Manual de uso de la librería matrix.h

La librería matrix.h permite al usuario hacer uso de la estructura tipo `matrix`, cuya funcionalidad reside en poder trabajar en C con este tipo de estructura matemática de forma sencilla y sin tener que preocuparse de la inicialización correcta de arrays de arrays cada vez que se desea instancia una matriz. También cuenta con un conjunto de operaciones aritméticas y de manejo de datos que facilitan aún más su uso.

Esta librería forma parte del trabajo de fin de grado de Álvaro González Méndez, estudiante de ingeniería informática en Escuela Técnica Superior de Ingenieros Informáticos de la Universidad Politécnica de Madrid. El código y la documentación de esta se encuentra en el repositorio de GitHub: https://github.com/itsTwoFive/gml_nn.

Podemos dividir los métodos y operaciones de la librería en los siguientes campos

- **Creación y borrado** de matrices:
  - Asignación de espacio dinámico
  - Liberado de espacio dinámico
- **Manejado de los valores** de la matriz:
  - Asignación de valores
  - Inicialización de matriz con todos los valores en cero
  - Inicialización de matriz con todos los valores aleatorios
  - Desplazado de dirección de valor dentro de la matriz
- **Representación de valores** de la matriz:
  - Mostrado por consola de la matriz
- **Operaciones aritméticas** de la matriz:
  - Suma
  - Resta
  - Multiplicación matricial
  - Multiplicación escalar
  - Suma por columnas
- **Transformaciones** de tipo
  - Transformación de matriz a array de arrays
  - Transformación de array de arrays a matriz

El tipo de datos `matrix` implementado en la librería es definido de la siguiente forma:

```
typedef struct {
    int cols;
    int rows;
    double *d;
}matrix;
```

Donde `cols` es un entero que representa el número de columnas que se encuentran en la matriz, `rows` el número de filas y el puntero a número real de precisión double `d` actúa como cadena de números reales y antes de usar una matriz debe ser inicializado mediante el asignado de memoria dinámica.

Interfaz de uso (listado de métodos)

```
matrix *mat_alloc(int rows, int cols);
```

La rutina **mat_alloc** crea una estructura de tipo matrix de tamaño `rows` x `cols`, reservando en el heap (montículo) de la computadora memoria dinámica suficiente para poder almacenar todos los valores que permitiría una matriz del mismo tamaño. Si la operación falla durante la asignación de memoria se mostrará un error indicando que esto ha sucedido. El espacio reservado se da por: `sizeof(double) * rows * cols`. Después de la asignación de espacio, se establecen todos los valores de la matriz a cero. Finalmente, la rutina acaba devolviendo un puntero a la estructura creada.

```
void mat_free(matrix* mat);
```

Este método vacío permite liberar el espacio reservado a la matriz apuntada por el puntero `mat`, es imprescindible hacer uso de este cada vez que se deja de usar un tipo matrix ya que si existen muchas instancias de la estructura se puede llegar a desbordar la memoria del montículo. Una buena práctica para la programación usando los dos métodos **mat_alloc** y **mat_free** es igualar el número de veces que se usan en el código ya que así cercioramos que no se quedan reservadas en memoria dinámica estructuras que se han dejado de usar.

```
double* mat_seek(matrix m, int i, int j);
```

Esta función devuelve un puntero que apunta dentro de la matriz `m` a la posición `i` fila y `j` columna. El uso de esta puede ser útil para recolectar un valor o para modificar este. Se recomienda usar la función **mat_set_number** si solo se quiere modificar una vez puesto que puede ser más sencillo y acorta el código.

```
void mat_set_number(matrix m, int i, int j, double value);
```

La rutina **mat_set_number** permite al programador alterar el valor dentro de la celda `i` x `j` de la matriz `m` al nuevo valor `value`. Esta función es ideal para cambiar valores de forma momentánea, si se va a alterar un valor varias veces se recomienda usar **mat_seek** para obtener el puntero de la posición y trabajar sobre este ya que solo se realiza una operación de recolectado de datos dentro de la cadena de reales de esta forma.

```
void mat_print(matrix m);
```

La función vacía **mat_print** imprime en la consola de la maquina donde se este ejecutando el Código la información sobre la matriz `m` (parámetros `rows` y `cols`) seguidos de los valores contenidos en la matriz. Para una matriz 4x3 con todos los valores a 0 el resultado de ejecutar la función **mat_print** seria:

```
[4, 3]
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
```

```
void mat_set_zeros(matrix m);
```

La rutina **mat_set_zeros** reestablece todos los valores de la matriz `m` a cero.

```
void mat_randf(matrix m);
```

El método **mat_randf** recorre la estructura `m` de tipo `matrix` y establece todos los valores de esta de forma aleatoria entre -1.0 y 1.0, estos límites se pueden cambiar cambiando el valor de las macros `MIN_R` y `MAX_R` respectivamente de la cabecera de la librería. La función hace uso del método de la librería stdlib.h `rand()`, si el programador quiere que los resultados siempre sean los mismo debe establecer su propia semilla con el método `srand(int seed)` de la misma librería para replicar los resultados.

```
void mat_sumf(matrix m1, matrix m2, matrix *result);
```

La función **mat_sumf** permite realizar la suma de valores 1 a 1 entre dos matrices `m1` y `m2`, el resultado queda guardado en la estructura tipo matrix apuntada por el puntero `result`. Antes del uso de la función **mat_sumf** es importante alocar el espacio correcto a la matriz en la posición `result` con el método **mat_alloc**. Importante recordar:

$$\underset{3\times 2}{\begin{bmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{bmatrix}}_{m1} + \underset{3\times 2}{\begin{bmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{bmatrix}}_{m2} = \underset{3\times 2}{\begin{bmatrix} a_1+b_1 & a_4+b_4 \\ a_2+b_2 & a_5+b_5 \\ a_3+b_3 & a_6+b_6 \end{bmatrix}}_{*result}$$

Si el espacio no esta bien reservado o las matrices a sumar no son del mismo tamaño se mostrará por el canal de error un mensaje especificando que se ha cometido un error a la hora de realizar la operación de suma.

```
void mat_subsf(matrix m1, matrix m2, matrix *result);
```

La rutina **mat_subsf**, al igual que la función **mat_sumf** recibe dos matrices de entrada `m1` y `m2` que deben ser restadas y almacenado su valor en la matriz guardada en el espacio de memoria que el puntero `result` guarda.

$$\underset{3\times 2}{\begin{bmatrix} a_1 & a_4 \\ a_2 & a_5 \\ a_3 & a_6 \end{bmatrix}}_{m1} - \underset{3\times 2}{\begin{bmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{bmatrix}}_{m2} = \underset{3\times 2}{\begin{bmatrix} a_1-b_1 & a_4-b_4 \\ a_2-b_2 & a_5-b_5 \\ a_3-b_3 & a_6-b_6 \end{bmatrix}}_{*result}$$

También se deben respetar los tamaños necesarios para la resta (todas las matrices deben compartir el mismo número de filas y columnas) o sino se mostrará un mensaje de error especificando que se ha cometido este.

```
void mat_productf(matrix m1, matrix m2, matrix *result);
```

El método **mat_productf** realiza la multiplicación matricial entre dos matrices `m1` y `m2`, el resultado queda guardado en la estructura tipo matrix apuntada por el puntero `result`. Antes del uso de la función **mat_productf** es importante alocar el espacio correcto a la matriz en la posición `result` con el método **mat_alloc**. Siguiendo la descripción matemática de la operación de multiplicación entre matrices, el número de columnas en la primera matriz `m1` debe ser igual al número de filas en la segunda `m2`, esto hace que el resultado producido sea una matriz con el número de filas igual a `m1` y el número de columnas a `m2` de la siguiente forma:

$$\underset{1\times 3}{\overset{m1}{\begin{bmatrix}a_1 & a_2 & a_3\end{bmatrix}}} \times \underset{3\times 2}{\overset{m2}{\begin{bmatrix}b_1 & b_4\\ b_2 & b_5\\ b_3 & b_6\end{bmatrix}}} = \underset{1\times 2}{\overset{*result}{\begin{bmatrix}a_1b_1+a_2b_2+a_3b_3 & a_1b_4+a_2b_5+a_3b_6\end{bmatrix}}}$$

Si el espacio no está bien reservado o las matrices a multiplicar no son del tamaño correcto se mostrará por el canal de error un mensaje especificando que se ha cometido un error a la hora de realizar la operación de multiplicación.

```
void mat_dot_productf(matrix m, double coeficent, matrix *result);
```

La función **mat_dot_productf** realiza el producto escalar entre una matriz `m` y un valor `coeficent` de la misma forma que se podría hacer en otros lenguajes de programación que contienen la implementación de matrices como Python o Matlab. El resultado queda guardado como en todas las demás operaciones aritméticas en la matriz apuntada por el puntero `result` (que debe ser alocada previamente con el método **mat_alloc**). La única restricción respecto al tamaño que deben tener las matrices `m` y `*result` es que deben ser del mismo tamaño. Analíticamente podemos observar usando `coeficent` como λ que el comportamiento de la función es:

$$\underset{3\times 2}{\overset{m}{\begin{bmatrix}a_1 & a_4\\ a_2 & a_5\\ a_3 & a_6\end{bmatrix}}} \times \lambda = \underset{3\times 2}{\overset{*result}{\begin{bmatrix}\lambda a_1 & \lambda a_4\\ \lambda a_2 & \lambda a_5\\ \lambda a_3 & \lambda a_6\end{bmatrix}}}$$

```
double mat_column_sum(matrix m, int col);
```

El método **mat_column_sum** realiza el sumatorio de todos los valores guardados en la columna `col` de la matriz `m`, así devolviendo como número real en coma flotante el resultado de este. Es importante no pasarle como parámetro `col` un valor mayor al número de columnas dentro de la matriz `m`, esto puede causar un error de desbordamiento y terminar con la ejecución del código o devolver resultados erróneos.

$$mcs(M,j) = \sum_{i=0}^{M_r} M(i,j)$$

Siendo M_r el número de filas en la matriz M, y j la columna `col` a sumar por completo.

```
matrix* mat_fromarray(int length,double array[]);
```

La rutina **mat_fromarray** permite transformar una cadena de números reales `array` en una matriz de longitud `length` vector con una fila y devuelve un puntero apuntando a la dirección de memoria donde se ha reservado el espacio dinámico de esta. Es importante usar el método **mat_free** cuando se deje de usar la matriz creada.

```
double* mat_toarray(matrix m);
```

El método **mat_toarray** funciona de forma inversa al ya descrito **mat_fromarray**, dada una estructura tipo `matrix` `m`, aloca espacio dinámico para una cadena de reales en punto flotante y devuelve la dirección a esta. Como se sigue usando memoria dinámica es recomendable liberar la matriz si ya no se va a volver a usar con método **mat_free** y cuando se acabe de usar la cadena

de reales llamar a `free` de la stdlib.h para liberar esta también y evitar posibles colapsos del heap.

Ejemplos de código

En esta sección se mostrarán varios ejemplos de uso de la librería con el fin de mostrar al programador que vaya a hacer uso de este código de ejemplo por el cual poder guiarse.

Programa 1. Suma de dos matrices

```c
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    /* Se crean 2 matrices de tamano 4x3 */
    matrix * mat1 = mat_alloc(4,3);
    matrix * mat2 = mat_alloc(4,3);

    /* Establecemos valores aleatorios para todas la entradas de la matriz1 */
    srand(25);
    mat_randf(*mat1);

    /* La segunda matriz la transformamos sus valores a todo unos */
    for (int i = 0; i < mat2->rows; i++)
    {
        for (int j = 0; j < mat2->cols; j++)
        {
            mat_set_number(*mat2,i,j,1.0);
        }
    }

    /* Sumamos los mat1 y mat2 */
    matrix * mat3 = mat_alloc(4,3);
    mat_sumf(*mat1,*mat2,mat3);

    /* Imprimimos valores de ambas matrices y el resultado por consola */
    mat_print(*mat1);
    mat_print(*mat2);
    mat_print(*mat3);

    /* Liberamos la memoria alocada a las tres matrices */
    mat_free(mat1);
    mat_free(mat2);
    mat_free(mat3);

    return 0;
}
```

Programa 2. Multiplicación matricial y multiplicación escalar

```c
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    /* Se crean 2 matrices de tamano 2x5 y 5x3 */
    matrix * mat1 = mat_alloc(2,5);
    matrix * mat2 = mat_alloc(5,3);

    /* Establecemos valores aleatorios ara todas la entradas de ambas matrices */
    srand(25);
```

```
    mat_randf(*mat1);
    mat_randf(*mat2);

    /* Multiplicamos los mat1 y mat2 */
    matrix * mat3 = mat_alloc(2,3);
    mat_productf(*mat1,*mat2,mat3);

    /* Imprimimos valores de ambas matrices y el resultado por consola */
    mat_print(*mat1);
    mat_print(*mat2);
    mat_print(*mat3);

    /* Realizamos una multipplicacion escalar por 10 */
    mat_dot_productf(*mat3,10,mat3);

    /* Volvemos a imprimir el resultado para comprobar su funcionamiento */
    mat_print(*mat3);

    /* Liberamos la memoria alocada a las tres matrices */
    mat_free(mat1);
    mat_free(mat2);
    mat_free(mat3);

    return 0;
}
```

Programa 3. Operación de sumatorio ponderado usado en los algoritmos de feed-forward de las redes neuronales $A = X \times W + B$

```
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    int num_inputs = 3;
    int num_neurons = 4;

    /* Se crean las matrices X, W, B y A */
    matrix *X = mat_alloc(1,num_inputs);
    matrix *W = mat_alloc(num_inputs,num_neurons);
    matrix *B = mat_alloc(1,num_neurons);
    matrix *A = mat_alloc(1,num_neurons);

    /* Se le agregan valores a X */
    *mat_seek(*X,0,0) = 1.5;
    *mat_seek(*X,0,1) = 1.0;
    *mat_seek(*X,0,2) = 2.25;

    /* Se inicializan los valores de W y B de forma aleatoria */
    mat_randf(*W);
    mat_randf(*B);

    /* Realizamos la multiplicacion X * W */
    matrix * temp = mat_alloc(1,num_neurons);
    mat_productf(*X,*W,temp);

    /* Sumamos el resultado de la multiplicacion con B */
    mat_sumf(*temp,*B,A);

    /* Liberamos la matriz temporal */
    mat_free(temp);

    /* Imprimimos el resultado */
    mat_print(*A);
```

```
    /* Liberamos las demas matrices */
    mat_free(X);
    mat_free(W);
    mat_free(B);
    mat_free(A);

    return 0;
}
```

Programa 4. Cálculo de la media de 100 valores aleatorios usando `mat_column_sum`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "matrix.h"

int main(int argc, char const *argv[])
{
    /* Establecemos el numero de valores a sumar*/
    int num_values = 100;

    /* Creamos un vector vertical de tamano 100 */
    matrix * mat = mat_alloc(num_values,1);

    /* Establecemos sus valores a valores aleatorios */
    time_t t;
    time(&t);
    srand((unsigned int) t);

    mat_randf(*mat);

    /* Visualizamos en consola los 100 valores */
    mat_print(*mat);

    /* Realizamos la suma de toda la columna */
    double sum = mat_column_sum(*mat,0);

    /* Dividimos entre el numero de valores para calcular la media*/
    double mean = sum/num_values;
    printf("Media = %f\n",mean);

    /* Liberamos la matriz */
    mat_free(mat);

    return 0;
}
```