1.1 Fundamentals

1.1.1 Notations

The notations in this course try to adhere to established conventions. Since these may not be universal, idiosyncrasies cannot be avoided completely. Notations in textbooks may be different, beware!

✎ notation for generic field of numbers: $\mathbb{K}$

In this course, $\mathbb{K}$ will designate either $\mathbb{R}$ (real numbers) or $\mathbb{C}$ (complex numbers); complex arithmetic [14, Sect. 2.5] plays a crucial role in many applications, for instance in signal processing.

(1.1.1) Notations for vectors

✦ Vectors = are n-tuples ($n \in \mathbb{N}$) with components $\in \mathbb{K}$.

vector = one-dimensional array (of real/complex numbers)

✎ vectors will usually be denoted by small **bold** symbols: $\mathbf{a}, \mathbf{b}, \ldots, \mathbf{x}, \mathbf{y}, \mathbf{z}$

✦ Default in this lecture: vectors $=$ column vectors

$$\underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{K}^n}_{\text{column vector}} \qquad \underbrace{\begin{bmatrix} x_1 \cdots x_n \end{bmatrix} \in \mathbb{K}^{1,n}}_{\text{row vector}}$$

$\mathbb{K}^n \triangleq$ vector space of *column vectors* with n components in $\mathbb{K}$.

✎ notation for column vectors: **bold** small roman letters, e.g. $\mathbf{x}, \mathbf{y}, \mathbf{z}$

✦ Transposing: $\begin{cases} \text{column vector} & \mapsto & \text{row vector} \\ \text{row vector} & \mapsto & \text{column vector} \end{cases}$.

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}^\top = \begin{bmatrix} x_1 \cdots x_n \end{bmatrix} \quad , \quad \begin{bmatrix} x_1 \cdots x_n \end{bmatrix}^\top = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

✎ Notation for row vectors: $\mathbf{x}^\top, \mathbf{y}^\top, \mathbf{z}^\top$

✦ Addressing vector components:

✎ two notations: $\mathbf{x} = [x_1 \ldots x_n]^\top \rightarrow x_i, \quad i = 1, \ldots, n$

$\mathbf{x} \in \mathbb{K}^n \rightarrow (\mathbf{x})_i, \quad i = 1, \ldots, n$

✦ Selecting sub-vectors:

✎ notation: $\mathbf{x} = [x_1 \dots x_n]^\top \;\succ\; (\mathbf{x})_{k:l} = (x_k, \dots, x_l)^\top, 1 \le k \le l \le n$

✦ j-th unit vector: $\mathbf{e}_j = [0, \dots, 1, \dots, 0]^\top, \quad (\mathbf{e}_j)_i = \delta_{ij}, i, j = 1, \dots, n.$

✎ notation: Kronecker symbol $\delta_{ij} := 1$, if $i = j$, $\delta_{ij} := 0$, if $i \neq j$.

(1.1.2) Notations and notions for matrices

✦ Matrices = two-dimensional arrays of real/complex numbers

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix} \in \mathbb{K}^{n,m}, \quad n, m \in \mathbb{N}.$$

vector space of $n \times m$-matrices: ($n \mathrel{\hat{=}}$ number of rows, $m \mathrel{\hat{=}}$ number of columns)

✎ notation: **bold** CAPITAL roman letters, e.g., $\mathbf{A}, \mathbf{S}, \mathbf{Y}$

$\mathbb{K}^{n,1} \leftrightarrow$ column vectors, $\mathbb{K}^{1,n} \leftrightarrow$ row vectors

✦ Writing a matrix as a tuple of its columns or rows

$$\mathbf{c}_i \in \mathbb{K}^n, \; i = 1, \dots, m \;\triangleright\; \mathbf{A} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_m] \in \mathbb{K}^{n,m},$$

$$\mathbf{r}_i \in \mathbb{K}^m, \; i = 1, \dots, n \;\triangleright\; \mathbf{A} = \begin{bmatrix} \mathbf{r}_1^\top \\ \vdots \\ \mathbf{r}_n^\top \end{bmatrix} \in \mathbb{K}^{n,m}.$$

✦ Addressing matrix entries & sub-matrices (✎ notations):

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}$$

→ entry $(\mathbf{A})_{i,j} = a_{ij}$, $1 \le i \le n, 1 \le j \le m$,
→ i-th row, $1 \le i \le n$: $a_{i,:} = (\mathbf{A})_{i,:}$,
→ j-th column, $1 \le j \le m$: $a_{:,j} = (\mathbf{A})_{:,j}$,
→ matrix block (sub-matrix) $(a_{ij})_{\substack{i=k,\dots,l \\ j=r,\dots,s}} = (\mathbf{A})_{k:l,r:s}$, $1 \le k \le l \le n$, $1 \le r \le s \le m$.

The colon (:) range notation is inspired by MATLAB's matrix addressing conventions, see Section 1.2.1. $(\mathbf{A})_{k:l,r:s}$ is a matrix of size $(l-k+1) \times (s-r+1)$.

✦ Transposed matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}^\top := \begin{bmatrix} a_{11} & \dots & a_{n1} \\ \vdots & & \vdots \\ a_{1m} & \dots & a_{mn} \end{bmatrix} \in \mathbb{K}^{m,n} .$$

✦ Adjoint matrix (Hermitian transposed):

$$\mathbf{A}^{\mathrm{H}} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}^{\mathrm{H}} := \begin{bmatrix} \bar{a}_{11} & \dots & \bar{a}_{n1} \\ \vdots & & \vdots \\ \bar{a}_{1m} & \dots & \bar{a}_{mn} \end{bmatrix} \in \mathbb{K}^{m,n} .$$

✎ notation: $\bar{a}_{ij} = \mathfrak{Re}(a_{ij}) - i\mathfrak{Im}(a_{ij})$ complex conjugate of a_{ij}.

1.1.2 Classes of matrices

Most matrices occurring in mathematical modelling have a special structure. This section presents a few of these. More will come up throughout the remainder of this chapter; see also [2, Sect. 4.3].

(1.1.3) Special matrices

Terminology and notations for a few very special matrices:

Identity matrix: $\mathbf{I} = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix} \in \mathbb{K}^{n,n}$,

Zero matrix: $\mathbf{O} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} \in \mathbb{K}^{n,m}$,

Diagonal matrix: $\mathbf{D} = \begin{bmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{bmatrix} \in \mathbb{K}^{n,n}$, $d_j \in \mathbb{K}$, $j = 1, \dots, n$.

The creation of special matrices can usually be done by special commands or functions in the various languages or libraries dedicated to numerical linear algebra, see § 1.2.5, § 1.2.13.

(1.1.4) Diagonal and triangular matrices

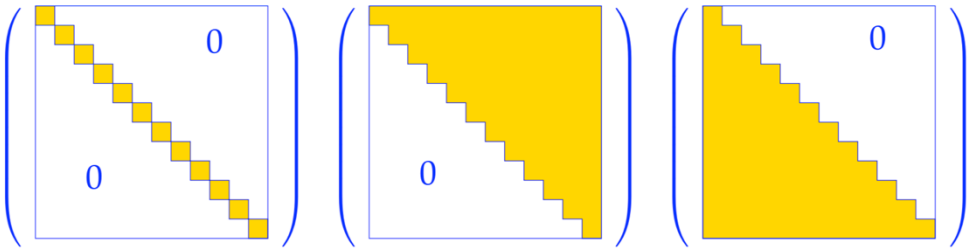
A little terminology to quickly refer to matrices whose non-zero entries occupy special locations:

Definition 1.1.5. Types of matrices

A matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{m,n}$ is

- diagonal matrix, if $a_{ij} = 0$ for $i \neq j$,
- upper triangular matrix if $a_{ij} = 0$ for $i > j$,
- lower triangular matrix if $a_{ij} = 0$ for $i < j$.

A triangular matrix is normalized, if $a_{ii} = 1, i = 1, \ldots, \min\{m, n\}$.



diagonal matrix | upper triangular | lower triangular

(1.1.6) Symmetric matrices

Definition 1.1.7. Hermitian/symmetric matrices

A matrix $\mathbf{M} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is Hermitian, if $\mathbf{M}^{\mathrm{H}} = \mathbf{M}$. If $\mathbb{K} = \mathbb{R}$, the matrix is called symmetric.

Definition 1.1.8. Symmetric positive definite (s.p.d.) matrices → [4, Def. 3.31], [12, Def. 1.22]

$\mathbf{M} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is symmetric (Hermitian) positive definite (s.p.d.), if

$$\mathbf{M} = \mathbf{M}^{\mathrm{H}} \quad \text{and} \quad \forall \mathbf{x} \in \mathbb{K}^n: \ \mathbf{x}^{\mathrm{H}}\mathbf{M}\mathbf{x} > 0 \ \Leftrightarrow \ \mathbf{x} \neq 0 \, .$$

If $\mathbf{x}^{\mathrm{H}}\mathbf{M}\mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{K}^n$ ▷ $\mathbf{M}$ positive semi-definite.

Lemma 1.1.9. Necessary conditions for s.p.d. → **[4, Satz 3.33], [12, Prop. 1.18]**

For a symmetric/Hermitian positive definite matrix $\mathbf{M} = \mathbf{M}^{\mathrm{H}} \in \mathbb{K}^{n,n}$ *holds true:*

1. $m_{ii} > 0$, $i = 1, \ldots, n$,
2. $m_{ii}m_{jj} - |m_{ij}|^2 > 0 \quad \forall 1 \le i < j \le n$,
3. *all eigenvalues of* $\mathbf{M}$ *are positive.* *(← also sufficient for symmetric/Hermitian* $\mathbf{M}$*)*

Remark 1.1.10 (S.p.d. Hessians)

Recall from analysis: in an isolated local minimum x^* of a C^2-function $f : \mathbb{R}^n \mapsto \mathbb{R}$ ➤ Hessian $\mathrm{D}^2 f(x^*)$ s.p.d. (see Def. 8.4.11 for the definition of the Hessian)

To compute the minimum of a C^2-function iteratively by means of Newton's method (→ Sect. 8.4) a linear system of equations with the s.p.d. Hessian as system matrix has to be solved in each step.

The solutions of many equations in science and engineering boils down to finding the minimum of some (energy, entropy, etc.) function, which accounts for the prominent role of s.p.d. linear systems in applications.

?! *Review question(s) 1.1.11.* (Special matrices)

- We consider two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,m}$, both with at most $N \in \mathbb{N}$ non-zero entries. What is the maximal number of non-zero entries of $\mathbf{A} + \mathbf{B}$?
- A matrix $\mathbf{A} \in \mathbb{R}^{n,m}$ enjoys the following property (banded matrix):

$$i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\},\ i - j \notin \{-B_-, \ldots, B_+\} \quad \Rightarrow \quad (\mathbf{A})_{ij} = 0\,,$$

  for given $B_-, B_+ \in \mathbb{N}_0$. What is the maximal number of non-zero entries of $\mathbf{A}$.

1.2 Software and Libraries

Whenever algorithms involve matrices and vectors (in the sense of linear algebra) it is advisable to rely on suitable code libraries or numerical programming environments.

1.2.1 MATLAB

MATLAB ("matrix laboratory") is a commercial (sold by MathWorks Corp.)

- full fledged high level programming language designed for numerical algoritms (domain specific language: DSL)
- integrated development environment (IDE) offering editor, debugger, profiler, tracing facilities,
- rather comprehensive collection of numerical libraries.

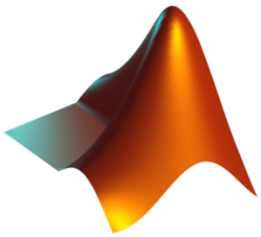


Fig. 25

Many textbooks, for instance [2] and [8] rely on MATLAB to demonstrate the actual implementation of numerical algorithms. So did earlier versions of this course. The current version has dumped MATLAB and, hence, this section **can be skipped** safely.

In its basic form MATLAB is an interpreted scripting language without strict type-binding. This, together with its uniform IDE across many platforms, makes it a very popular tool for *rapid prototyping* and *testing* in CSE.

Plenty of resources are available for MATLAB's users, among them

- MATLAB documentation accessible through the Help menu ot through this link,
- MATLAB's help facility through the commands **help** <**function**> or `doc` <**function**>,
- A concise MATLAB primer, one of many available online, see also here

(1.2.1) Fetching the dimensions of a matrix

☞ `v = size(A)` yields a row vector `v` of length 2 with `v(1)` containing the number of rows and `v(1)` containing the number of columns of the matrix `A`.

☞ **numel**`(A)` returns the total number of entries of `A`; if `A` is a (row or column) vector, we get the length of A.

- MATLAB-Einführung (in German) by P. Arbenz.

"In MATLAB everything is a matrix"

(Fundamental "data type" in MATLAB = matrix of complex numbers)

➢ In MATLAB vectors are represented as $n \times 1$-matrices (column vectors) or $1 \times n$-matrices (row vectors).

Note: The treatment of vectors as special matrices is consistent with the basic operations from matrix calculus.

(1.2.2) Access to matrix and vector components in MATLAB

Access (rvalue & lvalue) to components of a vector and entries of a matrix in MATLAB is possible through the `()-operator`:

- ☞ `r = v(i)`: retrieve `i`-th entry of vector `v`. `i` must be an integer and smaller or equal **numel**`(v)`.
- ☞ `r = A(i,j)`: get matrix entry $(\mathbf{A})_{i,j}$ for two (valid) integer indices `i` and `j`.
- ☞ `r = A(end-1,end-2)`: get matrix entry $(\mathbf{A})_{n-1,m-2}$ of an $n \times m$-matrix `A`.

! In case the matrix `A` is too small to contain an entry $(\mathbf{A})_{i,j}$, write access to `A(i,j)` will automatically trigger a dynamic adjustment of the matrix size to hold the accessed entry. The other new entries are filled with zeros.

```
% Caution: matrices are dynamically
   expanded when
% out of range entries are accessed
M = [1,2,3;4,5,6]; M(4,6) = 1.0; M,
```

Output: `M=`

1	2	3	0	0	0
4	5	6	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1

➢ Danger of accidental exhaustion of memory!

(1.2.3) Access to submatrices in MATLAB

For any two (row or column) vectors `I`, J of positive integers `A(I,J)` selects the submatrix

$$\left[(\mathbf{A})_{i,j}\right]_{\substack{i\in \mathtt{I}\\ j\in \mathtt{J}}} \in \mathbb{K}^{\sharp \mathtt{I},\sharp \mathtt{J}} .$$

`A(I,J)` can be used as both r-value and l-value; in the former case the maximal components of `I` and `J` have to be smaller or equal the corresponding matrix dimensions, lest MATLAB issue the error message

`Index exceeds matrix dimensions`. In the latter case, the size of the matrix is grown, if needed, see § 1.2.2.

(1.2.4) Initialization of matrices in MATLAB by concatenation

Inside square brackets `[ ]` the following two matrix construction operators can be used:

- `,`-operator $\hat{=}$ adding another matrix to the right (horizontal concatenation)
- `;`-operator $\hat{=}$ adding another matrix at the bottom (vertical concatenation)

(The `,`-operator binds more strongly than the `;`-operator!)

! Matrices joined by the `,`-operator must have the same number of rows.
Matrices concatenated vertically must have the same number of columns

▷ Filling a small matrix:
```
A = [1,2;3,4;5,6];
A = [[1;3;5],[2;4;6]];
```
→ 3×2 matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$.

▷ Initialization of vectors in MATLAB:

column vectors `x = [1;2;3];`
row vectors `y = [1,2,3];`

▷ Building a matrix from blocks:

```
% MATLAB script demonstrating the construction
   of a matrix from blocks
A = [1,2;3,4]; B = [5,6;7,8];
C = [A,B;-B,A], % use concatenation
```

Output: `C=`

```
     1     2     5     6
     3     4     7     8
    -5    -6     1     2
    -7    -8     3     4
```

(1.2.5) Special matrices in MATLAB

[Special matrices in MATLAB → § 1.1.3]

- ☞ $n \times n$ identity matrix: `I = eye(n);`
- ☞ $m \times n$ zero matrix: `O = zeros(m,n);`
- ☞ $m \times n$ random matrix with entries equidistributed in $[0,1]$: `R = rand(m,n);`
- ☞ $n \times n$ diagonal matrix with components of n-vector `d` (both row or column vectors are possible) on its diagonal: `D = diag(d);`

(1.2.6) Initialization of equispaced vectors ("loop index vectors")

In MATLAB `v = (a:s:b)`, where `a,b,s` are real numbers, creates a *row vector* as initialised by the following code

```
if ((b >= a) && (s > 0))
  v = [a]; while (v(end)+s <= b), v = [v,v(end)+s]; end

elseif ((b <= a) && (s < 0))
  v = [a]; while (v(end)+s >= b), v = [v,v(end)+s]; end
else v = [];
end
```

Examples:

```
>> v = (3:-0.5:-0.3)
v =  3.0000 2.5000 2.0000 1.5000 1.0000  0.5000 0
>> v = (1:2.5:-13)
v = Empty matrix: 1-by-0
```

These vectors can be used to program loops in MATLAB

```
for i = (a:s:b)
% Do something with the loop variable i
end
```

In general we could also pass a matrix as "loop index vector". In this case the loop variable with run through the *columns* of the matrix

```
% MATLAB loop over columns of a matrix
M = [1,2,3;4,5,6];
for i = M; i, end
```

Output:

```
i =  1    i =  2    i =  3
     4         5         6
```

(1.2.7) Special structural operations on matrices in MATLAB

- `A'` $\hat{=}$ Hermitian transpose of a matrix `A`, transposing without complex conjugation done by `transpose(A)`.
- **triu** `(A)` and **tril** `(A)` return the upper and lower triangular parts of a matrix `A` as r-value (copy): If $\mathbf{A} \in \mathbb{K}^{m,n}$

$$(\texttt{triu}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & \text{, if } i \leq j \text{ ,} \\ 0 & \text{else.} \end{cases} \quad (\texttt{tril}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & \text{, if } i \geq j \text{ ,} \\ 0 & \text{else.} \end{cases}$$

- **diag** `(A)` for a *matrix* $\mathbf{A} \in \mathbb{K}^{m,n}$, $\min\{m,n\} \geq 2$ returns the column vector $[(\mathbf{A})_{i,i}]_{i=1,\min\{m,n\}} \in \mathbb{K}^{\min\{m,n\}}$.

1.2.2 PYTHON

PYTHON is a widely used general-purpose and open source programming language. Together with the packages like NUMPY and MATPLOTLIB it delivers similar functionality like MATLAB for free. For interactive computing IPYTHON can be used. All those packages belong to the SCIPY ecosystem.

PYTHON features a good documentation and several scientific distributions are available (e.g. Anaconda, Enthought) which contain the most important packages. On most Linux-distributions the SCIPY ecosystem is also available in the software repository, as well as many other packages including for example the Spyder IDE delivered with Anaconda.

A good introduction tutorial to numerical PYTHON are the SCIPY-lectures. The full documentation of NUMPY and SCIPY can be found here. For former MATLAB-users there's also a guide. The scripts in this lecture notes follow the official PYTHON style guide.

Note that in PYTHON we have to import the numerical packages explicitly before use. This is normally done at the beginning of the file with lines like **import** `numpy` **as** `np` and **from** `matplotlib` **import** `pyplot` **as** `plt`. Those import statements are often skipped in this lecture notes to focus on the actual computations. But you can always assume the import statements as given here, e.g. `np.ravel(A)` is a call to a NUMPY function and `plt.loglog(x, y)` is a call to a MATPLOTLIB pyplot function.

PYTHON is not used in the current version of the lecture. Nevertheless a few PYTHON codes are supplied in order to convey similarities and differences to implementations in MATLAB and C++.

(1.2.8) Matrices and Vectors in PYTHON

The basic numeric data type in PYTHON are NUMPY's n-dimensional arrays. Vectors are normally implemented as 1D arrays and no distinction is made between row and column vectors. Matrices are represented as 2D arrays.

- ☞ `v = np.array([1, 2, 3])` creates a 1D array with the three elements 1, 2 and 3.
- ☞ `A = np.array([[1, 2], [3, 4]]` creates a 2D array.
- ☞ `A.shape` gives the n-dimensional size of an array.
- ☞ `A.size` gives the total number of entries in an array.

Note: There's also a matrix class in NUMPY with different semantics but its use is officially discouraged and it might even be removed in future release.

(1.2.9) Manipulating arrays in PYTHON

There are many possibilities listed in the documentation how to create, index and manipulate arrays.

An important difference to MATLAB is, that all arithmetic operations are normally performed element-wise, e.g. `A * B` is not the matrix-matrix product but element-wise multiplication (in MATLAB: `A.*A`). Also `A * v` does a broadcasted element-wise product. For the matrix product one has to use `np.dot(A, B)` or `A.dot(B)` explicitly.

1.2.3 EIGEN

Currently, the most widely used programming language for the development of new simulation software in scientific and industrial high-performance computing is C++. In this course we are going to use and discuss EIGEN as an example for a C++ library for numerical linear algebra ("embedded" domain specific language: DSL).

EIGEN is a header-only C++ template library designed to enable easy, natural and efficient numerical linear algebra: it provides data structures and a wide range of operations for matrices and vectors, see

below. EIGEN also implements many more fundamental algorithms (see the documentation page or the discussion below).

EIGEN relies on expression templates to allow the efficient evaluation of complex expressions involving matrices and vectors. Refer to the example given in the EIGEN documentation for details.

➥ Link to an "EIGEN Cheat Sheet" (quick reference relating to MATLAB commands)

(1.2.10) Compilation of codes using EIGEN

Compiling and linking on Mac OS X 10.10:

```
clang -D_HAS_CPP0X -std=c++11 -Wall -g \
     -Wno-deprecated-register -DEIGEN3_ACTIVATED \
     -I/opt/local/include -I/usr/local/include/eigen3  \
     -o main.cpp.o -c main.cpp
/usr/bin/c++ -std=c++11 -Wall -g -Wno-deprecated-register \
-DEIGEN3_ACTIVATED -Wl,-search_paths_first \
-Wl,-headerpad_max_install_names main.cpp.o \
-o executable /opt/local/lib/libboost_program_options-mt.dylib
```

Of course, different compilers may be used on different platforms. In all cases, basic EIGEN functionality can be used without linking with a special library. Usually the generation of such elaborate calls of the compiler is left to a build system like CMAKE.

(1.2.11) Matrix and vector data types in EIGEN

A generic matrix data type is given by the templated class

```
Matrix<typename Scalar,
       int RowsAtCompileTime, int ColsAtCompileTime>
```

Here **Scalar** is the underlying scalar type of the matrix entries, which must support the usual operations '+','-','*','/', and '+=', '*=', '', etc. Usually the scalar type will be either `double`, `float`, or `complex<>`. The cardinal template arguments `RowsAtCompileTime` and `ColsAtCompileTime` can pass a fixed size of the matrix, if it is known at compile time. There is a specialization selected by the template argument `Eigen::Dynamic` supporting variable size "dynamic" matrices.

C++11-code 1.2.12: Vector type and their use in EIGEN

```
#include <Eigen/Dense>

template<typename Scalar>
void eigenTypeDemo(unsigned int dim)
{
  // General dynamic (variable size) matrices
  using dynMat_t =
     Eigen::Matrix<Scalar,Eigen::Dynamic,Eigen::Dynamic>;
  // Dynamic (variable size) column vectors
  using dynColVec_t = Eigen::Matrix<Scalar,Eigen::Dynamic,1>;
  // Dynamic (variable size) row vectors
  using dynRowVec_t = Eigen::Matrix<Scalar,1,Eigen::Dynamic>;
  using index_t = typename dynMat_t::Index;
  using entry_t = typename dynMat_t::Scalar;

  // Declare vectors of size 'dim'; not yet initialized
  dynColVec_t colvec(dim);
  dynRowVec_t rowvec(dim);
  // Initialisation through component access
  for(index_t i=0; i< colvec.size(); ++i) colvec(i) = (Scalar)i;
  for(index_t i=0; i< rowvec.size(); ++i) rowvec(i) = (Scalar)1/(i+1);
  colvec[0] = (Scalar)3.14; rowvec[dim-1] = (Scalar)2.718;
  // Form tensor product, a matrix, see Section 1.3.1
  dynMat_t vecprod = colvec*rowvec;
  const int nrows = vecprod.rows();
  const int ncols = vecprod.cols();
}
```

Note that in Line 23 we could have relied on automatic type deduction via **auto** `vectprod = ...`. However, often it is safer to forgo this option and specify the type directly.

The following convenience data types are provided by EIGEN, see documentation:

- **MatrixXd** $\hat{=}$ generic variable size matrix with **d**ouble precision entries
- **VectorXd**, **RowVectorXd** $\hat{=}$ dynamic column and row vectors
  (**=** dynamic matrices with one dimension equal to 1)
- **MatrixNd** with $N = 2, 3, 4$ for small fixed size square $N \times N$-matrices (type `double`)
- **VectorNd** with $N = 2, 3, 4$ for small column vectors with fixed length N.

The `d` in the type name may be replaced with `i` (for `int`), `f` (for `float`), and `cd` (for `complex<double>`) to select another basic scalar type.

All matrix type feature the methods `cols()`, `rows()`, and `size()` telling the number of columns, rows, and total number of entries.

Access to individual matrix entries and vector components, both as Rvalue and Lvalue, is possible through the `()`-operator taking two arguments of type `index_t`. If only one argument is supplied, the matrix is accessed as a linear array according to its memory layout. For vectors, that is, matrices where one dimension is fixed to 1, the `[]`-operator can replace `()` with one argument, see Line 21 of Code 1.2.12.

(1.2.13) Initialization of *dense* matrices in EIGEN

The entry access operator `(int i,int j)` allows the most direct setting of matrix entries; there is hardly any runtime penalty.

Of course, in EIGEN dedicated functions take care of the initialization of the special matrices introduced in **??**:

```
Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n,n);
Eigen::MatrixXd O = Eigen::MatrixXd::Zero(n,m);
Eigen::MatrixXd D = d_vector.asDiagonal();
```

C++11-code 1.2.14: Initializing special matrices in EIGEN

```
#include <Eigen/Dense>
// Just allocate space for matrix, no initialisation
Eigen::MatrixXd A(rows,cols);
// Zero matrix.  Similar to matlab command zeros(rows, cols);
Eigen::MatrixXd B = MatrixXd::Zero(rows, cols);
// Ones matrix.  Similar to matlab command ones(rows, cols);
Eigen::MatrixXd C = MatrixXd::Ones(rows, cols);
// Matrix with all entries same as value.
Eigen::MatrixXd D = MatrixXd::Constant(rows, cols, value);
// Random matrix, entries uniformly distributed in [0,1]
Eigen::MatrixXd E = MatrixXd::Random(rows, cols);
// (Generalized) identity matrix, 1 on main diagonal
Eigen::MatrixXd I = MatrixXd::Identity(rows,cols);
std::cout << "size of A = (" << A.rows() << ',' << A.cols() << ')' <<
   std::endl;
```

A versatile way to initialize a matrix relies on a combination of the operators « and `,`, which allows the construction of a matrix from blocks:

```
MatrixXd mat3(6,6);
mat3 <<
    MatrixXd::Constant(4,2,1.5), // top row, first block
    MatrixXd::Constant(4,3,3.5), // top row, second block
    MatrixXd::Constant(4,1,7.5), // top row, third block
    MatrixXd::Constant(2,4,2.5), // bottom row, left block
    MatrixXd::Constant(2,2,4.5); // bottom row, right block
```

The matrix is filled top to bottom left to right, block dimensions have to match (like in MATLAB).

(1.2.15) Access to submatrices in EIGEN (→ documentation)

The method `block(int i,int j,int p,int q)` returns a reference to the submatrix with upper left corner at position (i,j) and size $p \times q$.

The methods `row(int i)` and `col(int j)` provide a reference to the corresponding row and column of the matrix. Even more specialised access methods are

```
topLeftCorner(p,q), bottomLeftCorner(p,q),
topRightCorner(p,q), bottomRightCorner(p,q),
topRows(q), bottomRows(q),
leftCols(p), and rightCols(q),
```

with obvious purposes.

C++11 code 1.2.16: Demonstration code for access to matrix blocks in EIGEN ⇝ GITLAB

```
template<typename MatType>
void blockAccess(Eigen::MatrixBase<MatType> &M)
{
  using index_t = typename Eigen::MatrixBase<MatType>::Index;
  using entry_t = typename Eigen::MatrixBase<MatType>::Scalar;
  const index_t nrows(M.rows()); // No. of rows
  const index_t ncols(M.cols()); // No. of columns

  cout << "Matrix M = " << endl << M << endl; // Print matrix
  // Block size half the size of the matrix
  index_t p = nrows/2,q = ncols/2;
  // Output submatrix with left upper entry at position (i,i)
  for(index_t i=0; i < min(p,q); i++)
    cout << "Block (" << i << ',' << i << ',' << p << ',' << q
         << ") = " << M.block(i,i,p,q) << endl;
  // l-value access: modify sub-matrix by adding a constant
  M.block(1,1,p,q) += Eigen::MatrixBase<MatType>::Constant(p,q,1.0);
  cout << "M = " << endl << M << endl;
  // r-value access: extract sub-matrix
  MatrixXd B = M.block(1,1,p,q);
  cout << "Isolated modified block = " << endl << B << endl;
  // Special sub-matrices
  cout << p << " top rows of m = " << M.topRows(p) << endl;
  cout << p << " bottom rows of m = " << M.bottomRows(p) << endl;
  cout << q << " left cols of m = " << M.leftCols(q) << endl;
  cout << q << " right cols of m = " << M.rightCols(p) << endl;
  // r-value access to upper triangular part
  const MatrixXd T = M.template triangularView<Upper>(); //
  cout << "Upper triangular part = " << endl << T << endl;
  // l-value access to upper triangular part
  M.template triangularView<Lower>() *= -1.5; //
  cout << "Matrix M = " << endl << M << endl;
}
```

EIGEN offers views for access to triangular parts of a matrix, see Line 29 and Line 32, according to

```
M.triangularView<XX>()
```

where `XX` can stand for one of the following: `Upper`, `Lower`, `StrictlyUpper`, `StrictlyLower`, `UnitUpper`, `UnitLower`, see documentation.

For column and row vectors references to sub-vectors can be obtained by the methods `head(int length)`, `tail(int length)`, and `segment(int pos,int length)`.

Note: Unless the preprocessor switch `NDEBUG` is set, EIGEN performs range checks on all indices.

Since operators like MATLAB's `.*` are not available, EIGEN uses the Array concept to furnish entry-wise operations on matrices. An EIGEN-Array contains the same data as a matrix, supports the same methods for initialisation and access, but replaces the operators of matrix arithmetic with entry-wise actions. Matrices and arrays can be converted into each other by the `array()` and `matrix()` methods, see documentation for details.

C++11 code 1.2.18: Using Array in EIGEN ⇝ GITLAB

```
2  void matArray(int nrows,int ncols){
3    Eigen::MatrixXd m1(nrows,ncols),m2(nrows,ncols);
4    for(int i = 0; i < m1.rows(); i++)
5      for(int j=0; j < m1.cols(); j++) {
6        m1(i,j) = (double)(i+1)/(j+1);
7        m2(i,j) = (double)(j+1)/(i+1);
8      }
9    // Entry-wise product, not a matrix product
10   Eigen::MatrixXd m3 = (m1.array() * m2.array()).matrix();
11   // Explicit entry-wise operations on matrices are possible
12   Eigen::MatrixXd m4(m1.cwiseProduct(m2));
13   // Entry-wise logarithm
14   cout << "Log(m1) = " << endl << log(m1.array()) << endl;
15   // Entry-wise boolean expression, true cases counted
16   cout << (m1.array() > 3).count() << " entries of m1 > 3" << endl;
17 }
```

The application of a functor (→ Section 0.2.3) to all entries of a matrix can also been done via the `unaryExpr()` method of a matrix:

```
// Apply a lambda function to all entries of a matrix
auto fnct = [](double x) { return (x+1.0/x); };
cout << "f(m1) = " << endl << m1.unaryExpr(fnct) << endl;
```

Remark 1.2.19 (EIGEN in use)

- ☞ EIGEN is used as one of the base libraries for the Robot Operating System (ROS), an open source project with strong ETH participation.
- ☞ The geometry processing library libigl uses EIGEN as its basic linear algebra engine. At ETH it is being used and developed at ETH Zurich, at the Interactive Geometry Lab and Advanced Technologies Lab.

1.2.4 (Dense) Matrix storage formats

All numerical libraries store the entries of a (generic = *dense*) matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ in a linear array of length mn (or longer). Accessing entries entails suitable index computations.

Two natural options for "vectorisation" of a matrix: *row major, column major*

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

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

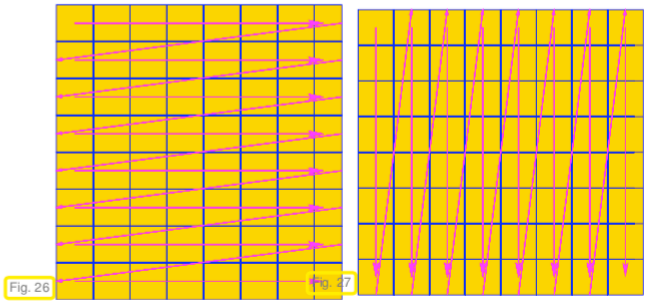
Access to entry $(\mathbf{A})_{ij}$ of $\mathbf{A} \in \mathbb{K}^{n,m}$, $i = 1,\ldots,n$, $j = 1,\ldots,m$:

row major:

$$(\mathbf{A})_{ij} \leftrightarrow \texttt{A\_arr(m*(i-1)+(j-1))}$$

column major:

$$(\mathbf{A})_{ij} \leftrightarrow \texttt{A\_arr(n*(j-1)+(i-1))}$$



row major column major

Example 1.2.20 (Accessing matrix data as a vector)

Both in MATLAB and EIGEN the single index access operator relies on the linear data layout: In MATLAB

```
A = [1 2 3;4 5 6;7 8 9]; A(:)',
```

produces the terminal output

```
1     4     7     2     5     8     3     6     9
```

which clearly reveals the *column major* storage format.

In PYTHON the default data layout is row major, but it can be explicitly set. Further, array transposition does not change any data, but only the memory order and array shape.

In PYTHON the default data layout is row major, but it can be explicitly set. Further, array transposition does not change any data, but only the memory order and array shape.

PYTHON-code 1.2.21: Storage order in PYTHON

```
# array creation
A = np.array([[1, 2], [3, 4]])  # default (row major) storage
B = np.array([[1, 2], [3, 4]], order='F')  # column major storage

# show internal storage
np.ravel(A, 'K')  # array elements as stored in memory:  [1, 2, 3, 4]
np.ravel(B, 'K')  # array elements as stored in memory:  [1, 3, 2, 4]

# nothing happens to the data on transpose, just the storage order changes
np.ravel(A.T, 'K')  # array elements as stored in memory:  [1, 2, 3, 4]
np.ravel(B.T, 'K')  # array elements as stored in memory:  [1, 3, 2, 4]

# storage order can be accessed by checking the array's flags
A.flags['C_CONTIGUOUS']  # True
B.flags['F_CONTIGUOUS']  # True
A.T.flags['F_CONTIGUOUS']  # True
B.T.flags['C_CONTIGUOUS']  # True
```

In EIGEN the data layout can be controlled by a template argument; default is column major.

C++11 code 1.2.22: Single index access of matrix entries in EIGEN ⇢ GITLAB

```
2  void storageOrder(int nrows=6,int ncols=7)
3  {
4    cout << "Different matrix storage layouts in Eigen" << endl;
5    // Template parameter ColMajor selects column major data layout
6    Matrix<double,Dynamic,Dynamic,ColMajor> mcm(nrows,ncols);
7    // Template parameter RowMajor selects row major data layout
8    Matrix<double,Dynamic,Dynamic,RowMajor> mrm(nrows,ncols);
9    // Direct initialization; lazy option:  use int as index type
10   for (int l=1,i=0; i<nrows; i++)
11     for (int j=0; j<ncols; j++,l++)
12       mcm(i,j) = mrm(i,j) = l;
13
14   cout << "Matrix mrm = " << endl << mrm << endl;
15   cout << "mcm linear = ";
16   for (int l=0;l < mcm.size(); l++) cout << mcm(l) << ',';
17   cout << endl;
18
19   cout << "mrm linear = ";
20   for (int l=0;l < mrm.size(); l++) cout << mrm(l) << ',';
21   cout << endl;
22 }
```

The function call `storageOrder(3,3)`, *cf.* Code 1.2.22 yields the output

```
Different matrix storage layouts in Eigen
Matrix mrm =
1 2 3
4 5 6
7 8 9
mcm linear = 1,4,7,2,5,8,3,6,9,
mrm linear = 1,2,3,4,5,6,7,8,9,
```

Remark 1.2.23 (Vectorisation of a matrix)

Mapping a column-major matrix to a column vector with the same number of entries is called vectorization or linearization in numerical linear algebra, in symbols

$$\mathrm{vec} : \mathbb{K}^{n,m} \to \mathbb{K}^{n\cdot m} \quad , \quad \mathrm{vec}(\mathbf{A}) = \begin{bmatrix} (\mathbf{A})_{:,1} \\ (\mathbf{A})_{:,2} \\ \vdots \\ (\mathbf{A})_{:,m} \end{bmatrix} . \tag{1.2.24}$$

Remark 1.2.25 (MATLAB command reshape)

matlab offers the built-in command **reshape** for changing the dimensions of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$:

```
B = reshape(k,l,A);     % error, in case kl ≠ mn
```

This command will create an $k \times l$-matrix by just reinterpreting the linear array of entries of `A` as data for a matrix with `k` rows and `l` columns. Regardless of the size and entries of the matrices the following test will always produce a `equal = true` result

```
if (()prod(size(A)) ~= (k*l)), error('Size mismatch'); end
B = reshape(A,k,l);
equal = (B(:) == A(:));
```

Remark 1.2.26 (NUMPY function reshape)

NUMPY offers the function `np.reshape` for changing the dimensions of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$:

```
# read elements of A in row major order (default)
B = np.reshape(A, (k, l))   # error, in case kl ≠ mn
B = np.reshape(A, (k, l), order='C')   # same as above
# read elements of A in column major order
B = np.reshape(A, (k, l), order='F')
# read elements of A as stored in memory
B = np.reshape(A, (k, l), order='A')
```

This command will create an $k \times l$-array by reinterpreting the array of entries of `A` as data for an array with `k` rows and `l` columns. The order in which the elements of `A` are be read can be set by the `order` argument to row major (default, `'C'`), column major (`'F'`) or `A`'s internal storage order, i.e. row major if `A` is row major or column major if `A` is column major (`'A'`).

Remark 1.2.27 (Reshaping matrices in EIGEN)

If you need a reshaped view of a matrix' data in EIGEN you can obtain it via the raw data vector belonging to the matrix. Then use this information to create a matrix view by means of `Map` → documentation.

C++11 code 1.2.28: Demonstration on how reshape a matrix in EIGEN ↬ GITLAB

```
2 template<typename MatType>
3 void reshapetest(MatType &M)
4 {
5   using index_t = typename MatType::Index;
6   using entry_t = typename MatType::Scalar;
7   const index_t nsize(M.size());
```

```
8
9   // reshaping possible only for matrices with non-prime dimensions
10    if ((nsize %2) == 0) {
11      entry_t *Mdat = M.data(); // raw data array for M
12      // Reinterpretation of data of M
13      Map<Eigen::Matrix<entry_t,Dynamic,Dynamic>> R(Mdat,2,nsize/2);
14      // (Deep) copy data of M into matrix of different size
15      Eigen::Matrix<entry_t,Dynamic,Dynamic> S =
16        Map<Eigen::Matrix<entry_t,Dynamic,Dynamic>>(Mdat,2,nsize/2);
17
18      cout << "Matrix M = " << endl << M << endl;
19      cout << "reshaped to " << R.rows() << 'x' << R.cols()
20        << " = " << endl << R << endl;
21      // Modifying R affects M, because they share the data space !
22      R *= -1.5;
23      cout << "Scaled (!) matrix M = " << endl << M << endl;
24      // Matrix S is not affected, because of deep copy
25      cout << "Matrix S = " << endl << S << endl;
26    }
27  }
```

This function has to be called with a mutable (l-value) matrix type object. A sample output is printed next:

```
1  Matrix M =
2   0 -1 -2 -3 -4 -5 -6
3   1  0 -1 -2 -3 -4 -5
4   2  1  0 -1 -2 -3 -4
5   3  2  1  0 -1 -2 -3
6   4  3  2  1  0 -1 -2
7   5  4  3  2  1  0 -1
8  reshaped to 2x21 =
9   0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1 -6 -4 -2
10  1  3  5  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1
11 Scaled (!) matrix M =
12   -0  1.5    3  4.5    6  7.5    9
13 -1.5   -0  1.5    3  4.5    6  7.5
14   -3 -1.5   -0  1.5    3  4.5    6
15 -4.5   -3 -1.5   -0  1.5    3  4.5
16   -6 -4.5   -3 -1.5   -0  1.5    3
17 -7.5   -6 -4.5   -3 -1.5   -0  1.5
18 Matrix S =
19  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1 -6 -4 -2
20  1  3  5  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1
```

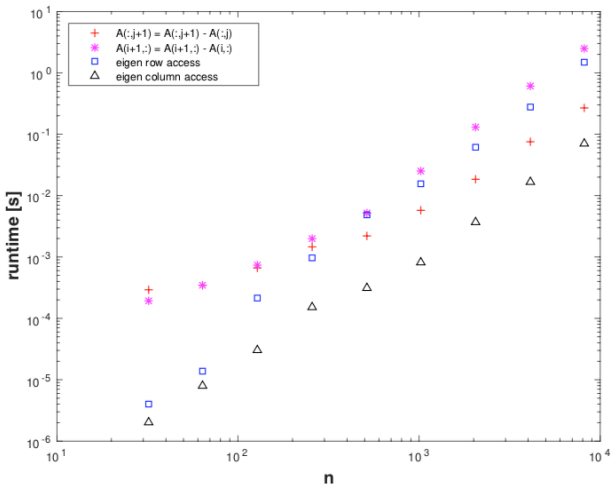
C++11 code 1.2.31: Timing for row and column oriented matrix access for EIGEN ↪ GITLAB

```
void rowcolaccesstiming(void)
{
  const int K = 3; // Number of repetitions
  const int N_min = 5; // Smalles matrix size 32
  const int N_max = 13; // Scan until matrix size of 8192
  unsigned long n = (1L << N_min);
  Eigen::MatrixXd times(N_max-N_min+1,3);

  for(int l=N_min; l<= N_max; l++, n*=2) {
    Eigen::MatrixXd A = Eigen::MatrixXd::Random(n,n);
    double t1 = 1000.0;
    for(int k=0;k<K;k++) {
      auto tic =  high_resolution_clock::now();
      for(int j=0; j < n-1; j++) A.row(j+1) -= A.row(j); // row access
      auto toc = high_resolution_clock::now();
      double t =
         (double)duration_cast<microseconds>(toc-tic).count()/1E6;
      t1 = std::min(t1,t);
    }
    double t2 = 1000.0;
    for(int k=0;k<K;k++) {
      auto tic =  high_resolution_clock::now();
      for(int j=0; j < n-1; j++) A.col(j+1) -= A.col(j); //column
          access
      auto toc = high_resolution_clock::now();
      double t =
         (double)duration_cast<microseconds>(toc-tic).count()/1E6;
      t2 = std::min(t2,t);
    }
    times(l-N_min,0) = n;      times(l-N_min,1) = t1;
       times(l-N_min,2) = t2;
  }
  std::cout << times << std::endl;
}
```

PYTHON-code 1.2.32: Timing for row and column oriented matrix access in PYTHON

```
import numpy as np
import timeit
from matplotlib import pyplot as plt

def col_wise(A):
    for j in range(A.shape[1] - 1):
        A[:, j + 1] -= A[:, j]

def row_wise(A):
    for i in range(A.shape[0] - 1):
        A[i + 1, :] -= A[i, :]

```

```
# Timing for row/column-wise operations on matrix, we conduct k runs in order
# to reduce risk of skewed measurements due to OS activity during run.

k = 3
res = []
for n in 2**np.mgrid[4:14]:
    A = np.random.normal(size=(n, n))

    t1 = min(timeit.repeat(lambda: col_wise(A), repeat=k, number=1))
    t2 = min(timeit.repeat(lambda: row_wise(A), repeat=k, number=1))

    res.append((n, t1, t2))

# plot runtime versus matrix sizes
ns, t1s, t2s = np.transpose(res)

plt.figure()
plt.plot(ns, t1s, '+', label='A[:, j + 1] -= A[:, j]')
plt.plot(ns, t2s, 'o', label='A[i + 1, :] -= A[i, :]')
plt.xlabel(r'$n$')
plt.ylabel(r'runtime [s]')
plt.legend(loc='upper left')
plt.savefig('../PYTHON_PICTURES/accessrtlin.eps')

plt.figure()
plt.loglog(ns, t1s, '+', label='A[:, j + 1] -= A[:, j]')
plt.loglog(ns, t2s, 'o', label='A[i + 1, :] -= A[i, :]')
plt.xlabel(r'$n$')
plt.ylabel(r'runtime [s]')
plt.legend(loc='upper left')
plt.savefig('../PYTHON_PICTURES/accessrtlog.eps')

plt.show()
```



Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz $\times$ 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3, -DNDEBUG

The compiler flags `-O3` and `-DNDEBUG` are essential. The C++ code would be significantly slower if the default compiler options were used!

For both MATLAB and EIGEN codes we observe a glaring discrepancy of CPU time required for accessing entries of a matrix in rowwise or columnwise fashion. This reflects the impact of features of the unterlying hardware architecture, like cache size and memory bandwidth:

Interpretation of timings: Since matrices in MATLAB are stored column major all the matrix elements in a column occupy contiguous memory locations, which will all reside in the cache together. Hence, column oriented access will mainly operate on data in the cache even for large matrices. Conversely, row oriented access addresses matrix entries that are stored in distant memory locations, which incurs frequent cash misses (cache thrashing).

The impact of hardware architecture on the performance of algorithms will **not** be taken into account in this course, because hardware features tend to be both intricate and ephemeral. However, for modern high performance computing it is essential to adapt implementations to the hardware on which the code is supposed to run.

1.3 Basic linear algebra operations

First we refresh the basic rules of vector and matrix calculus. Then we will learn about a very old programming interface for simple dense linear algebra operations.

1.3.1 Elementary matrix-vector calculus

What you should know from linear algebra [10, Sect. 2.2]:

- vector space operations in matrix space $\mathbb{K}^{m,n}$ (addition, multiplication with scalars)
- dot product: $\mathbf{x},\mathbf{y} \in \mathbb{K}^n$, $n \in \mathbb{N}$: $\mathbf{x}\cdot\mathbf{y} := \mathbf{x}^{\mathrm{H}}\mathbf{y} = \sum_{i=1}^{n} \bar{x}_i y_i \in \mathbb{K}$
  (in EIGEN: `x.dot(y)` or `x.adjoint()*y`, `x`,`y` $\hat{=}$ column vectors)
- tensor product: $\mathbf{x} \in \mathbb{K}^m, \mathbf{y} \in \mathbb{K}^n$, $n \in \mathbb{N}$: $\mathbf{x}\mathbf{y}^{\mathrm{H}} = \left(x_i \bar{y}_j\right)_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} \in \mathbb{K}^{m,n}$
  (in EIGEN: `x*y.adjoint()`, `x`,`y` $\hat{=}$ column vectors)
- All are special cases of the matrix product:

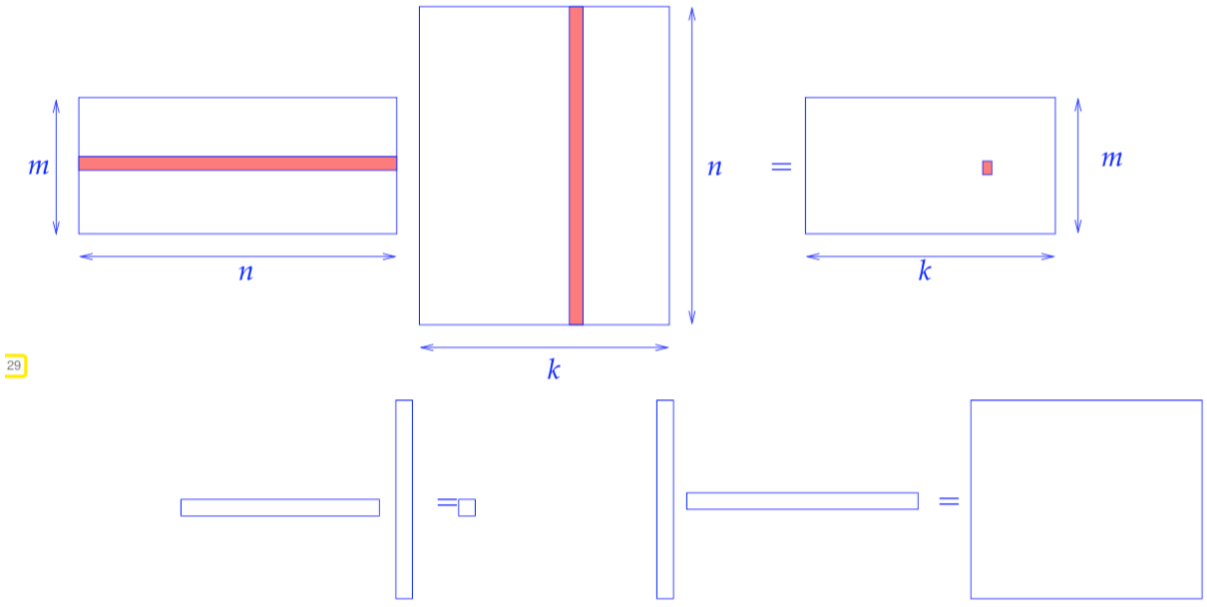
$$\mathbf{A} \in \mathbb{K}^{m,n}, \quad \mathbf{B} \in \mathbb{K}^{n,k}: \qquad \mathbf{AB} = \left[\sum_{j=1}^{n} a_{ij} b_{jl}\right]_{\substack{i=1,\ldots,m \\ l=1,\ldots,k}} \in \mathbb{R}^{m,k}. \tag{1.3.1}$$

Recall from linear algebra basic properties of the matrix product: for all $\mathbb{K}$-matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ (of suitable sizes), $\alpha, \beta \in \mathbb{K}$

$$\begin{aligned} \text{associative:} &\quad (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}), \\ \text{bi-linear:} &\quad (\alpha\mathbf{A}+\beta\mathbf{B})\mathbf{C} = \alpha(\mathbf{AC}) + \beta(\mathbf{BC}), \quad \mathbf{C}(\alpha\mathbf{A}+\beta\mathbf{B}) = \alpha(\mathbf{CA}) + \beta(\mathbf{CB}), \\ \text{non-commutative:} &\quad \mathbf{AB} \neq \mathbf{BA} \quad \text{in general}. \end{aligned}$$

(1.3.2) Visualisation of (special) matrix products

Dependency of an entry of a product matrix:



29

Remark 1.3.3 (Row-wise & column-wise view of matrix product)

To understand what is going on when forming a matrix product, it is often useful to decompose it into matrix×vector operations in one of the following two ways:

$\mathbf{A} \in \mathbb{K}^{m,n}$, $\mathbf{B} \in \mathbb{K}^{n,k}$:

$$\mathbf{AB} = \underbrace{\begin{bmatrix} \mathbf{A}(\mathbf{B})_{:,1} & \dots & \mathbf{A}(\mathbf{B})_{:,k} \end{bmatrix}}_{\text{matrix assembled from columns}} \quad , \quad \mathbf{AB} = \underbrace{\begin{bmatrix} (\mathbf{A})_{1,:}\mathbf{B} \\ \vdots \\ (\mathbf{A})_{m,:}\mathbf{B} \end{bmatrix}}_{\text{matrix assembled from rows}} . \tag{1.3.4}$$

For notations refer to Sect. 1.1.1.

Remark 1.3.5 (Understanding the structure of product matrices)

A "mental image" of matrix multiplication is useful for telling special properties of product matrices.

For instance, zero blocks of the product matrix can be predicted easily in the following situations using the idea explained in Rem. 1.3.3 (try to understand how):

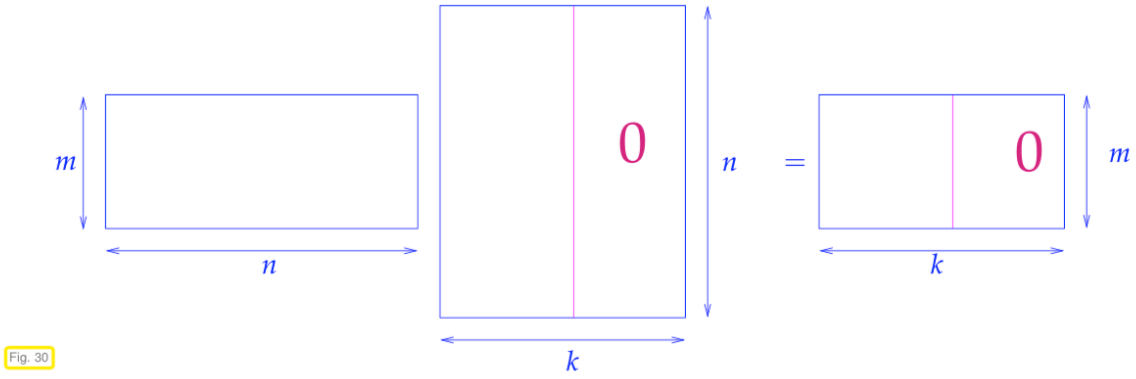


Fig. 30

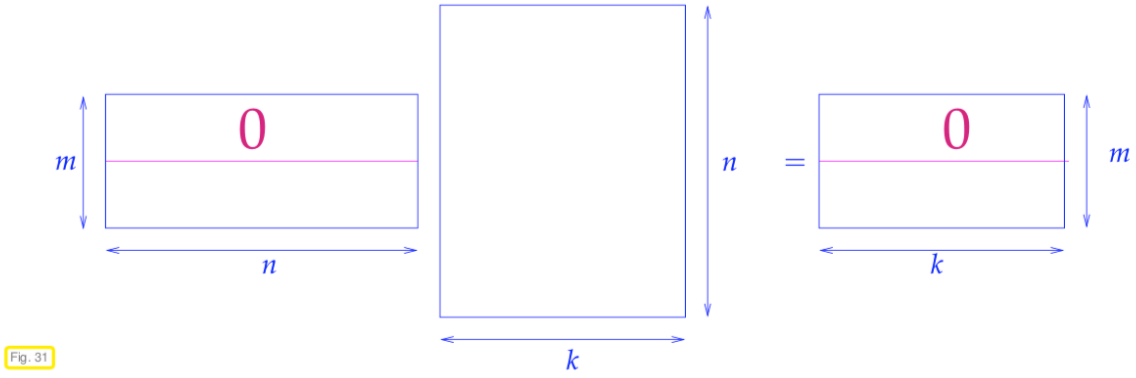


Fig. 31

A clear understanding of matrix multiplication enables you to "see", which parts of a matrix factor matter in a product:

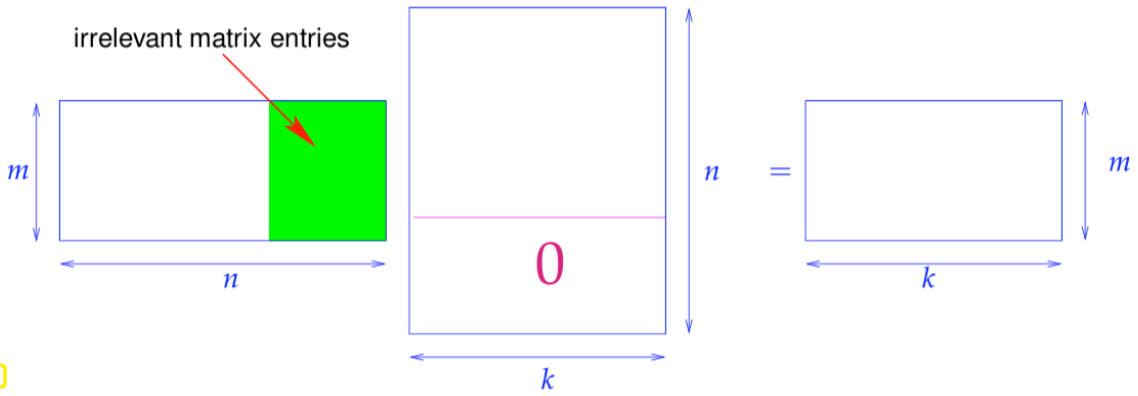
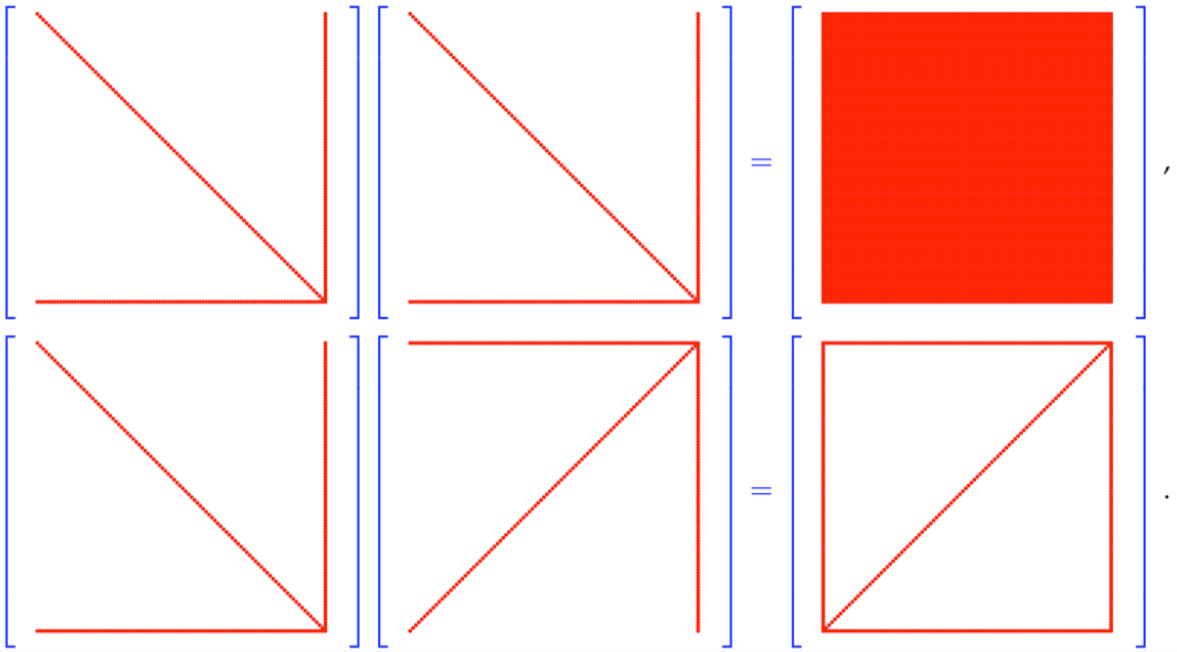


Fig. 32

"Seeing" the structure/pattern of a matrix product:



These nice renderings of the so-called patterns of matrices, that is, the distribution of their non-zero entries have been created by a special EIGEN/**Figure**-command for visualizing the structure of a matrix: `fig.spy(M)`

C++11 code 1.3.6: Visualizing the structure of matrices in EIGEN → GITLAB

```
2  #include <Eigen/Dense>
3
4  #include <figure/figure.hpp>
5
6  using namespace Eigen;
7
8  int main() {
9      int n = 100;
10     MatrixXd A(n,n), B(n,n); A.setZero(); B.setZero();
11     A.diagonal() = VectorXd::LinSpaced(n,1,n);
12     A.col(n-1) = VectorXd::LinSpaced(n,1,n);
```

```
13      A.row(n-1) = RowVectorXd::LinSpaced(n,1,n);
14      B = A.colwise().reverse();
15      MatrixXd C = A*A, D = A*B;
16      mgl::Figure fig1, fig2, fig3, fig4;
17      fig1.spy(A);    fig1.save("Aspy_cpp");
18      fig2.spy(B);    fig2.save("Bspy_cpp");
19      fig3.spy(C);    fig3.save("Cspy_cpp");
20      fig4.spy(D);    fig4.save("Dspy_cpp");
21      return 0;
22  }
```

This code also demonstrates the use of `diagonal()`, `col()`, `row()` for L-value access to parts of a matrix.

PYTHON/MATPLOTLIB-command for visualizing the structure of a matrix: `plt.spy(M)`

PYTHON-code 1.3.7: Visualizing the structure of matrices in PYTHON

```
1 n = 100
2 A = np.diag(np.mgrid[:n])
3 A[:, -1] = A[-1, :] = np.mgrid[:n]
4 plt.spy(A)
5 plt.spy(A[::-1, :])
6 plt.spy(np.dot(A, A))
7 plt.spy(np.dot(A, B))
```

Remark 1.3.8 (Multiplying triangular matrices)

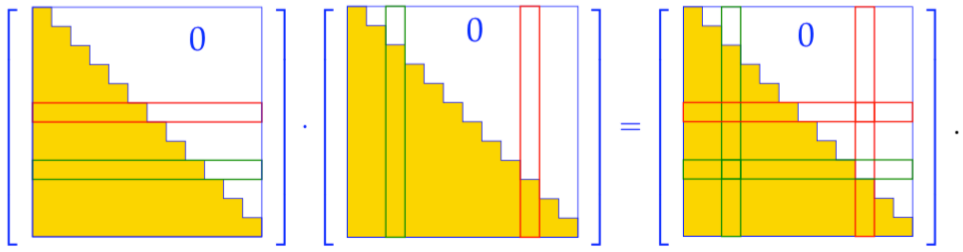
The following result is useful when dealing with matrix decompositions that often involve triangular matrices.

Lemma 1.3.9. Group of regular diagonal/triangular matrices

$$\mathbf{A},\mathbf{B} \begin{Bmatrix} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{Bmatrix} \Rightarrow \mathbf{AB} \text{ and } \mathbf{A}^{-1} \begin{Bmatrix} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{Bmatrix}.$$

(assumes that $\mathbf{A}$ is regular)

"Proof by visualization" → Rem. 1.3.5



Experiment 1.3.10 (Scaling a matrix)

Scaling = multiplication with diagonal matrices (with non-zero diagonal entries):

It is important to know the different effect of multiplying with a diagonal matrix from left or right:

- multiplication with diagonal matrix *from left* ➤ row scaling

$$\begin{bmatrix} d_1 & 0 & & 0 \\ 0 & d_2 & & 0 \\ & & \ddots & \\ 0 & 0 & & d_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} = \begin{bmatrix} d_1 a_{11} & d_1 a_{12} & \dots & d_1 a_{1m} \\ d_2 a_{21} & d_2 a_{22} & \dots & d_2 a_{2m} \\ \vdots & & & \vdots \\ d_n a_{n1} & d_n a_{n2} & \dots & d_n a_{nm} \end{bmatrix} = \begin{bmatrix} d_1 (\mathbf{A})_{1,:} \\ \vdots \\ d_n (\mathbf{A})_{n,:} \end{bmatrix}.$$

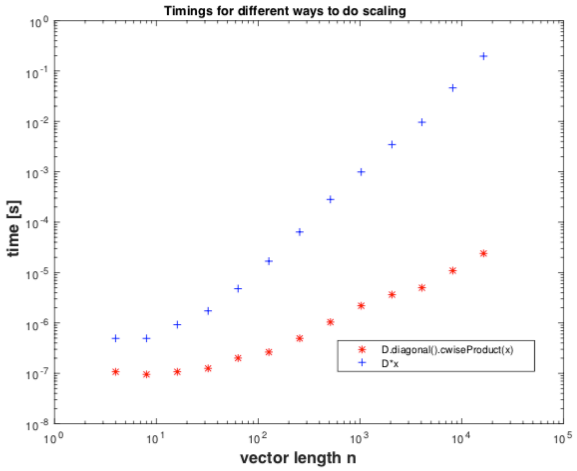
- multiplication with diagonal matrix *from right* ➤ column scaling

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \begin{bmatrix} d_1 & 0 & & 0 \\ 0 & d_2 & & 0 \\ & & \ddots & \\ 0 & 0 & & d_m \end{bmatrix} = \begin{bmatrix} d_1 a_{11} & d_2 a_{12} & \dots & d_m a_{1m} \\ d_1 a_{21} & d_2 a_{22} & \dots & d_m a_{2m} \\ \vdots & & & \vdots \\ d_1 a_{n1} & d_2 a_{n2} & \dots & d_m a_{nm} \end{bmatrix}$$

$$= \begin{bmatrix} d_1 (\mathbf{A})_{:,1} & \dots & d_m (\mathbf{A})_{:,m} \end{bmatrix}.$$

Multiplication with a scaling matrix $\mathbf{D} := \mathrm{diag}(d_1, \ldots, d_n) \in \mathbb{R}^{n,n}$ in EIGEN can be realised in three ways, see Code 1.3.11, **??** 9-**??** 1 **??** 13, and **??** 15.

The code will be slowed down massively in cas a temporary dense matrix is created inadvertentl Notice that EIGEN's expression templates avoid th pointless effort, see **??** 15.



C++11 code 1.3.11: Timing multiplication with scaling matrix in EIGEN ↪ GITLAB

```
2   int nruns = 3, minExp = 2, maxExp = 14;
3   MatrixXd tms(maxExp-minExp+1,4);
4   for(int i = 0; i <= maxExp-minExp; ++i){
5     Timer tbad, tgood, topt;  // timer class
6     int n = std::pow(2, minExp + i);
7     VectorXd d = VectorXd::Random(n,1), x = VectorXd::Random(n,1),
          y(n);
8     for(int j = 0; j < nruns; ++j) {
9       MatrixXd D = d.asDiagonal(); //
10      // matrix vector multiplication
11      tbad.start(); y = D*x; tbad.stop(); //
12      // componentwise multiplication
13      tgood.start(); y= d.cwiseProduct(x); tgood.stop(); //
14      // matrix multiplication optimized by Eigen
15      topt.start(); y = d.asDiagonal()*x; topt.stop(); //
16    }
17    tms(i,0)=n;
18    tms(i,1)=tgood.min(); tms(i,2)=tbad.min(); tms(i,3)= topt.min();
19  }
```

PYTHON-code 1.3.12: Timing multiplication with scaling matrix in PYTHON

```
import numpy as np
from matplotlib import pyplot as plt
import timeit

# script for timing a smart and foolish way to carrry out
# multiplication with a scaling matrix

nruns = 3
res = []
for n in 2**np.mgrid[2:15]:
    d = np.random.uniform(size=n)
    x = np.random.uniform(size=n)

    tbad = min(timeit.repeat(lambda: np.dot(np.diag(d), x),
                             repeat=nruns, number=1))
    tgood = min(timeit.repeat(lambda: d * x, repeat=nruns,
                              number=1))

    res.append((n, tbad, tgood))

ns, tbads, tgoods = np.transpose(res)
plt.figure()
plt.loglog(ns, tbads, '+', label='using np.diag')
plt.loglog(ns, tgoods, 'o', label='using *')
plt.legend(loc='best')
plt.title('Timing for different ways to do scaling')
plt.savefig('../PYTHON_PICTURES/scaletiming.eps')
plt.show()
```

Hardly surprising, the component-wise multiplication of the two vectors is way faster than the intermittent initialisation of a diagonal matrix (main populated by zeros) and the computation of a matrix×vector product. Nevertheless, such blunders keep on haunting numerical codes. Do not rely solely on EIGEN optimizations!

Remark 1.3.13 (Row and column transformations)

Simple operations on rows/columns of matrices, *cf.* what was done in Exp. 1.2.29, can often be expressed as multiplication with special matrices: For instance, given $\mathbf{A} \in \mathbb{K}^{n,m}$ we obtain $\mathbf{B}$ by adding row $(\mathbf{A})_{j,:}$ to row $(\mathbf{A})_{j+1,:}$, $1 \leq j < n$.

Realisation through matrix product ⟹

$$\mathbf{B} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & 1 & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix} \mathbf{A} \,.$$

The matrix multiplying $\mathbf{A}$ from the left is a specimen of a transformation matrix, a matrix that coincides with the identity matrix $\mathbf{I}$ except for a single off-diagonal entry.

left-multiplication right-multiplication	with transformation matrices	↣	row transformations column transformations

row/column transformations will play a central role in Sect. 2.3

Remark 1.3.14 (Matrix algebra)

A vector space $(V, \mathbb{K}, +, \cdot)$, where V is additionally equipped with a bi-linear and associative "multiplication" is called an algebra. Hence, the vector space of square matrices $\mathbb{K}^{n,n}$ with matrix multiplication is an algebra with *unit element* $\mathbf{I}$.

(1.3.15) Block matrix product

Given matrix dimensions $M, N, K \in \mathbb{N}$ block sizes $1 \leq n < N$ ($n' := N - n$), $1 \leq m < M$ ($m' := M - m$), $1 \leq k < K$ ($k' := K - k$) we start from the following matrices:

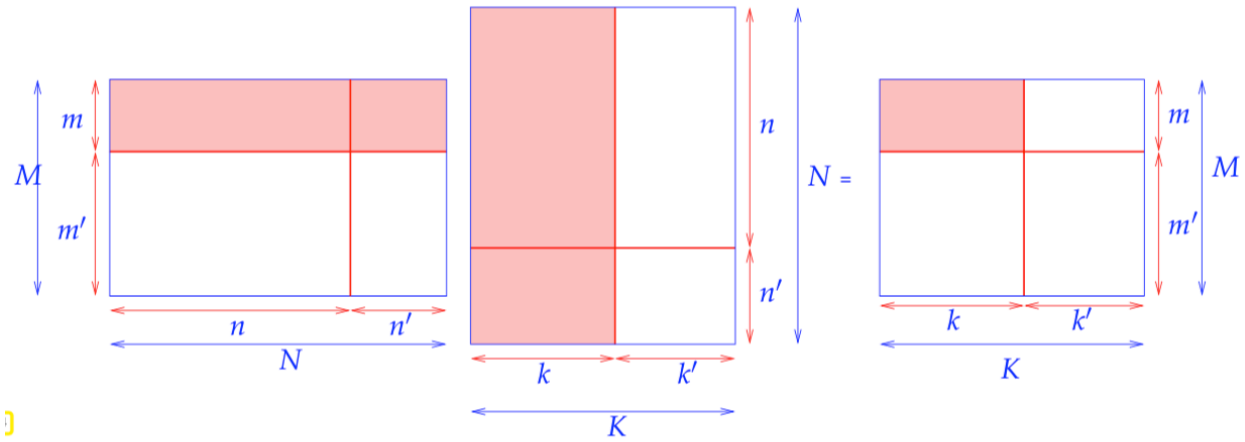
$$\begin{matrix} \mathbf{A}_{11} \in \mathbb{K}^{m,n} & \mathbf{A}_{12} \in \mathbb{K}^{m,n'} \\ \mathbf{A}_{21} \in \mathbb{K}^{m',n} & \mathbf{A}_{22} \in \mathbb{K}^{m',n'} \end{matrix} \quad , \quad \begin{matrix} \mathbf{B}_{11} \in \mathbb{K}^{n,k} & \mathbf{B}_{12} \in \mathbb{K}^{n,k'} \\ \mathbf{B}_{21} \in \mathbb{K}^{n',k} & \mathbf{B}_{22} \in \mathbb{K}^{n',k'} \end{matrix} .$$

This matrices serve as sub-matrices or matrix blocks and are assembled into larger matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \in \mathbb{K}^{M,N} \quad , \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \in \mathbb{K}^{N,K} .$$

It turns out that the matrix product $\mathbf{AB}$ can be computed by the same formula as the product of simple 2×2-matrices:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix} . \tag{1.3.16}$$



Bottom line: one can compute with block-structured matrices in *almost* ($*$) the same ways as with matrices with real/complex entries, see [12, Sect. 1.3.3].

($*$): you must not use the commutativity of multiplication (because matrix multiplication is not commutative).

1.3.2 BLAS – Basic Linear Algebra Subprograms

BLAS (Basic Linear Algebra Subprograms) is a specification (API) that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. They are the de facto low-level routines for linear algebra libraries (Wikipedia).

The BLAS API is standardised by the BLAS technical forum and, due to its history dating back to the 70s, follows conventions of FORTRAN 77, see the Quick Reference Guide for examples. However, wrappers for other programming languages are available. CPU manufacturers and/or developers of operating systems usually supply highly optimised implementations:

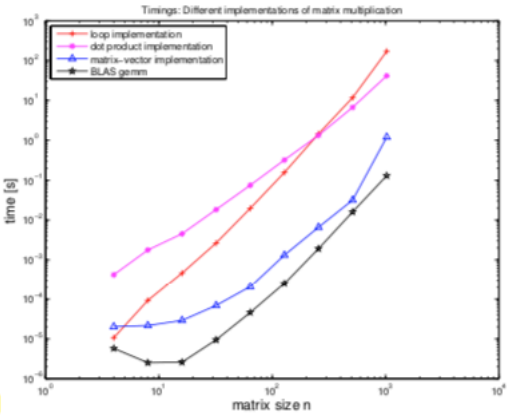
- OpenBLAS: open source implementation with some general optimisations, available under BSD license.
- ATLAS (Automatically Tuned Linear Algebra Software): open source BLAS implementation with auto-tuning capabilities. Comes with C and FORTRAN interfaces and is included in Linux distributions.
- Intel MKL (Math Kernel Library): commercial highly optimised BLAS implemetation available for all Intel CPUs. Used by most proprietory simulation software and also MATLAB.

Experiment 1.3.17 (Multiplying matrices in MATLAB)

MATLAB-code 1.3.18: Timing different implementations of matrix multiplication in MATLAB

```
% MATLAB script for timing different implementations of matrix
    multiplications
nruns = 3; times = [];
for n=2.^(2:10) % n = size of matrices
  fprintf('matrix size n = %d\n',n);
```

```
  A = rand(n,n); B = rand(n,n); C = zeros(n,n);
  t1 = realmax;
  % loop based implementation (no BLAS)
  for l=1:nruns
    tic;
    for i=1:n, for j=1:n
        for k=1:n, C(i,j) = C(i,j) + A(i,k)*B(k,j); end
      end, end
    t1 = min(t1,toc);
    end
  t2 = realmax;
  % dot product based implementation (BLAS level 1)
  for l=1:nruns
    tic;
    for i=1:n
      for j=1:n, C(i,j) = dot(A(i,:),B(:,j));  end
    end
    t2 = min(t2,toc);
  end
  t3 = realmax;
  % matrix-vector based implementation (BLAS level 2)
  for l=1:nruns
    tic;
    for j=1:n, C(:,j) = A*B(:,j); end
    t3 = min(t3,toc);
  end
  t4 = realmax;
  % BLAS level 3 matrix multiplication
  for l=1:nruns
    tic; C = A*B; t4 = min(t4,toc);
  end
  times = [ times; n t1 t2 t3 t4];
end

figure('name','mmtiming');
loglog(times(:,1),times(:,2),'r+-',...
     times(:,1),times(:,3),'m*-',...
     times(:,1),times(:,4),'b^-',...
     times(:,1),times(:,5),'kp-');
title('Timings: Different implementations of matrix
   multiplication');
xlabel('matrix size n','fontsize',14);
ylabel('time [s]','fontsize',14);
legend('loop implementation','dot product implementation',...
       'matrix-vector implementation','BLAS gemm (MATLAB *)',...
       'location','northwest');

print -depsc2 '../PICTURES/mvtiming.eps';
```



Platform:

- Mac OS X 10.6
- Intel Core 7, 2.66 GHz
- L2 256 kB, L3 4 MB, Mem 4 GB
- MATLAB 7.10.0 (R 2010a)

In MATLAB we can achieve a tremendous gain in execution speed by relying on compact matrix/vector operations that invoke efficient BLAS routine.

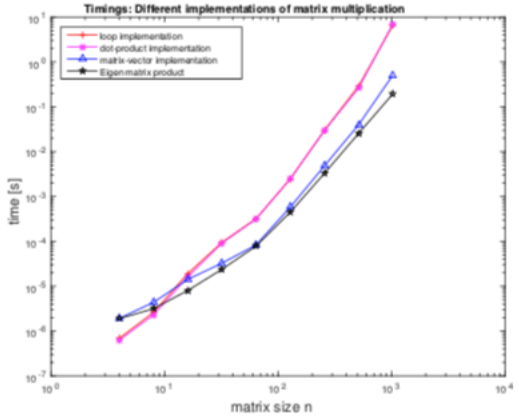
Advise: avoid loops in MATLAB and replace them with vectorised operations.

To some extent the same applies to EIGEN code, a corresponding timing script is given here:

C++11 code 1.3.19: Timing different implementations of matrix multiplication in EIGEN
➥ GITLAB

```
2  //!  script for timing different implementations of matrix
       multiplications
3  //!  no BLAS is used in Eigen!
4  void mmtiming(){
5    int nruns = 3, minExp = 2, maxExp = 10;
6    MatrixXd timings(maxExp-minExp+1,5);
7    for(int p = 0; p <= maxExp-minExp; ++p){
8      Timer t1, t2, t3, t4;     // timer class
9      int n = std::pow(2, minExp + p);
10     MatrixXd A = MatrixXd::Random(n,n);
11     MatrixXd B = MatrixXd::Random(n,n);
12     MatrixXd C = MatrixXd::Zero(n,n);
13     for(int q = 0; q < nruns; ++q){
14       // Loop based implementation no template magic
15       t1.start();
16       for(int i = 0; i < n; ++i)
17         for(int j = 0; j < n; ++j)
18           for(int k = 0; k < n; ++k)
19             C(i,j) += A(i,k)*B(k,j);
20       t1.stop();
21       // dot product based implementation little template magic
22       t2.start();
23       for(int i = 0; i < n; ++i)
24         for(int j = 0; j < n; ++j)
25           C(i,j) = A.row(i).dot( B.col(j) );
26       t2.stop();
27       // matrix-vector based implementation middle template magic
28       t3.start();
29       for(int j = 0; j < n; ++j)
30         C.col(j) = A * B.col(j);
31       t3.stop();
32       // Eigen matrix multiplication template magic optimized
```

```
33            t4.start();
34            C = A * B;
35            t4.stop();
36        }
37        timings(p,0)=n; timings(p,1)=t1.min(); timings(p,2)=t2.min();
38        timings(p,3)=t3.min(); timings(p,4)=t4.min();
39    }
40    std::cout << std::scientific << std::setprecision(3) << timings <<
          std::endl;
41    //Plotting
42    mgl::Figure fig;
43    fig.setFontSize(4);
44    fig.setlog(true, true);
45    fig.plot(timings.col(0),timings.col(1), " +r-").label("loop
          implementation");
46    fig.plot(timings.col(0),timings.col(2)," *m-").label("dot-product
          implementation");
47    fig.plot(timings.col(0),timings.col(3)," ^b-").label("matrix-vector
          implementation");
48    fig.plot(timings.col(0),timings.col(4)," ok-").label("Eigen matrix
          product");
49    fig.xlabel("matrix size n");  fig.ylabel("time [s]");
50    fig.legend(0.05,0.95);  fig.save("mmtiming");
51 }
```



Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

In EIGEN we can achieve some gain in execution speed by relying on compact matrix/vector operations that invoke efficient EIGEN routine. Notice that loops are not as punished as in MATLAB!

The same applies to PYTHON code, a corresponding timing script is given here:

PYTHON-code 1.3.20: Timing different implementations of matrix multiplication in PYTHON

```
# script for timing different implementations of matrix multiplications
import numpy as np
from matplotlib import pyplot as plt
import timeit

def mm_loop_based(A, B, C):
    m, n = A.shape
```

```
    _, p = B.shape
    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]
    return C

def mm_blas1(A, B, C):
    m, n = A.shape
    _, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = np.dot(A[i, :], B[:, j])
    return C

def mm_blas2(A, B, C):
    m, n = A.shape
    _, p = B.shape
    for i in range(m):
        C[i, :] = np.dot(A[i, :], B)
    return C

def mm_blas3(A, B, C):
    C = np.dot(A, B)
    return C

def main():
    nruns = 3
    res = []
    for n in 2**np.mgrid[2:11]:
        print('matrix size n = {}'.format(n))
        A = np.random.uniform(size=(n, n))
        B = np.random.uniform(size=(n, n))
        C = np.random.uniform(size=(n, n))

        tloop = min(timeit.repeat(lambda: mm_loop_based(A, B, C),
                                  repeat=nruns, number=1))
        tblas1 = min(timeit.repeat(lambda: mm_blas1(A, B, C),
                                   repeat=nruns, number=1))
        tblas2 = min(timeit.repeat(lambda: mm_blas2(A, B, C),
                                   repeat=nruns, number=1))
        tblas3 = min(timeit.repeat(lambda: mm_blas3(A, B, C),
                                   repeat=nruns, number=1))
        res.append((n, tloop, tblas1, tblas2, tblas3))

    ns, tloops, tblas1s, tblas2s, tblas3s = np.transpose(res)
    plt.figure()
    plt.loglog(ns, tloops, 'o', label='loop implementation')
    plt.loglog(ns, tblas1s, '+', label='dot product
       implementation')
```

```
57      plt.loglog(ns, tblas2s, '*', label='matrix-vector
            implementation')
58      plt.loglog(ns, tblas3s, '^', label='BLAS gemm (np.dot)')
59      plt.legend(loc='upper left')
60      plt.savefig('../PYTHON_PICTURES/mvtiming.eps')
61      plt.show()
62
63  if __name__ == '__main__':
64      main()
```

BLAS routines are grouped into "levels" according to the amount of data and computation involved (asymptotic complexity, see Section 1.4.1 and [5, Sect. 1.1.12]):

- **Level 1**: vector operations such as scalar products and vector norms.
  asymptotic complexity $O(n)$, (with $n \triangleq$ vector length),
  e.g.: dot product: $\rho = \mathbf{x}^\top \mathbf{y}$
- **Level 2**: vector-matrix operations such as matrix-vector multiplications.
  asymptotic complexity $O(mn)$,(with $(m,n) \triangleq$ matrix size),
  e.g.: matrix×vector multiplication: $\mathbf{y} = \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}$
- **Level 3**: matrix-matrix operations such as matrix additions or multiplications.
  asymptotic complexity often $O(nmk)$,(with $(n,m,k) \triangleq$ matrix sizes),
  e.g.: matrix product: $\mathbf{C} = \mathbf{A}\mathbf{B}$

Syntax of BLAS calls:
The functions have been implemented for different types, and are distinguished by the first letter of the function name. E.g. *sdot* is the dot product implementation for single precision and *ddot* for double precision.

✦ **BLAS LEVEL 1**: vector operations, asymptotic complexity $O(n)$, $n \triangleq$ vector length

- dot product $\rho = \mathbf{x}^\top \mathbf{y}$

  xDOT(N, X, INCX, Y, INCY)

  - x $\in \{$S, D$\}$, scalar type: S $\triangleq$ type `float`, D $\triangleq$ type `double`
  - $N \triangleq$ length of vector (modulo stride INCX)
  - $X \triangleq$ vector $\mathbf{x}$: array of type x
  - `INCX` $\triangleq$ stride for traversing vector X
  - $Y \triangleq$ vector $\mathbf{y}$: array of type x
  - `INCY` $\triangleq$ stride for traversing vector Y
- vector operations $\mathbf{y} = \alpha \mathbf{x} + \mathbf{y}$

  xAXPY(N, ALPHA, X, INCX, Y, INCY)

  - x $\in \{$S, D, C, Z$\}$, S $\triangleq$ type `float`, D $\triangleq$ type `double`, C $\triangleq$ type `complex`

- $N \triangleq$ length of vector (modulo stride INCX)
- ALPHA $\triangleq$ scalar α
- $X \triangleq$ vector $\mathbf{x}$: array of type x
- INCX $\triangleq$ stride for traversing vector X
- $Y \triangleq$ vector $\mathbf{y}$: array of type x
- INCY $\triangleq$ stride for traversing vector Y

✦ **BLAS LEVEL 2**: matrix-vector operations, asymptotic complexity $O(mn)$, $(m,n) \triangleq$ matrix size

- matrix×vector multiplication $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$

```
xGEMV(TRANS,M,N,ALPHA,A,LDA,X,
      INCX,BETA,Y,INCY)
```

  - x $\in \{S,D,C,Z\}$, scalar type: S $\triangleq$ type `float`, D $\triangleq$ type `double`, C $\triangleq$ type `complex`
  - $M, N \triangleq$ size of matrix $\mathbf{A}$
  - ALPHA $\triangleq$ scalar parameter α
  - A $\triangleq$ matrix $\mathbf{A}$ stored in *linear array* of length $M \cdot N$ (column major arrangement)

$$(\mathbf{A})_{i,j} = \mathrm{A}[N * (j-1) + i] \,.$$

  - LDA $\triangleq$ "leading dimension" of $\mathbf{A} \in \mathbb{K}^{n,m}$, that is, the number n of rows.
  - $X \triangleq$ vector $\mathbf{x}$: array of type x
  - INCX $\triangleq$ stride for traversing vector X
  - BETA $\triangleq$ scalar paramter β
  - $Y \triangleq$ vector $\mathbf{y}$: array of type x
  - INCY $\triangleq$ stride for traversing vector Y
- **BLAS LEVEL 3**: matrix-matrix operations, asymptotic complexity $O(mnk)$, $(m,n,k) \triangleq$ matrix sizes
  - matrix×matrix multiplication $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$

```
xGEMM(TRANSA,TRANSB,M,N,K,
      ALPHA,A,LDA,X,B,LDB,
      BETA,C,LDC)
```

    (☞ meaning of arguments as above)

Remark 1.3.21 (BLAS calling conventions)

The BLAS calling syntax seems queer in light of modern object oriented programming paradigms, but it is a legacy of FORTRAN77, which was (and partly still is) the programming language, in which the BLAS

routines were coded.

It is a very common situation in scientific computing that one has to rely on old codes and libraries implemented in an old-fashioned style.

Example 1.3.22 (Calling BLAS routines from C/C++)

When calling BLAS library functions from C, all arguments have to be passed by reference (as pointers), in order to comply with the argument passing mechanism of FORTRAN77, which is the model followed by BLAS.

C++-code 1.3.23: BLAS-based SAXPY operation in C++

```
#define daxpy_  daxpy
#include <iostream>

// Definition of the required BLAS function.  This is usually done
// in a header file like blas.h that is included in the EIGEN3
// distribution
extern "C" {
  int daxpy_(const int* n, const double* da, const double* dx,
             const int* incx, double* dy, const int* incy);
}

using namespace std;

int main(){
  const int    n     = 5; // length of vector
  const int    incx  = 1; // stride
  const int    incy  = 1; // stride
  double alpha = 2.5;     // scaling factor

  // Allocated raw arrays of doubles
  double* x  = new double [n];
  double* y  = new double [n];

  for (size_t i=0; i<n; i++){
    x[i] = 3.1415 * i;
    y[i] = 1.0 / (double)(i+1);
  }

  cout << "x=["; for (size_t i=0; i<n; i++) cout << x[i] << ' ';
  cout << "]" << endl;
  cout << "y=["; for (size_t i=0; i<n; i++) cout << y[i] << ' ';
  cout << "]" << endl;

  // Call the BLAS library function passing pointers to all arguments
  // (Necessary when calling FORTRAN routines from C
  daxpy_(&n, &alpha, x, &incx, y, &incy);

  cout << "y = " << alpha << " * x + y = ]";
  for (int i=0; i<n; i++) cout << y[i] << ' '; cout << "]" << endl;
  return(0);
}
```

When using EIGEN in a mode that includes an external BLAS library, all this calls are wrapped into EIGEN methods.

Example 1.3.24 (Using Intel Math Kernel Library (Intel MKL) from EIGEN)

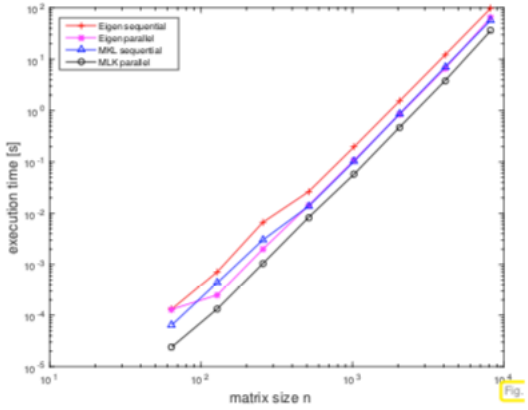
The Intel Math Kernel Library is a highly optimized math library for Intel processors and can be called directly from EIGEN (Using Intel® Math Kernel Library from Eigen) using the correct compiler flags.

C++-code 1.3.25: Timing of matrix multiplication in EIGEN for MKL comparison ↪ GITLAB

```
2  //!  script for timing different implementations of matrix
       multiplications
3  void mmeigenmkl(){
4    int nruns = 3, minExp = 6, maxExp = 13;
5    MatrixXd timings(maxExp-minExp+1,2);
6    for(int p = 0; p <= maxExp-minExp; ++p){
7      Timer t1; // timer class
8      int n = std::pow(2, minExp + p);
9      MatrixXd A = MatrixXd::Random(n,n);
10     MatrixXd B = MatrixXd::Random(n,n);
11     MatrixXd C = MatrixXd::Zero(n,n);
12     for(int q = 0; q < nruns; ++q){
13           t1.start();
14           C = A * B;
15           t1.stop();
16     }
17     timings(p,0)=n; timings(p,1)=t1.min();
18   }
19   std::cout << std::scientific << std::setprecision(3) << timings <<
       std::endl;
20 }
```

Timing results:

n	EIGEN sequential [s]	EIGEN parallel [s]	MKL sequential [s]	MKL parallel [s]
64	1.318e-04	1.304e-04	6.442e-05	2.401e-05
128	7.168e-04	2.490e-04	4.386e-04	1.336e-04
256	6.641e-03	1.987e-03	3.000e-03	1.041e-03
512	2.609e-02	1.410e-02	1.356e-02	8.243e-03
1024	1.952e-01	1.069e-01	1.020e-01	5.728e-02
2048	1.531e+00	8.477e-01	8.581e-01	4.729e-01
4096	1.212e+01	6.635e+00	7.075e+00	3.827e+00
8192	9.801e+01	6.426e+01	5.731e+01	3.598e+01



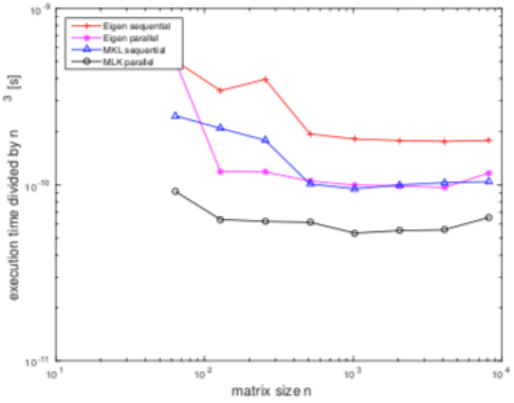


Fig.

Timing environment:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

1.4 Computational effort

Large scale numerical computations require immense resources and execution time of numerical codes often becomes a central concern. Therefore, much emphasis has to be put on

1. designing algorithms that produce a desired result with (nearly) minimal computational effort (defined precisely below),
2. exploit possibilities for parallel and vectorised execution,
3. organising algorithms in order to make them fit memory hierarchies,
4. implementing codes that make optimal use of hardware resources and capabilities,

While Item 2–Item 4 are out of the scope of this course and will be treated in more advanced lectures, Item 1 will be a recurring theme.

The following definition encapsulates what is regarded as a measure for the "cost" of an algorithm in computational mathematics.

Definition 1.4.1. Computational effort

The computational effort required by a numerical code amounts to the number of elementary operations (additions,subtractions,multiplications,divisions,square roots) executed in a run.

(1.4.2) What computational effort does not tell us

Fifty years ago counting elementary operations provided good predictions of runtimes, but nowadays this is no longer true.

(1.4.2) What computational effort does not tell us

Fifty years ago counting elementary operations provided good predictions of runtimes, but nowadays this is no longer true.

"Computational effort $\not\sim$ runtime"

The computational effort involved in a run of a numerical code is only loosely related to overall execution time on modern computers.

This is conspicuous in Exp. 1.2.29, where algorithms incurring exactly the same computational effort took different times to execute.

The reason is that on today's computers a key bottleneck for fast execution is latency and bandwidth of memory, *cf.* the discussion at the end of Exp. 1.2.29 and [9]. Thus, concepts like I/O-complexity [1, 6] might be more appropriate for gauging the efficiency of a code, because they take into account the pattern of memory access.

1.4.1 (Asymptotic) complexity

The concept of computational effort from Def. 1.4.1 is still useful in a particular context:

Definition 1.4.4. (Asymptotic) complexity

The asymptotic complexity of an algorithm characterises the worst-case dependence of its computational effort on one or more problem size parameter(s) when these tend to ∞.

- *Problem size parameters* in numerical linear algebra usually are the lengths and dimensions of the vectors and matrices that an algorithm takes as inputs.
- *Worst case* indicates that the maximum effort over a set of admissible data is taken into account.

When dealing with asymptotic complexities a mathematical formalism comes handy:

Definition 1.4.5. Landau symbol [2, p. 7]

We write $F(n) = O(G(n))$ for two functions $F, G : \mathbb{N} \to \mathbb{R}$, if there exists a constant $C > 0$ and $n_* \in \mathbb{N}$ such that

$$F(n) \leq CG(n) \quad \forall n \geq n_* .$$

More generally, $F(n_1, \ldots, n_k) = O(G(n_1, \ldots, n_k))$ for two functions $F, G : \mathbb{N}^k \to \mathbb{R}$ implies the existence of a constant $C > 0$ and a threshold value $n_* \in \mathbb{N}$ such that

$$F(n_1, \ldots, n_k) \leq CG(n_1, \ldots, n_k) \quad \forall n_1, \ldots, n_k \in \mathbb{N} , \quad n_\ell \geq n_*, \ \ell = 1, \ldots, k .$$

Remark 1.4.6 (Meaningful "O-bounds" for complexity)

Of course, the definition of the Landau symbol leaves ample freedom for stating meaningless bounds; an algorithm that runs with linear complexity $O(n)$ can be correctly labelled as possessing $O(\exp(n))$ complexity.

Yet, whenever the Landau notation is used to describe asymptotic complexities, the bounds have to be sharp in the sense that no function with slower asymptotic growth will be possible inside the O. To make this precise we stipulate the following.

Sharpness of a complexity bound

Whenever the asymptotic complexity of an algorithm is stated as $O(n^\alpha \log^\beta n \exp(\gamma n^\delta))$ with non-negative parameters $\alpha, \beta, \gamma, \delta \geq 0$ in terms of the problem size parameter n, we take for granted that choosing a smaller value for any of the parameters will no longer yield a valid (or provable) asymptotic bound.

In particular

- complexity $O(n)$ means that the complexity is not $O(n^\alpha)$ for any $\alpha < 1$,
- complexity $O(\exp(n))$ excludes asymptotic complexity $O(n^p)$ for any $p \in \mathbb{R}$.

Terminology: If the asymptotic complexity of an algorithm is $O(n^p)$ with $p = 1, 2, 3$ we say that it is of "linear", "quadratic", and "cubic" complexity, respectively.

Remark 1.4.8 (Relevance of asymptotic complexity)

§ 1.4.2 warned us that computational effort and, thus, asymptotic complexity, of an algorithm for a concrete problem on a particular platform may not have much to do with the actual runtime (the blame goes to memory hierarchies, internal pipelining, vectorisation, etc.).

Then, why do we pay so much attention to asymptotic complexity in this course?

To a certain extent, the asymptotic complexity allows to predict the *dependence of the runtime* of a particular implementation of an algorithm *on the problem size* (for large problems).

For instance, an algorithm with asymptotic complexity $O(n^2)$ is likely to take $4\times$ as much time when the problem size is doubled.

(1.4.9) Concluding polynomial complexity from runtime measurements

Available: "Measured runtimes" $t_i = t_i(n_i)$ for different values $n_1, n_2, \ldots, n_N$, $n_i \in \mathbb{N}$, of the problem size parameter

Conjectured: power law dependence $t_i \approx C n_i^\alpha$ (also "algebraic dependence"), $\alpha \in \mathbb{R}$

How can we glean evidence that supports or refutes our conjecture from the data? Look at the data in doubly logarithmic scale!

$$t_i = C n_i^\alpha \quad \Rightarrow \quad \log(t_i) \approx \log C + \alpha \log(n_i) \,, \quad i = 1, \ldots, N \,.$$

If the conjecture holds true, then the points (n_i, t_i) will approximately lie on a *straight line* with slope α in a doubly logarithmic plot (which can be created in MATLAB by the **loglog** plotting command and in EIGEN with the **Figure**-command `fig.setlog(true, true);`).

➢ quick "visual test" of conjectured asymptotic complexity

More rigorous: Perform linear regression on $(\log n_i, \log t_i)$, $i = 1, \ldots, N$ (→ Chapter 3)

1.4.2 Cost of basic operations

Performing elementary BLAS-type operations through simple (nested) loops, we arrive at the following obvious complexity bounds:

operation	description	#mul/div	#add/sub	asymp. complexity
dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H \mathbf{y}$	n	$n-1$	$O(n)$
tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}\mathbf{y}^H$	nm	0	$O(mn)$
matrix product$^{(*)}$	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{A}\mathbf{B}$	mnk	$mk(n-1)$	$O(mnk)$

Remark 1.4.10 ("Fast" matrix multiplication)

$(*)$: The $O(mnk)$ complexity bound applies to "straightforward" matrix multiplication according to (1.3.1).

For $m = n = k$ there are (sophisticated) variants with better asymptotic complexity, e.g., the divide-and-conquer Strassen algorithm [13] with asymptotic complexity $O(n^{\log_2 7})$:

Start from $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{n,n}$ with $n = 2\ell$, $\ell \in \mathbb{N}$. The idea relies on the block matrix product (1.3.16) with $\mathbf{A}_{ij}, \mathbf{B}_{ij} \in \mathbb{K}^{\ell,\ell}$, $i,j \in \{1,2\}$. Let $\mathbf{C} := \mathbf{AB}$ be partitioned accordingly: $\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{22} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$. Then tedious elementary computations reveal

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{Q}_0 + \mathbf{Q}_3 - \mathbf{Q}_4 + \mathbf{Q}_6 \,, \\ \mathbf{C}_{21} &= \mathbf{Q}_1 + \mathbf{Q}_3 \,, \\ \mathbf{C}_{12} &= \mathbf{Q}_2 + \mathbf{Q}_4 \,, \\ \mathbf{C}_{22} &= \mathbf{Q}_0 + \mathbf{Q}_2 - \mathbf{Q}_1 + \mathbf{Q}_5 \,, \end{aligned}$$

where the $\mathbf{Q}_k \in \mathbb{K}^{\ell,\ell}$, $k = 1, \ldots, 7$ are obtained from

$$\begin{aligned} \mathbf{Q}_0 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) * (\mathbf{B}_{11} + \mathbf{B}_{22}) \,, \\ \mathbf{Q}_1 &= (\mathbf{A}_{21} + \mathbf{A}_{22}) * \mathbf{B}_{11} \,, \\ \mathbf{Q}_2 &= \mathbf{A}_{11} * (\mathbf{B}_{12} - \mathbf{B}_{22}) \,, \\ \mathbf{Q}_3 &= \mathbf{A}_{22} * (-\mathbf{B}_{11} + \mathbf{B}_{21}) \,, \\ \mathbf{Q}_4 &= (\mathbf{A}_{11} + \mathbf{A}_{12}) * \mathbf{B}_{22} \,, \\ \mathbf{Q}_5 &= (-\mathbf{A}_{11} + \mathbf{A}_{21}) * (\mathbf{B}_{11} + \mathbf{B}_{12}) \,, \\ \mathbf{Q}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{22}) * (\mathbf{B}_{21} + \mathbf{B}_{22}) \,. \end{aligned}$$

Beside a considerable number of matrix additions (computational effort $O(n^2)$) it takes only **7** multiplications of matrices of size $n/2$ to compute $\mathbf{C}$! Strassen's algorithm boils down to the *recursive application* of these formulas for $n = 2^k$, $k \in \mathbb{N}$.

A refined algorithm of this type can achieve complexity $O(n^{2.36})$, see [3].

1.4.3 Reducing complexity in numerical linear algebra: Some tricks

In computations involving matrices and vectors complexity of algoritms can often be reduced by performing the operations in a particular order:

Example 1.4.11 (Efficient associative matrix multiplication)

We consider the multiplication with a rank-1-matrix. Matrices with rank 1 can always be obtained as the tensor product of two vectors, that is, the matrix product of a column vector and a row vector. Given $\mathbf{a} \in \mathbb{K}^m$, $\mathbf{b} \in \mathbb{K}^n$, $\mathbf{x} \in \mathbb{K}^n$ we may compute the vector $\mathbf{y} = \mathbf{a}\mathbf{b}^\top\mathbf{x}$ in two ways:

$$\mathbf{y} = \left(\mathbf{a}\mathbf{b}^\top\right)\mathbf{x}\,. \qquad (1.4.12) \qquad\qquad \mathbf{y} = \mathbf{a}\left(\mathbf{b}^\top\mathbf{x}\right)\,. \qquad (1.4.13)$$

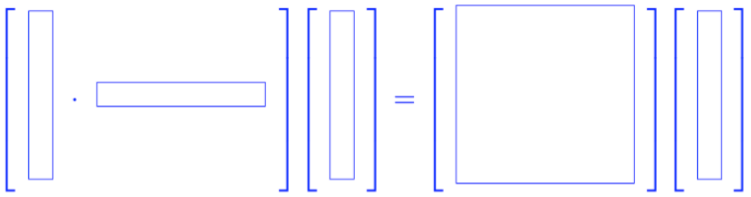
```
T = (a*b.transpose())*x;
```

➤ complexity $O(mn)$

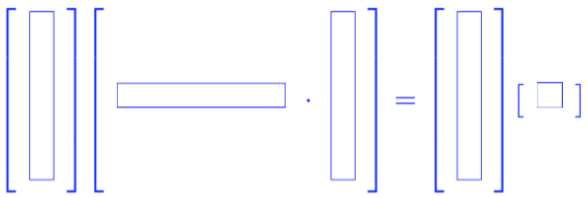
```
t = a*b.dot(x);
```

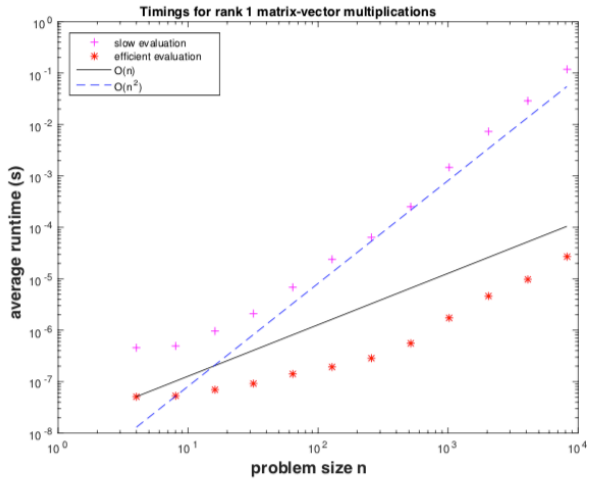
➤ complexity $O(n+m)$ ("linear complexity")

Visualization of evaluation according to (1.4.12):



Visualization of evaluation according to (1.4.13):





◁ average runtimes for efficient/inefficient matrix×vector multiplication with rank-1 matrices , see § 1.4.9 for the rationale behind choosing a *doubly logarithmic plot*.

Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB,
- 8 GB main memory
- gcc 4.8.4, -O3

C++11 code 1.4.14: EIGEN code for Ex. 1.4.11 ↪ GITLAB

```
2  //!  This function compares the runtimes for the multiplication
3  //!  of a vector with a rank-1 matrix ab^T, a,b ∈ R^n
4  //!  using different associative evaluations.
5  //!  Runtime measurements consider minimal time for
6  //!  several (nruns) runs
7  MatrixXd dottenstiming() {
8    const int nruns = 3, minExp = 2, maxExp = 13;
9    // Matrix for storing recorded runtimes
10   MatrixXd timings(maxExp-minExp+1,3);
11   for(int i = 0; i <= maxExp-minExp; ++i){
12     Timer tfool, tsmart; // Timer objects
13     const int n = std::pow(2, minExp + i);
14     VectorXd a = VectorXd::LinSpaced(n,1,n);
15     VectorXd b = VectorXd::LinSpaced(n,1,n).reverse();
16     VectorXd x = VectorXd::Random(n,1), y(n);
17     for(int j = 0; j < nruns; ++j){
18       // Grossly wasteful evaluation
19       tfool.start(); y = (a*b.transpose())*x;      tfool.stop();
20       // Efficient implementation
21       tsmart.start(); y = a * b.dot(x); tsmart.stop();
22     }
23     timings(i,0)=n;
24     timings(i,1)=tsmart.min(); timings(i,2)=tfool.min();
25   }
26   return timings;
27 }
```

Complexity can sometimes be reduced by reusing intermediate results.

Example 1.4.15 (Hidden summation)

The asymptotic complexity of the EIGEN code

```
Eigen::MatrixXd AB = A*B.transpose();
y = AB.triangularView<Eigen::Upper>()*x;
```

when supplied with two low-rank matrices $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{n,p}$, $p \ll n$, in terms of $n \to \infty$ obviously is $O(n^2)$, because an intermediate $n \times n$-matrix $\mathbf{A}\mathbf{B}^T$ is built.

First, consider the case of a tensor product (= rank-1) matrix, that is, $p = 1$, $\mathbf{A} \leftrightarrow \mathbf{a} = [a_1, \ldots, a_n]^\top \in \mathbb{K}^n$, $\mathbf{B} \leftrightarrow \mathbf{b} = [b_1, \ldots, b_n] \in \mathbb{K}^n$. Then

$$\mathbf{y} = \operatorname{triu}(\mathbf{a}\mathbf{b}^T)\mathbf{x} = \begin{bmatrix} a_1b_1 & a_1b_2 & \ldots & & \ldots & a_1b_n \\ 0 & a_2b_2 & a_2b_3 & \ldots & \ldots & a_2b_n \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \ldots & & \ldots & 0 & a_nb_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_n \end{bmatrix}$$

$$= \begin{bmatrix} a_1 & & \\ & \ddots & \\ & & a_n \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & \ldots & & \ldots & 1 \\ 0 & 1 & 1 & \ldots & \ldots & 1 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \ldots & & \ldots & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} b_1 & & \\ & \ddots & \\ & & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_n \end{bmatrix} .$$

Thus, the core problem is the fast multiplication of a vector with an upper triangular matrix $\mathbf{T}$ described in EIGEN syntax by `Eigen::MatrixXd::Ones(n,n).triangularView<Eigen::Upper>()`. Note that multiplication of a vector $\mathbf{x}$ with $\mathbf{T}$ yields a vector of partial sums of components of $\mathbf{x}$ starting from last component. This can be achieved by invoking the special C++ command `std::partial_sum` in the numeric header (documentation). We also observe that

$$\mathbf{A}\mathbf{B}^T = \sum_{\ell=1}^{p} (\mathbf{A})_{:,\ell} ((\mathbf{B})_{:,\ell})^\top ,$$

so that that the computations for the special case $p = 1$ discussed above can simply be reused p times!

C++11 code 1.4.16: Efficient multiplication with the upper diagonal part of a rank-p-matrix in EIGEN ➪ GITLAB

```
//!  Computation of y = triu(AB^T)x
//!  Efficient implementation with backward cumulative sum
//!  (partial_sum)
template<class Vec, class Mat>
void lrtrimulteff(const Mat& A, const Mat& B, const Vec& x, Vec& y){
  const int n = A.rows(), p = A.cols();
  assert( n == B.rows() && p == B.cols()); // size missmatch
  for(int l = 0; l < p; ++l){
    Vec tmp = (B.col(l).array() * x.array()).matrix().reverse();
    std::partial_sum(tmp.data(), tmp.data()+n, tmp.data());
    y += (A.col(l).array() * tmp.reverse().array()).matrix();
  }
}
```

This code enjoys the obvious complexity of $O(pn)$ for $p, n \to \infty$, $p < n$. The code offers an example of a function templated with its argument types, see § 0.2.5. The types **Vec** and **Mat** must fit the concept of EIGEN vectors/matrices.

The next concept from linear algebra is important in the context of computing with multi-dimensional arrays.

Definition 1.4.17. Kronecker product

The Kronecker product $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$, $m, n, l, k \in \mathbb{N}$, is the $(ml) \times (nk)$-matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{ml,nk} .$$

Example 1.4.18 (Multiplication of Kronecker product with vector)

The function $(\mathbf{A} \otimes \mathbf{B})\mathbf{x}$ when invoked with two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$ and a vector $\mathbf{x} \in \mathbb{K}^{nk}$, will suffer an asymptotic complexity of $O(m \cdot n \cdot l \cdot k)$, determined by the size of the intermediate dense matrix $\mathbf{A} \otimes \mathbf{B} \in \mathbb{K}^{ml,nk}$.

Using the partitioning of the vector $\mathbf{x}$ into n equally long sub-vectors

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^n \end{bmatrix} \quad , \quad \mathbf{x}^j \in \mathbb{K}^k \, ,$$

we find the representation

$$(\mathbf{A} \otimes \mathbf{B})\mathbf{x} = \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{1,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{1,n}\mathbf{B}\mathbf{x}^n \\ (\mathbf{A})_{2,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{2,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{2,n}\mathbf{B}\mathbf{x}^n \\ \vdots \\ \vdots \\ (\mathbf{A})_{m,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{m,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{m,n}\mathbf{B}\mathbf{x}^n \end{bmatrix} .$$

The idea is to form the products $\mathbf{B}\mathbf{x}^j$, $j = 1, \ldots, n$, once, and then combine them linearly with coefficients given by the entries in the rows of $\mathbf{A}$:

C++11 code 1.4.19: Efficient multiplication of Kronecker product with vector in EIGEN
➥ GITLAB

```
//!  @brief Multiplication of Kronecker product with vector y = (A⊗B)x.
   Elegant way using reshape
//!  WARNING: using Matrix::Map we assume the matrix is in ColMajor
   format, *beware* you may incur bugs if matrix is in RowMajor isntead
//!  @param[in] A Matrix m × n
//!  @param[in] B Matrix l × k
//!  @param[in] x Vector of dim nk
//!  @param[out] y Vector y = kron(A,B)*x of dim ml
template <class Matrix, class Vector>
```

```
9  void kronmultv(const Matrix &A, const Matrix &B, const Vector &x,
       Vector &y){
10   unsigned int m = A.rows(); unsigned int n = A.cols();
11   unsigned int l = B.rows(); unsigned int k = B.cols();
12   // 1st matrix mult.  computes the products Bx^j
13   // 2nd matrix mult.  combines them linearly with the coefficients
       of A
14   Matrix t = B * Matrix::Map(x.data(),k,n) * A.transpose(); //
15   y = Matrix::Map(t.data(), m*l, 1);
16 }
```

Recall the reshaping of a matrix in EIGEN in order to understand this code: **??**.

The asymptotic complexity of this code is determined by the two matrix multiplications in Line 14. This yields the asymptotic complexity $O(lkn + mnl)$ for $l, k, m, n \to \infty$.

PYTHON-code 1.4.20: Efficient multiplication of Kronecker product with vector in PYTHON

```
1 def kronmultv(A, B, x):
2     n, k = A.shape[1], B.shape[1]
3     assert x.size == n * k, 'size mismatch'
4     xx = np.reshape(x, (n, k))
5     Z = np.dot(xx, B.T)
6     yy = np.dot(A, Z)
7     return np.ravel(yy)
```

Note that different reshaping is used in the PYTHON code due to the default row major storage order.

1.5 Machine Arithmetic

1.5.1 Experiment: Loss of orthogonality

(1.5.1) Gram-Schmidt orthogonalisation

From linear algebra [10, Sect. 4.4] or Ex. 0.2.41 we recall the fundamental algorithm of Gram-Schmidt orthogonalisation of an ordered finite set $\{\mathbf{a}^1, \ldots, \mathbf{a}^k\}$, $k \in \mathbb{N}$, of vectors $\mathbf{a}^\ell \in \mathbb{K}^n$:

Input: $\{\mathbf{a}^1,\ldots,\mathbf{a}^k\} \subset \mathbb{K}^n$

1: $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|_2}$ *% 1st output vector*
2: **for** $j = 2,\ldots,k$ **do**
{ *% Orthogonal projection*
3: $\mathbf{q}^j := \mathbf{a}^j$
4: **for** $\ell = 1,2,\ldots,j-1$ **do** (GS)
5: { $\mathbf{q}^j \leftarrow \mathbf{q}^j - \mathbf{a}^j \cdot \mathbf{q}^\ell \mathbf{q}^\ell$ }
6: **if** ($\mathbf{q}^j = \mathbf{0}$) **then** STOP
7: **else** { $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|_2}$ }
8: }

Output: $\{\mathbf{q}^1,\ldots,\mathbf{q}^j\}$

In linear algebra we have learnt that, if it does not STOP prematurely, this algorithm will compute orthonormal vectors $\mathbf{q}^1,\ldots,\mathbf{q}^k$ satisfying

$$\mathrm{Span}\{\mathbf{q}^1,\ldots,\mathbf{q}^\ell\} = \mathrm{Span}\{\mathbf{a}^1,\ldots,\mathbf{a}^\ell\}\,, \tag{1.5.2}$$

for all $\ell \in \{1,\ldots,k\}$.

More precisely, if $\mathbf{a}^1,\ldots,\mathbf{a}^\ell$, $\ell \le k$, are linearly independent, then the Gram-Schmidt algorithm will not terminate before the $\ell+1$-th step.

✎ Notation: $\|\cdot\|_2 \triangleq$ Euclidean norm of a vector $\in \mathbb{K}^n$

The following code implements the Gram-Schmidt orthonormalization of a set of vectors passed as the columns of a matrix $\mathbf{A} \in \mathbb{R}^{n,k}$.

C++11 code 1.5.3: Gram-Schmidt orthogonalisation in EIGEN ➙ GITLAB

```
2  template <class Matrix>
3  Matrix gramschmidt( const Matrix & A ) {
4      Matrix Q = A;
5      // First vector just gets normalized, Line 1 of (GS)
6      Q.col(0).normalize();
7      for(unsigned int j = 1; j < A.cols(); ++j) {
8          // Replace inner loop over each previous vector in Q with fast
9          // matrix-vector multiplication (Lines 4, 5 of (GS))
10         Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() *
               A.col(j));//
11         // Normalize vector, if possible.
12         // Otherwise colums of A must have been linearly dependent
13         if( Q.col(j).norm() <= 10e-9 * A.col(j).norm() ) { //
14             std::cerr << "Gram-Schmidt failed: A has lin. dep
                   columns." << std::endl;
15             break;
16         } else { Q.col(j).normalize(); } // Line 7 of (GS)
17     }
18     return Q;
19 }
```

PYTHON-code 1.5.4: Gram-Schmidt orthogonalisation in PYTHON

```
def gramschmidt(A):
    _, k = A.shape
    Q = A[:, [0]] / np.linalg.norm(A[:, 0])
    for j in range(1, k):
        q = A[:, j] - np.dot(Q, np.dot(Q.T, A[:, j]))
        nq = np.linalg.norm(q)
        if nq < 1e-9 * np.linalg.norm(A[:, j]):
            break
        Q = np.column_stack([Q, q / nq])
    return Q
```

Note the different loop range due to the zero-based indexing in PYTHON.

Experiment 1.5.5 (Unstable Gram-Schmidt orthonormalization)

If $\{\mathbf{a}^1,\ldots,\mathbf{a}^k\}$ are linearly independent we expect the output vectors $\mathbf{q}^1,\ldots,\mathbf{q}^k$ to be orthonormal:

$$(\mathbf{q}^\ell)^\top \mathbf{q}^m = \delta_{\ell,m}\,, \quad \ell,m \in \{1,\ldots,k\}\,. \tag{1.5.6}$$

This property can be easily tested numerically, for instance by computing $\mathbf{Q}^\top\mathbf{Q}$ for a matrix $\mathbf{Q} = \left[\mathbf{a}^1,\ldots,\mathbf{q}^k\right] \in \mathbb{R}^{n,k}$.

C++11 code 1.5.7: Wrong result from Gram-Schmidt orthogonalisation EIGEN ↪ GITLAB

```
void gsroundoff(MatrixXd& A){
  // Gram-Schmidt orthogonalization of columns of A, see Code 1.5.3
  MatrixXd Q = gramschmidt(A);
  // Test orthonormality of columns of Q, which should be an
  // orthogonal matrix according to theory
  cout << setprecision(4) << fixed << "I = "
       << endl << Q.transpose()*Q << endl;
  // EIGEN's stable internal Gram-Schmidt orthogonalization by
  // QR-decomposition, see Rem. 1.5.9 below
  HouseholderQR<MatrixXd> qr(A.rows(),A.cols()); //
  qr.compute(A); MatrixXd Q1 = qr.householderQ(); //
  // Test orthonormality
  cout << "I1 = " << endl << Q1.transpose()*Q1 << endl;
  // Check orthonormality and span property (1.5.2)
  MatrixXd R1 = qr.matrixQR().triangularView<Upper>();
  cout << scientific << "A-Q1*R1 = " << endl << A-Q1*R1 << endl;
}
```

We test the orthonormality of the output vectors of Gram-Schmidt orthogonalization for a special matrix $\mathbf{A} \in \mathbb{R}^{10,10}$, a so-called Hilbert matrix, defined by $(\mathbf{A})_{i,j} = (i+j-1)^{-1}$. Then Code 1.5.7 produces the follwing output:

```
I =
   1.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
   0.0000    1.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
  -0.0000   -0.0000    1.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
   0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
  -0.0000   -0.0000   -0.0000   -0.0000    1.0000    0.0000   -0.0008   -0.0007   -0.0007   -0.0006
   0.0000    0.0000    0.0000    0.0000    0.0000    1.0000   -0.0540   -0.0430   -0.0360   -0.0289
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0008   -0.0540    1.0000    0.9999    0.9998    0.9996
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0007   -0.0430    0.9999    1.0000    1.0000    0.9999
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0007   -0.0360    0.9998    1.0000    1.0000    1.0000
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0006   -0.0289    0.9996    0.9999    1.0000    1.0000
```

Obviously, the vectors produced by the function `gramschmidt` fail to be orthonormal, contrary to the predictions of rigorous results from linear algebra!

However, Line 11, Line 12 of Code 1.5.7 demonstrate another way to orthonormalize the columns of a matrix using EIGEN's built-in class template **HouseholderQR**:

```
I1 =
 1.0000 -0.0000  0.0000 -0.0000 -0.0000 -0.0000 -0.0000 -0.0000 -0.0000  0.0000
-0.0000  1.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000 -0.0000  0.0000 -0.0000
 0.0000 -0.0000  1.0000 -0.0000 -0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
-0.0000  0.0000 -0.0000  1.0000  0.0000 -0.0000 -0.0000  0.0000  0.0000  0.0000
-0.0000  0.0000 -0.0000  0.0000  1.0000 -0.0000  0.0000 -0.0000 -0.0000  0.0000
-0.0000 -0.0000  0.0000 -0.0000 -0.0000  1.0000 -0.0000 -0.0000  0.0000 -0.0000
-0.0000 -0.0000  0.0000 -0.0000  0.0000 -0.0000  1.0000  0.0000  0.0000 -0.0000
-0.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000  1.0000 -0.0000  0.0000
-0.0000  0.0000  0.0000  0.0000 -0.0000  0.0000  0.0000 -0.0000  1.0000 -0.0000
 0.0000 -0.0000  0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000 -0.0000  1.0000
```

Now we observe apparently perfect orthogonality (1.5.6) of the columns of the matrix `Q1` in Code 1.5.7. There is another algorithm that reliably yields the theoretical output of Gram-Schmidt orthogonalization.

"Computers cannot compute"

Computers cannot compute "properly" in $\mathbb{R}$: numerical computations may not respect the laws of analysis and linear algebra!

This introduces an important new aspect in the study of numerical algorithms.

Remark 1.5.9 (Stable orthonormalization by QR-decomposition)

In Code 1.5.7 we saw the use of the EIGEN class `HousholderQR<MatrixType>` for the purpose of Gram-Schmidt orthogonalisation. The underlying theory and algorithms will be explained later in Section 3.3.3. There we will have the following insight:

➢ Up to signs the columns of the matrix $\mathbf{Q}$ available from the QR-decomposition of $\mathbf{A}$ are the same vectors as produced by the Gram-Schmidt orthogonalisation of the columns of $\mathbf{A}$.

! Code 1.5.7 demonstrates a case where a desired result can be obtained by two *algebraically equivalent* computations, that is, they yield the same result in a mathematical sense. Yet, when implemented on a computer, the results can be *vastly different*. One algorithm may produce junk ("unstable algorithm"), whereas the other lives up to the expectations ("stable algorithm")

Supplement to Exp. 1.5.5: despite its ability to produce orthonormal vectors, we get as output for `D=A-Q1*R1` in Code 1.5.7:

```
D =
 2.2204e-16  3.3307e-16  3.3307e-16  1.9429e-16  1.9429e-16  5.5511e-17  1.3878e-16  6.9389e-17  8.3267e-17  9.7145e-17
 0.0000e+00  1.1102e-16  8.3267e-17  5.5511e-17  0.0000e+00  5.5511e-17 -2.7756e-17  0.0000e+00  0.0000e+00  4.1633e-17
-5.5511e-17  5.5511e-17  2.7756e-17  5.5511e-17  0.0000e+00  0.0000e+00  0.0000e+00 -1.3878e-17  1.3878e-17  1.3878e-17
 0.0000e+00  5.5511e-17  2.7756e-17  2.7756e-17  0.0000e+00  1.3878e-17 -1.3878e-17  0.0000e+00  1.3878e-17  2.7756e-17
 0.0000e+00  2.7756e-17  0.0000e+00  1.3878e-17  1.3878e-17  1.3878e-17  0.0000e+00  1.3878e-17  1.3878e-17  4.1633e-17
-2.7756e-17  2.7756e-17  1.3878e-17  4.1633e-17  2.7756e-17  1.3878e-17  0.0000e+00 -1.3878e-17  2.7756e-17  2.7756e-17
 0.0000e+00  2.7756e-17  0.0000e+00  2.7756e-17  2.7756e-17  1.3878e-17  0.0000e+00  1.3878e-17  2.7756e-17  2.0817e-17
 0.0000e+00  2.7756e-17  2.7756e-17  1.3878e-17  1.3878e-17  1.3878e-17  0.0000e+00  1.3878e-17  2.0817e-17  2.7756e-17
 1.3878e-17  1.3878e-17  1.3878e-17  2.7756e-17  1.3878e-17  0.0000e+00 -1.3878e-17  6.9389e-18 -6.9389e-18  1.3878e-17
 0.0000e+00  2.7756e-17  1.3878e-17  1.3878e-17  1.3878e-17  0.0000e+00  0.0000e+00  0.0000e+00  1.3878e-17  1.3878e-17
```

➥ The computed QR-decomposition apparently fails to meet the exact algebraic requirements stipulated by Thm. 3.3.9. However, note the tiny size of the "defect".

Remark 1.5.10 (QR-decomposition in EIGEN)

The two different QR-decompositions (3.3.3.1) and (3.3.3.1) of a matrix $\mathbf{A} \in \mathbb{K}^{n,k}$, $n,k \in \mathbb{N}$, can be computed in EIGEN as follows:

```
HouseholderQR<MatrixXd> qr(A);
// Full QR-decomposition (3.3.3.1)
Q = qr.householderQ();
// Economical QR-decomposition (3.3.3.1)
thinQ = qr.householderQ() *
   MatrixXd::Identity(A.rows(),std::min(A.rows(), A.cols()));
```

The returned matrices `Q` and `R` correspond to the QR-factors $\mathbf{Q}$ and $\mathbf{R}$ as defined above. See the discusssion, whether the "economical" decomposition is truly economical: some expression template magic.

Remark 1.5.11 (QR-decomposition in PYTHON)

The two different QR-decomposition (3.3.3.1) and (3.3.3.1) of a matrix $\mathbf{A} \in \mathbb{K}^{n,k}$, $n, k \in \mathbb{N}$, can be computed in PYTHON as follows:

```
Q, R = np.linalg.qr(A, mode='reduced')   # Economical (3.3.3.1)
Q, R = np.linalg.qr(A, mode='complete')  # Full (3.3.3.1)
```

The returned matrices `Q` and `R` correspond to the QR-factors $\mathbf{Q}$ and $\mathbf{R}$ as defined above.

1.5.2 Machine Numbers

(1.5.12) The finite and discrete set of machine numbers

The reason, why computers must fail to execute exact computations with real numbers is clear:

Computer = finite automaton ➢ can handle only *finitely many* numbers, not $\mathbb{R}$

machine numbers, set $\mathbb{M}$

Essential property: $\mathbb{M}$ is a finite, discrete subset of $\mathbb{R}$ (its numbers separated by gaps)

The set of machine numbers $\mathbb{M}$ cannot be closed under elementary arithmetic operations $+, -, \cdot, /$, that is, when adding, multiplying, etc., two machine numbers the result may not belong to $\mathbb{M}$.

The results of elementary operations with operands in $\mathbb{M}$ have to be mapped back to $\mathbb{M}$, an operation called rounding.

◤ roundoff errors (*ger.:* Rundungsfehler) are inevitable

The impact of roundoff means that mathematical identities may not carry over to the computational realm. As we have seen above in Exp. 1.5.5

Computers cannot compute "properly" !

numerical computations $\neq$ **analysis linear algebra**

This introduces a *new* and *important* aspect in the study of numerical algorithms!

(1.5.13) Internal representation of machine numbers

Now we give a brief sketch of the internal structure of machine numbers $\in \mathbb{M}$. The main insight will be that

"Computers use floating point numbers (scientific notation)"

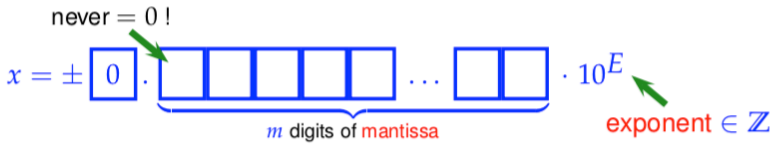
Example 1.5.14 (Decimal floating point numbers)

Some 3-digit normalized decimal floating point numbers:

valid: $0.723 \cdot 10^2$, $0.100 \cdot 10^{-20}$, $-0.801 \cdot 10^5$

invalid: $0.033 \cdot 10^2$, $1.333 \cdot 10^{-4}$, $-0.002 \cdot 10^3$

General form of an m-digit normalized decimal floating point number:



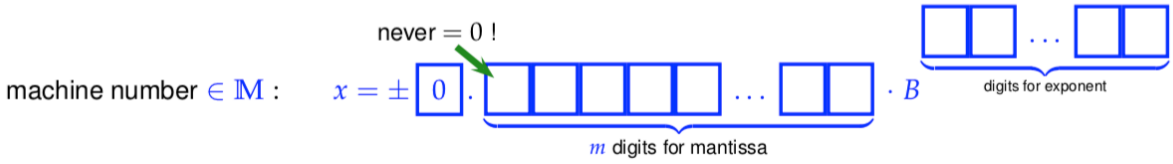
Of course, computers are restricted to a *finite range* of exponents:

Definition 1.5.15. Machine numbers/floating point numbers → [2, Sect. 2.1]

Given
- basis $B \in \mathbb{N} \setminus \{1\}$,
- exponent range $\{e_{\min}, \ldots, e_{\max}\}$, $e_{\min}, e_{\max} \in \mathbb{Z}$, $e_{\min} < e_{\max}$,
- number $m \in \mathbb{N}$ of digits (for mantissa),

the corresponding set of machine numbers is

$$\mathbb{M} := \{d \cdot B^E : d = i \cdot B^{-m}, i = B^{m-1}, \ldots, B^m - 1, E \in \{e_{\min}, \ldots, e_{\max}\}\}$$



Remark 1.5.16 (Extremal numbers in $\mathbb{M}$)

Clearly, there is a largest element of $\mathbb{M}$ and two that are closest to zero. These are mainly determined by the range for the exponent E, *cf.* Def. 1.5.15.

Largest machine number (in modulus) : $x_{\max} = \max|\mathbb{M}| = (1 - B^{-m}) \cdot B^{e_{\max}}$
Smallest machine number (in modulus) : $x_{\min} = \min|\mathbb{M}| = B^{-1} \cdot B^{e_{\min}}$

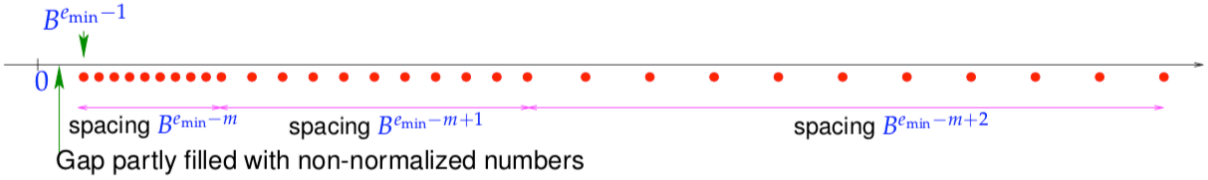
In C++ these extremal machine numbers are accessible through the

```
std::numeric_limits<double>::max()
```
and `std::numeric_limits<double>::min()`

functions. Other properties of arithmetic types can be queried accordingly from the numeric_limits header.

Remark 1.5.17 (Distribution of machine numbers)

From Def. 1.5.15 it is clear that there are equi-spaced sections of $\mathbb{M}$ and that the gaps between machine numbers are bigger for larger numbers:



Non-normalized numbers violate the lower bound for the mantissa i in Def. 1.5.15.

No surprise: for modern computers $B = 2$ (binary system), the other parameters of the universally implemented machine number system are

single precision : $m = 24^*, E \in \{-125, \ldots, 128\}$ ➢ 4 bytes

double precision : $m = 53^*, E \in \{-1021, \ldots, 1024\}$ ➢ 8 bytes

$*$: including bit indicating sign

The standardisation of machine numbers is important, because it ensures that the same numerical algorithm, executed on different computers will nevertheless produce the same result.

Remark 1.5.19 (Special cases in IEEE standard)

```
double x = exp(1000), y = 3/x, z = x*sin(M_PI), w = x*log(1);
cout << x << endl << y << endl << z << endl << w << endl;
```

Output:

```
1 inf
2 0
3 inf
4 -nan
```

$E = e_{\max}$, $M \neq 0$ $\hat{=}$ NaN = *Not a number* $\rightarrow$ exception
$E = e_{\max}$, $M = 0$ $\hat{=}$ Inf = *Infinity* $\rightarrow$ overflow
$E = 0$ $\hat{=}$ Non-normalized numbers $\rightarrow$ underflow
$E = 0$, $M = 0$ $\hat{=}$ number 0

(1.5.20) Characteristic parameters of IEEE floating point numbers (double precision)

☞ C++ does not always fulfill the requirements of the IEEE 754 standard and it needs to be checked with `std::numeric_limits<T>::is_iec559`.

C++11-code 1.5.21: Querying characteristics of `double` numbers ⇝ GITLAB

```
2  #include <limits>
3  #include <iostream>
4  #include <iomanip>
5
6  using namespace std;
7
8  int main(){
9       cout << std::numeric_limits<double>::is_iec559 << endl
10      << std::defaultfloat << numeric_limits<double>::min() << endl
11      << std::hexfloat << numeric_limits<double>::min() << endl
12      << std::defaultfloat << numeric_limits<double>::max() << endl
13      << std::hexfloat << numeric_limits<double>::max() << endl;
14 }
```

Output:

```
1  true
2  2.22507e-308
3  0010000000000000
4  1.79769e+308
5  7fefffffffffffff
```

1.5.3 Roundoff errors

Experiment 1.5.22 (Input errors and roundoff errors)

The following computations would always result in 0, if done in exact arithmetic.

C++11-code 1.5.23: Demonstration of roundoff errors ↪ GITLAB

```
#include <iostream>
int main(){
      std::cout.precision(15);
      double a = 4.0/3.0, b = a-1, c = 3*b, e = 1-c;
      std::cout << e << std::endl;
      a = 1012.0/113.0; b = a-9; c = 113*b; e = 5+c;
      std::cout << e << std::endl;
      a = 83810206.0/6789.0; b = a-12345; c = 6789*b; e = c-1;
      std::cout << e << std::endl;
}
```

Output:

```
2.22044604925031e-16
6.75015598972095e-14
-1.60798663273454e-09
```

Can you devise a similar calculation, whose result is even farther off zero? Apparently the rounding that inevitably accompanies arithmetic operations in $\mathbb{M}$ can lead to results that are far away from the true result.

For the discussion of errors introduced by rounding we need important notions.

Definition 1.5.24. Absolute and relative error → [2, Sect. 1.2]

Let $\widetilde{x} \in \mathbb{K}$ be an approximation of $x \in \mathbb{K}$. Then its absolute error is given by

$$\epsilon_{\text{abs}} := |x - \widetilde{x}| ,$$

and its relative error is defined as

$$\epsilon_{\text{rel}} := \frac{|x - \widetilde{x}|}{|x|} .$$

Remark 1.5.25 (Relative error and number of correct digits)

The number of correct (significant, valid) digits of an approximation $\widetilde{x}$ of $x \in \mathbb{K}$ is defined through the relative error:

$$\text{If } \epsilon_{\text{rel}} := \frac{|x-\widetilde{x}|}{|x|} \leq 10^{-\ell}, \text{ then } \widetilde{x} \text{ has } \ell \text{ correct digits, } \ell \in \mathbb{N}_0$$

(1.5.26) Floating point operations

We may think of the elementary binary operations $+, -, *, /$ in $\mathbb{M}$ comprising two steps:

❶ Compute the exact result of the operation.

❷ Perform rounding of the result of ❶ to map it back to $\mathbb{M}$.

Definition 1.5.27. Correct rounding

Correct rounding ("rounding up") is given by the function

$$\mathrm{rd} : \begin{cases} \mathbb{R} & \to & \mathbb{M} \\ x & \mapsto & \max \operatorname{argmin}_{\widetilde{x}\in\mathbb{M}} |x - \widetilde{x}| \, . \end{cases}$$

(Recall that $\operatorname{argmin}_x F(x)$ is the set of arguments of a real valued function F that makes it attain its (global) minimum.)

Of course, ❶ above is not possible in a strict sense, but the effect of both steps can be realised and yields a floating point realization of $\star \in \{+, -, \cdot, /\}$.

✎ Notation: write $\widetilde{\star}$ for the floating point realization of $\star \in \{+, -, \cdot, /\}$:

Then ❶ and ❷ may be summed up into

$$\text{For } \star \in \{+, -, \cdot, /\}: \quad x \,\widetilde{\star}\, y := \mathrm{rd}(x \star y) \, .$$

Remark 1.5.28 (Breakdown of associativity)

As a consequence of rounding addition $\widetilde{+}$ and multiplication $\widetilde{*}$ as implemented on computers fail to be associative. They will usually be commutative, though this is not guaranteed.

(1.5.29) Estimating roundoff errors → [2, p. 23]

Let us denote by EPS the largest relative error (→ Def. 1.5.24) incurred through rounding:

$$\mathrm{EPS} := \max_{x \in I \setminus \{0\}} \frac{|\operatorname{rd}(x) - x|}{|x|} , \tag{1.5.30}$$

where $I = [\min|\mathbb{M}|, \max|\mathbb{M}|$ is the range of positive machine numbers.

For machine numbers according to Def. 1.5.15 EPS can be computed from the defining parameters B (base) and m (length of mantissa) [2, p. 24]:

$$\mathrm{EPS} = \tfrac{1}{2} B^{1-m} . \tag{1.5.31}$$

However, when studying roundoff errors, we do not want to delve into the intricacies of the internal representation of machine numbers. This can be avoided by just using a single bound for the relative error due to rounding, and, thus, also for the relative error potentially suffered in each elementary operation.

Assumption 1.5.32. "Axiom" of roundoff analysis

There is a small positive number EPS, the machine precision, such that for the elementary arithmetic operations $\star \in \{+, -, \cdot, /\}$ and "hard-wired" functions* $f \in \{\exp, \sin, \cos, \log, \ldots\}$ holds

$$x \mathbin{\widetilde{\star}} y = (x \star y)(1+\delta) \quad , \quad \widetilde{f}(x) = f(x)(1+\delta) \quad \forall x, y \in \mathbb{M} ,$$

with $|\delta| < \mathrm{EPS}$.

*: this is an ideal, which may not be accomplished even by modern CPUs.

relative roundoff errors of elementary steps in a program bounded by machine precision !

Example 1.5.33 (Machine precision for IEEE standard)

C++ tells the machine precision as following:

C++11-code 1.5.34: Finding out EPS in C++ ↪ GITLAB

```
#include <iostream>
#include <limits> // get various properties of arithmetic types
int main() {
  std::cout.precision(15);
  std::cout << std::numeric_limits<double>::epsilon() << std::endl;
}
```

Output:

```
2.22044604925031e-16
```

Knowing the machine precision can be important for checking the validity of computations or coding termination conditions for iterative approximations.

```
cout.precision(25);
double eps =
   numeric_limits<double>::epsilon();
cout << fixed << 1.0 + 0.5*eps << endl
     << 1.0 - 0.5*eps << endl
     << (1.0 + 2/eps) - 2/eps << endl;
```

Output:

```
1.0000000000000000000000000
0.9999999999999998889776975
0.0000000000000000000000000
```

In fact, the following "definition" of EPS is sometimes used:

EPS is the smallest positive number $\in \mathbb{M}$ for which $1 \tilde{+} \mathrm{EPS} \neq 1$ (in $\mathbb{M}$):

We find that $1 \tilde{+} \mathrm{EPS} = 1$ actually complies with the "axiom" of roundoff error analysis, Ass. 1.5.32:

$$1 = (1+\mathrm{EPS})(1+\delta) \quad \Rightarrow \quad |\delta| = \left|\frac{\mathrm{EPS}}{1+\mathrm{EPS}}\right| < \mathrm{EPS}\,,$$

$$\frac{2}{\mathrm{EPS}} = (1+\frac{2}{\mathrm{EPS}})(1+\delta) \quad \Rightarrow \quad |\delta| = \left|\frac{\mathrm{EPS}}{2+\mathrm{EPS}}\right| < \mathrm{EPS}\,.$$

Do we have to worry about these tiny roundoff errors **?**

YES (→ Exp. 1.5.5):

- accumulation of roundoff errors
- amplification of roundoff errors

Remark 1.5.36 (Testing equality with zero)

Since results of numerical computations are almost always polluted by roundoff errors: Tests like `if (x == 0)` are pointless and even dangerous, if x contains the result of a numerical computation.

Remedy: Test `if (abs(x) < eps*s) ...`,
$s \triangleq$ positive number, compared to which $|x|$ should be small.

We saw a first example of this practise in Code 1.5.3, Line 13.

Remark 1.5.37 (Overflow and underflow)

overflow) $\triangleq$ |result of an elementary operation| $> \max\{\mathbb{M}\}$
$\triangleq$ IEEE standard $\Rightarrow$ `Inf`
underflow $\triangleq$ $0 <$ |result of an elementary operation| $< \min\{|\mathbb{M} \setminus \{0\}|\}$
$\triangleq$ IEEE standard $\Rightarrow$ use non-normalized numbers (!)

The Axiom of roundoff analysis Ass. 1.5.32 does *not hold* once non-normalized numbers are encountered:

C++11-code 1.5.38: Demonstration of over-/underflow ↪ GITLAB

```
2 #include <iostream>
3 #define _USE_MATH_DEFINES
```

```cpp
#include <cmath>
#include <limits>
using namespace std;
int main(){
    cout.precision(15);
    double min = numeric_limits<double>::min();
    double res1 = M_PI*min/123456789101112;
    double res2 = res1*123456789101112/min;
    cout << res1 << endl << res2 << endl;
}
```

Output:

```
5.68175492717434e-322
3.15248510554597
```

Try to avoid underflow and overflow

A simple example teaching how to avoid overflow during the computation of the norm of a 2D vector [2, Ex. 2.9]:

$$r = \sqrt{x^2 + y^2}$$

straightforward evaluation: overflow, when $|x| > \sqrt{\max|\mathbb{M}|}$ or $|y| > \sqrt{\max|\mathbb{M}|}$.

$$r = \begin{cases} |x|\sqrt{1+(y/x)^2} & \text{, if } |x| \geq |y| \text{ ,} \\ |y|\sqrt{1+(x/y)^2} & \text{, if } |y| > |x| \text{ .} \end{cases}$$

➢ no overflow!

1.5.4 Cancellation

In general, predicting the impact of roundoff errors on the result of a multi-stage computation is very difficult, if possible at all. However, there is a constellation that is particularly prone to dangerous amplification of roundoff and still can be detected easily.

Example 1.5.39 (Computing the zeros of a quadratic polynomial)

The following simple EIGEN code computes the real roots of a quadratic polynomial $p(\xi) = \xi^2 + \alpha\xi + \beta$ by the discriminant formula

$$p(\xi_1) = p(\xi_2) = 0\,, \quad \xi_{1/2} = \frac{1}{2}\left(\alpha \pm \sqrt{D}\right) \text{, if } \quad D := \alpha^2 - 4\beta \geq 0\,. \tag{1.5.40}$$

C++11-code 1.5.41: Discriminant formula for the real roots of $p(\xi) = \xi^2 + \alpha\xi + \beta$ ↪ GITLAB

```
//!  C++ function computing the zeros of a quadratic polynomial
//!  ξ → ξ^2 + αξ + β by means of the familiar discriminant
//!  formula ξ_{1,2} = ½(−α ± √(α^2 − 4β)).  However
//!  this implementation is vulnerable to round-off!  The zeros are
//!  returned in a column vector
```

```
Vector2d zerosquadpol(double alpha, double beta){
  Vector2d z;
  double D = std::pow(alpha,2) −4*beta; // discriminant
  if(D < 0) throw "no real zeros";
  else{
    // The famous discriminant formula
    double wD = std::sqrt(D);
    z << (−alpha−wD)/2, (−alpha+wD)/2; //
  }
  return z;
}
```

This formula is applied to the quadratic polynomial $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$ after its coefficients α, β have been computed from γ, which will have introduced *small* relative roundoff errors (of size EPS).

C++11-code 1.5.42: Testing the accuracy of computed roots of a quadratic polynomial
→ GITLAB

```
//!  Eigen Function for testing the computation of the zeros of a parabola
void compzeros(){
  int n = 100;
  MatrixXd res(n,4);
  VectorXd gamma = VectorXd::LinSpaced(n, 2,992);
  for(int i = 0; i < n; ++i){
    double alpha = -(gamma(i) + 1./gamma(i));
    double beta = 1.;
    Vector2d z1 =  zerosquadpol(alpha, beta);
    Vector2d z2 = zerosquadpolstab(alpha, beta);
    double ztrue = 1./gamma(i), z2true = gamma(i);
    res(i,0) = gamma(i);
    res(i,1) = std::abs((z1(0)-ztrue)/ztrue);
    res(i,2) = std::abs((z2(0)-ztrue)/ztrue);
    res(i,3) = std::abs((z1(1)-z2true)/z2true);
  }
  // Graphical output of relative error of roots computed by unstable implementation
  mgl::Figure fig1;
  fig1.setFontSize(3);
  fig1.title("Roots of a parabola computed in an unstable manner");
  fig1.plot(res.col(0),res.col(1), " +r").label("small root");
  fig1.plot(res.col(0),res.col(3), " *b").label("large root");
  fig1.xlabel("\\gamma");
  fig1.ylabel("relative errors in \\xi_1, \\xi_2");
  fig1.legend(0.05,0.95);
  fig1.save("zqperrinstab");
  // Graphical output of relative errors (comparison), small roots
  mgl::Figure fig2;
  fig2.title("Roundoff in the computation of zeros of a parabola");
  fig2.plot(res.col(0), res.col(1), " +r").label("unstable");
  fig2.plot(res.col(0), res.col(3), " *m").label("stable");
  fig2.xlabel("\\gamma");
  fig2.ylabel("relative errors in \\xi_1");
  fig2.legend(0.05,0.95);
  fig2.save("zqperr");
}
```

Observation:

Roundoff incurred during the computation of α and β leads to "wrong" roots.
For large γ the computed small root may be fairly inaccurate as regards its *relative error*, which can be several orders of magnitude larger than machine precision EPS.

The large root always enjoys a small relative error about the size of EPS.

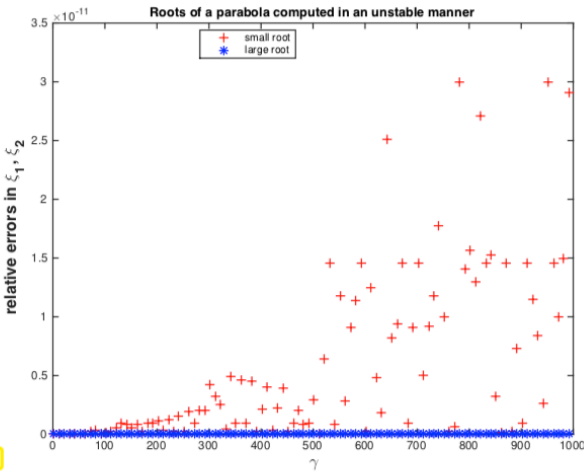


Fig. 40

In order to understand why the small root is much more severely affected by roundoff, note that its computation involves the subtraction of two large numbers, if γ is large. This is the typical situation, in which cancellation occurs.

(1.5.43) Visualisation of cancellation effect

We look at the *exact* subtraction of two almost equal positive numbers both of which have small relative errors (red boxes) with respect to some desired exact value (indicated by blue boxes). The result of the subtraction will be small, but the errors may add up during the subtraction, ultimately constituting a large fraction of the result.

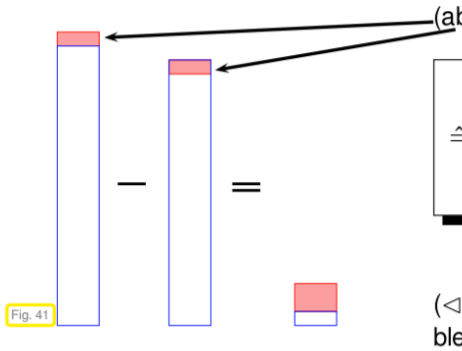


Fig. 41

Cancellation

≙ Subtraction of almost equal numbers

(➤ extreme *amplification* of *relative* errors)

(◁ Roundoff error introduced by subtraction itself is negligible.)

Example 1.5.44 (Cancellation in decimal system)

We consider two positive numbers x, y of about the same size afflicted with relative errors $\approx 10^{-7}$. This means that their seventh decimal digits are perturbed, here indicated by $*$. When we subtract the

two numbers the perturbed digits are shifted to the left, resulting in a possible relative error of $\approx 10^{-3}$:

$$\begin{array}{rcll} x & = & 0.123467* & \leftarrow \text{7th digit perturbed} \\ y & = & 0.123456* & \leftarrow \text{7th digit perturbed} \\ \hline x-y & = & 0.000011* = 0.11*000 \cdot 10^{-4} & \leftarrow \text{3rd digit perturbed} \end{array}$$

padded zeroes

Again, this example demonstrates that cancellation wreaks havoc through error amplification, not through the roundoff error due to the subtraction.

Example 1.5.45 (Cancellation when evaluating difference quotients → [4, Sect. 8.2.6], [2, Ex. 1.3])

From analysis we know that the derivative of a differentiable function $f : I \subset \mathbb{R} \to \mathbb{R}$ at a point $x \in I$ is the limit of a difference quotient

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} .$$

This suggests the following approximation of the derivative by a difference quotient with small but finite $h > 0$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for} \quad |h| \ll 1 .$$

Results from analysis tell us that the approximation error should tend to zero for $h \to 0$. More precise quantitative information is provided by the Taylor formula for a twice continuously differentiable function [2, p. 5]

$$f(x+h) = f(x) + f'(x)h + \tfrac{1}{2} f''(\xi) h^2 \quad \text{for some} \quad \xi = \xi(x,h) \in [\min\{x, x+h\}, \max\{x, x+h\}] , \tag{1.5.46}$$

from which we infer

$$\frac{f(x+h) - f(x)}{h} - f'(x) = \tfrac{1}{2} h f''(\xi) \quad \text{for some} \quad \xi = \xi(x,h) \in [\min\{x, x+h\}, \max\{x, x+h\}] . \tag{1.5.47}$$

We investigate the approximation of the derivative by difference quotients for $f = \exp$, $x = 0$, and different values of $h > 0$:

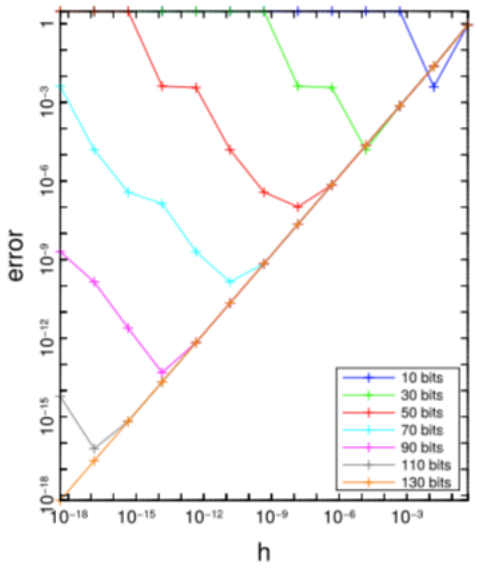
C++11-code 1.5.48: Difference quotient approximation of the derivative of exp → GITLAB

```
//!  Difference quotient approximation
//!  of the derivative of exp
void diffq(){
  double h = 0.1, x = 0.0;
  for(int i = 1; i <= 16; ++i){
    double df = (exp(x+h)-exp(x))/h;
    cout << setprecision(14) << fixed;
    cout  << setw(5) << -i
       << setw(20) << df-1 << endl;
    h /= 10;
  }
}
```

$\log_{10}(h)$	relative error
-1	0.05170918075648
-2	0.00501670841679
-3	0.00050016670838
-4	0.00005000166714
-5	0.00000500000696
-6	0.00000049996218
-7	0.00000004943368
-8	-0.00000000607747
-9	0.00000008274037
-10	0.00000008274037
-11	0.00000008274037
-12	0.00008890058234
-13	-0.00079927783736
-14	-0.00079927783736
-15	0.11022302462516
-16	-1.00000000000000

Measured relative errors ▷

We observe an initial decrease of the relative approximation error followed by a steep increase when h drops below 10^{-8}.



That the observed errors are really due to round-off errors is confirmed by the following numerical results reported besides, using a variable precision floating point module of EIGEN, the MPFRC++ Support module.

C++11-code 1.5.49: Evaluation of difference quotients with variable precision → GITLAB

```
// This module provides support for multi precision floating point
    numbers via the MPFR C++ library which itself is built upon
    MPFR/GMP.
#include <unsupported/Eigen/MPRealSupport>
//!  Numerical differentiation of exponential function with extended
    precision arithmetic
```

```
//!  Uses the unsupported EIGEN MPFRC++ Support module
void numericaldifferentiation(){
     // declare numeric type
     typedef mpfr::mpreal numeric_t;
     // bit number that should be evaluated
     int n = 7, k = 13;
     VectorXi bits(n); bits << 10,30,50,70,90,110,130;
     MatrixXd experr(13, bits.size()+1);
     for(int i = 0; i < n; ++i){
          // set precision to bits(i) bits (double has 53 bits)
          numeric_t::set_default_prec( bits(i) );
          numeric_t x = "1.1";// Evaluation point in extended precision
          for(int j=0;j<k;++j) {
               numeric_t h = mpfr::pow("2", -1-5*j);// width of
                  difference quotient in extended precision
               // compute (absolute) error
               experr(j,i+1) = mpfr::abs(( mpfr::exp(x+h) -
                  mpfr::exp(x) ) / h - mpfr::exp(x)).toDouble();
               experr(j,0) = h.toDouble();
          }
     }
     // Plotting
     // ...
```

Line 16: The literal `"1.1"` in instead of `1.1` prevents the conversion to double.

Obvious culprit: cancellation when computing the numerator of the difference quotient for small $|h|$ leads to a strong amplification of inevitable errors introduced by the evaluation of the transcendent exponential function.

We witness the competition of two opposite effects: Smaller h results in a better approximation of the derivative by the difference quotient, but the impact of cancellation is the stronger the smaller $|h|$.

$$\left.\begin{array}{l}\text{Approximation error } f'(x) - \dfrac{f(x+h)-f(x)}{h} \to 0 \\ \text{Impact of roundoff} \to \infty\end{array}\right\} \text{ as } h \to 0\,.$$

In order to provide a rigorous underpinning for our conjecture, in this example we embark on our first roundoff error analysis merely based on the "Axiom of roundoff analysis" Ass. 1.5.32: As in the computational example above we study the approximation of $f'(x) = e^x$ for $f = \exp$, $x \in \mathbb{R}$.

$$\begin{aligned} \texttt{df} &= \frac{e^{x+h}\boxed{(1+\delta_1)} - e^x\boxed{(1+\delta_2)}}{h} \\ &= e^x\left(\frac{e^h-1}{h} + \frac{\delta_1 e^h - \delta_2}{h}\right) \\ \Rightarrow\ |\texttt{df}| &\le e^x\left(\tfrac{e^h-1}{h} + \texttt{eps}\,\tfrac{1+e^h}{h}\right) \end{aligned}$$

correction factors take into account roundoff:
(→ "'axiom of roundoff analysis", Ass. 1.5.32)

$$|\delta_1|, |\delta_2| \le \texttt{eps}\,.$$

$1 + O(h)$ $\quad O(h^{-1})$ for $h \to 0$

(Note that the estimate for the term $(e^h - 1)/h$ is a particular case of (1.5.47).)

relative error: $\left|\dfrac{e^x - \texttt{df}}{e^x}\right| \approx h + \dfrac{2\texttt{eps}}{h} \to \min$ for $h = \sqrt{2\,\texttt{eps}}$.

In double precision: $\sqrt{2\texttt{eps}} = 2.107342425544702 \cdot 10^{-8}$

Remark 1.5.50 (Cancellation during the computation of relative errors)

In the numerical experiment of Ex. 1.5.45 we computed the relative error of the result by subtraction, see Code 1.5.48. Of course, massive cancellation will occur! Do we have to worry?

In this case cancellation can be tolerated, because we are *interested only* in the magnitude of the relative error. Even if it was affected by a large relative error, this information is still not compromised.

For example, if the relative error has the exact value 10^{-8}, but can be computed only with a huge relative error of 10%, then the perturbed value would still be in the range $[0.9 \cdot 10^{-8}, 1.1 \cdot 10^{-8}]$. Therefore it will still have the correct magnitude and still permit us to conclude the number of valid digits correctly.

Remark 1.5.51 (Cancellation in Gram-Schmidt orthogonalisation of Exp. 1.5.5)

The matrix created by the MATLAB command `A = hilb(10)`, the so-called Hilbert matrix, has columns that are almost linearly dependent.

Cancellation when computing orthogonal projection of vector $\mathbf{a}$ onto space spanned by vector $\mathbf{b}$ ▷

$$\mathbf{p} = \mathbf{a} - \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \mathbf{b} \, .$$

If $\mathbf{a}$, $\mathbf{b}$ point in almost the same direction, $\|\mathbf{p}\| \ll \|\mathbf{a}\|, \|\mathbf{b}\|$, so that a "tiny" vector $\mathbf{p}$ is obtained by subtracting two "long" vectors, which implies cancellation.

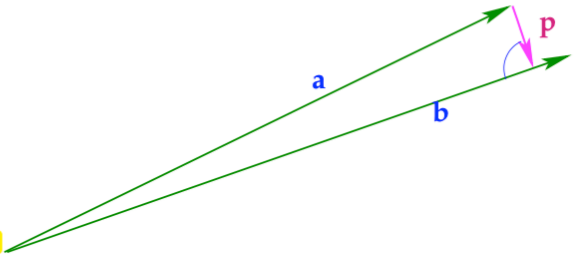


Fig. 43

This can happen in Line 10 of Code 1.5.3.

Example 1.5.52 (Cancellation: roundoff error analysis)

We consider a simple arithmetic expression written in two ways:

$$a^2 - b^2 = (a+b)(a-b) \, , \quad a, b \in \mathbb{R}; .$$

We evaluate this term by means of two algebraically equivalent algorithms for the input data $a = 1.3$, $b = 1.2$ in 2-digit decimal arithmetic with standard rounding. ("Algebraically equivalent" means that two algorithms will produce the same results in the absence of roundoff errors.

Algorithm A	Algorithm B
$x := a \tilde{\cdot} a = 1.7$ (rounded)	$x := a \tilde{+} b = 2.5$ (exact)
$y := b \tilde{\cdot} b = 1.4$ (rounded)	$y := a \tilde{-} b = 0.1$ (exact)
$x \tilde{-} y = 0.30$ (exact)	$x * y = 0.25$ (exact)

Algorithm B produces the exact result, whereas Algorithm A fails to do so. Is this pure coincidence or an indication of the superiority of algorithm B? This question can be answered by roundoff error analysis. We demonstrate the approach for the two algorithms A & B and general input $a, b, \in \mathbb{R}$.

Roundoff error analysis heavily relies on Ass. 1.5.32 and dropping terms of "higher order" in the machine precision, that is terms that behave like $O(\text{EPS}^q)$, $q > 1$. It involves introducing the relative roundoff error for every elementary operation through a factor $(1+\delta)$, $|\delta| \leq \text{EPS}$.

Algorithm A:

$$x = a^2(1+\delta_1)\,,\ y = b^2(1+\delta_2)$$
$$\tilde{f} = (a^2(1+\delta_1) - b^2(1+\delta_2))(1+\delta_3) = f + a^2\delta_1 - b^2\delta_2 + (a^2-b^2)\delta_3 + O(\text{EPS}^2)$$

$$\frac{|\tilde{f}-f|}{|f|} \leq \text{EPS}\frac{a^2+b^2+|a^2-b^2|}{|a^2-b^2|} + O(\text{EPS}^2) = \text{EPS}\left(1 + \frac{|a^2+b^2|}{|a^2-b^2|}\right) + O(\text{EPS}^2)\,. \qquad (1.5.53)$$

will be neglected

For $a \approx b$ the relative error of the result of Algorithm A will be much larger than the machine precision EPS. This reflects cancellation in the last subtraction step.

Algorithm B:

$$x = (a+b)(1+\delta_1)\,,\ y = (a-b)(1+\delta_2)$$
$$\tilde{f} = (a+b)(a-b)(1+\delta_1)(1+\delta_2)(1+\delta_3) = f + (a^2-b^2)(\delta_1+\delta_2+\delta_3) + O(\text{EPS}^2)$$

$$\frac{|\tilde{f}-f|}{|f|} \leq |\delta_1+\delta_2+\delta_3| + O(\text{EPS}^2) \leq 3\text{EPS} + O(\text{EPS}^2)\,. \qquad (1.5.54)$$

Relative error of the result of Algorithm B is always $\approx$ EPS !

In this example we see a general guideline at work:

> If inevitable, subtractions prone to cancellation should be done as early as possible.

The reason is that input data and and initial intermediate results are usually not as much tainted by roundoff errors as numbers computed after many steps.

(1.5.55) Avoiding disastrous cancellation

The following examples demonstrate a few fundamental techniques for steering clear of cancellation by using alternative formulas that yield the same value (in exact arithmetic), but do not entail subtracting two numbers of almost equal size.

Example 1.5.56 (Stable discriminant formula → Ex. 1.5.39, [2, Ex. 2.10])

If ξ_1 and ξ_2 are the two roots of the quadratic polynomial $p(\xi) = \xi^2 + \alpha\xi + \beta$, then $\xi_1 \cdot \xi_2 = \beta$ (Vieta's formula). Thus once we have computed a root, we can obtain the other by simple division.

Idea:

1. Depending on the sign of α compute "stable root" without cancellation.
2. Compute other root from Vieta's formula (avoiding subtraction)

C++11-code 1.5.57: Stable computation of real root of a quadratic polynomial ↪ GITLAB

```
//!  C++ function computing the zeros of a quadratic polynomial
//!  ξ → ξ^2+αξ+β by means of the familiar discriminant
//!  formula ξ_{1,2} = 1/2(−α ± √(α^2 − 4β)).
//!  This is a stable implementation based on Vieta's theorem.
//!  The zeros are returned in a column vector
VectorXd zerosquadpolstab(double alpha, double beta){
  Vector2d z(2);
  double D = std::pow(alpha,2) −4*beta; // discriminant
  if(D < 0) throw "no real zeros";
  else{
    double wD = std::sqrt(D);
    // Use discriminant formula only for zero far away from 0
    // in order to avoid cancellation.  For the other zero
    // use Vieta's formula.
    if(alpha >= 0){
      double t = 0.5*(−alpha−wD);  //
      z << t, beta/t;
    }
    else{
      double t = 0.5*(−alpha+wD);  //
      z << beta/t, t;
    }
  }
  return z;
}
```

➥ Invariably, we add numbers with the same sign in Line 17 and Line 21.

Numerical experiment based on the driver code Code 1.5.42.

Observation:

The new code can also compute the small root of the polynomial $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$ (expanded in monomials) with a relative error $\approx$ EPS.

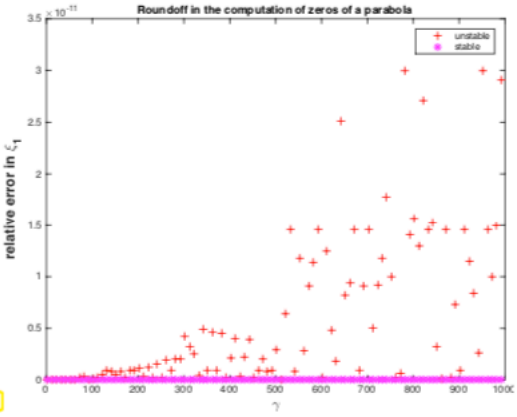


Fig. 44

Example 1.5.58 (Exploiting trigonometric identities to avoid cancellation)

The task is to evaluate the integral

$$\int_0^x \sin t \, dt = \underbrace{1 - \cos x}_{I} = \underbrace{2\sin^2(x/2)}_{II} \quad \text{for} \quad 0 < x \ll 1 \,, \tag{1.5.59}$$

and this can be done by the two different formulas I and II.

Relative error of expression I (`1-cos(x)`) with respect to equivalent expression II (`2*sin(x/2)^{2}`) ▷

Expression I is affected by cancellation for $|x| \ll 1$, since then $\cos x \approx 1$, whereas expression II can be evaluated with a relative error $\approx$ EPS for all x.

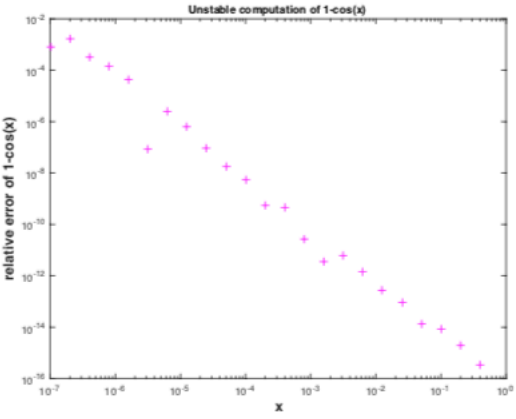


Fig.

Analytic manipulations offer ample opportunity to rewrite expressions in equivalent form immune to cancellation.

Example 1.5.60 (Switching to equivalent formulas to avoid cancellation)

Now we see an example of a computation allegedly dating back to Archimedes, who tried to approximate the area of a circle by the areas of inscribed regular polygons.

Approximation of circle by regular n-gon, $n \in \mathbb{N}$ ▷

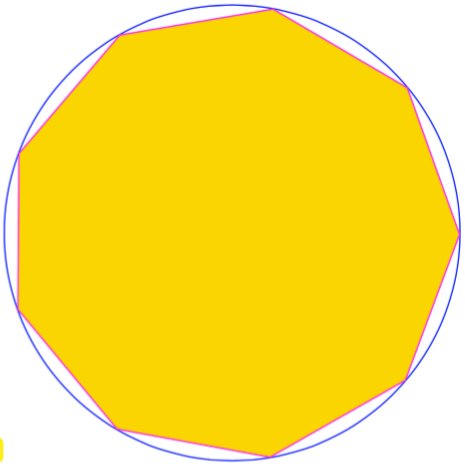


Fig. 46

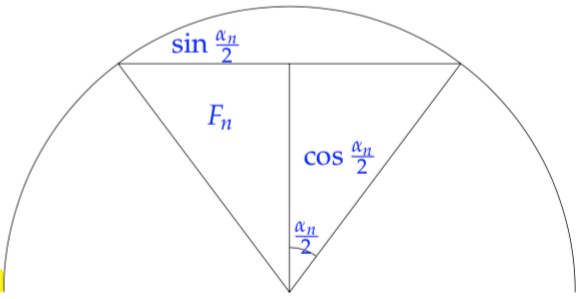


Fig. 47

Area of n-gon:

$$A_n = n \cos \frac{\alpha_n}{2} \sin \frac{\alpha_n}{2} = \frac{n}{2} \sin \alpha_n = \frac{n}{2} \sin\left(\frac{2\pi}{n}\right) .$$

Recursion formula for A_n derived from

$$\sin \frac{\alpha_n}{2} = \sqrt{\frac{1-\cos \alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2 \alpha_n}}{2}} ,$$

Initial approximation: $A_6 = \frac{3}{2}\sqrt{3}$.

C++11-code 1.5.61: Tentative computation of circumference of regular polygon ⇝ GITLAB

```
//!  Approximation of Pi by approximating the circumference of a
//!  regular polygon
MatrixXd ApproxPIinstable(double tol = 1e-8, int maxIt = 50){
  double s=sqrt(3)/2.; double An=3.*s;// initialization (hexagon case)
  unsigned int n = 6, it = 0;
  MatrixXd res(maxIt,4); // matrix for storing results
  res(it,0) = n; res(it,1) = An;
  res(it,2) = An - M_PI; res(it,3)=s;
  while( it < maxIt && s > tol ){// terminate when s is 'small enough'
    s = sqrt((1.- sqrt(1.-s*s))/2.);// recursion for area
    n *= 2; An = n/2.*s;     // new estimate for circumference
    ++it;
    res(it,0) =n; res(it,1) =An;// store results and (absolute) error
    res(it,2) = An - M_PI; res(it,3)=s;
  }
  return res.topRows(it);
}
```

The approximation deteriorates after applying the recursion formula many times:

n	A_n	$A_n - \pi$	$\sin \alpha_n$
6	2.598076211353316	-0.543516442236477	0.866025403784439
12	3.000000000000000	-0.141592653589794	0.500000000000000
24	3.105828541230250	-0.035764112359543	0.258819045102521
48	3.132628613281237	-0.008964040308556	0.130526192220052
96	3.139350203046872	-0.002242450542921	0.065403129230143
192	3.141031950890530	-0.000560702699263	0.032719082821776
384	3.141452472285344	-0.000140181304449	0.016361731626486
768	3.141557607911622	-0.000035045678171	0.008181139603937
1536	3.141583892148936	-0.000008761440857	0.004090604026236
3072	3.141590463236762	-0.000002190353031	0.002045306291170
6144	3.141592106043048	-0.000000547546745	0.001022653680353
12288	3.141592516588155	-0.000000137001638	0.000511326906997
24576	3.141592618640789	-0.000000034949004	0.000255663461803
49152	3.141592645321216	-0.000000008268577	0.000127831731987
98304	3.141592645321216	-0.000000008268577	0.000063915865994
196608	3.141592645321216	-0.000000008268577	0.000031957932997
393216	3.141592645321216	-0.000000008268577	0.000015978966498
786432	3.141593669849427	0.000001016259634	0.000007989485855
1572864	3.141592303811738	-0.000000349778055	0.000003994741190
3145728	3.141608696224804	0.000016042635011	0.000001997381017
6291456	3.141586839655041	-0.000005813934752	0.000000998683561
12582912	3.141674265021758	0.000081611431964	0.000000499355676
25165824	3.141674265021758	0.000081611431964	0.000000249677838
50331648	3.143072740170040	0.001480086580246	0.000000124894489
100663296	3.159806164941135	0.018213511351342	0.000000062779708
201326592	3.181980515339464	0.040387861749671	0.000000031610136
402653184	3.354101966249685	0.212509312659892	0.000000016660005
805306368	4.242640687119286	1.101048033529493	0.000000010536712
1610612736	6.000000000000000	2.858407346410207	0.000000007450581

Where does cancellation occur in Line 11 of Code 1.5.61? Since $s \ll 1$, computing $1 - s$ will not trigger cancellation. However, the subtraction $1 - \sqrt{1 - s^2}$ will, because $\sqrt{1 - s^2} \approx 1$ for $s \ll 1$:

Where does cancellation occur in Line 11 of Code 1.5.61? Since $s \ll 1$, computing $1 - s$ will not trigger cancellation. However, the subtraction $1 - \sqrt{1 - s^2}$ will, because $\sqrt{1 - s^2} \approx 1$ for $s \ll 1$:

$$\sin \frac{\alpha_n}{2} = \sqrt{\frac{1 - \cos \alpha_n}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}}$$

For $\alpha_n \ll 1$: $\sqrt{1 - \sin^2 \alpha_n} \approx 1$

Cancellation here!

We arrive at an *equivalent* formula not vulnerable to cancellation essentially using the identity $(a + b)(a - b) = a^2 - b^2$ in order to eliminate the difference of square roots in the numerator.

$$\begin{aligned} \sin \frac{\alpha_n}{2} &= \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2} \cdot \frac{1 + \sqrt{1 - \sin^2 \alpha_n}}{1 + \sqrt{1 - \sin^2 \alpha_n}}} \\ &= \sqrt{\frac{1 - (1 - \sin^2 \alpha_n)}{2(1 + \sqrt{1 - \sin^2 \alpha_n})}} = \frac{\sin \alpha_n}{\sqrt{2\left(1 + \sqrt{1 - \sin^2 \alpha_n}\right)}}. \end{aligned}$$

C++11-code 1.5.62: Stable recursion for area of regular n-gon ↪ GITLAB

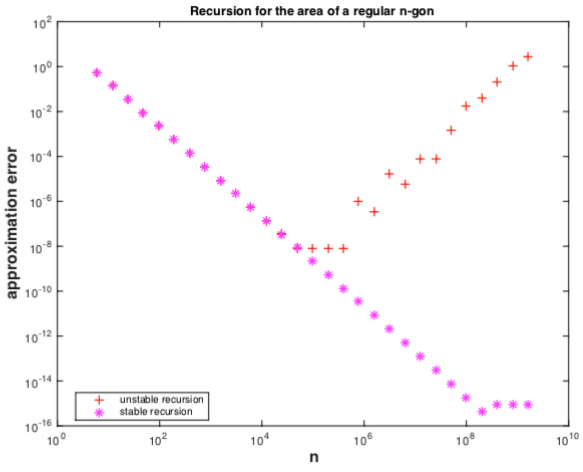
```
2 //!  Approximation of Pi by approximating the circumference of a
```

```
3 //!  regular polygon
4 MatrixXd apprpistable(double tol = 1e-8, int maxIt = 50){
5   double s=sqrt(3)/2.; double An=3.*s;// initialization (hexagon case)
6   unsigned int n = 6, it = 0;
7   MatrixXd res(maxIt,4); // matrix for storing results
8   res(it,0) = n; res(it,1) = An;
9   res(it,2) = An - M_PI; res(it,3)=s;
10  while( it < maxIt && s > tol ){// terminate when s is 'small enough'
11    s = s/sqrt(2*(1+sqrt((1+s)*(1-s))));// Stable recursion without
        cancellation
12    n *= 2; An = n/2.*s;     // new estimate for circumference
13    ++it;
14    res(it,0) =n; res(it,1) =An;// store results and (absolute) error
15    res(it,2) = An - M_PI; res(it,3)=s;
16  }
17  return res.topRows(it);
18 }
```

Using the stable recursion, we observe better approximation for polygons with more corners:

n	A_n	$A_n - \pi$	$\sin \alpha_n$
6	2.598076211353316	-0.543516442236477	0.866025403784439
12	3.000000000000000	-0.141592653589793	0.500000000000000
24	3.105828541230249	-0.035764112359544	0.258819045102521
48	3.132628613281238	-0.008964040308555	0.130526192220052
96	3.139350203046867	-0.002242450542926	0.065403129230143
192	3.141031950890509	-0.000560702699284	0.032719082821776
384	3.141452472285462	-0.000140181304332	0.016361731626487
768	3.141557607911857	-0.000035045677936	0.008181139603937
1536	3.141583892148318	-0.000008761441475	0.004090604026235
3072	3.141590463228050	-0.000002190361744	0.002045306291164
6144	3.141592105999271	-0.000000547590522	0.001022653680338
12288	3.141592516692156	-0.000000136897637	0.000511326907014
24576	3.141592619365383	-0.000000034224410	0.000255663461862
49152	3.141592645033690	-0.000000008556103	0.000127831731976
98304	3.141592651450766	-0.000000002139027	0.000063915866118
196608	3.141592653055036	-0.000000000534757	0.000031957933076
393216	3.141592653456104	-0.000000000133690	0.000015978966540
786432	3.141592653556371	-0.000000000033422	0.000007989483270
1572864	3.141592653581438	-0.000000000008355	0.000003994741635
3145728	3.141592653587705	-0.000000000002089	0.000001997370818
6291456	3.141592653589271	-0.000000000000522	0.000000998685409
12582912	3.141592653589663	-0.000000000000130	0.000000499342704
25165824	3.141592653589761	-0.000000000000032	0.000000249671352
50331648	3.141592653589786	-0.000000000000008	0.000000124835676
100663296	3.141592653589791	-0.000000000000002	0.000000062417838
201326592	3.141592653589794	0.000000000000000	0.000000031208919
402653184	3.141592653589794	0.000000000000001	0.000000015604460
805306368	3.141592653589794	0.000000000000001	0.000000007802230
1610612736	3.141592653589794	0.000000000000001	0.000000003901115

Plot of errors for approximations of π as computed by the two algebraically equivalent recursion formulas▷



Observation, *cf.* Ex. 1.5.45

Amplified roundoff errors due to cancellation supersedes approximation error for $n \geq 10^5$.

Roundoff errors merely of magnitude EPS in the case of stable recursion

Example 1.5.63 (Summation of exponential series)

In principle, the function value $\exp(x)$ can be approximated up to any accuracy by summing sufficiently many terms of the globally convergent exponential series.

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots .$$

C++11-code 1.5.64: Summation of exponential series ↪ GITLAB

```
2  double expeval(double x,
3                 double tol=1e-8){
4    // Initialization
5    double y = 1.0, term = 1.0;
6    long int k = 1;
7    // Termination criterion
8    while(abs(term) > tol*y) {
9      term *= x/k;     // next summand
10     y += term; // Summation
11     ++k;
12   }
13   return y;
14 }
```

Results for `tol` $= 10^{-8}$, $\widetilde{\exp}$ designates the approximate value for $\exp(x)$ returned by the function from Code 1.5.64. Rightmost column lists relative errors, which tells us the number of valid digits in the approximate result.

x	Approximation $\widetilde{\exp}(x)$	$\exp(x)$	$\frac{\lvert\exp(x)-\widetilde{\exp}(x)\rvert}{\exp(x)}$
-20	6.1475618242e-09	2.0611536224e-09	1.982583033727893
-18	1.5983720359e-08	1.5229979745e-08	0.049490585500089
-16	1.1247503300e-07	1.1253517472e-07	0.000534425951530
-14	8.3154417874e-07	8.3152871910e-07	0.000018591829627
-12	6.1442105142e-06	6.1442123533e-06	0.000000299321453
-10	4.5399929604e-05	4.5399929762e-05	0.000000003501044
-8	3.3546262812e-04	3.3546262790e-04	0.000000000662004
-6	2.4787521758e-03	2.4787521767e-03	0.000000000332519
-4	1.8315638879e-02	1.8315638889e-02	0.000000000530724
-2	1.3533528320e-01	1.3533528324e-01	0.000000000273603
0	1.0000000000e+00	1.0000000000e+00	0.000000000000000
2	7.3890560954e+00	7.3890560989e+00	0.000000000479969
4	5.4598149928e+01	5.4598150033e+01	0.000000001923058
6	4.0342879295e+02	4.0342879349e+02	0.000000001344248
8	2.9809579808e+03	2.9809579870e+03	0.000000002102584
10	2.2026465748e+04	2.2026465795e+04	0.000000002143799
12	1.6275479114e+05	1.6275479142e+05	0.000000001723845
14	1.2026042798e+06	1.2026042842e+06	0.000000003634135
16	8.8861105010e+06	8.8861105205e+06	0.000000002197990
18	6.5659968911e+07	6.5659969137e+07	0.000000003450972
20	4.8516519307e+08	4.8516519541e+08	0.000000004828737

Observation:

Large relative approximation errors for $x \ll 0$.

For $x \ll 0$ we have $\exp(x)| \ll 1$, but this value is computed by summing large numbers of opposite sign.

Terms summed up for $x = -20$ ▷

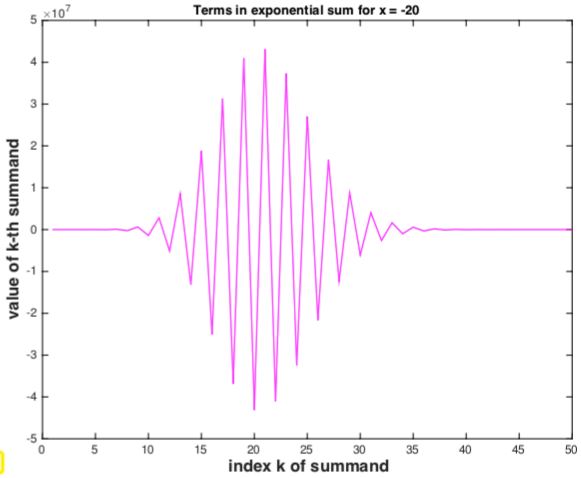


Fig. 49

Remedy: Cancellation can be avoided by using identity

$$\exp(x) = \frac{1}{\exp(-x)} \quad \text{, if} \quad x < 0 \,.$$

Example 1.5.65 (Combat cancellation by approximation)

In a computer code we have to provide a routine for the evaluation of

$$\int_0^1 e^{at}\,\mathrm{d}t = \frac{\exp(a)-1}{a} \quad \text{for any} \quad a > 0\,, \tag{1.5.66}$$

cf. the discussion of cancellation in the context of numerical differentiation in Ex. 1.5.45. There we observed massive cancellation.

Recall the Taylor expansion formula in one dimension for a function that is $m+1$ times continuously differentiable in a neighborhood of x [14, Satz 5.5.1]

$$f(x+h) = \sum_{k=0}^{m} \frac{1}{k!} f^{(k)}(x)h^k + R_m(x,h)\,, \quad R_m(x,h) = \frac{1}{(m+1)!} f^{(m+1)}(\xi)h^{m+1}\,,$$

for some $\xi \in [\min\{x, x+h\}, \max\{x, x+h\}]$, and for all sufficiently small $|h|$. Here $R(x,h)$ is called the remainder term and $f^{(k)}$ denotes the k derivative of f.

Cancellation in (1.5.66) can be avoided by replacing $\exp(a)$, $a > 0$, with a suitable Taylor expansion around $a = 0$ and then dividing by a:

$$\frac{\exp(a)-1}{a} = \sum_{k=0}^{m} \frac{1}{(k+1)!} a^k + R_m(a)\,, \quad R_m(a) = \frac{1}{(m+1)!}\exp(\xi)a^m \text{ for some } 0 \le \xi \le a\,.$$

For a similar discussion see [2, Ex. 2.12].

Issue: How to choose the number m of terms to be retained in the Taylor expansion? We have to pick m large enough such that the relative approximation error remains below a prescribed threshold `tol`. To *estimate* the relative approximation error, we use the expression for the remainder together with the simple estimate $(\exp(a)-1)/a > 1$ for all $a > 0$:

$$\text{rel. err.} = \frac{(e^a-1)/a - \sum_{k=0}^{m} \frac{1}{(k+1)!}a^k}{(e^a-1)/a} \le \frac{1}{(m+1)!}\exp(\xi)a^m \le \frac{1}{(m+1)!}\exp(a)a^m\,.$$

For $a = 10^{-3}$ we get

m	1	2	3	4	5
	1.0010e-03	5.0050e-07	1.6683e-10	4.1708e-14	8.3417e-18

Hence, keeping $m = 3$ terms is enough for achieving about 10 valid digits.

Relative error of unstable formula `(exp(a)-1.0)/a` and relative error, when using a Taylor expansion approximation for small a ▷

```
if (abs(a) < 1E-3)
  v = 1.0 + (1.0/2 + 1.0/6*a)*a;
else
  v = (exp(a)-1.0)/a;
end
```

Error computed by comparison with MATLAB's built-in function `expm1` that provides a stable implementation of $\exp(x)-1$.

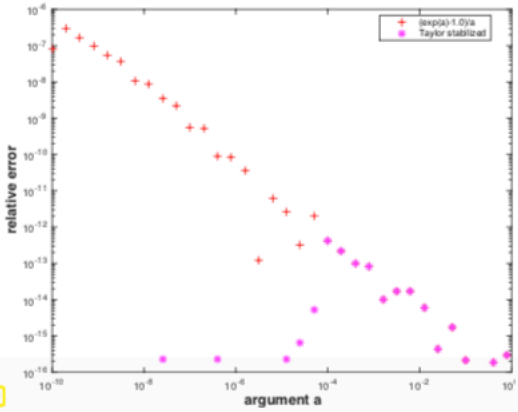


Fig. 50

1.5.5 Numerical stability

We have seen that a particular "problem" can be tackled by different "algorithms", which produce different results due to roundoff errors. This section will clarify what distinguishes a "good" algorithm from a rather abstract point of view.

(1.5.67) The "problem"

A mathematical notion of "problem":

- data space X, usually $X \subset \mathbb{R}^n$
- result space Y, usually $Y \subset \mathbb{R}^m$
- mapping (problem function) $F : X \mapsto Y$

A problem is a well defined *function* that assigns to each datum a result.

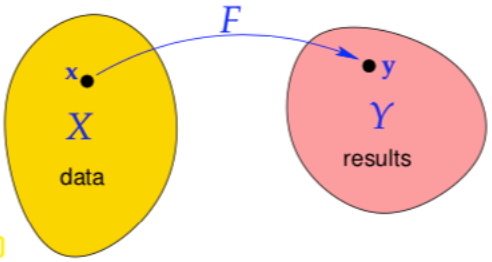


Fig. 51

Note: In this course, both the data space X and the result space Y will always be subsets of finite dimensional vector spaces.

Example 1.5.68 (The "matrix×vector-multiplication problem")

We consider the "problem" of computing the product $\mathbf{Ax}$ for a given matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ and a given vector $\mathbf{x} \in \mathbb{K}^n$.

➢
- Data space $X = \mathbb{K}^{m,n} \times \mathbb{K}^n$ (input is a matrix and a vector)
- Result space $Y = \mathbb{R}^m$ (space of column vectors)
- Problem function $F : X \to Y$, $F(\mathbf{a}, \mathbf{x}) := \mathbf{Ax}$

(1.5.69) Norms on spaces of vectors and matrices

Norms provide tools for measuring errors. Recall from linear algebra and calculus [10, Sect. 4.3], [7, Sect. 6.1]:

Definition 1.5.70. Norm

X = vector space over field $\mathbb{K}$, $\mathbb{K} = \mathbb{C}, \mathbb{R}$. A map $\|\cdot\| : X \mapsto \mathbb{R}_0^+$ is a norm on X, if it satisfies

(i) $\forall \mathbf{x} \in X: \ \mathbf{x} \neq 0 \ \Leftrightarrow \ \|\mathbf{x}\| > 0$ (definite),
(ii) $\|\lambda \mathbf{x}\| = |\lambda| \|\mathbf{x}\| \quad \forall \mathbf{x} \in X, \lambda \in \mathbb{K}$ (homogeneous),
(iii) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in X$ (triangle inequality).

Examples: (for vector space $\mathbb{K}^n$, vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{K}^n$)

name	:	definition	EIGEN function
Euclidean norm	:	$\|\mathbf{x}\|_2 := \sqrt{\lvert x_1\rvert^2 + \cdots + \lvert x_n\rvert^2}$	`x.norm()`
1-norm	:	$\|\mathbf{x}\|_1 := \lvert x_1\rvert + \cdots + \lvert x_n\rvert$	`x.lpNorm<1>()`
∞-norm, max norm	:	$\|\mathbf{x}\|_\infty := \max\{\lvert x_1\rvert, \ldots, \lvert x_n\rvert\}$	`x.lpNorm<Eigen::Infinity>()`

Remark 1.5.71 (Inequalities between vector norms)

All norms on the vector space $\mathbb{K}^n$, $n \in \mathbb{N}$, are equivalent in the sense that for arbitrary two norms $\|\cdot\|_1$ and $\|\cdot\|_2$ we can always find a constant $C > 0$ such that

$$\|\mathbf{v}\|_1 \leq C \|\mathbf{v}\|_2 \quad \forall \mathbf{v} \in \mathbb{K}^n \, . \tag{1.5.72}$$

Of course, the constant C will usually depend on n and the norms under consideration.

For the vector norms introduced above, explicit expressions for the constants "C" are available: for all $\mathbf{x} \in \mathbb{K}^n$

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n} \|\mathbf{x}\|_2 \, , \tag{1.5.73}$$
$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n} \|\mathbf{x}\|_\infty \, , \tag{1.5.74}$$
$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n \|\mathbf{x}\|_\infty \, . \tag{1.5.75}$$

The matrix space $\mathbb{K}^{m,n}$ is a vector space, of course, and can also be equipped with various norms. Of particular importance are norms *induced by vector norms* on $\mathbb{K}^n$ and $\mathbb{K}^m$.

Definition 1.5.76. Matrix norm

Given vector norms $\|\cdot\|_1$ and $\|\cdot\|_2$ on $\mathbb{K}^n$ and $\mathbb{K}^m$, respectively, the associated matrix norm is defined by

$$\mathbf{M} \in \mathbb{R}^{m,n}: \quad \|\mathbf{M}\| := \sup_{\mathbf{x}\in\mathbb{R}^n\setminus\{0\}} \frac{\|\mathbf{M}\mathbf{x}\|_2}{\|\mathbf{x}\|_1} .$$

By virtue of definition the matrix norms enjoy an important property, they are sub-multiplicative:

$$\forall \mathbf{A} \in \mathbb{K}^{n,m}, \mathbf{B} \in \mathbb{K}^{m,k}: \quad \|\mathbf{A}\mathbf{B}\| \le \|\mathbf{A}\|\|\mathbf{B}\| . \tag{1.5.77}$$

✎ notations for matrix norms for *quadratic matrices* associated with standard vector norms:

$$\|\mathbf{x}\|_2 \to \|\mathbf{M}\|_2, \quad \|\mathbf{x}\|_1 \to \|\mathbf{M}\|_1, \quad \|\mathbf{x}\|_\infty \to \|\mathbf{M}\|_\infty$$

Example 1.5.78 (Matrix norm associated with ∞-norm and 1-norm)

Rather simple formulas are available for the matrix norms induced by the vector norms $\|\cdot\|_\infty$ and $\|\cdot\|_1$

$$\text{e.g. for } \mathbf{M} \in \mathbb{K}^{2,2}: \quad \begin{aligned} \|\mathbf{M}\mathbf{x}\|_\infty &= \max\{|m_{11}x_1 + m_{12}x_2|, |m_{21}x_1 + m_{22}x_2|\} \\ &\le \max\{|m_{11}| + |m_{12}|, |m_{21}| + |m_{22}|\}\, \|x\|_\infty , \\ \|\mathbf{M}\mathbf{x}\|_1 &= |m_{11}x_1 + m_{12}x_2| + |m_{21}x_1 + m_{22}x_2| \\ &\le \max\{|m_{11}| + |m_{21}|, |m_{12}| + |m_{22}|\}(|x_1| + |x_2|) . \end{aligned}$$

For general $\mathbf{M} \in \mathbb{K}^{m,n}$

➢ matrix norm $\leftrightarrow \|\cdot\|_\infty$ = row sum norm $\quad \|\mathbf{M}\|_\infty := \max_{i=1,\ldots,m} \sum_{j=1}^{n} |m_{ij}|$, (1.5.79)

➢ matrix norm $\leftrightarrow \|\cdot\|_1$ = column sum norm $\quad \|\mathbf{M}\|_1 := \max_{j=1,\ldots,n} \sum_{i=1}^{m} |m_{ij}|$. (1.5.80)

Sometimes special formulas for the Euclidean matrix norm come handy [5, Sect. 2.3.3]:

Lemma 1.5.81. Formula for Euclidean norm of a Hermitian matrix

$$\mathbf{A} \in \mathbb{K}^{n,n}, \mathbf{A} = \mathbf{A}^H \quad \Rightarrow \quad \|\mathbf{A}\|_2 = \max_{\mathbf{x} \neq 0} \frac{|\mathbf{x}^H \mathbf{A} \mathbf{x}|}{\|\mathbf{x}\|_2^2} .$$

Proof. Recall from linear algebra: Hermitian matrices (a special class of normal matrices) enjoy unitary similarity to diagonal matrices:

$$\exists \mathbf{U} \in \mathbb{K}^{n,n}, \text{ diagonal } \mathbf{D} \in \mathbb{R}^{n,n}: \quad \mathbf{U}^{-1} = \mathbf{U}^H \quad \text{and} \quad \mathbf{A} = \mathbf{U}^H \mathbf{D} \mathbf{U} .$$

Since multiplication with an unitary matrix preserves the 2-norm of a vector, we conclude

$$\|\mathbf{A}\|_2 = \left\| \mathbf{U}^H \mathbf{D} \mathbf{U} \right\|_2 = \|\mathbf{D}\|_2 = \max_{i=1,\ldots,i} |d_i| , \quad \mathbf{D} = \operatorname{diag}(d_1, \ldots, d_n) .$$

On the other hand, for the same reason:

$$\max_{\|\mathbf{x}\|_2 = 1} \mathbf{x}^H \mathbf{A} \mathbf{x} = \max_{\|\mathbf{x}\|_2 = 1} (\mathbf{U}\mathbf{x})^H \mathbf{D} (\mathbf{U}\mathbf{x}) = \max_{\|\mathbf{y}\|_2 = 1} \mathbf{y}^H \mathbf{D} \mathbf{y} = \max_{i=1,\ldots,i} |d_i| .$$

Hence, both expressions in the statement of the lemma agree with the largest modulus of eigenvalues of $\mathbf{A}$.

□

Corollary 1.5.82. Euclidean matrix norm and eigenvalues

For $\mathbf{A} \in \mathbb{K}^{m,n}$ the Euclidean matrix norm $\|\mathbf{A}\|_2$ is the square root of the largest (in modulus) eigenvalue of $\mathbf{A}^H \mathbf{A}$.

For a normal matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ (that is, $\mathbf{A}$ satisfies $\mathbf{A}^H \mathbf{A} = \mathbf{A}\mathbf{A}^H$) the Euclidean matrix norm agrees with the modulus of the largest eigenvalue.

(1.5.83) (Numerical) algorithm

When we talk about an "algorithm" we have in mind a concrete code function in MATLAB or C++; the only way to describe an algorithm is through a piece of code. We assume that this function defines another mapping $\widetilde{F} : X \to Y$ on the data space of the problem. Of course, we can only feed data to the MATLAB/C++-function, if they can be represented in the set $\mathbb{M}$ of machine numbers. Hence, implicit in the definition of $\widetilde{F}$ is the assumption that input data are subject to *rounding* before passing them to the code function proper.

Problem	Algorithm
$F : X \subset \mathbb{R}^n \to Y \subset \mathbb{R}^m$	$\widetilde{F} : X \to \widetilde{Y} \subset \mathbb{M}$

(1.5.84) Stable algorithm → **[2, Sect. 1.3]**

[Stable algorithm]

- We study a problem (→ § 1.5.67) $F : X \to Y$ on data space X into result space Y.
- We assume that both X and Y are equipped with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$, respectively (→ Def. 1.5.70).
- We consider a concrete algorithm $\widetilde{F} : X \to Y$ according to § 1.5.83.

We write $w(\mathbf{x})$, $\mathbf{x} \in X$, for the computational effort (→ Def. 1.4.1) required by the algorithm for input $\mathbf{x}$.

Definition 1.5.85. Stable algorithm

An algorithm $\widetilde{F}$ for solving a problem $F : X \mapsto Y$ is numerically stable if for all $\mathbf{x} \in X$ its result $\widetilde{F}(\mathbf{x})$ (possibly affected by roundoff) is the exact result for "slightly perturbed" data:

$$\exists C \approx 1: \quad \forall \mathbf{x} \in X: \quad \exists \tilde{\mathbf{x}} \in X: \quad \|\mathbf{x} - \tilde{\mathbf{x}}\|_X \leq C w(\mathbf{x}) \,\mathrm{EPS}\, \|\mathbf{x}\|_X \quad \wedge \quad \widetilde{F}(\mathbf{x}) = F(\tilde{\mathbf{x}}) \,.$$

Here EPS should be read as machine precision according to the "Axiom" of roundoff analysis Ass. 1.5.32.

Illustration of Def. 1.5.85
($\mathbf{y} \mathrel{\hat{=}}$ exact result for exact data $\mathbf{x}$) ▷

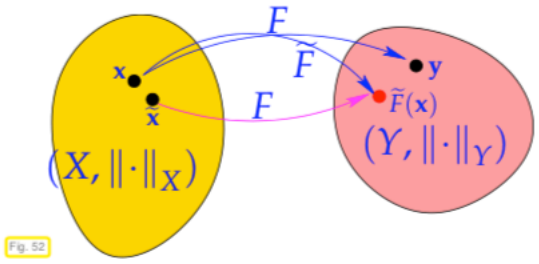


Fig. 52

Terminology:
Def. 1.5.85 introduces stability in the sense of backward error analysis

Sloppily speaking, the impact of roundoff $(*)$ on a *stable algorithm* is of the same order of magnitude as the effect of the inevitable perturbations due to rounding the input data.

➢ For stable algorithms roundoff errors are "harmless".

$(*)$ In some cases the definition of $\widetilde{F}$ will also involve some approximations as in Ex. 1.5.65. Then the above statement also includes approximation errors.

Example 1.5.86 (Testing stability of matrix×vector multiplication)

Assume you are given a black box implementation of a function

```
VectorXd mvmult(const MatrixX &A, const VectorXd &x)
```

that purports to provide a stable implementation of $\mathbf{Ax}$ for $\mathbf{A} \in \mathbb{K}^{m,n}$, $\mathbf{x} \in \mathbb{K}^n$, *cf.* Ex. 1.5.68. How can we verify this claim for particular data. Both, $\mathbb{K}^{m,n}$ and $\mathbb{K}^n$ are equipped with the Euclidean norm.

The task is, given $\mathbf{y} \in \mathbb{K}^n$ as returned by the function, to find conditions on $\mathbf{y}$ that ensure the existence of a $\widetilde{\mathbf{A}} \in \mathbb{K}^{m,n}$ such that

$$\widetilde{\mathbf{A}}\mathbf{x} = \mathbf{y} \quad \text{and} \quad \left\|\widetilde{\mathbf{A}} - \mathbf{A}\right\|_2 \leq Cmn \,\mathrm{EPS}\, \|\mathbf{A}\|_2 \,, \tag{1.5.87}$$

for a small constant ≈ 1.

In fact we can choose (easy computation)

$$\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{z}\mathbf{x}^T \quad , \quad \mathbf{z} := \frac{\mathbf{y} - \mathbf{A}\mathbf{x}}{\|\mathbf{x}\|_2^2} \in \mathbb{K}^m \, ,$$

and we find

$$\left\|\widetilde{\mathbf{A}} - \mathbf{A}\right\|_2 = \left\|\mathbf{z}\mathbf{x}^T\right\|_2 = \sup_{\mathbf{w}\in\mathbb{K}^n\setminus\{\mathbf{0}\}} \frac{\mathbf{x}\cdot\mathbf{w}\|\mathbf{z}\|_2}{\|\mathbf{w}\|_2} \leq \|\mathbf{x}\|_2\|\mathbf{z}\|_2 = \frac{\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2} \, .$$

Hence, in principle stability of an algorithm for computing $\mathbf{A}\mathbf{x}$ is confirmed, if for every $\mathbf{x} \in X$ the computed result $\mathbf{y} = \mathbf{y}(\mathbf{x})$ satisfies

$$\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2 \leq C\, mn\, \mathrm{EPS}\, \|\mathbf{x}\|_2 \|\mathbf{A}\|_2 \, ,$$

with a small constant $C > 0$ independent of data and problem size.

Remark 1.5.88 (Numerical stability and sensitive dependence on data)

A problem shows sensitive dependence on the data, if small perturbations of input data lead to large perturbations of the output. Such problems are also called ill-conditioned. For such problems stability of an algorithm is easily accomplished.

Example: The problem is the prediction of the position of the billard ball after ten bounces given the initial position, velcity, and spin.

It is well known, that tiny changes of the initial conditions can shift the final location of the ball to virtually any point on the table: the billard problem is chaotic.

Hence, a stable algorithm for its solution may just output a *fixed* or *random* position without even using the initial conditions!

Learning Outcomes

Principal take-home knowledge and skills from this chapter:

- Knowledge about the syntax of fundamental operations on matrices and vectors in EIGEN.
- Understanding of the concepts of computational effort/cost and asymptotic complexity in numerics.
- Awareness of the asymptotic complexity of basic linear algebra operations
- Ability to determine the (asymptotic) computational effort for a concrete (numerical linear algebra) algorithm.
- Ability to manipulate simple expressions involving matrices and vectors in order to reduce the computational cost for their evaluation.