linear algebra. One of the built in functions of linalg is linalg.norm which returns one of an infinite number of vector norms. Linalg.det(a) computes the determinant of an array, a. trace(a) returns the sum along diagonals of the array. Linalg.inv(a) computes the multiplicative inverse of a matrix. And linalg.solve(a,b) solves a linear matrix equation or system of linear scalar equations.

Fun Stuff

Of course, as this is a math course, it is no fun to just use built in function like linalg.solv and call it a day. How can we combine these tool in conjunction with logic from our brain to actually perform these computations ourselves? After all, solving a linear system of equations using the Gauss-Jordan elimination method is an algorithmic process that could hypothetically be written as a series of instructions that a computer could execute. I tried to do just that… and I failed. HOWEVER, I can walk you through my logic. Here’s the code:



First I got the size of the row and column from the user. Then I created he an array with that specified shape. Then I initialized the matrix with random values using a random number generator. Then I created a nest for loop. The outer loop accounts for the column and the inner loop accounts for the row. The first condition that I specified is the case where the index where we are in the loop is equal to the pivot and is equal to zero. Ideally, we want our pivot to be one so we would swap out the row by finding an element in that particular column with a non-zero entry. If all the entries in the column are zero I raise an error saying there is no unique solution. The second condition that looked for is where we are at our pivot and the pivot is not equal to one. Usually to you would set this entry equal to that same entry divided by itself. Since that is equal to 1, I just set it equal to 1. The second case I looked for was the case where the entry does not equal 0 and is not the pivot which is set of all entries excluding the diagonal. I set these values equal to itself minus the scalar value at the entry times the row of the pivot, which would convert that entry to a zero and change other values such that we are closer to the solution at the last column. I think this would have work if I found a clever way to iterate through the matrix, since these are essentially the main three operations one performs for Gauss-Jordan elimination. However, the nest for loop combs through the problem once but does not finish the job. One of the things I observed is that it creates a upper-triangular matrix when the size of the matrix is a square matrix(meaning the row and column size are equal). However, when you are doing Gauss-Jordan elimination, ideally, you want to be working with a matrix such that the number of rows is one less than the number of columns to account for the column with the solutions, or b in the equation Ax=b. The output I have might be part of the solution but not anywhere near the solution unfortunately.

Another application that I used with matrices that actually works (and that I didn’t use numpy for) is computing Pascals in Python. The main operation that is performed in addition on matrices. You start off with a row that has n number of entries, n being the depth of the triangle, the middle entry being one, and all the other entries being zero. The binomial coefficients are then computed by summing the previous values of the tree. We are using matrices as a framework to compute this mathematic concept. Here is the code:

|  |
| --- |
| def bicoef(n, m, memory={}): |
|  | """ |
|  | find the binary coefficient C(n,m) |
|  | using the recursive Pascal's identity: |
|  | C(n,m) = C(n-1,m-1) + C(n-1,m) |
|  |  |
|  | Note: this is not an efficient method because |
|  | the recursive function calls result in duplication |
|  | of many calculations. |
|  | """ |
|  |  |
|  | """ |
|  | Tracing it out for 4choose2: |
|  | bicoef(4,2)= bicoef(3,1,\_) + bicoef(3,2,\_) |
|  | / \ / \ |
|  | bicoef(2,0) bicoef(2,1) bicoef(2,1) bicoef(2,2) |
|  | does not get added / \ / \ | |
|  | as a key because bicoef(1,0) bicoef(1,1) bicoef(1,0) bicoef(1,1) | |
|  | base-case is reached / \ / \ | |
|  | / / \ / \ | |
|  | 1 + 1 + 1 + 1 + 1 + 1 = 6 |
|  | """ |
|  | #base cases |
|  | if(n < 0): |
|  | return 0 |
|  | if(m < 0): |
|  | return 0 |
|  | if(m > n): |
|  | return 0 |
|  | if(n == 0): |
|  | return 1 |
|  | if(m == 0): |
|  | return 1 |
|  | if(m == n): |
|  | return 1 |
|  | #to store the previous computed values of n and m to update n and m for next computation |
|  | if (n,m) in memory.keys(): |
|  | #print("a ",memory.keys()) |
|  | #print("b ",memory) |
|  | return memory[(n,m)] |
|  | #print("c ",bicoef(n-1,m-1, memory)) |
|  | #print("d ",bicoef(n-1,m,memory)) |
|  | memory[(n,m)]=bicoef(n-1,m-1, memory)+bicoef(n-1,m,memory) |
|  | #print("e ", memory) |
|  | return memory[(n,m)] |

|  |  |
| --- | --- |
|  | P\_tri = [[bicoef(n,m) for m in range(N+1)] for n in range(N+1)] |

The last line of code constructs a matrix representing pascal’s triangle by iteratively invoking a bionomial coefficient generator function.