Comparative study of row major ordering and column major ordering in accessing elements of 2 Dimensional Arrays

Yogita Yashveer Raghav, Assistant professor, K.R Mangalam University, SohnaRoad, Gurgaon
ygtraghav@gmail.com

Abstract : ***The main focus of this paper is to compare the execution time between row major ordering and column major ordering for accessing matrix elements. The aim is to design a program, which generates two matrices with various dimensions, and accessing the matrix elements using both ways row major and column major ordering.. The execution time of each algorithm is recorded to evaluate the performance of each algorithm. The programming language used this project is C Language. The overall finding is that the row major ordering algorithm is more efficient than column major ordering on large size of matrices. However, in scientific computing, memory has to be considered. To implement this we need need large size of matrix.***

Keywords: ***Row major ordering, Column major ordering, Two-dimensional arrays***

Introduction: Before starting discussion about multi-dimensional array, we need to know about the functioning of one dimensional array, like how the elements are stored and accessed .For batter understand of this we will understand it with an example:

A1, A2, A3….An are the contents that I need to store in one dimensional array, so it will be stored as shown in the figure 1.

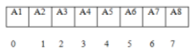


Figure 1

One thing you need to remember is, no matter what type of array it is (1D or multi-dimensional), they will be always stored linearly in the memory space as above.

For accessing 1D elements we need the index, or number at which the element is stored. For example if we need to store any value at the 4th position in the array then the syntax in C language would be like:

A [4] =10;

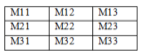
To find the address of any element in one dimensional array is very easy. Just we need to know the Base address and the type of elements that are stored in array. Suppose the element of which we need to find the address is 6 th element of array. We assume that the base address is 1000, and type is integer, then

Base address+element of which we need address* Size of integer

Address of 6th element= 1000+6*2=1012

Two-dimensional arrays are generally arranged in memory in row-major order (for C, Pascal etc) or column-major order (for FORTRAN).

Let's jump to the case of multi-dimensional arrays now. Imagine I've a 3×3 matrix like this:



M11	M12	M13
M21	M22	M23
M31	M32	M33

Figure 2

The real problem of storing elements arises here, since elements have to be stored linearly in the memory space, we have many possible ways to store them. Here are some of the possibilities I could've had for the matrix above.

1. M11 M13 M12 M21 M23 M22 M31 M33 M32 M33
2. M11 M22 M33 M12 M32 M13 M23 M31 M32 M21

… and I could go on filling randomly depending on the no. of elements I've in my 2-D array. Out of all these possible ways, there are two main ways of storing them, they are:

Row Major Ordering
Column Major Ordering

Row Major Ordering:

In this method, the elements of an array are filled up row-by-row such that the first row elements are stored first, then the second row and so on. Most of the high level programming languages like C/C++, Java, Pascal, etc uses this method for storing multidimensional arrays.

As you can see in the 3×3 Matrix array we have above, I want to know at what position is my element A12 located in the linear memory space. This position is called the OFFSET.

One way to do it is to make up a linear diagram like the one below, count manually using 0 as the starting index and go on till the element I need. But imagine you are given an array like A[20][40], definitely you can't go over all these elements if you wanted to know, say where A[15][20] is located.

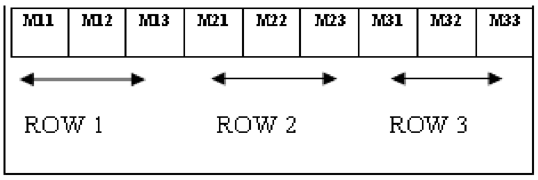


Figure 3

For this, we resort to a fairly easy mathematical method of calculating the offset. Here is the formula to calculate the offset for row major ordering.

Offset for row major ordering

$$\text{OFFSET} = (\text{row} \times \text{NUMCOL}) + \text{col}$$

We will elaborate it with the help of a example. Consider we want to search the address of element [2][1].First count how many rows and columns we have crossed, 2 rows and 1 column, then how many cells in each row i.e. 3,so 2*3=6 and column =1 so 6+1=7.So total 7 cells are there. So multiply it with 2 or 4 whatever the storage size of data type .Here we consider it 2 so 7*2=14, add it with the Base address, here we consider it 1000.

Address of [2][1]=(2*3+1)*2+1000 =1014.

Column Major Method

In Column Major ordering, all the elements of the first column are stored first, then the next column elements and so on till we are left with no columns in our 2-D array.

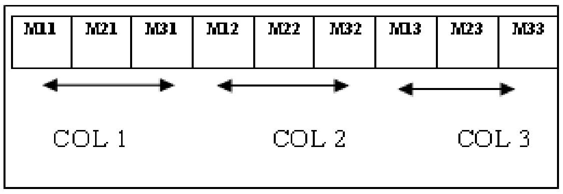


Figure 4

Offset for column major method

$$\text{OFFSET} = (\text{col} \times \text{NUMROW}) + \text{row}$$

Calculating address of a particular element

Depending on the ordering method the elements are stored in the memory, we will get different positions of an element in the linear memory and consequently a different address for each method. To calculate the address we use the following procedure:

Step 1: Get the offset value of the element under consideration; make sure you use the correct formula.

Step 2: Multiply the offset with the size of the element's data type.

Step 3: Add this to the base address to get the final address.

We will elaborate it with the help of a example.
Consider we want to search the address of element [2][3].First count how many rows and columns we have crossed, 2 rows and 3 column, then how many cells in each column i.e. 3,so 3*3=6 and rows =2 so 6+2=8.So total 8 cells are there. So multiply it with 2 or 4 whatever the storage size of data type .Here we consider it 2 so 8*2=16,add it with the Base address, here we consider it 1000.
Address of [2][3]=(3*3+2)*2+1000 =1022.

Performance analysis of Row major and Column major order of accessing arrays elements in C Program

In computing, row-major order and column-major order are methods for storing multidimensional arrays in linear storage such as random access memory.

The two mentioned ways differ from each other with respect to the order in which elements are stored contiguously in the memory. The elements in row-major order are arranged consecutively along the row and that in the column-major order are arranged consecutively along the column

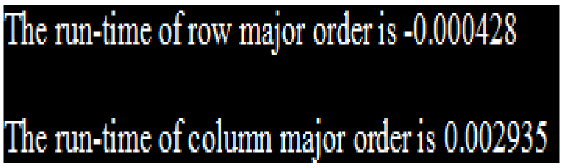
As exchanging the indices of an array is the essence of array transposition, an array stored as row-major but read as column-major (or vice versa) will appear transposed. As actually performing this rearrangement in memory is typically an expensive operation, some systems provide options to specify individual matrices as being stored transposed. The programmer must then decide whether or not to rearrange the elements in memory, based on the actual usage (including the number of times that the array is reused in a computation).

Note that the difference between row-major and column-major order is simply that the order of the dimensions is reversed. Equivalently, in row-major order the rightmost indices vary faster as one steps through consecutive memory locations, while in column-major order the leftmost indices vary faster.

Below program illustrates that row major order storing of arrays in C is more efficient than column-major order (though Pascal and FORTRAN follows column major order):

```
#include <stdio.h>
#include <time.h>
int sample[1000][1000];
void main()
{
int i, j;
clock_t start, stop;
double d = 0.0;
start = clock();
for (i = 0; i < 1000; i++)
for (j = 0; j < 1000; j++)
sample[i][j] = sample[i][j] + (sample[i][j] *
sample[i][j]);
d = (double)(stop - start) /
CLOCKS_PER_SEC;
printf("The run-time of row major order is
%lf\n", d);
start = clock();
for (j = 0; j < 1000; j++)
for (i = 0; i < 1000; i++)
sample[i][j] = sample[i][j] + (sample[i][j] *
sample[i][j]);
stop = clock();
d = (double)(stop - start) /
CLOCKS_PER_SEC;
printf("The run-time of column major order is
%lf", d);
}
```

OUTPUT:



```
The run-time of row major order is -0.000428

The run-time of column major order is 0.002935
```

Conclusion: Performance depends upon the way that how we are going to access the elements and how the elements of matrix are represented and stored in memory. Often a matrix is stored in row-major order, so that consecutive elements of a row are contiguous in memory. Reading memory in contiguous locations is faster than jumping around among locations. As a result, if the matrix is stored in row-major order, then iterating through its elements sequentially in row-major order may be faster than iterating through its elements in column-major order. Of course, if the matrix is stored in column-major order then iterating through its elements sequentially in column-major order may be faster than iterating through its elements in row-major order.

References:

[1] Juby Mathew, 2 Dr. R Vijaya Kumar,Comparative Study of Strassen's Matrix Multiplication Algorithm, ISSN : 0976-8491 (Online) | ISSN : 2229-4333 (Print), IJCST Vol. 3, Issue 1, Jan. - March 2012

[2] D. G. Shin, and K. B. Irani, "Fragmenting
relations horizontally using knowledge based
approach," IEEE Transactions on Software Engineering (TSE), Vol. 17, No. 9, pp. 872–883, 1991.

[3] https://www.geeksforgeeks.org

[4] http://www.cbseguy.com

[5] www.researchgate.net

[6] www.programmingsimplified.com

[7] https://en.wikipedia.org.