Technische Universität München

Informatics I for MSE

MSE IN8011
Prof. Bjoern Menze, Giles Tetteh
Philipp Kaeß

WS 19/20
Homework 4

Due Date: 01.12.19, 23:59

Release Date: 22.11.19

Homework 4: Matrix Calculations

Topics: Multi-Dimensional Arrays, `Struct`s / `Union`s, Matrix Calculus, Functions

Introduction

In your mathematics module you already have been faced by diverse mathematical operations performed on matrices, e.g. the calculation of a matrix determinant. In this homework, you will create a library of a few important matrix functions. This will not only provide you a tool to check your future matrix calculations you did by hand for mathematics, but also give you a deeper understanding of the calculations itself and especially the multi-dimensional array definition in C.

Even though you should actually already know these operations, the mathematical definitions are given below:

Determinant Calculation (here recursively defined):

- for $A \in \mathbb{R}^{1x1}$ with $A = (\ a\ )$ holds: $\det(A) = a$

- for $A \in \mathbb{R}^{(d+1)\ x\ (d+1)}$ the determinant can be computed as follows:

  We define the names of the $(d+1)$ elements of the **first** row of A as a, b, c .... The determinant is then defined recursively by

$$\det(A) = a * \det(A_a) - b * \det(A_b) + c * \det(A_c) - d * \det(A_d) + \cdots ,$$

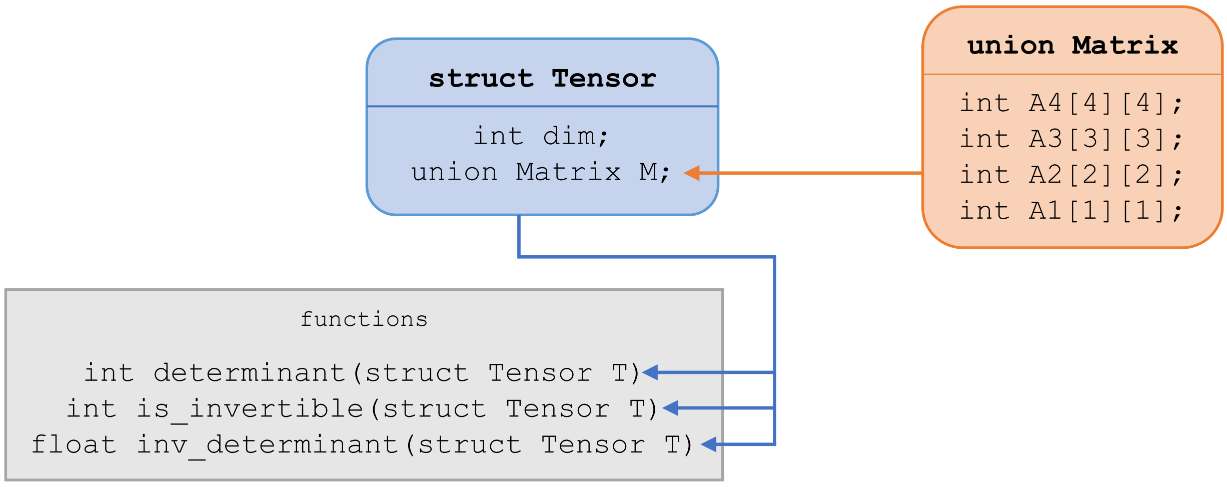
  where A_i is the sub-matrix that results when we "cross out" the row and column of A that corresponds to the current element i.

Invertibility

- any matrix $A \in \mathbb{R}^{dxd}$ is invertible iff (if and only if) the determinant of A is not 0. The inverse of A (if existent) is called A^{-1}
- $\det(A^{-1}) = \det(A)^{-1}$

In this assignment, you will create functions doing calculations with square matrices from sizes `1x1` to `4x4`. When a function in C receives a one-dimensional array as an argument, we do not have to specify the size of this array. This is actually an exception, and does not hold for higher dimensional arrays. So when dealing with two dimensional arrays, we at least will have to specify the size of the second dimension. As we would like to create one function for all sizes from `1x1` to `4x4`, we have to think of another solution:

We define the `struct Tensor`, which contains the dimension of the array as an `int`, and a `union Matrix`, which is the matrix itself and therefore is either a `1x1`, a `2x2`, a `3x3`, or a `4x4` array of `int`s. The following graphic may help you to understand this:



When we create a new matrix which we want to process with our functions, we first declare a new `struct Tensor new_tensor`, then specify the dimension (e.g. `3`) by storing it into `dim`. To store the matrix values itself, we then access the respective array (dependent on the dimension we chose) in `union Matrix` by `new_tensor.M.A3[0][0] = 42;` (repeat for all elements). You can assume that we will only test your functions with correctly and completely initialized `struct Tensor`s. We will also provide you with a main file with a nice UI, which enables you to enter the matrices you want to test on your functions in the terminal.

Specifications
When the function `determinant(struct Tensor T)` is called by a main function, the following should happen:
- The correct determinant of the matrix in `T` according to the above mathematical definition is calculated (considering the dimension) and returned as an `int`
- The values in `struct Tensor T` must not be changed

When the function `is_invertible(struct Tensor T)` is called by a main function, the following should happen:
- The matrix in `T` is tested for invertibility. If it is invertible, the function should return `1`, otherwise `0`.
- The values in `struct Tensor T` must not be changed

When the function `inv_determinant(struct Tensor T)` is called by a main function, the following should happen:
- If the matrix in `T` is invertible, the determinant of the inverse matrix is calculated and returned as a float. If it is not invertible, the function should return `0.0`.
- The values in `struct Tensor T` must not be changed

Tool Box

The explanations below **may** help you to solve this problem.

STRUCTS/UNIONS

The difference between a struct and a union is quite simple: You can imagine a struct as an empty box, where you can store different things at the same time. Using it can make sense when there are different pieces of data (eg. variables) that are closely related. (Like here the dimension of a matrix and the matrix itself). Every variable in the struct can be considered as a different part of an instance of the struct.

A union is similar, but has an important difference: Only one variable of the union can be stored at one moment in time. You can imagine a union as a placeholder for different possible variables. In our example, we don't know how large the matrix will be, and therefore use the placeholder `union Matrix` instead. This union can then be **either** a `4x4`, a `3x3`, a `2x2` or a `1x1` `int`-Matrix. So in the analogy of the struct-box, you can imagine a union as an empty box that is just large enough to hold the largest thing that has to fit in, but can only hold one element at a time.

Even though that is actually not really correct, it can help to think of structs and unions as of new datatypes you just defined. When you want to use e.g. a struct you just defined, you have to create an instance of it with a variable name, just as you would do it with an `int`. (e.g. `struct Tensor t1;`) You can then access the variables by using the dot operator: `t1.dim = 3;`

2D-ARRAYS

A two dimensional array in C works just the same as a one dimensional does, but has an additional pair of square brackets (`[]`) due to the new dimension. Just as in the 1D-Case you can access the elements by writing the indices into the square brackets, where now the first index depicts the row, and the second index depicts the column of the element to access. Example:

```
//Create a 3x3 array and initialize the first two rows. Third
//is automatically initialized to 0
int my_array[3][3] = {{1,2,1},{0,1,2}};

//Print element in second row (index 1) and third column (index
//2). Should print "2"
printf("%d",my_array[1][2]);
```

CALLING FUNCTIONS

You can of course call functions you defined before. Example:

```
int det = determinant(T);
```

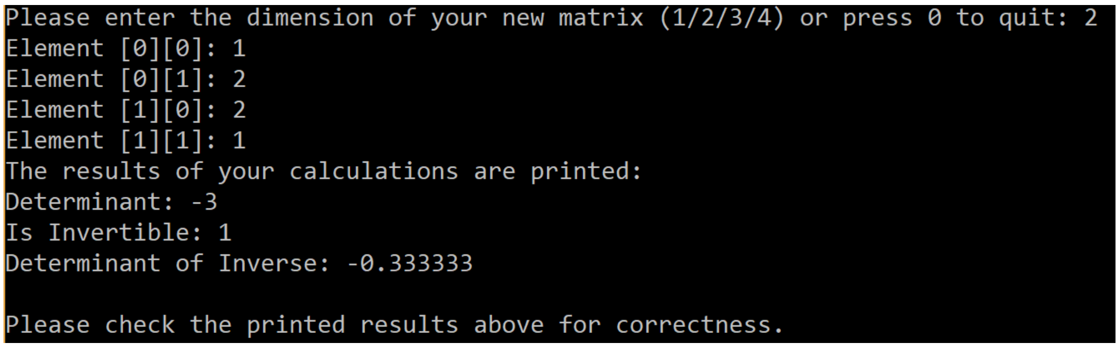
Tasks

We uploaded the template file ***matrix_functions.c*** on Moodle. Complete the functions *determinant(), is_invertible() and inv_determinant()*, so that the program works as specified above. Please note:

- You are only allowed to write code in the spots indicated in the functions. Do not change anything else in the functions or the struct and union definitions. Also do not include any other libraries than the ones already included.
- Do not print anything
- You are provided with an additional C-File ***matrix_main.c***. This file contains a main-Function, which you can use to test your functions. Store the two C-Files in the same directory on your computer, and compile and run the main file. The main-Function will then ask you to enter the dimension and the elements of a matrix of your choice. With your entered values, the main will create a `struct Tensor` to call your functions with. The return values of your functions will then be displayed. Please check by yourself whether these values are correct. You can find tools for calculating determinants online.
- When you are finished, put ***matrix_functions.c*** directly in a zip-File named "HW4_Mustermann_Max" (with your name of course) and upload this file to Moodle. Do NOT upload your main-File and do NOT put any folders into the zip-File.

Example Output

Your terminal could look something like that after an example run of your program with the provided main:



```
Please enter the dimension of your new matrix (1/2/3/4) or press 0 to quit: 2
Element [0][0]: 1
Element [0][1]: 2
Element [1][0]: 2
Element [1][1]: 1
The results of your calculations are printed:
Determinant: -3
Is Invertible: 1
Determinant of Inverse: -0.333333

Please check the printed results above for correctness.
```

Threshold

To pass this homework, your program has to fulfill the following conditions:

- Your file must be compilable and runnable without any errors and with the flags -Wall -Werror enabled for any correctly initialized `struct Tensor`
- Each of your three functions will be tested with 20 different matrices, which totals to 60 testcases. To pass this assignment, you have to succeed in at least 80% (=48) of the test cases

Information on Plagiarism and Copyright

As already announced in the lecture and the tutorials, you are supposed to do this homework alone. This homework is an official part of the exam at the end of this semester, and therefore fraud or plagiarism will be pursued and punished.

You are encouraged to discuss **specific** problems with other students (like e.g. “I don’t know how to print a string, can you help me?”), but you are forbidden to share ideas on the approach/structure of your functions, and of course to show other students your solution. This will result in a similar control-flow and structure of your submissions, which we will be able to detect even if you change variable names or loops etc.

We will compare your submissions with a code-optimized plagiarism scanner and if we find plagiarized code, you will fail this submission. If you are found guilty of plagiarism twice, you will lose the homework bonus, and in severe cases we will take further steps.

This homework is intellectual property of the TUM, namely the Chair of Image-Based Biomedical Computing. Publishing either this homework task itself or your solutions to it in any form (including on the MSE-Cloud) is strictly forbidden and can be pursued TUM-internally as well as legally.