# HPC1: Lecture 2

Andreas F. Borchert and Michael C. Lehn

18 October 2019

## Contents

**1 Required skills** **1**

**2 Motivation: Understanding performance issues in Matlab** **1**

**3 Representing matrices and vectors** **2**
3.1 Address of a matrix element . . . 2
3.2 Using pointer arithmetic in an implementation . . . 3

**4 More on memory storage of matrices** **3**
4.1 Matrix views (submatrices) . . . 4
4.2 Vector views (special case of submatrices) . . . 5

**5 Functions operating on matrices (and vectors)** **5**

**6 Allocating and deallocating memory for matrices** **7**
6.1 Using memory from the data segment (Fortran 77 style) . . . 7
6.2 Using dynamically allocated memory . . . 8
6.3 Typical bugs related to dynamic memory allocation . . . 9
6.4 Danger of significant runtime overhead . . . 10

**7 Review: Understanding performance issues in Matlab** **10**

## 1 Required skills

In this session we use pointers more intensively. Hence, you should be very familiar with pointers (including const pointers) and operations related to pointers (e. g. dereferencing a pointer, pointer arithmetic, address operator).

## 2 Motivation: Understanding performance issues in Matlab

Consider the following Matlab script *matrixinit.m*:

```
1  clear
2  m = 1000;
3  n = 1000;
4
5  tic
6  for i=1:m
7      for j=1:n
8          A(i,j) = (i-1)*n + j;
9      end
10 end
```

```
11  toc
12
13  tic
14  B = zeros(m,n);
15  for i=1:m
16      for j=1:n
17          B(i,i) = (i−1)*n + j;
18      end
19  end
20  toc
```

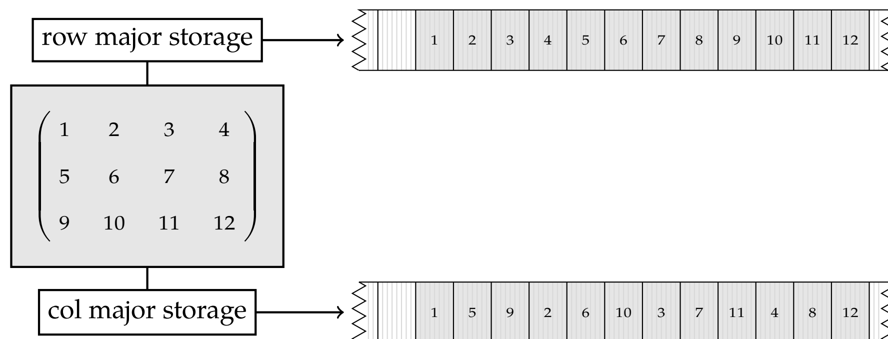
This script initializes two $m \times n$ matrices $A$ and $B$ (in this case with $m = n = 4000$). Running the script shows that the *wall time* required for the initialization is significantly different for the two matrices:

```
1  >> matrixinit
2  Elapsed time is 1.540211 seconds.
3  Elapsed time is 0.023684 seconds.
```

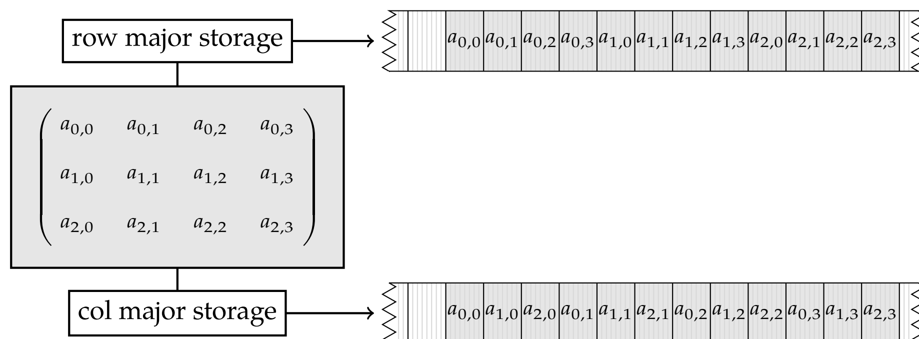
One aim of this lecture is to understand this behavior!

# 3 Representing matrices and vectors

In order to store the two dimensional structure of a matrix it is a natural choice to store elements either row or column-wise in memory. For example:




If we make the example a little bit more general this is equivalent to




Note that we the indices for the matrix elements start at zero, i. e.

$$A = \left(a_{i,j}\right)_{0 \le i < m,\, 0 \le j < n}.$$

Using this convention for indexing will make it easier to realize an implementation of numerical algorithms in C. In case we have an algorithm that follows a different convention (e. g. it is common in linear algebra that indices start at one) we assume you rewrite the algorithm first by applying an index shift.

## 3.1 Address of a matrix element

Assume each matrix element requires $c$ bytes for storage. For an $m \times n$ matrix the address of a matrix element can then be determined relatively to the address used for $a_{0,0}$ as follows:

- If the matrix is stored row-wise (we will denote this as *row major storage*) we have

  $$\text{address_of}(a_{i,j}) = \text{address_of}(a_{0,0}) + c \cdot (i \cdot n + j) \qquad \text{with } \; 0 \leq i < m, \; 0 \leq j < n.$$

  You can interpret this as going from $a_{0,0}$ to $a_{i,j}$ by

  - jumping down $i$ rows to $a_{i,0}$ and then
  - jumping right $j$ columns to $a_{i,j}$.

  Jumping down a row means skipping $n$ elements in memory and jumping to the right one element means skipping one element.

- If the matrix is stored column-wise (we will denote this as *col major storage*) we have

  $$\text{address_of}(a_{i,j}) = \text{address_of}(a_{0,0}) + c \cdot (i + j \cdot m). \qquad \text{with } \; 0 \leq i < m, \; 0 \leq j < n.$$

  This formula can be derived analogously.

By defining

$$\text{incRow} := \begin{cases} n, & \text{row major,} \\ 1, & \text{col major} \end{cases} \quad \text{and} \quad \text{incCol} := \begin{cases} 1, & \text{row major,} \\ m, & \text{col major} \end{cases}$$

we can combine both cases through

$$\text{address_of}(a_{i,j}) = \text{address_of}(a_{0,0}) + c \cdot (i \cdot \text{incRow} + j \cdot \text{incCol}) \qquad \text{with } \; 0 \leq i < m, \; 0 \leq j < n.$$

## 3.2 Using pointer arithmetic in an implementation

In order to work with a matrix $A$ we need the following information:

- the matrix dimensions $m$ and $n$,
- a pointer to the element $a_{0,0}$, and
- the increments *incRow* and *incCol*.

For an implementation we need to specify what types we use for these quantities:

- For the matrix dimension $m$ and $n$ we will use the unsigned integer type *size_t* and
- for the increments *incRow* and *inCol* the signed integer type *ptrdiff_t*. Both types are actually just typedefs declared in *stddef.h*. On a 64-bit machine both types are 64-bit integers.
- Mostly we will use **double** as element type. In this case each element is stored in $c = 8$ bytes (check **sizeof**(**double**)). However, the pointer arithmetic in C implicitly takes this into account.

So for example, in an implementation we define and initialize corresponding variables:

```c
size_t m, n;
double *A;
ptrdiff_t incRow, incCol;

/* initialize m, n, A, incRow, incCol: */
```

Because of the pointer arithmetic in C the expression

```c
    A + i*incRow + j*incCol
```

is the address of $a_{i,j}$. For dereferencing (and accessing values) we can either use the star-operator or the bracket-operators (which we prefer for numerical code):

| Address of $a_{i,j}$ | Value of $a_{i,j}$ |
|---|---|
| A + i*incRow + j*incCol | *(A + i*incRow + j*incCol) |
| &A[i*incRow + j*incCol] | A[i*incRow + j*incCol] |

# 4 More on memory storage of matrices

Recall how matrices can be stored in row or column major order. In general we will implement a numerical linear algebra method such that it handles both cases, i.e. matrices stored in row or column major order. Hereby two aspects are relevant:

1. **Functionality (and correctness)**

   An implementation should be able to handle matrices in any storage order and should produce the same numerical results. That means round off errors might differ depending on the storage order by they should be of the same magnitude.

2. **Performance**

   The performance should not depend on the storage order. In the next lecture we will see that this requires a better understanding on how the actual hardware architecture really works.

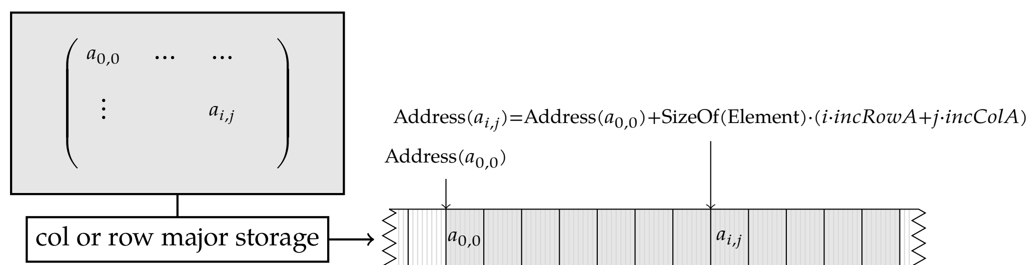
Of course, performance of an implementation is not relevant unless we can ensure its correctness. So we first will only focus on how code can handle different storage orders transparently.

## 4.1 Matrix views (submatrices)

We assume that for a $m \times n$ matrix $A$ the following information is available:

1. the number of rows $m$,
2. the number of columns $n$,
3. the address of the first element $a_{0,0}$ (i. e. a pointer to $a_{0,0}$),
4. the row increment *incRowA*, and
5. the column increment *incColA*.

As we know, the information about the storage order is incorporated into the increments and, independently of that, the address of any other element $a_{i,j}$ can be specified as depicted:




With the term *matrix view* we will express two things:

- A matrix view is a submatrix (e. g. a matrix block) of a (known) matrix.
- A matrix view is *not* a copy of this submatrix. Instead it is a reference to a submatrix. That means for instance, changing elements of a matrix view changes elements of the original matrix.

For example, in this picture $A$ is an $m \times n$ matrix and $B$ an $r \times s$ matrix view:

$$
A = \begin{pmatrix}
a_{0,0} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{0,n-1} \\
\vdots & & a_{i,j} & \cdots & a_{i,j+s-1} & & \vdots \\
\vdots & & \vdots & & \vdots & & \vdots \\
\vdots & & a_{i+r-1,j} & \cdots & a_{i+r-1,j+s-1} & & \vdots \\
a_{m-1,0} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{m-1,n-1}
\end{pmatrix}
$$

$$
B = \begin{pmatrix}
a_{i,j} & \cdots & a_{i,j+s-1} \\
\vdots & & \vdots \\
a_{i+r-1,j} & \cdots & a_{i+r-1,j+s-1}
\end{pmatrix}
$$

Obviously the row and column increments of $B$ are the same as the increments of $A$. So assuming that $A$ is described by

```
size_t m, n;
double *A;
ptrdiff_t incRowA, incColA;

/*
   some code for initializing m, n, A, incRowA and incColA
*/
```

we can describe $B$ through

```
size_t r /* = ... */;
size_t s /* = ... */;
double *B = &A[i*incRowA+j*incColA]; // for some i, j
ptrdiff_t incRowB = incRowA, incColB = incColA;
```

## 4.2 Vector views (special case of submatrices)

Vector views are special cases of matrix views with a single row or a single column. We will not distinguish explicitly between row or column vectors. Hence, we only need three pieces of information:

- the length of the vector,
- the address of its first element, and
- the increment (also denoted as stride) between the elements in memory.

One of the most common cases will be that a vector references a row or column of a matrix. Let $x$ be a vector view and denote its length with $k$:

- If $x$ references the row of $A$ with index $i$ we can describe this view as follows:

```
size_t k = n;
double *x = &A[i*incRowA];
ptrdiff_t incX = incColA;
```

- If $x$ references the column of $A$ with index $j$ we can specify this as follows:

```
size_t k = m;
double *x = &A[j*incColA];
ptrdiff_t incX = incRowA;
```

# 5 Functions operating on matrices (and vectors)

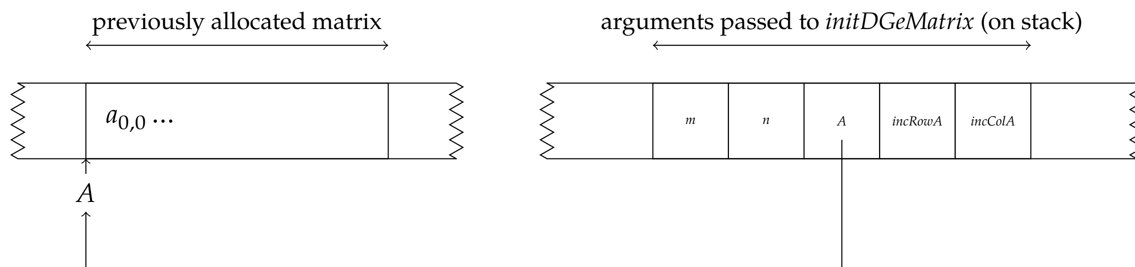
We know that in C a function always receives copies of its arguments (*call by value*). That means if a function modifies its arguments it modifies its local copy but not the original value of the parameter. See following example:

```
#include <stdio.h>

void
foo(int n)
{
    n = 666;
}

int
main()
{
    int n = 42;
    foo(n);
    printf("n=%d\n", n); // prints: n=42
}
```

However, function *initDGeMatrix* receives a matrix and modifies it:

```
void
initDGeMatrix(size_t m, size_t n,
              double *A,
              ptrdiff_t incRowA, ptrdiff_t incColA)
{
    for (size_t i=0; i<m; ++i) {
        for (size_t j=0; j<n; ++j) {
            A[i*incRowA + j*incColA] = i*n + j + 1;
        }
    }
}
```

How is that possible? Arguments are still passed as copies, however, a copy of an address still points to the same memory location. So before *initDGeMatrix* gets called (but after its arguments are pushed on the stack) the stack looks like this:




So make yourself clear: In *initDGeMatrix* we do not modify the pointer $A$ itself. We modify values to which the pointer arithmetic expression $A + i{*}incRowA + j{*}incColA$ points to.

In some cases we want some kind of guarantee that a function does not modify a matrix. For example, function *printDGeMatrix* is supposed to print a matrix without modifying it. If the programmer of this function accidentally modifies it we want the compiler to issue an error. We do this by declaring a const-pointer. So here is some code that resulted by copying and pasting function *initDGeMatrix* but the programmer forgot to remove the assignment from the inner loop body:

```
void
printDGeMatrix(size_t m, size_t n,
               const double *A,
               ptrdiff_t incRowA, ptrdiff_t incColA)
{
    for (size_t i=0; i<m; ++i) {
        for (size_t j=0; j<n; ++j) {
            A[i*incRowA + j*incColA] = i*n + j + 1; // <-- gives an error
            printf("%10.2lf ", A[i*incRowA + j*incColA]);
        }
        printf("\n");
    }
    printf("\n");
}
```

Be aware that **const** has in C the meaning of *read-only*. Using a cast a programmer still can (willingly) modify the underlying data:

```
void
printDGeMatrix(size_t m, size_t n,
               const double *A,
               ptrdiff_t incRowA, ptrdiff_t incColA)
{
    for (size_t i=0; i<m; ++i) {
        for (size_t j=0; j<n; ++j) {
            ((double *)A)[i*incRowA + j*incColA] = 666; // <-- compiler will accept this
            printf("%10.2lf ", A[i*incRowA + j*incColA]);
        }
        printf("\n");
    }
    printf("\n");
}
```

# 6 Allocating and deallocating memory for matrices

In the previous sections we talked about how matrices can be part of existing matrices. However, there obviously has to be some kind of initial matrix which is not a submatrix. The typical structure of a numerical linear algebra program is as follows:

1. allocate memory for matrices (and vectors),
2. initialize matrices …, do all the computations …, and
3. release memory.

## 6.1 Using memory from the data segment (Fortran 77 style)

We can use some globally defined arrays for the initial matrices and vectors. The dimensions of these arrays have to be known at compile time. Using macros default values for the maximal dimensions can be overridden when the program gets compiled, for example:

```
#ifndef A_MAX_M
#define A_MAX_M 1024
#define

#ifndef A_MAX_N
#define A_MAX_N 1024
#define

double A[A_MAX_M * A_MAX_N];

/* ... */

int
main()
{
    size_t m, n;
    ptrdiff_t incRowA, incColA;

    /* initialize m, n, incRowA, incColA (requires m<=A_MAX_M, n<=A_MAX_N) */

    initDGeMatrix(m, n, A, incRowA, incColA);
    printDGeMatrix(m, n, A, incRowA, incColA);
}
```

Constructs like these were used (and were the only option) in Fortran 77. Usually it requires two steps:

1. In a first *dry run* or *workspace query* one specifies only the desired problem size and the program only computes the amount of memory required for this purpose. E. g. computing eigenvalues and eigenvectors of a $n \times n$ matrix $A$ usually requires beside for storing the matrix some additional memory.

2. The result of workspace query is then used to define array dimensions for the computational run (where the actual problem gets solved).

The advantage of this pattern is that allocation of global arrays is basically free, i. e. it does not induce any runtime overhead. That is because the data segment gets initialized when the program gets loaded into memory (and before it gets started).

However, the disadvantage is the lack of flexibility. Eventually we have to recompile the program after a workspace query if default dimensions for arrays are not large enough.

## 6.2 Using dynamically allocated memory

If the size of an array is not known at compile time it is possible to allocate memory at runtime. We know that functions are free to allocate memory on the stack. However, using the stack has at least two drawbacks:

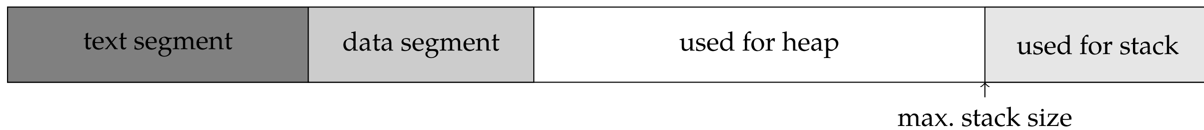
- The memory from the stack is bound to a function (local memory). Once the function returns it gets reused for other function calls.

- The maximal size of the stack is limited by the operating system. You can check this (and also change the stack limit) in the terminal with *ulimit*. Here an example on thales:

```
thales> ulimit -a
core file size (blocks, -c) unlimited
data seg size (kbytes, -d) unlimited
file size (blocks, -f) unlimited
open files (-n) 512
pipe size (512 bytes, -p) 10
```

```
7       stack size (kbytes, -s) 10240
8       cpu time (seconds, -t) unlimited
9       max user processes (-u) 1024
10      virtual memory (kbytes, -v) unlimited
```

When we talk about dynamically allocated memory we usually refer to the memory allocated from the so-called *heap*. The heap is the memory between the data segment and the stack:




The operating system provides functions for allocating and releasing memory from the heap. For us the following three standard library functions are sufficient for the moment:

```
1  void *malloc(size_t size);
2
3  void free(void *ptr);
4
5  void *calloc(size_t nelem, size_t elsize);
```

So basically these functions are doing book keeping:

- *malloc* (or *calloc*) are used to query a free memory block on the heap. While *calloc* initializes the allocated storage with zeroes, *malloc* leaves it uninitialized.
- *free* releases a memory block so that it can be reused.

## 6.3 Typical bugs related to dynamic memory allocation

- Not checking whether allocation was successful.

  If *malloc* (or *calloc*) fails the program will not abort automatically. The allocation function simply returns a null pointer. A null pointer is actually a pointer that contains address 0 as its value. Hence, we can use it in a logical expression and abort the program manually if allocation failed:

```
1  // allocate memory for A
2  double *A = malloc(m*n*sizeof(double));
3  if (!A) {
4      abort();
5  }
```

- Memory leaks (not releasing memory).

  Maybe you have experienced that sometimes applications slow down if you keep them running for several days. The reason for that might be a memory leak, i. e. memory gets allocated but unintentionally never released.

  Here is an example related to a future exercise for this lecture. Note that in line 18 the function might be left without releasing previously allocated memory.

```
1  void
2  dgemm_simple_blk(size_t m, size_t n, size_t k,
3                   double alpha,
4                   const double *A, ptrdiff_t incRowA, ptrdiff_t incColA,
5                   const double *B, ptrdiff_t incRowB, ptrdiff_t incColB,
6                   double beta,
7                   double *C, ptrdiff_t incRowC, ptrdiff_t incColC)
```

```
{
    // allocate buffers
    double *A_ = malloc(DGEMM_BLK_M*DGEMM_BLK_K*sizeof(double));
    double *B_ = malloc(DGEMM_BLK_K*DGEMM_BLK_N*sizeof(double));
    double *C_ = malloc(DGEMM_BLK_M*DGEMM_BLK_N*sizeof(double));

    /* some code */

    // check if we are already done
    if (alpha==0) {
        return; /* memory leak! */
    }

    /* more code */

    // release buffers
    free(C_);
    free(B_);
    free(A_);
}
```

- Trying to release memory more then once.

  Releasing accidentally memory more than once leads to undefined behaviour. Here an academic example, where the error is obvious:

```
    double *A_ = malloc(DGEMM_BLK_M*DGEMM_BLK_K*sizeof(double));
    free(A_);
    free(A_); // <-- undefined
```

  More realistic examples, where it is harder to keep track how often *free* gets called for a pointer, often are related to implementations of dynamic data structures.

- Assuming that *malloc* returns properly initialized memory.

  Initially, when a program starts all memory is properly initialized. However, when storage areas are allocated and released using *malloc* und *free* and then returned again by *malloc* the contents of previous data structures may be visible again. This could include NaNs (*not a number* values) or theoretically even sNaNs (*signaling* NaNs which trigger a trap when used). Hence, all data must be properly initialized before being used.

## 6.4 Danger of significant runtime overhead

In general the call of *malloc* can be expensive, i. e. time consuming. That's because a free block of the required size has to be found. So a typical worst case scenario for performance would be using *malloc* inside a loop with different block sizes.

The time required for *malloc* can be rather fast if a block of same size was previously allocated and released. For example, in

```
void
foo()
{
    double *buffer = malloc(FOO_SIZE*sizeof(double));

    /* do something */

    free(buffer);
}
```

The first call of *foo* might be expensive. The next call of *foo* might be rather cheap if the previously released memory block is still available.

Note that the pattern described at the beginning of this section (allocating all memory when the program starts, then doing all computations and releasing memory before the program terminates) avoids this problem.

# 7 Review: Understanding performance issues in Matlab

We now review the initial Matlab example:

- In the initialization of matrix $A$ the matrix gets resized in the loop. For resizing Matlab has to
  1. allocate a new (and bigger) memory block,
  2. copy elements from the old block to the new block and
  3. release the old block
- For matrix $B$ no resizing is required. That's because for matrix $B$ a sufficient large memory block gets allocated before the loop when we assign a $m \times n$ zero matrix to it.