

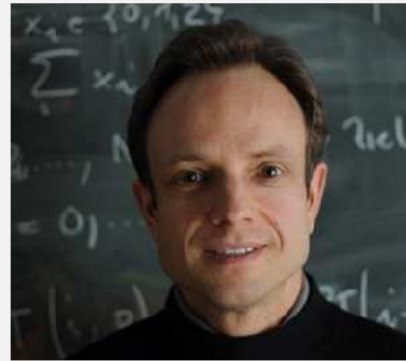
Course Video

Section 1.2.3: (Dense) Matrix Storage Formats

Prof. R. Hiptmair, SAM, ETH Zurich

Date: August 24, 2020

(C) Seminar für Angewandte Mathematik, ETH Zürich



Prerequisites.

- Matrices in EIGEN

Dependencies.

- Section 1.2.1

Duration: 18 minutes

I. Computing with Matrices and Vectors

1.2. Software and Libraries

1.2.3. (Dense) Matrix Storage Formats

Always: The entries of a (generic, dense) $A \in \mathbb{K}^{m,n}$ are stored in a *contiguous linear array of size $m \cdot n$*

Exception: structured / sparse
 $\hookrightarrow$ diagonal / banded / triangular...

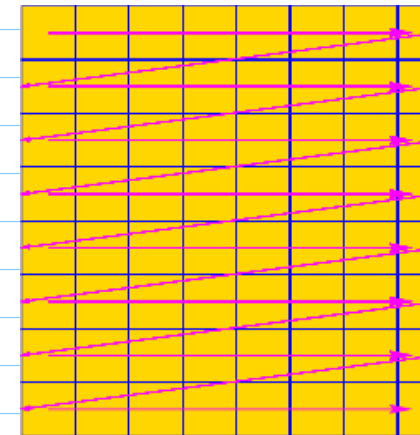
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Row major (C-arrays, bitmaps, Python):

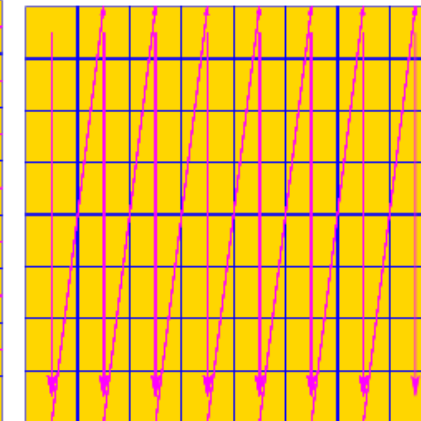
A_arr	1	2	3	4	5	6	7	8	9
-------	---	---	---	---	---	---	---	---	---

Column major (Fortran, MATLAB, EIGEN):

A_arr	1	4	7	2	5	8	3	6	9
-------	---	---	---	---	---	---	---	---	---



row major



column major

Index mapping
(row major)

$$(i, j) \rightarrow i \cdot n + j$$

C++-indexing:
 $i \in \{0, \dots, m-1\}$
 $j \in \{0, \dots, n-1\}$

②

Eigen: accessing underlying array

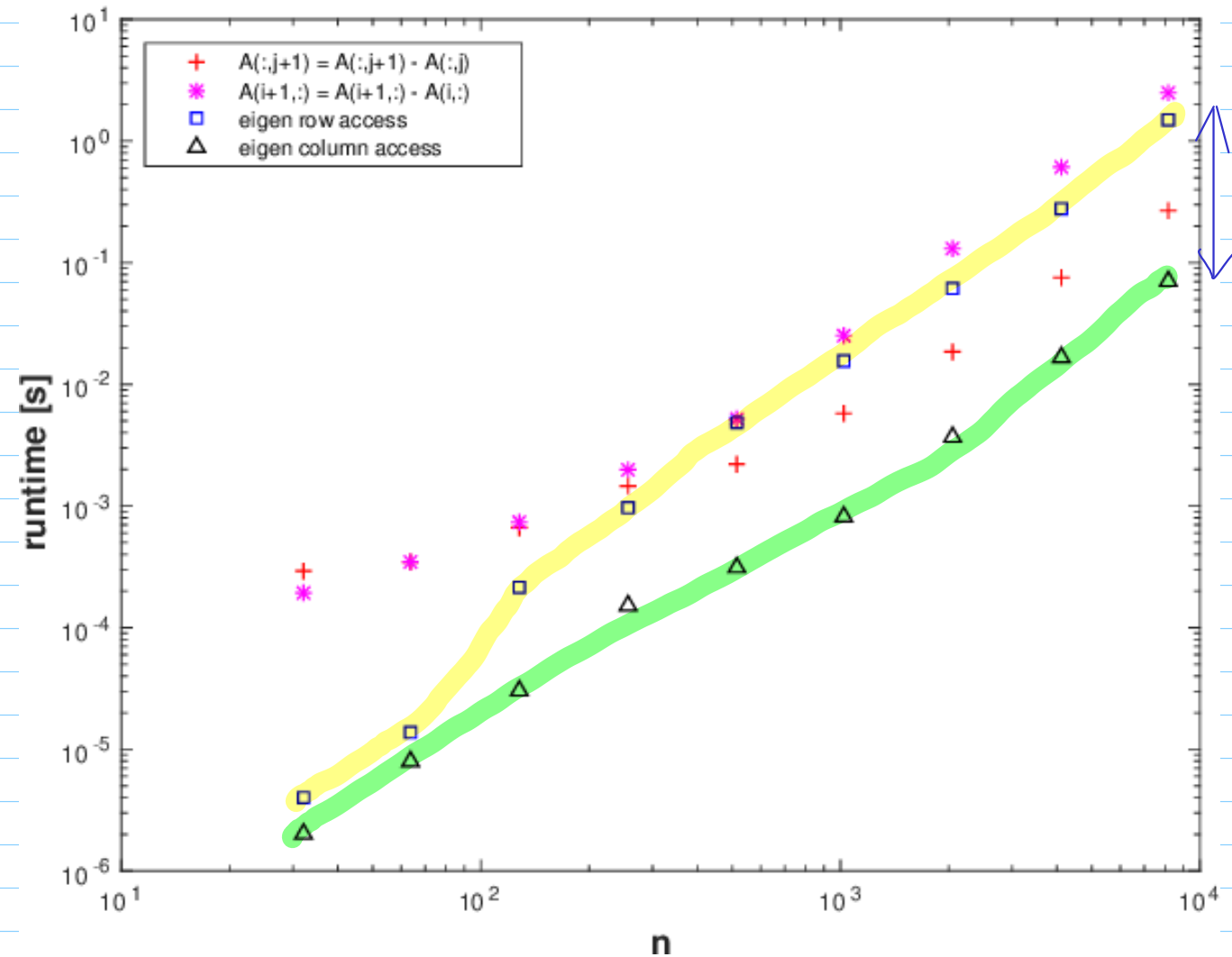
- $A(i) \hat{=}$ reference to the i -th element of the array
- $A.data()$ $\hat{=}$ raw pointer, e.g. `double *`

Exp. 1.2.3.9: Data layout impacts efficiency

C++ code 1.2.3.10: Timing for row and column oriented matrix access for EIGEN → GITLAB

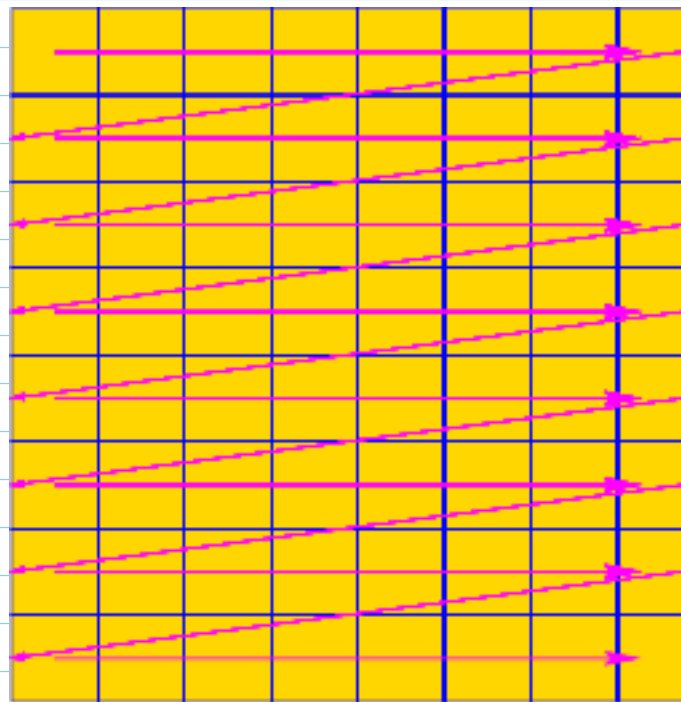
```

2 void rowcolaccesstiming(void)
3 {
4     const int K = 3; // Number of repetitions
5     const int N_min = 5; // Smallest matrix size 32
6     const int N_max = 13; // Scan until matrix size of 8192
7     unsigned long n = (1L << N_min);
8     Eigen::MatrixXd times(N_max-N_min+1,3);
9
10    for(int l=N_min; l<= N_max; l++, n*=2) {
11        Eigen::MatrixXd A = Eigen::MatrixXd::Random(n,n); → declare matrix object
12        double t1 = 1000.0;
13        for(int k=0;k<K;k++) {
14            auto tic = high_resolution_clock::now();
15            for(int j=0; j < n-1; j++) A.row(j+1) -= A.row(j); // row access
16            auto toc = high_resolution_clock::now();
17            double t = (double)duration_cast<microseconds>(toc-tic).count()/1E6;
18            t1 = std::min(t1,t);
19        }
20        double t2 = 1000.0;
21        for(int k=0;k<K;k++) {
22            auto tic = high_resolution_clock::now();
23            for(int j=0; j < n-1; j++) A.col(j+1) -= A.col(j); //column access
24            auto toc = high_resolution_clock::now();
25            double t = (double)duration_cast<microseconds>(toc-tic).count()/1E6;
26            t2 = std::min(t2,t);
27        }
28        times(l-N_min,0) = n;    times(l-N_min,1) = t1;    times(l-N_min,2) = t2;
29    }
30    std::cout << times << std::endl;
31 }
  
```

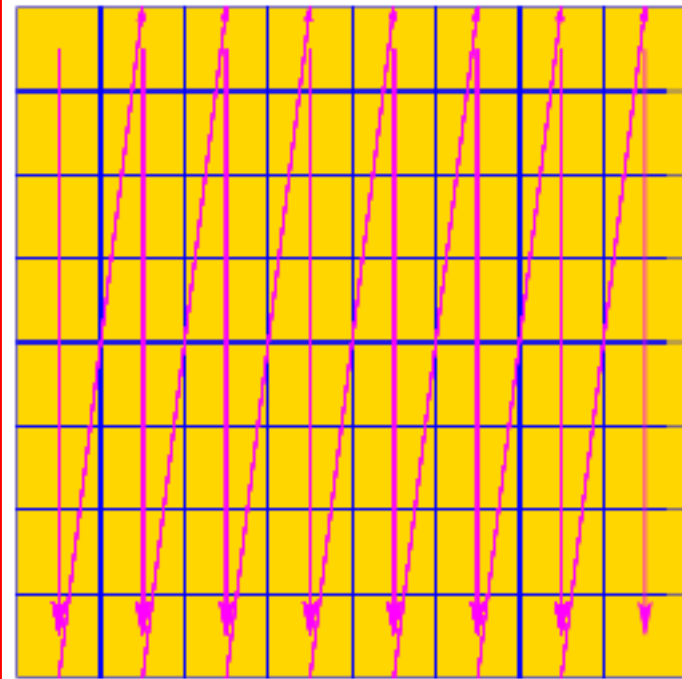


⇒ Row-wise ops. are much slower than col-wise access
 ↓
 incur cache missing

③



row major



column major



elements of all columns are
contiguously in memory

▷ Review questions 1.2.3.11

4

5

7