***Laboratorio 1: Tipos Abstractos de Datos***

***Diseño e implementación***

Se desea implementar una clase que represente una matriz bidimensional de números reales en doble precisión que permita realizar algunas operaciones de álgebra lineal básica.

Deben ofrecerse como mínimo las siguientes funcionalidades:

* Constructor a partir de número de filas y columnas. Todos los valores de la matriz se inician a cero. La matriz almacena el número de filas y de columnas y un búfer en los que almacenan todos los elementos de la matriz. El búfer debe ser un único array unidimensional reservado en memoria dinámica.

matrix a{n,n};

* Constructor por defecto. Se crea una matriz de cero filas y cero columnas que no consume memoria dinámica, pero sobre la que posteriormente se puede copiar otra matriz.

matrix::matrix(**int** i, **int** j) :  
 rows\_{(i>0)?(i):0},  
 cols\_{(j>0)?(j):0},  
 buffer\_{((i==0) && (j==0))?(**nullptr**):**new double**[i\*j]{}}{std::cout<< **"CONSTRUCTOR POR DEFECTO"** << **"\n"**;}

* Debe soportar operaciones de copia y movimiento.

matrix(**const** matrix & v);  
 *//asignacion copia*matrix & **operator**=(**const** matrix & v);  
  
 *//contructor movimiento*matrix(matrix && m) :  
 buffer\_{m.buffer\_},  
 rows\_{m.getRows()},  
 cols\_{m.getCols()}  
{std::cout<< **"CONSTRUCTOR DE MOVIMIENTO"** << **"\n"**;  
 m.buffer\_ = **nullptr**;  
 m.rows\_ = 0;  
 m.cols\_ = 0; }  
 *//asignacion movimiento* matrix & **operator**=(**const** matrix && m);

* Destructor que debe liberar la memoria que pueda ser propiedad de la matriz.

~matrix() { **delete** []buffer\_; }

* El operador paréntesis se sobrecargará con dos argumentos para acceder a una posición de la matriz.

**double operator**()(**int** i, **int** j) **const** { **return** buffer\_[i \* cols\_ + j]; }

* Los operadores + y \* se sobrecargarán para implementar la suma y el producto de matrices.

Sobrecarga de operador +

matrix & matrix::**operator**+(**const** matrix & m){  
 **if**(rows\_ == m.getRows() && cols\_ == m.getCols())  
 **for**(**int** i = 0 ; i < m.getSize() ; i++){  
 **this**->buffer\_[i] = **this**->buffer\_[i] + m.getBuffer()[i];}  
 **return** \***this**}

Sobrecarga de operador \*

matrix & matrix::**operator**\*(**const** matrix & m){  
 **if**(cols\_ == m.getRows()) {  
 matrix aux{rows\_,m.getCols()};  
 **int** z = 0;  
 **int** zIni = 0;  
 **int** k=0;  
 **for** (**int** i = 0; i < getSize(); i++) {  
 zIni = k \* aux.cols\_;  
 z = zIni;  
 **if**((i-1) % aux.cols\_ == 0){  
 z = zIni;  
 i = z;}  
 **for** (**int** j = 0; j < m.getSize(); j++) {  
 **if**(j==0)k++;  
 aux.set(z, aux.get(z) + get(i) \* m.get(j));  
 z++;  
 **if**((j+1) % aux.cols\_ == 0 ){  
 i++;  
 z = zIni;}}}  
 \***this** = aux;}  
 **return** \***this**;}

***Evaluación***

Se implemento un programa en el cual se generaron tres matrices cuadradas con un valor constante n, con sus valores iniciales a cero, posteriormente se rellenaron con números aleatorios con una distribución normal de media 10.00 y desviación estándar de 2.5, posteriormente se calcula la operación solicitada, midiendo el tiempo de ejecución y luego visualizando el mismo para evaluar la rapidez de este con las diversas modificaciones de parámetros que se exigen, finalmente se calcula e imprime el valor S que es el promedio de los elementos de la matriz resultado. El código implementado es el siguiente:

**void** evaluacion(){  
 matrix a{n,n};  
 matrix b{n,n};  
 matrix c{n,n};  
 matrix d{n,n};

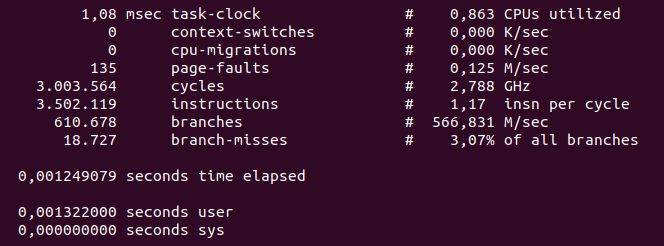
**double** s = 0;  
 std::random\_device rd{};  
 std::mt19937 gen{rd()};  
 std::normal\_distribution<> dis(10, 2.5);  
 **for** (**int** i = 0; i < a.getSize(); ++i) {  
 a.set( i , dis(gen) );  
 b.set( i , dis(gen) );  
 c.set( i , dis(gen) ); }  
 **auto** start = chrono::high\_resolution\_clock::now();  
 d = a + b \* c;  
 **auto** end = chrono::high\_resolution\_clock::now();  
 chrono::duration<**double**> diff = end-start;  
 cout << **"Tiempo de cálculo de la matriz D = "** << diff.count() << endl;  
 **for** (**int** i = 0; i < d.getSize(); ++i) { s = s + d.get(i); }  
 s = s/(d.getSize());  
 cout << **"\n"** << **"s = "** << s << **"\n"**; }

**Resultados**

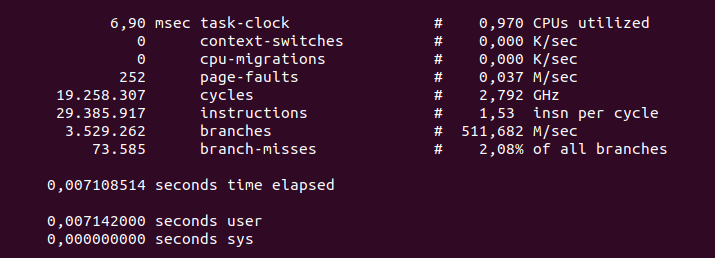
A continuación, se presentan los distintos resultados obtenidos para los distintos valores de las matrices.

**Sin operaciones de movimiento**

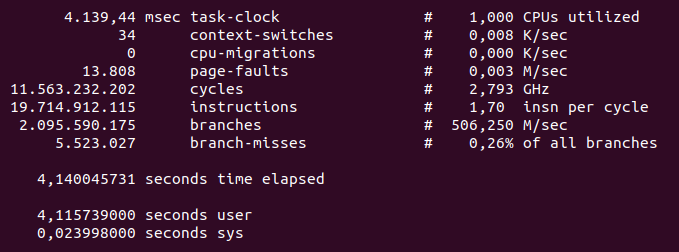
Para n = 10. Tiempo de cálculo de la matriz D = 1.4169e-05



Para n=100. Tiempo de cálculo de la matriz D = 0.00425548

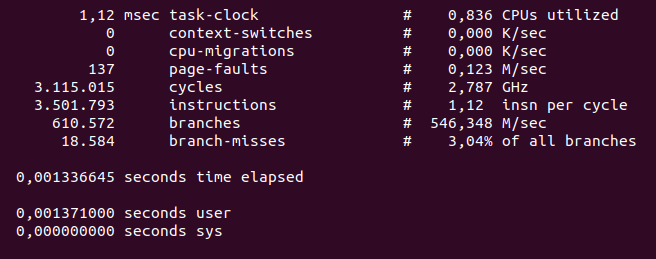


Para n = 1000. Tiempo de cálculo de la matriz D = 4.00756

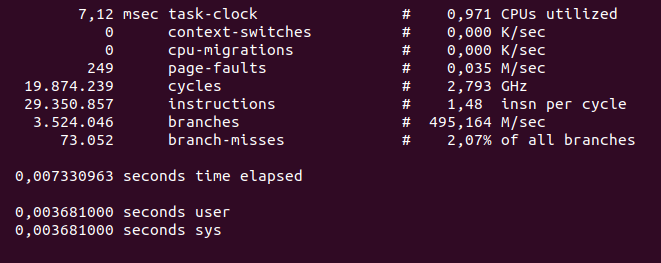


**Con operaciones de movimiento**

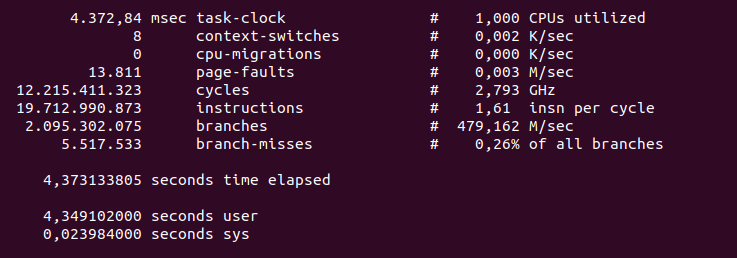
Para n = 10**.** Tiempo de cálculo de la matriz D = 2.0546e-05



Para n = 100 Tiempo de cálculo de la matriz D = 0.00457016



Para n = 1000. Tiempo de cálculo de la matriz D = 4.25516



Evaluando las distintas ejecuciones se aprecia que con movimiento la ejecución es más rápida. La diferencia se aprecia mejor a medida que se aumenta el valor del tamaño.

En la carpeta de la entrega se encuentran los siguientes archivos:

• Dos carpetas comprimidas con movimiento y sin movimiento

• el archivo de la memoria Laboratorio 1.