Part I

Systems of Equations

1 Computing with Matrices and Vectors

The implementation of most numerical algorithms relies on array type data structures modelling concepts from linear algebra (matrices and vectors).

Related information can be found in [20, Ch. 1].

1.1 Notations

1.1.1 Vectors

- Vectors = are n-tuples ($n \in \mathbb{N}$) with components $x_i \in \mathbb{K}$, over field $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$.

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

- Default in this lecture: vectors = column vectors

$$\underbrace{\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{K}^n}_{\text{column vector}} \quad \underbrace{(x_1 \cdots x_n)}_{\text{row vector}}$$

  vector space of column vectors with n components

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

- Initialization of vectors in MATLAB:

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

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

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}^T = (x_1 \cdots x_n) \quad , \quad (x_1 \cdots x_n)^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

  Transposing in MATLAB: `x_T = x';`

- Adressing vector components:

  ✎ notation: $\mathbf{x} = (x_1 \ldots x_n)^T \rightarrow x_i, i = 1, \ldots, n$
  $\mathbf{x} \in \mathbb{K}^n \rightarrow (\mathbf{x})_i, i = 1, \ldots, n$

  MATLAB: `x(i)` selects i-th compoent of vector `x`

- Selecting sub-vectors:

  ✎ notation: $\mathbf{x} = (x_1 \ldots x_n)^T \succ (\mathbf{x})_{k:l} = (x_k, \ldots, x_l)^T, 1 \le k \le l \le n$

  MATLAB:
  `x(4:7)` $\longleftrightarrow$ `[x(4);x(5);x(6);x(7)]`,
  `x(7:-1:4)` $\longleftrightarrow$ `[x(7);x(6);x(5);x(4)]`,
  `x(3:2:10)` $\longleftrightarrow$ `[x(3);x(5);x(7);x(9)]`

- j-th unit vector: $\mathbf{e}_j = (0, \ldots, 1, \ldots, 0)^T, \quad e_i = \delta_{ij}$.

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

1.1.2 Matrices

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

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

  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., $\mathbf{A}, \mathbf{S}, \mathbf{Y}$

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

MATLAB: ▷ vectors are $1 \times n$/$n \times 1$-matrices

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

- Accessing matrix entries & sub-matrices (✎ notations):

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

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

MATLAB:

Matrix A	↦ entry at position (i,j)	=	`A(i,j)`
	↦ i-th row	=	`A(i,:)`
	↦ j-th column	=	`A(:,j)`
	↦ matrix block (sub-matrix) $(a_{ij})_{\substack{i=k,\dots,l \\ j=r,\dots,s}} = (\mathbf{A})_{k:l,r:s}$	=	`A(k:l,r:s)`

- Transposed matrix:

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

- Adjoint matrix (Hermitian transposed):

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

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

- Special matrices:

Identity matrix: $\mathbf{I} = \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix} \in \mathbb{K}^{n,n}$, MATLAB: `I = eye(n);`

Zero matrix: $\mathbf{O} = \begin{pmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{pmatrix} \in \mathbb{K}^{n,m}$, MATLAB: `O = zeros(n,m);`

Diagonal matrix: $\mathbf{D} = \begin{pmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{pmatrix} \in \mathbb{K}^{n,n}$, MATLAB: `D = diag(d);` with vector `d`

Remark 1.1.1 (Matrix storage formats). (for dense/full matrices, *cf.* Sect. 2.5)

$\mathbf{A} \in \mathbb{K}^{m,n}$ ▶ linear array (size mn) + index computations

(Note: leading dimension (*row major, column major*))

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

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

A_arr	1	2	3	4	5	6	7	8	9

Column major (Fortran, MATLAB, OpenGL):

A_arr	1	4	7	2	5	8	3	6	9

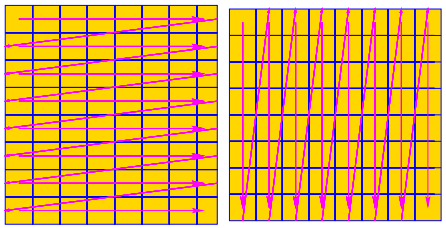
Access to entry a_{ij} of $\mathbf{A} \in \mathbb{K}^{n,m}$, $i = 1,\dots,n$, $j = 1,\dots,m$:

row major:

$a_{ij} \leftrightarrow$ `A_arr(m*(i-1)+(j-1))`

column major:

$a_{ij} \leftrightarrow$ `A_arr(n*(j-1)+(i-1))`



row major column major

△

Example 1.1.2 (Impact of data access patterns on runtime).

Cache hierarchies ⇝ slow access of "remote" memory sites !

column oriented

```
A = randn(n,n);
for j = 1:n-1,
  A(:,j+1) = A(:,j+1) - A(:,j);
end
```

access

$n = 3000 \rightsquigarrow 0.1s$

row oriented

```
A = randn(n);
for i = 1:n-1,
  A(i+1,:) = A(i+1,:) - A(i,:);
end
```

access

$n = 3000 \rightsquigarrow 0.3s$

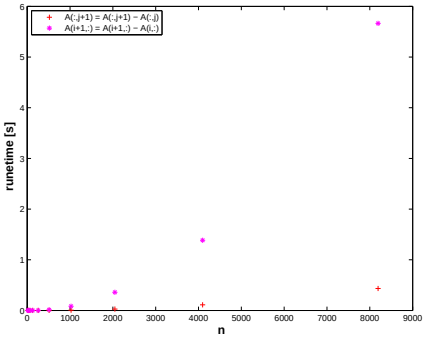
Code 1.1.3: timing for row and column oriented matrix access in MATLAB

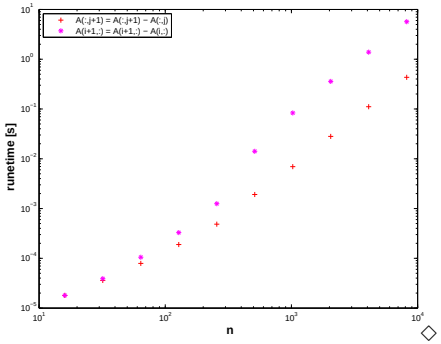
```
1  % Timing for row/column operations
2  K = 3; res = [];
3  for n=2.^(4:13)
4    A = randn(n,n);
5
6    t1 = 1000;
7    for k=1:K,  tic;
8      for j = 1:n-1,  A(:,j+1) = A(:,j+1) - A(:,j); end;
9      t1 = min(toc,t1);
10   end
11   t2 = 1000;
12   for k=1:K,  tic;
13     for i = 1:n-1,  A(i+1,:) = A(i+1,:) - A(i,:); end;
```

```
    t2 = min(toc,t2);
  end
  res = [res; n, t1 , t2];
end

figure; plot(res(:,1),res(:,2),'r+', res(:,1),res(:,3),'m*');
xlabel('{\bf n}','fontsize',14);
ylabel('{\bf runetime [s]}','fontsize',14);
legend('A(:,j+1) = A(:,j+1) - A(:,j)','A(i+1,:) = A(i+1,:) - A(i,:)',...
       'location','northwest');
print -depsc2 '../PICTURES/accessrtlin.eps';

figure; loglog(res(:,1),res(:,2),'r+', res(:,1),res(:,3),'m*');
xlabel('{\bf n}','fontsize',14);
ylabel('{\bf runetime [s]}','fontsize',14);
legend('A(:,j+1) = A(:,j+1) - A(:,j)','A(i+1,:) = A(i+1,:) - A(i,:)',...
       'location','northwest');
print -depsc2 '../PICTURES/accessrtlog.eps';
```





1.2 Complexity/computational effort

complexity/computational effort of an algorithm :⇔ number of elementary operators

additions/multiplications

Crucial: dependence of (worst case) complexity of an algorithm on (integer) problem size parameters (worst case ↔ maximum for all possible data)

Usually studied: asymptotic complexity ≙ "leading order term" of complexity w.r.t *large* problem size parameters

The usual choice of problem size parameters in numerical linear algebra is the number of independent real variables needed to describe the input data (vector length, matrix sizes).

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(*)	$(\mathbf{A}\in\mathbb{R}^{m,n}, \mathbf{B}\in\mathbb{R}^{n,k})\mapsto \mathbf{AB}$	mnk	$mk(n-1)$	$O(mnk)$

✎ notation ("Landau-O"): $f(n)=O(g(n)) \Leftrightarrow \exists C>0, N>0: |f(n)|\le Cg(n)$ for all $n>N$.

Example 1.2.1 (Efficient associative matrix multiplication).

$\mathbf{a}\in\mathbb{K}^m$, $\mathbf{b}\in\mathbb{K}^n$, $\mathbf{x}\in\mathbb{K}^n$:

$$\mathbf{y}=\left(\mathbf{a}\mathbf{b}^T\right)\mathbf{x}\,.$$

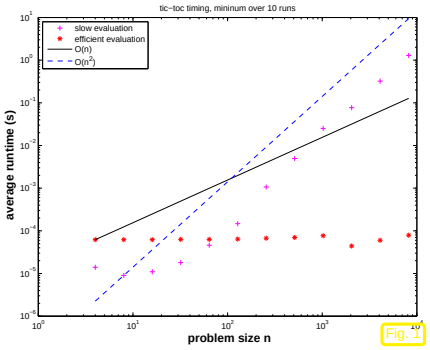
```
T = a*b'; y = T*x;
```

➤ complexity $O(mn)$

$$\mathbf{y}=\mathbf{a}\left(\mathbf{b}^T\mathbf{x}\right)\,.$$

```
t = b'*x; y = a*t;
```

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



◁ average runtimes for efficient/inefficient matrix×vector multiplication with rank-1 matrices (MATLAB `tic-toc` timing)

Platform:

- MATLAB7.4.0.336 (R2007a)
- Genuine Intel(R) CPU T2500 @ 2.00GHz
- Linux 2.6.16.27-0.9-smp

Code 1.2.2: MATLAB code for Ex. 1.2.1

```
function dottenstiming(N,nruns)
% This function compares the runtimes for the
% multiplication of a vector with a rank-1 matrix ab^T, a,b ∈ R^n
% using different associative evaluations

if (nargin < 1), N = 2.^(2:13); end
if (nargin < 2), nruns = 10; end

times = []; % matrix for storing recorded runtimes
```

```
for n=N
  %Initialize dense vectors a, b, x (column vectors!)
  a = (1:n)'; b = (n:-1:1)'; x = rand(n,1);

  %Measuring times using MATLAB tic-toc commands
  tfool = 1000; for i=1:nruns, tic; y = (a*b')*x; tfool =
    min(tfool,toc); end;
  tsmart = 1000; for i=1:nruns, tic; y = a*dot(b',x); tsmart =
    min(tsmart,toc); end;
  times = [times;n, tfool, tsmart];
end

%log-scale plot for investigation of asymptotic complexity
figure('name','dottenstiming');
loglog(times(:,1),times(:,2),'m+',...
       times(:,1),times(:,3),'r*',...
       times(:,1),times(:,1)*times(1,3)/times(1,1),'k-',...
       times(:,1),(times(:,1).^2)*times(2,2)/(times(2,1)^2),'b--');
xlabel('{\bf problem size n}','fontsize',14);
ylabel('{\bf average runtime (s)}','fontsize',14);
title('tic-toc timing, mininum over 10 runs');
legend('slow evaluation','efficient evaluation',...
       'O(n)','O(n^2)','location','northwest');
```

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

◇

Remark 1.2.3 (Reading off complexity).

Available: "Maesurements" $t_i = t_i(n_i)$ for different $n_1, n_2, \ldots, n_N$, $n_i \in \mathbb{N}$

Conjectured: Algebraic dependence $t_i = Cn_i^\alpha$, $\alpha \in \mathbb{R}$

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

If the conjecture holds true, then the points (n_i, t_i) will lie on a *straight line* with *slope* α in a doubly logarithmic plot.

➢ quick "visual test" of conjectured asymptotic complexity

More rigorous: Perform linear regression on $(\log n_i, \log t_i)$, $i = 1, \ldots, N$ (→ Ch. 6)

△

Remark 1.2.4 (Relevance of asymptotic complexity).

Runtimes in Ex. 1.2.1 illustrate that the

> asymptotic complexity of an algorithm need not be closely correlated with its overall runtime on a particular platform,

because on modern computer architectures with multi-level memory hierarchies the *memory access pattern* may be more important for efficiency than the mere number of floating point operations, see [31].

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.

△

1.3 Essential Skills Learned in Chapter 1

You should know:

- how to use vectors and marices in Matlab
- what is the impact of data acces patterns on runtime
- what is the complexity of an algorithm with examples
- the meaning of Landau-O notation