Computing Matrix Determinant

The determinant of a matrix is a very important concept in mathematics and coordinate geometry. The famous German polymath Gottfried Leibniz invented the notion of the determinant of a square matrix for the first time and gave a formula to compute it. The idea is that you take an element from each row and column of the matrix where neither the row index nor the column index can repeat. In other word, all row and column indices must be unique. This gives us exactly n elements for a permutation of rows and columns of the matrix. Then you multiply the elements in the permutation to get a product P. Afterwards, you multiply the product by +1 or -1 depending on the **sign** of the permutation of columns, which we will explain in a moment. Once you add all such products, you get the determinant.

If we take elements from the row indices of an n-by-n matrix sequentially from 0 to n -1, then there is n! permutations possible of the column indices where each index is different. Therefore, the determinant of the matrix is a sum of n! products. The equation of determinant calculation may sound complex. However, the whole process can be broken into a three-step algorithm that we are going to investigate now. This should serve as a good exercise for implementing a complex algorithm piece by piece using simple parts.

Step 1: output all permutations of numbers from 0 to n – 1

We can output the permutations of 0 to n – 1 using a recursive function. First, we will fill an array of size n from 0 to n – 1 incrementally. Then in a recursive routine, that will start from index i = 0, we will swap the element at i^{th} index with the element at any of j = i + 1 to n – 1 index. Then we will call the recursive routine for the next index. After the nested call returns, then we will restore the swapped elements at index i and j. The recursive routine will output a permutation when it reaches index n – 1. Below is a Python code of the permutation printing algorithm.

```
def permute(i, array):
  length = array.shape[0]
  if i == length - 1:
    # print a permutation when the recursive call reach an end
    print(array)
  else:
    for j in range(i, length):
      # swap elements at index i and j
      array[i],array[j] = array[j],array[i]
      # call the same function recursively
      permute(i + 1, array)
      # swap elements at index i and j again to restore the old order
      array[i],array[j] = array[j],array[i]
```

For example, the above function should be called as follows to print all permutation of 0 to 3.

```
arraySize = 4
columns = np.zeros([arraySize], dtype=int)
for i in range(arraySize):
  columns[i] = i
permute(0, columns)
```

Step 2: compute the sign of a permutation of column indices

The sign of a permutation of numbers from 0 to n – 1 is the parity of the number of inversions inside the permutation. Here the inversion count for a number x in the permutation is the count of numbers that are larger than x that appear in the permutation before x. We can write a simple code using double for/while loop to compute the parity of an input permutation as follows:

```
def getParity(array):
  length = array.shape[0]
  totalInversions = 0

  # iterate over all the elements of the array
  for i in range(length):
    currentValue = array[i]
    # count the number of inversion for ith element
    iInversions = 0
    for j in range(i):
      if array[j] > currentValue:
        iInversions += 1
    # add the inversion of the current element to total inversions
    totalInversions += iInversions

  # return the parity of the inversions
  return totalInversions % 2
```

Step 3: use permutations to compute determinant of the matrix

We now know how to compute all permutations of unique column indices and also the sign of a permutation. Then writing an algorithm for computing a product of n elements with unique row and column indices is simple. We will write a function that will take the matrix and a column permutation as input and return the product multiplied by sign. A Python implementation is as follows.

```
def getProduct(matrix, columnPermutation):
  rowCount = matrix.shape[0]
  colCount = matrix.shape[1]
  productWithoutSign = 1

  # iterate over the rows incrementally
  for r in range(rowCount):
    # retrieve  the column index from the column permutation
    c = columnPermutation[r]
    # multiply the specific element at row,column index with the current product
    productWithoutSign *= matrix[r][c]

  # get the parity of the column permutation
  parity = getParity(columnPermutation)
  if parity == 1:
    return productWithoutSign * -1
  else:
    return productWithoutSign
```

Now we have all the pieces needed to complete the implementation of matrix determinant calculation. What we will do is:

1. Create a global variable called determinant and initialize it to zero.
2. Then update the recursive function for permutation to take the matrix as the third input, and
    a. Instead of printing a permutation, we will compute the product of the unique row-column element sequence represented by the permutation.
    b. Then add the product to the global variable.
3. At the end of recursion, the global variable will have the matrix determinant.

Below is the updated recursive function.

```
# global variable for determinant
determinant = 0

# name of the permutation function is changed and it also take the matrix
# as an input
def computeDeterminant(i, array, matrix):
  length = array.shape[0]
  if i == length - 1:
    # instead of printing the column permutation; compute product here
    product = getProduct(matrix, array)
    # add product to the global determinant
    global determinant
    determinant += product
  else:
    for j in range(i, length):
      # swap elements at index i and j
      array[i],array[j] = array[j],array[i]
      # call the same function recursively
      computeDeterminant(i + 1, array, matrix)
      # swap elements at index i and j again to restore the old order
      array[i],array[j] = array[j],array[i]
```

Below is a driver code to experiment the algorithm.

```
# driver code
import random
print(random.randint(3, 9))
n = 4
matrix = np.zeros([n, n], dtype=int)
columns = np.zeros([n], dtype=int)
for r in range(n):
  columns[r] = r
  for c in range(n):
    matrix[r][c] = random.randint(1, 10)
determinant = 0
computeDeterminant(0, columns, matrix)
print(matrix)
print(determinant)
```