# Notes for Programming in C Lab Session #8

September 13, 2021

## 1 Introduction

The purpose of this lab session is to write matrix manipulation code to see how different memory access patterns can affect performance.

## 2 Overview

A *matrix* is a rectangular array of numbers, and also one of the fundamental concepts of mathematics. Matrices can represent linear transformations between vector spaces, extensive-form games in game theory, graph connectivity in graph theory, the systems of differential equations arising in control theory, just to list a few applications. As a result, high-performance implementations of matrices and operations on them are of great importance to a wide variety of scientific and engineering domains.

In this lab, we will work use the following datatype for matrices:

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

Here, a matrix is represented by a structure containing a number of rows, a number of columns, and an array of doubles `elts` containing the elements of the array. As programmers, we immediately face a choice in how to represent arrays. An array is a two-dimensional object like:

$$A \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

However, a C array is *one-dimensional*. So we have to decide how to place the 12 elements of the $4 \times 3$ matrix $A$ in memory. In C, it is typical to represent arrays in *row-major order*. This means that the `elts` array will have the following shape:

$$\texttt{elts} \mapsto \boxed{1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 10 \mid 11 \mid 12}$$

So the `elts` array stores the rows of $A$ one after another in memory.[1]

As a result, if we have a matrix $B$ of size $n \times m$, and we want to find $B(i, j)$ – the $j$-th column of the $i$-th row will be the $(n \times i) + j$-th element of the array.

[1] The choice of row-major order is purely conventional; historically Fortran has made the opposite choice!

One of the most important matrix operations is *matrix multiplication*. Given an $n \times m$ matrix $A$, and an $m \times o$ matrix $B$, we define the following $n \times o$ matrix $A \times B$ as the product:

$$(A \times B)(i, j) = \sum_{k \in \{0 \ldots n\}} A(i, k) \times B(k, j)$$

In the calculation of $A(i, j)$, we will touch the following entries:

$$\begin{pmatrix} A_{(0,0)} & \cdots & \cdots & A_{(0,m-1)} \\ \vdots & & & \vdots \\ A_{(i,0)} & \cdots & \cdots & A_{(i,m-1)} \\ \vdots & & & \vdots \\ A_{(n-1,0)} & \cdots & \cdots & A_{(n-1,m-1)} \end{pmatrix} \times \begin{pmatrix} B_{(0,0)} & \cdots & B_{(0,j)} & \cdots & B_{(0,o-1)} \\ \vdots & & & & \vdots \\ \vdots & & & & \vdots \\ B_{(m-1,0)} & \cdots & B_{(m-1,j)} & \cdots & B_{(m-1,o-1)} \end{pmatrix}$$

Note that we are accessing the elements of $A_{(i,k)}$ in a row-wise order, but accessing the elements of $B_{(k,j)}$ in a column-wise order. As a result, we risk a *cache miss* on each access to $B$!

However, if $B$ were *transposed* – i.e., if rows and columns were interchanged – then we would be accessing the elements of $B$ in a row-wise order as well. In equational form, we can make the following observation (writing $B^T$ for the transpose of $B$):

$$\begin{aligned} (A \times B^T)(i, j) &= \sum_{k \in \{0 \ldots n\}} A(i, k) \times B^T(k, j) \\ &= \sum_{k \in \{0 \ldots n\}} A(i, k) \times B(j, k) \end{aligned}$$

By making use of the observation that $B^T(k, j) = B(j, k)$, we can replace a column-wise traversal with a row-wise traversal.

So in this exercise, you will implement naive multiplication, transpose, and transposed multiplication, and compare the performance of naive multiplication to building a transpose and then doing a transposed multiplication.

# 3 Instructions

1. Download the `lab8.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab8.tar.gz`.
3. This will extract the `lab8/` directory. Change into this directory using the `cd lab8/` command.
4. In this directory, there will be files `lab8.c`, `matrix.h`, and `matrix.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab8` executable.
6. There is a test routine to check if you have implemented matrix multiplication probably works, together with expected correct output in the `lab8.c` file.
7. Once it works, run the timing functions on your two matrix multiplication routines to see which one is faster.

# 4 The Types and Functions to Implement

- `matrix_t matrix_create(int rows, int cols);`

  Given integer arguments `rows` and `cols`, return a new matrix of size `rows` × `cols`. Initializing the elements of the array is optional, but may help you debug.

- `void matrix_free(matrix_t m);`

  Deallocate the storage associated with the matrix `m`.

- `void matrix_print(matrix_t m);`

  You don't have to implement this – it comes for free to help you test your code.

- `double matrix_get(matrix_t m, int r, int c);`

  Return the value in the $r$-th row and $c$-th column of $m$.

- `void matrix_set(matrix_t m, int r, int c, double d);`

  Modify the value in the $r$-th row and $c$-th column of $m$ to $d$.

- `matrix_t matrix_multiply(matrix_t m1, matrix_t m2);`

  Given an $n \times m$ matrix `m1` and an $m \times k$ matrix `m2`, return the $n \times k$ matrix that is the matrix product of `m1` and `m2`.

  You should be able to implement this with a simple triply-nested for-loop.

- `matrix_t matrix_transpose(matrix_t m);`

  Given an $n \times m$ matrix `m` as an argument, return the $m \times n$ transposed matrix. (That is, if $A$ is the argument and $B$ is the return value, then $A(i, j) = B(j, i)$.)

- `matrix_t matrix_multiply_transposed(matrix_t m1, matrix_t m2);`

  Given an $n \times m$ matrix `m1` and an $k \times m$ matrix `m2`, return the $n \times k$ matrix that corresponds to `m1` times the transpose of `m2`.

- `matrix_t matrix_multiply_fast(matrix_t m1, matrix_t m2);`

  This function should also implement matrix multiplication, but do it by constructing the transpose of `m2`, and then passing that to `matrix_multiply_fast`. Don't forget to free the transposed matrix when you are done!