Coef

Function that receives a year as an input

We call the previously defined $getio()$ function and assign **IOmatrix** and **output** with its results.

Then assign **n** the value of the number of elements(sectors) in the matrix. We repeat this process in this the next functions in order to let our program to work with any matrix of any size.

Create an empty matrix **result** of size $n \times n$

Nested for to iterate through the entire matrix **result**, i being the columns and j the rows

Using the definition of Technical Coefficient Matrix: $A_{ij} = \frac{Z_{ij}}{x_j}$
where A is the result matrix, Z is the Input-Output Matrix and x is the output vector we fill the matrix with its values.

Finally, we return the matrix as the result of our *coef* function

leon

Function that receives a matrix as an input

Again, define **n** as the number of sectors in our matrix

Using the definition on a Leontief Inverse Matrix: $x = (I - A)^{-1}$
where x is the result matrix, I is an Identity matrix of size n (so you can substract A from it), A is our input matrix (in our case, a technical coefficient matrix)

We're using the numpy function $linalg.inv()$ which takes a matrix and returns its inverse for ease of use

highest_impact

Function that receives a matrix as input

This function adds each column and returns the index of the column that had the highest sum (?)

We declare **indexHighest** which is the index of the column with the current highest sum and its 0 by default

We also declare **highestColSum** which is the value of the sum of the current highest column

Nested for, to traverse the entire matrix in order.

At the end of the nested fors, the values of our variables **indexHighest** and **highestColSum** should be correct

The variable **accum** is used to store the current accumulated sum of a column, its declared between the fors so its reset to zero when we go to the next column

The operator $+=$ is used as a "shortcut", its the same as saying *accum = accum + mat[row,column]*

```
accum+= mat[row,column]     is equivalent to     accum = accum + mat[row,column]
```

The if is used to check if the sum of our current column is greater than our previous highest, if it is we update the value of our **indexHighest** and **highestColSum** variables

Finally we return **indexHighest**

minaij and maxaij

Both are basically the same function with just a few changes and opposite results.

Both functions receive a list of matrices *mats* and *i,j* which correspond to a position in a matrix

Minaij returns the minimum value found in all the matrices in said list at the position *[i , j]*

Maxaij returns the maximum value found in all the matrices in said list at the position *[i , j]*

We create a variable *Max/Min*

simA

Function that receives a list of matrices as input

Define **n** by using the first matrix we have on the list

Create an empty matrix **result** of size $n \times n$

Nested for to traverse the entire **result** matrix

Our goal is to fill said matrix for each position $[i, j]$ where each value is uniformly distributed random number between the minimum and maximum value found in that same position in all the matrices.

To achieve this we use our previously defined functions maxaij and minaij

After all the matrix has been filled, we return the resultant matrix as an output of the function

CreateSimulations

Function that takes the parameter *times* which is the number of simulations we are going to create, and mats (list of matrices)

Define **n** by using the first matrix we have on the list

create the list **simulations**, which is a list of the result of running the *highest_impact* function on a simulated matrix created by using our other function *simA*. We do this "*times*" times and store the results in this **simulations** list.

Then, we create a results list filled by zeros where we will count the times each sector is selected as the most impactful by our *highest_impact* function.

In the next for "for elem in simulations" we iterate through the elements of simulations and add 1 to results[elem]. At the end of the for, **results** would represent the number of occurrences of each sector being the most impactful for all simulations. **Ex.**   [10, 5, 4, 0, 0, 0] means that Sector 0 was the most impactful sector in 10 different simulations, Sector 1 in 5 simulations, Sector 2 in 4 and the next Sectors weren't impactful in any simulations at all.

In "for i in range(n)" we just modify the results list by dividing all its elements by **times** so the new values represent percentages.

Finally we return **simulations** and **results**