Universidad de Castilla La Mancha

Escuela Superior de Informática

Caso de Estudio

Escribir procedimientos en lenguaje C para manipular matrices y crear una librería que pueda integrarse en un programa SWI-Prolog.

Autores: *Alejandro del Hoyo Abad y Sergio Pozuelo Martín-Consuegra*
Grupo: *Cas-ET6*
Asignatura: *Programación declarativa*
Fecha: *3-05-2024*

Índice

1. Introducción

En este trabajo, se explicará el desarrollo de diversos predicados externos para crear una librería externa destinada a la manipulación de matrices, programada en el lenguaje C. Estos predicados externos están diseñados para ser utilizados en SWI-Prolog, un entorno de desarrollo y sistema de implementación del lenguaje de programación Prolog.

En este contexto, podemos definir como predicados externos (*foreign predicates*) a aquellas funciones de C que poseen el mismo número de argumentos que el predicado que representan, es decir, son capaces de interactuar con programas Prolog siendo posible que sean invocadas desde este como si se tratara de cualquier predicado estándar. De este modo, la interfaz SWI-Prolog proporciona a C una inmensa cantidad de funciones y tipos para poder manipular términos de Prolog y poder comunicarse con el núcleo de Prolog.

Por lo tanto, este documento estará dividido en distintas secciones que abarcan distintos aspectos relevantes a la interfaz de lenguaje externo de que usa Prolog (Foreign language interface) tales como: los tipos de biblioteca externos, su compilación y ejecución, los tipos de datos de la interfaz junto con la implementación de los predicados externos y su justificación en el uso de las mismas dentro del contexto de las matrices.

1.1. Enlaces de módulos externos o bibliotecas externas (Linking foreign Modules)

En primer lugar, todo predicado externo que quiera ser utilizado en Prolog debe estar enlazado a un módulo externo para el correcto funcionamiento del programa. En este sentido, se distinguen los siguientes tipos:

- **Enlace estático (*static linking*).** En este caso, el enlace al fichero de SWI-Prolog se lleva a cabo en tiempo de compilación, de forma que tras compilar los predicados externos, se crea un fichero ejecutable que posee tanto esos predicados como el entorno de SWI-Prolog. Por lo tanto, los predicados que se hayan definido ya son parte del entorno, lo que implica que no es necesaria ninguna inicialización para disponer de dichos predicados. Asimismo, este tipo de enlace se considera altamente portable, lo que facilita el proceso de depuración en diversas plataformas. No obstante, es relativamente incómodo de usar y las bibliotecas que necesitas pasar al enlazador pueden variar de un sistema a otro, aunque el programa de utilidad swipl-ld a menudo oculta estos problemas al usuario.

- **Enlace dinámico (*dynamic linking*)** Por otro lado, en las dinámicas el enlace se produce en tiempo de ejecución tras haber inicializado SWI-Prolog. Esto implica que al compilar nuestros predicados externos, se generará en un archivo con extensión de librería dinámica que en el caso de SO Windows será *.dll* y en SO tipo Unix será *.so*. Debido a esto, se deberá de cargar previamente con el predicado *load_foreign_library/1* donde el argumento corresponde al nombre de la librería.

Asimismo, cargar objetos compartidos proporciona compartición y protección, siendo generalmente mejor opción que la del enlace estático.

Es importante tener en cuenta que el tipo de enlace no influye en la implementación de los predicados externos, es decir, el código de los predicados es el mismo independientemente del tipo de enlace. Asimismo, en este informe se explicará en detalle el enlace dinámico con SWI-Prolog puesto que ha sido el que hemos usado.

1.2. Tipos de predicados externos: deterministas y no deterministas

En la interfaz SWI-Prolog se ofrece soporte para predicados deterministas y no deterministas. Los deterministas son aquellos que no permiten distintos puntos de elección, es decir, sólo tienen éxito una única vez. De hecho, este tipo de predicados son los que se han utilizado para el desarrollo de la librería de las matrices de esta práctica.

Por otro lado, los no deterministas sí que permiten varios puntos de elección, pudiendo tener que ser reevaluados en otro punto de decisión, lo que los hace un poco más complicados ya que la función externa necesita información de contexto para generar la próxima solución. De esta forma, el predicado externo tiene que estar preparado para activar simultáneamente más de un objetivo.

2. Compilación y ejecución de la librería dinámica

Antes de detallar los tipos de datos de interfaz y la implementación de los diversos predicados externos en C, creemos que es necesario explicar brevemente cómo se puede compilar y ejecutar dichos predicados. Debido a esto, vamos a proceder a explicar la compilación y la ejecución de estos mediante una librería dinámica.

2.1. Compilación

En primer lugar, antes de utilizar el comando correspondiente, es necesario destacar que para llevar a cabo la compilación de los predicados externos, será imprescindible incluir el archivo *SWI-Prolog.h*, que es el lugar donde se declaran los tipos y funciones de interfaz con *SWI-Prolog*. En este sentido, el comando necesario para realizar dicho proceso es el de *swipl-ld* que está ya instalado junto a la interfaz *SWI-Prolog*, siendo este:

- ***swipl-ld –I/include file.c -o file.o***

En este escenario, el comando actúa como una interfaz con el compilador C *gcc*, que identifica e incorpora los archivos de cabecera necesarios de SWI-Prolog. Por lo tanto, para realizar el enlace y generar la biblioteca dinámica *extlib.so*, combinando el código objeto de los archivos C con las funciones necesarias de las bibliotecas de interfaz tanto de SWI-Prolog como de C, se deberá de utilizar el siguiente comando:

- ***swipl-ld -o -shared dynamic_library.so -o file.o***

En este caso, *file.o* será el archivo o los archivos que han sido generados con el comando anterior y *dynamic_library.so* será el nombre de la librería que utilizaremos para cargar los distintos predicados externos dentro de la interfaz *SWI-Prolog*. En este sentido, la siguiente imagen resume de forma visual lo que se ha explicado previamente:

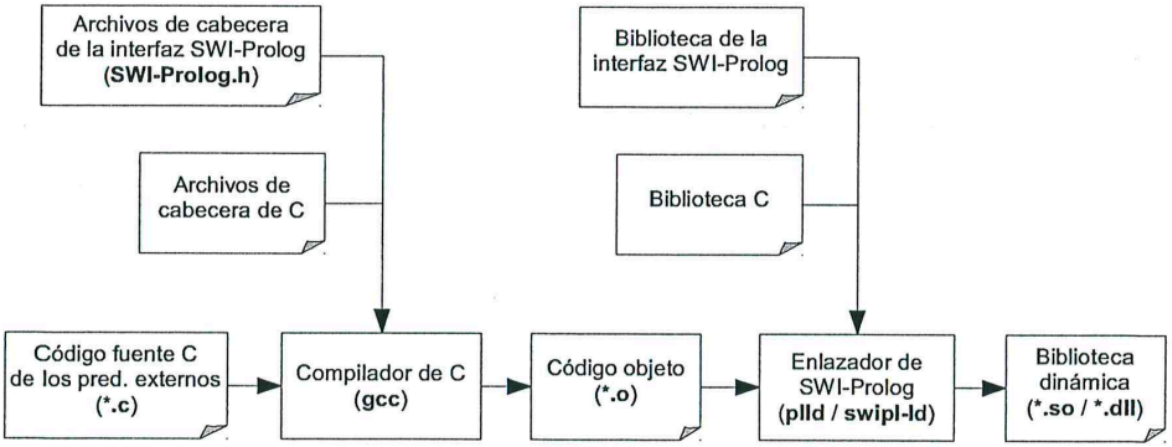

Figura 1: Proceso de compilación de una biblioteca dinámica.

2.2. Ejecución

Para invocar los predicados externos mediante el enlace dinámico, primero se llama al predicado predefinido *load_foreign_library(+library)* donde "library" es el parámetro de ruta del archivo *.so* o *.dll* creado al compilar la biblioteca. En este sentido, cuando se carga dicha biblioteca en memoria, la interfaz SWI-Prolog se encargará de llamar a la función de instalación, la cual registra su nombre y aridad. De este modo, todos los predicados que han sido definidos dentro de la librería estarán disponibles en la interfaz como si se tratara de un predicado nativo.

De forma resumida, el proceso consiste en que el predicado externo utiliza la interfaz con SWI-Prolog para convertir los términos de entrada del predicado a variables para que puedan manejarse desde C. Tras esto, se realizan los cómputos que sean necesarios y finalmente se comunica de nuevo con la interfaz SWI-Prolog para convertir dichos resultados obtenidos a términos que sean devueltos al programa Prolog. Asimismo, dentro de SWI-Prolog es posible cargar más de una biblioteca dinámica, cada uno con diversos predicados externos, siempre que las cabeceras de los predicados de todas las bibliotecas sean únicas. Esto permite el acceso a distintas funciones y predicados como:

- Funciones C de la propia biblioteca externa.
- Funciones C de la interfaz con SWI-Prolog.

- Funciones C de la biblioteca estándar de C, con las que se manejan cadenas, leer o escribir archivos.

- Predicados Prolog de librerías predefinidas de SWI-Prolog.

- Predicados Prolog definidos por el usuario en el programa cargado en memoria.

3. Predicados externos implementados en la librería

Antes de entrar en detalle sobre los tipos de datos de interfaz y la correspondiente explicación de dichos predicados, vamos a enumerar los distintos predicados externos que hemos implementado y que pueden ser usados en la interfaz SWI-Prolog. Debido a esto, es destacable tener en cuenta que para representar una matriz en Prolog, se realizará como una lista de listas. Por ejemplo, la variable: *A = [[0,2,3], [1,-2,3]]*, representa a una matriz que tiene dos filas y tres columnas. De hecho, para nombrar una matriz de una dimensión se deberá hacer de esta forma: *A = [[0,2,3]]*, puesto que si no tendremos un error de sintaxis en el término de Prolog. Asimismo, la matriz devuelta estará compuesta por números de tipo *double* para garantizar la capacidad de trabajar con cualquier tipo de dato numérico.

En esta librería, se han implementado los siguientes:

- ***sumar_matrices (+Matriz1, +Matriz2, -ResultadoMatriz)***. Realiza la adición entre matriz1 y matriz2, y lo devuelve en resultado matriz.

- ***restar_matrices (+Matriz1, +Matriz2, -ResultadoMatriz)***. Realiza la resta entre matriz1 y matriz2 y lo devuelve en resultado matriz.

- ***multiplicar_matrices(+Matriz1, +Matriz2, -ResultadoMatriz)***. Realiza la multiplicación entre matriz1 y matriz 2 y lo devuelve en resultado matriz.

- **transponer (+Matriz, -MatrizTraspuesta)**. Devuelve la matriz traspuesta.

- ***producto_escalar (+Vector1, +Vector2, -Resultado)***. Realiza el producto escalar entre el vector1 y el vector2.

- ***obtener_valor_maximo (+Matriz, -Valor)***. Obtiene el valor máximo de una matriz.

- ***es_diagonal (+Matriz)***. Comprueba si la matriz es diagonal, esto es una matriz en donde todos los elementos menos la diagonal principal son 0.

- ***multiplicar_matriz_por_factor (+Matriz, +Factor, -ResultadoMatriz)***. Multiplica cada elemento de la matriz por un factor y devuelve dicha matriz resultante.

- **dividir_matriz_por_factor (+Matriz, +Factor, -ResultadoMatriz)**. Divide cada elemento de la matriz por un factor y devuelve la matriz resultante. Se ha tenido en cuenta que el factor no sea igual a 0.

- ***sumar_elementos_de_matriz (+Matriz, -Valor)***. Devuelve la suma de todos los elementos de la matriz.

- ***es_matriz_diagonal_superior (+Matriz)*.** Comprueba que debajo de la diagonal de la matriz, todos los elementos son 0.

- ***matrices_mismas_dimensions (+Matriz1, +Matriz2)*.** Comprueba que dos matrices tienen el mismo número de filas y de columnas.

3.1. Tipos de datos de interfaz

En primer lugar, es crucial tener en cuenta que el tipo principal de dato con el que trabaja Prolog se denomina *term_t* el cual es un tipo de dato que representa un identificador para un término en lugar del término en sí. En este sentido, los términos solamente pueden ser representados y manipulados utilizando este tipo, ya que es la única forma segura de garantizar que el kernel de Prolog esté al tanto de todos los términos referenciados por código externo y, por lo tanto, de que permita la recolección de basura y cambios de pila mientras el código externo está activo, por ejemplo, durante una llamada desde C. En este sentido, una referencia de término es un *uintptr_t* en C que representa el desplazamiento de una variable en la pila del entorno Prolog.

Por lo tanto, *tem_t* puede representar todo tipo de valores como: una cadena de texto, una lista, un átomo, un valor *integer* o *double*, un *compound term*, un diccionario o una variable, entre otros.

Por lo tanto, es necesario usar referencias para resultados intermedios, pudiendo estas ser creadas mediante las siguientes funciones:

- **term_t PL_new_term_ref()**. Devuelve una referencia a *term_t*.

- **term_t PL_new_term_refs (+count)**. Devuelve *count* referencias a *term_t*. En este sentido, la primera referencia es la que se devuelve, las otras son t+1, t+2, etc. Esta función se utiliza porque *PL_open_query()* espera los argumentos como un conjunto consecutivo de referencias a términos. Por ejemplo:

```
pl_mypredicate(term_t a0, term_t a1)
{
term_t t0 = PL_new_term_refs(2);
term_t t1 = t0+1;
  ...
}
```

- **term_t PL_copy_ref (term_t from)**: Crea una referencia a un nuevo término, el cual apunta inicialmente a *term_t from*.

Asimismo, otros tipos de datos externos a tener en cuenta pueden ser los siguientes:

- ***atom_t***: Un átomo en representación interna de Prolog. Los átomos son punteros a una estructura opaca. Se utilizan para manejar cadenas y otros tipos de datos.

- ***predicate_t***: Enlace a un predicado de Prolog.

- ***foreign_t***: Tipo de retorno para una función que implementa un predicado Prolog.

- ***functor_t***: Un *functor* es una representación interna de un par nombre/aridad. Se usan para encontrar el nombre y la aridad de un término compuesto, así como para construir nuevos términos compuestos.

3.2. Definiciones de predicados externos

Teniendo ya las nociones básicas sobre los tipos de datos que se utilizan en el *Foreign Language Interface* de *SWI-Prolog*, pasamos a la explicación de cómo definir y utilizar estos predicados externos. En este sentido, todos los predicados externos que se tienen definidos están asociados a una función C, de forma que cuando se invoque desde la interfaz *SWI-Prolog*, este ejecutará la función C y, posteriormente, retornará el control a la interfaz mediante las llamadas *PL_suceed* y *PL_fail* que corresponden a los valores 1 (éxito) y 0 (fallo) respectivamente. Asimismo, esta función debe devolver un valor del tipo *foreign_t* y solo se acepta valores de argumento de tipo *term_t*. La estructura básica es la siguiente:

```
foreign_t pl_expred(term_t tList, term_t tNumber) {
 if (/.../) {
     PL_succeed;
} else
PL_fail;
     }
}
```

3.3. Función de instalación

Para que los predicados externos que hemos definido puedan ser usados en la interfaz *SWI-Prolog*, será imprescindible definir una función de instalación que registre en el núcleo de *Prolog* el nombre y aridad de estos. De este modo, dicha función posee las siguientes características:

- El tipo de retorno de esta se trata de *install_t*.

- No posee ningún parámetro de entrada.

- El nombre de esta función es *install()* o *install* seguido del nombre de la biblioteca.

Teniendo esto en cuenta, dentro de esta función, se llamará a la función *PL_register_foregin(+name, +arity, +function, 0)* por cada predicado externo de la biblioteca. Más concretamente, la explicación de los parámetros de esta función son los siguientes:

- **+*name***: nombre que tendrá el predicado externo a la hora de ser llamado en la interfaz SWI-Prolog.

- **+arity**: representa el número de argumentos de tipo term_t que tiene la función de C

- **+*function*:** es el nombre de la función de C la cual no debe necesariamente coincidir con el nombre que se le proporciona al predicado en SWI-Prolog.

3.4. Análisis y construcción de términos

Para analizar los términos de entrada de un predicado externo y construir los términos de salida existen diversas funciones que permitir conocer el tipo de los términos almacenados en la variable tipo *term_t*, y por otro, relacionan dichas variables con variables de los tipos de estándares de C. En este sentido, son muy útiles para comprobar el tipo de parámetros de entrada en los predicados externos. Algunos de estos términos son los siguientes:

- ***PL_is_atom (+term)***. Devuelve **TRUE** si se trata de un *átomo*.

- ***PL_is_number (+term)***. Devuelve **TRUE** si se trata de un *número*.

- ***PL_is_float (+term)***. Devuelve **TRUE** si se trata de un *float*.

- ***PL_is_string (+term)***. Devuelve **TRUE** si se trata de una cadena de *chars*.

- ***PL_is_compound* (+term)**. Devuelve **TRUE** si se trata de un *functor compuesto*.

Por otro lado, la librería externa también cuenta con funciones predeterminadas para obtener los distintos valores de los términos *term_t* de la interfaz SWI-Prolog. En este sentido, algunas de las funciones más destacables son las siguientes:

- ***PL_get_float (+term, -double_pointer)***: Obtiene un puntero *double* del término *term_t*.

- ***PL_get_integer (+term, -integer_pointer)***: Obtiene un puntero *integer* del término *term_t*.

- ***PL_get_atom_chars (+term, -char_pointer)***: Obtiene un puntero a un carácter *char*.

Del mismo modo, es posible realizar lo opuesto, es decir, guardar en un término term_t un tipo de variable. Esto se hará mediante el predicado *PL_put_<tipo_variable>* (-term, +<*tipo variable*>).

3.5. Manipulación de listas con predicados externos

Por otro lado, es destacable mencionar que la librería externa ofrece funciones para poder trabajar con listas. En este sentido, estas funciones han sido de gran ayuda durante este trabajo puesto que constantemente estamos trabajando con ellas, ya que una matriz se puede representar como una lista de listas en Prolog. Por lo tanto, algunas de las más destacables son las siguientes:

- ***PL_get_list (+list, -head, -tail)***: En este caso, cuando el término list representa una lista no vacía, los términos head y tail guardan la cabeza y la cola de la lista respectivamente, devolviendo True. Asimismo, es muy útil para poder recorrer una lista dentro de un bucle *while* cuando el término list contiene una lista vacía.

- ***PL_put_nil (+term)* y *PL_get_nil (+term)***: Se encargan de comprobar que el término representa la constante de terminación de la lista y guardar la lista vacía en el término term respectivamente.

- ***PL_cons_list (-list, +head, +tail)***: En el término list se guarda la lista resultante de concatenar la cabeza y la cola. En este sentido, los parámetros list y tail pueden ser el mismo término, lo que implica la facilidad de poder construir listas mediante un bucle.

3.6. Función de unificar

Como es bien sabido, el concepto de unificación de términos en Prolog es fundamental, puesto que esto implica encontrar valores para las variables en dos o más términos, de manera que estos se vuelvan iguales. En este caso, se utilizará la función *PL_unify (?term1, ?term2)* en la cual se intentará unificar el *term2* con el *term1* en donde devolverá **TRUE** si es posible hacer dicha operación o **FALSE** si no pueden ser unificables.

Debido a lo previamente mencionado, esta función resulta muy flexible para devolver resultados de un predicado externo, como se puede observar en el siguiente ejemplo implementado. En este caso, el objetivo es realizar un predicado externo en donde se lleve a cabo la suma de todos los elementos de una lista de *integers*. En este escenario, el código correspondiente a esto se ve reflejado en la siguiente imagen:

```c
#include <SWI-Prolog.h>
#include <stdio.h>

foreign_t pl_sum_list(term_t tList, term_t result) {
    term_t tTail = PL_copy_term_ref(tList);
    term_t tHead = PL_new_term_ref();
    int sum = 0;
    int current_value;
    while(PL_get_list(tTail, tHead, tTail)) {
        if(!PL_get_integer(tHead, &current_value))
            PL_fail;
        sum += current_value;
    }
    return PL_unify_integer(result, sum);
}

install_t
install() {
    PL_register_foreign("suma_elementos_de_list", 2, pl_sum_list, 0);
}
```

4. Explicación del código de las matrices

Para llevar a cabo una librería externa que pueda manipular matrices, hemos decidido dividir el código principalmente en dos ficheros distintos. Por un lado, *matricesLogic.c* se encarga de llevar a cabo la lógica de la manipulación de las matrices junto con los distintos métodos auxiliares para llevar a cabo las conversiones. Por otro lado, *matricesProlog*.c utiliza las funciones de la clase *matricesLogic.c* para crear las distintas matrices y poder llevar a cabo las llamadas a los predicados externos. No obstante, de manera general, en todo predicado externo se lleva a cabo las siguientes fases:

1. Comprobación de los tipos de términos de los argumentos de entrada del predicado y conversión de los tipos de SWI-Prolog a la estructura matrix.

2. Obtención de la matriz y realizar las operaciones respectivas con esta.

3. Conversión de la matriz resultante o resultado obtenido para llevar a cabo la unificación con el *term_t* result si es necesario o devolver macro *PL_succeed* o *PL_fail* si se trata de un predicado que verifica alguna característica de la matriz.

Teniendo esto en cuenta, vamos a proceder a detallar más en profundidad como se ha dividido el programa que hemos desarrollado:

4.1. Archivos

En este apartado se explicará los distintos ficheros que se han utilizado para llevar a cabo el proyecto. En este caso, se tienen los siguientes:

4.1.1 definitions.h

Lo más destacable de este archivo es la definición del *struct Matrix* que se compone de los siguientes elementos:

```
typedef struct {
      int rows;
      int columns;
      double* data;
} Matrix;
```

Para este caso, hemos decidido emplear dos atributos para representar las filas y columnas de la matriz, junto con un puntero unidimensional donde almacenaremos los elementos. La elección de utilizar un único puntero en lugar de un puntero doble se basa en mejorar la eficiencia en el acceso a los elementos de la matriz. Además, personalmente hemos encontrado que esta implementación simplifica significativamente el código, ya que evita la necesidad de gestionar múltiples bloques de memoria para almacenar cada fila por separado.

En este contexto, hemos utilizado la siguiente macro para facilitar el acceso y la modificación de los elementos de la matriz:

```
ACCESS(matrix, row, column) matrix -> data[column * matrix -> rows + row]
```

4.1.2 matricesLogic.c

Como se ha mencionado anteriormente, se pueden diferenciar dos tipos de funciones. Por un lado, aquellas que se encargan de hacer manipulaciones con las matrices y, por otro lado, los métodos auxiliares que se encargan de llevar a cabo la correspondiente conversión para poder operar con las matrices de forma correcta. En este caso, se utilizan cuatro:

- ***Matrix* parse_list_of_lists_into_matrix (term_t tList)***. Se encarga de transformar una lista de listas de *Prolog* a una *struct* Matrix de *C*. En este caso, se utilizan los predicados relacionados con las listas para recorrer las respectivas listas y, al ser una lista de listas, es necesario tener un *while* anidado para poder recorrer la lista de cada row.

- ***int parse_matrix_into_list_of_lists (Matrix* matrix, term_t resultListOfLists)***. Se encarga de transformar una *struct* de matrices a una lista de listas para la representación en *Prolog*. En este caso, se hace uso del predicado *PL_new_term_refs*,

puesto que estamos tratando con listas consecutivas y *PL_cons_list* para poder concatenar todas estas listas en el orden correspondiente.

- ***int assign_matrix_row_values_to_list (Matrix* matrix, term_t tList, int current_rows)***. Asigna los valores de una fila de una matriz a una lista que forma parte de la lista de listas del término. En este sentido, este es un método auxiliar para la función *parse_matrix_into_list_of_lists*.

- ***int get_correct_dimensions (term_t tList, int* rows, int* columns)***. Obtiene y comprueba que las dimensiones de una matriz son correctas, es decir, se comprueba que el número de valores que tiene cada lista es el mismo en todas ellas. En este sentido, se vuelve a usar el bucle *while* con el predicado *PL_get_list* para poder obtener dichos valores.

4.1.3 matricesProlog.c

En este caso, solo se utiliza el predicado externo *foreign_t* para los distintos métodos que se desean implementar en la librería externa, donde se emplean los métodos de la clase de la lógica de matrices junto con los métodos auxiliares para poder trabajar con los datos correspondientes. Además, en aquellos predicados en donde se obtiene una matriz o valor resultante, se deberá de proceder a usar el predicado encargado de la unificación de términos. Por último, se hace uso de la función *install_t()* para realizar las correspondientes llamadas y asignar el nombre que tendrán dichos predicados al ser invocadas en la interfaz SWI-Prolog.

4.2. Ejemplos de la implementación de los predicados

Los ejemplos son relativos a los predicados que han sido implementados en la librería externa *matrices.so*. En este sentido, algunas invocaciones de estos predicados en la interfaz SWI-Prolog con la librería *matrices.so* ya cargada son los siguientes:

- **Suma de matrices satisfactoria:**

```
?- sumar_matrices([[1,2,3],[2,4,2.42]],[[1,2,23.2],[1.23,2.2,3]],R).
R = [[2.0, 4.0, 26.2], [3.23, 6.2, 5.42]].
```

- **Multiplicación de matrices satisfactoria:**

```
?- multiplicar_matrices([[1,4,0]],[[2],[-1],[5]],R).
R = [[-2.0]].
```

- **Obtener elemento máximo de una matriz:**

```
?- obtener_valor_maximo([[1,3,2,5],[2.5,3.4,68.45,4]],M).
M = 68.45.
```

- **Comprobar que dos matrices tienen el mismo número de dimensiones**

```
?- matrices_mismas_dimensions([[1,2,3,4],[4,6,5,7]],[[3,4,6,5],[3,2,41,5]]).
true.
```

- **Error en la multiplicación de matrices**

```
?- multiplicar_matrices([[2,3,4,2]],[[1,2,3,4]],R).
Para realizar la multiplicación, asegúrate de que el número de columnas de la primera matriz: 4 sea igual al número de filas de la segunda: 1
false.
```

- **Error en suma de matrices**

```
?- sumar_matrices([[2,3,4,a]],[[1,2,3,4]],R).
Asegúrate que todos los valores que se introduce a la matriz son valores numéricos
false.
```

5. Conclusión

Durante el transcurso de este proyecto, hemos adquirido una comprensión profunda de las capacidades y la versatilidad que ofrece el lenguaje C para interactuar con Prolog a través de una Interfaz de Lenguaje Externo (Foreign Language Interface). Esta experiencia nos ha permitido apreciar cómo podemos combinar eficazmente programas escritos en Prolog y C, abriendo así la puerta a proyectos de gran interés que serían difíciles de realizar utilizando exclusivamente uno u otro lenguaje.

Nos hemos dado cuenta de que aprender a utilizar los predicados externos predefinidos y su sintaxis no ha sido tan desafiante como esperábamos, debido en parte a nuestra base previa en C obtenida en cursos anteriores y a la relativa simplicidad de manipular matrices en este contexto. Sin embargo, nos encontramos con dificultades significativas al principio al organizar el código. Inicialmente, cometimos el error de escribir todo el código de C en un solo archivo, lo que resultó en una mezcla confusa de la lógica de los procedimientos de las matrices y las llamadas a los predicados externos para la interfaz SWI-Prolog. Esta falta de estructura condujo a una repetición excesiva de código y a la aparición de numerosos errores. Esto nos llevó a realizar una reestructuración completa, centrándonos especialmente en la optimización de los métodos auxiliares esenciales.

Algunos métodos importantes que se diseñaron durante el proceso de reestructuración fueron aquellos utilizados para facilitar la conversión de una lista de listas de Prolog al *struct* correspondiente en *C*, así como para revertir este proceso y permitir la modificación de los valores de la matriz a través de punteros. Estos métodos nos han permitido tener un manejo más intuitivo y cómodo a la hora de programar el código evitando así realizar modificaciones directas sobre la matriz.

En definitiva, durante este trabajo no solo hemos aprendido cómo funciona el tipo *term_t* y la comunicación entre *SWI-Prolog* y *C*, sino que también hemos mejorado nuestro manejo con el lenguaje *C*, puesto que para llevar a cabo este proyecto se ha requerido un mayor nivel de organización que para los tipos de programas que estábamos acostumbrados a hacer.