An Introduction to Numerical Linear Algebra

P. de Groen

In these course notes for the course *Numerical Linear Algebra* in the second year bachelor in mathematics we explain the standard algorithms for the solution of a set of linear equations and a linear least squares problem. In order to enhance the understanding of the way algorithms work in practice, we first give an introduction to round-off error analysis. Moreover, we give a mini-tutorial to Matlab, which is an ideal programming and computing environment for experiments with numerical algorithms.
The standard reference for numerical linear algebra is the book
G.H. Golub & C.F. Van Loan, *Matrix Computations*,
The Johns Hopkins University Press, Baltimore, Maryland, USA, 3rd print, 1996.

Contents

1 A mini-tutorial to MATLAB **2**

2 Examples of unstable algorithms **3**
2.a Recursive computation of an exponential integral . . . 3
2.b How to compute the Variance . . . 4

3 Error analysis **5**
3.a Elementary definitions . . . 5
3.b Representation of real numbers and floating-point arithmetic . . . 6
3.c The unavoidable error . . . 7
3.d Examples of round-off error analysis . . . 7
3.e Exercises . . . 10

4 Linear Algebra **12**
4.a notations . . . 12
4.b Exercises . . . 14
4.c The singular value decomposition . . . 15
4.d The Condition Number of a Matrix . . . 17
4.e Exercises . . . 18
4.f Gaussian Elimination . . . 19
4.g The algorithm of Crout . . . 23
4.h Round-off error analysis . . . 24
4.i Exercises . . . 24

5 Linear Least Squares Problems **26**
5.a The normal equations . . . 27
5.b The method of Gram-Schmidt . . . 28
5.c Householder Transformations . . . 31
5.d Givens rotations . . . 35

1 A mini-tutorial to MATLAB

"Matlab" is an interactive computing environment, designed by Cleve Moler, that was started as a demonstration project, in which students easily could experiment with the newly developed computational methods for linear algebra, implemented in the packages LINPACK and EISPACK. The environment was so succesful, that Moler created the company Mathworks around it. It commercialised and extended the design into a very powerful programming and computing environment for solving and simulating mathematical and physical problems and graphical visualisation.

The basic data structure is the matrix. The instruction "`p=5; q=7; A = rand(p,q)`" creates a real matrix with 5 rows en 7 columns (in $\mathbb{R}^{5\times 7}$) consisting of random numbers uniformly distributed on $[0\,,\,1]$. A matrix containing only one column is a column vector, a matrix containing only one row is a row vector and a $1{\times}1$–matrix is identified with one single "real" (or complex) number. Hence, the types "real" and "vector" are not considered as separate data types. The basic "real" is implemented as a standard floating point IEEE 64-bits real and a complex number $z \in \mathbb{C}$ (represented by `''z''`) is implemented as a pair of reals with `u=real(z)` its real and `v=imag(z)` its imaginary parts. The floating point relative accuracy `eps` $= 2\times 10^{-16}$ is a standard variable in Matlab.
Let the matrices $A \in \mathbb{R}^{p\times q}$ and $B \in \mathbb{R}^{r\times s}$ be represented by the names `''A''` and `''B''` and let $\mu \in \mathbb{R}$ be a real represented by the name `''mu''`. Operations with those matrices and vectors follow the usual rules of linear algebra.

- Multiplication by scalars: `mu*A` represents the multiple $\mu A \in \mathbb{R}^{p\times q}$.
- Matrix addition: `A + B` represents the sum $A + B \in \mathbb{R}^{p\times q}$ provided their dimensions are equal, $p = r$ and $q = s$.
- Matrix addition and multiplication: `A * B` represents the product $A\,B \in \mathbb{R}^{p\times s}$ provided the number of columns in A is equal to the number of rows in B, $q = r$.
- Transposition: `A'` represents the transposed matrix A^T; if A is a complex matrix Hermitian transposition (transposition plus complex conjugation) is used.

Example (>> is the matlab-prompt):

```
>> x=[1+i,1-i]
x =
   1.0000 + 1.0000i   1.0000 - 1.0000i
>> x'
ans =
   1.0000 - 1.0000i
   1.0000 + 1.0000i
>> x'*x
ans =
   2.0000                  0 - 2.0000i
        0 + 2.0000i   2.0000
>> x*x'
ans = 4
>>
```

Submatrices can be selected in various ways; e.g., if $A \in \mathbb{C}^{p\times q}$, then
• `real(A)`$\in \mathbb{R}^{p\times q}$ is the real part, and `imag(A)`$\in \mathbb{R}^{p\times q}$ is the imaginary part.
• `A(:,k)` is the k-th column (provided $1 \le k \le q$) and

- `A(1:3,2:2:q)` is a matrix consisting of the elements from the first three rows of A that have even column index.

The command `x = A\b` solves the system of linear equations $A\mathbf{x} = \mathbf{b}$ using the best numerical method available: it uses Gaussian elimination with row pivoting if A is square and well conditioned, it uses a QR decomposition or a Singuar value decomposition if A is either badly conditioned or non-square. Obviously, the dimensions of $\mathbf{b}$ en A have to be compatible.
The whole body of standard matrix en vector routines is available, such as FFT, QR, LU, Cholesky, SVD and eigenvalues/eigenvectors.

The Matlab-primer of Kermit Sigmon can be found on my website,
http://homepages.vub.ac.be/~pdegroen/numeriek/matlab_primer.pdf .
The search on the internet for a 'matlab tutorial' results in a large number of links to very good introductions to the use of matlab, inclusive the tutorials of the "Mathworks" company.

2 Examples of unstable algorithms

2.a Recursive computation of an exponential integral.

Define the integral

$$E_n \;:=\; \int_0^1 x^n e^{x-1} dx \quad \text{for} \quad n = 0,\ 1,\ 2,\ 3,\ \cdots .$$

De value of E_0 is,

$$E_0 = \int_0^1 e^{x-1} dx \;=\; e^{x-1}\Big|_0^1 \;=\; 1 - e^{-1} \;=\; 0.63212055882856\ .$$

For all positive values of n we may use the following recursion, derived by integrating by parts:

$$E_n \;=\; \int_0^1 x^n e^{x-1} dx \;=\; x^n e^{x-1}\Big|_0^1 \;-\; n\int_0^1 x^{n-1} e^{x-1} dx \;=\; 1 \;-\; nE_{n-1}.$$

Forward recursion,

$$E_0 := 1 - e^{-1}\ , \quad E_n := 1 - nE_{n-1} \quad (n = 1,\ 2,\ \cdots),$$

is unstable, as we may infer from the (theoretically impossible) negative value for $n = 18$ in the table below. De reason is, that an error ε in E_{k-1} is ampified to an error $k\varepsilon$ in E_k. Hence, the error in E_{18} is approximately $18! \approx 10^{16}$ times the error in E_0.

The **backward** recursion,

$$\text{choose } E_m = \text{arbitray}\ , \quad E_{n-1} = (1 - E_n)/n \quad (n = m,\ m-1,\ \cdots),$$

is stable. For every starting value E_m it yields the correct value of E_n provided m is sufficiently large with respect to n. This is shown in column 4, where the starting value $E_{18} = 0$ is chosen to be zero; in every (backward) iteration step the error becomes smaller and in E_5 it has disappeared below the rounding error of the entry.

n	forward from $k = 0$	backward from $n = 50$	backward from $n = 18$	difference between columns 3 and 4
0	0.63212055882856	0.63212055882856	0.63212055882856	0.00000000000000
1	0.36787944117144	0.36787944117144	0.36787944117144	0.00000000000000
2	0.26424111765712	0.26424111765712	0.26424111765712	0.00000000000000
3	0.20727664702865	0.20727664702865	0.20727664702865	-0.00000000000000
4	0.17089341188538	0.17089341188538	0.17089341188538	0.00000000000000
5	0.14553294057308	0.14553294057308	0.14553294057308	-0.00000000000000
6	0.12680235656152	0.12680235656153	0.12680235656152	0.00000000000001
7	0.11238350406936	0.11238350406930	0.11238350406934	-0.00000000000004
8	0.10093196744509	0.10093196744559	0.10093196744528	0.00000000000032
9	0.09161229299417	0.09161229298966	0.09161229299250	-0.00000000000284
10	0.08387707005829	0.08387707010339	0.08387707007499	0.00000000002841
11	0.07735222935878	0.07735222886266	0.07735222917515	-0.00000000031248
12	0.07177324769464	0.07177325364803	0.07177324989825	0.00000000374978
13	0.06694777996972	0.06694770257562	0.06694775132275	-0.00000004874714
14	0.06273108042387	0.06273216394138	0.06273148148148	0.00000068245990
15	0.05903379364190	0.05901754087930	0.05902777777778	-0.00001023689848
16	0.05545930172957	0.05571934593124	0.05555555555556	0.00016379037568
17	0.05719187059731	0.05277111916899	0.05555555555556	-0.00278443638656
18	-0.02945367075154	0.05011985495809	0	0.05011985495809

2.b How to compute the Variance

The variance of a series of measurements can be computed by two **Mathematically** equivalent formulae. Given n measurements $\{x_1,\ x_2,\ \cdots\ ,\ x_n\}$ of a physical quantity X, then its mean g and variance S^2 are give by

$$g\ :=\ \frac{1}{n}\sum_{k=1}^{n} x_k\ ,\qquad S_n^2\ :=\ \frac{1}{n-1}\sum_{k=1}^{n}(x_k-g)^2\ =\ \frac{1}{n-1}\left(\sum_{k=1}^{n} x_k^2 - ng^2\right).$$

The second formula is potentially numerically unstable (if $S_n^2 << g^2$) and much more sensitive to small variations in the mean g, as can be seen in the following experiment.

Experiment (using Matlab, `>>` is the matlab prompt)

```
>> format short e
>> RelPerturbG=1e-12
RelPerturbG =
   1.0000e-012
>> n=10000;
>> x=randn(n,1)+1e8*ones(n,1);
>> g=sum(x)/n;
>> sig2=x'*x-n*g*g;
>> sig1=(x-g*ones(size(x)))'*(x-g*ones(size(x)));
>> g=sum(x)/n*(1+RelPerturbG);
>> sig2s=x'*x-n*g*g;
>> sig1s=(x-g*ones(size(x)))'*(x-g*ones(size(x)));
>> Values=[sig1,sig2,sig1s,sig2s]
Values =
```

```
    9.7946e+003 -8.1920e+004 9.7946e+003 -2.0008e+008
>> sprintf(['computed value using formula 1 :                                  %25.15e\ n',...
        'computed value using formula 1 and relative perturbation of g :  %25.15e\ n',...
        'computed value using formula 2 :                                  %25.15e\ n',...
        'computed value using formula 2 and relative perturbation of g :  %25.15e\ n']...
        ,sig1,sig2,sig1s,sig2s)
ans =
computed value using formula 1 :                                  9.794567005712350e+003
computed value using formula 1 and relative perturbation of g :   9.794567105990183e+003
computed value using formula 2 :                                 -8.192000000000000e+004
computed value using formula 2 and relative perturbation of g :  -2.000814080000000e+008
```

By chance the sum of squares computed using formula 2 in this experiment is even negative!

3 Error analysis

3.a Elementary definitions

We are given a real number X and an approximation $\widetilde{X}$ of it. The *absolute* en *relative* errors in the approximation $\widetilde{X}$ are given by:

$$\begin{array}{llll} \text{absolute error in } \widetilde{X}: & F_X := \widetilde{X} - X & \text{such that} & \widetilde{X} = X + F_X\,, \\ \text{relative error in } \widetilde{X}: & f_X := \dfrac{\widetilde{X} - X}{X} & \text{such that} & \widetilde{X} = X(1 + f_X) \quad (\text{provided } X \neq 0). \end{array} \tag{3.1}$$

The concept *"absolute error"* does not have any relation to "absolute values"; we use absolute as opposed to relative. The absolute error has the same dimensions (e.g. length, weight, time) as X has, while the relative error is dimensionless.

Exercise 1: Show that the absolute and relative errors in the quantities $\widetilde{X}$ en $\widetilde{Y}$ satisfy:

$$F_{X+Y} = F_X + F_Y \qquad \text{and} \qquad f_{X*Y} = f_X + f_Y + f_X f_Y\,.$$

When we know the (absolute or relative) error in a quantity $\widetilde{X}$ (as the result of a measurement or a computation), *then we also know the quantity exactly!* Unfortunately, this (almost) never happens; in general we do not know more than an *upper bound* on the absolute value of the error. In ordinary language we are used to talk about the *"error"* in a quantity, meaning *"an upper bound for such an error"*. Thus, for a given approximation $\widetilde{X}$ of a quantity X we define:

$$\begin{array}{lll} \Delta_X \text{ is (an upper bound) for the absolute error in } \widetilde{X} & \text{if} & |\,\widetilde{X} - X\,| \leq \Delta_X\,, \\ \delta_X \text{ is (an upper bound) for the relative error in } \widetilde{X} & \text{if} & \left|\dfrac{\widetilde{X} - X}{X}\right| \leq \delta_X\,. \end{array} \tag{3.2}$$

Exercise 2: Prove the following rules for the computation of "the errors" in the sum and the product of $\widetilde{X}$ and $\widetilde{Y}$:

$$\begin{array}{ll} \Delta_{X\pm Y} \leq \Delta_X + \Delta_Y\,, & \Delta_{XY} \leq |Y|\Delta_X + |X|\Delta_Y + \Delta_X\Delta_Y\,, \\ \delta_{X\pm Y} \leq \dfrac{|X|\delta_X + |Y|\delta_Y}{|X \pm Y|}\,, & \delta_{XY} \leq \delta_X + \delta_Y + \delta_X\delta_Y\,. \end{array} \tag{3.3}$$

Remark. You should read these lines as: If Δ_X and Δ_Y are upper bounds for the errors in X and Y respectively, then there is an upper bound $\Delta_{X\pm Y}$ for the error in $X \pm Y$ satisfying $\Delta_{X\pm Y} \leq \Delta_X + \Delta_Y$. This implies that $\Delta_X + \Delta_Y$ is an upper bound for the error in $X \pm Y$.
Find the corresponding rules for the computation of (upper bounds on) the absolute and relative errors in the quotient X/Y.

3.b Representation of real numbers and floating-point arithmetic

Real numbers generally are stored in a computer in "floating point" format as the product of a *mantissa* times an *exponent*. This implies a large dynamical range for those numbers. Given a base[1] β, a real number $x \in I\!R$ can be represented by a pair (m, e) satisfying

$$x = m \cdot \beta^e\,, \tag{3.4}$$

where m is the *mantissa* is and e the exponent. Since the pair $(m\cdot\beta\,,\, e-1)$ represents the same number, we may normalise the mantissa imposing a condition like $1/\beta \leq |\,m\,| < 1$. Obviously, the number of bits used in the representation must be finite. In the IEEE-standard for 64-bits *REAL*s a binary representation ($\beta = 2$) is chosen with 53 bits for the absolute value van mantisse, 10 bits for the absolute value of the exponent and 2 sign bits. Because the first bit of a normalised mantissa is always equal to 1 (why??), this first bit need not be stored. Because only 10 bits are used for the exponent, numbers whose exponent is larger than 2^{10} or smaller than 2^{-10} cannot be stored. Hence, only numbers with absolute values between 10^{-300} and 10^{300} (approximately) can be represented. If the result of an arithmetical operation ($+\,,\,-\,,\,\times\,,\,/$) is smaller or larger, it is called "underflow" and "overflow" respectively. Unless otherwise specified, the result of underflow is set to zero and the result of overflow to NaN (not a number). A further operation with a NaN results in a system error.

A real number x within the range $10^{-300} \leq |\,x\,| \leq 10^{300}$ most often cannot be represented exactly (only certain rationals can). It has to be approximated by a rational number that fits within the representation system, called a "**machine number**". Usually, the nearest machine number is chosen. It is denoted by $\mathbf{fl}(x)$. The difference $x - \mathbf{fl}(x)$ is the "round-off error".

Theorem. If in a processor the base β is chosen for the representation of numbers and if a mantissa carries t digits in that base, then the *relative* rounding error satisfies the inequality (over- and underflow omitted):

$$\left|\frac{x-\mathbf{fl}(x)}{x}\right| \leq \eta \quad \text{but also} \quad \left|\frac{x-\mathbf{fl}(x)}{\mathbf{fl}(x)}\right| \leq \eta \quad \text{met} \quad \eta := \tfrac{1}{2}\beta^{1-t}\,. \tag{3.5}$$

The symbol η denotes the *machineprecision.*

Exercise 3: Prove this theorem.
Prove also, that for every arithmetical operation $\odot \in \{+,-,\times,/\}$ involving two machine numbers x and y (except for over- and underflow) there exist real numbers ε_1 and ε_2, that satisfy (exactly)

$$\mathbf{fl}(x \odot y) = (x \odot y)(1+\varepsilon_1) = \frac{x \odot y}{1+\varepsilon_2} \quad \text{met} \quad |\,\varepsilon_1\,| \leq \eta \quad \text{and} \quad |\,\varepsilon_2\,| \leq \eta\,. \tag{3.6}$$

Remark. Check that η also can be defined as the largest real number such that $\mathbf{fl}(1+\eta) = 1$!

Exercise 4: the power series for the exponential function is: $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$.

- How many terms of the series are needed to compute e^{-5} with a relative error smaller than 10^{-3}?
- Is this possible using a (decimal) calculator (computer), where real numbers are stored in decimal format

[1] The standard value today is $\beta = 2$, but in earlier times other values have been used, such as $\beta = 8$ (CDC) and $\beta = 16$ (IBM).

with a mantissa of 4 digits? Why?
• Is there a way to circumvent the problems due to small mantissa length in the computation of e^{-5} using such a computer?

3.c The unavoidable error

Let us consider the problem to compute the value of $y := f(x)$ of a given smooth (C^2 at least) real function f for some value of the (real) argument x. A priori, we know that all computations have to be executed within the rounding environment of our computer. Hence, we know in advance, that the argument has to be converted to a (binary) 'machine number', and that we unavoidably start making an error in computing $\widetilde{y} = f(x + \xi)$ by changing the argument to the rounded value $x + \xi$ with $|\,\xi/x\,| \le \eta$. Even disregarding all other sources of errors that can arise by an implementation of an algorithm for computing the value of $f(x)$, this may cause an error in the computed value. Using a Taylor expansion we find

$$\widetilde{y} = f(x+\xi) = f(x) + \xi\, f'(x) + O(\xi^2) \quad \text{such that} \quad \widetilde{y} - y \approx \xi\, f'(x)\,.$$

We can estimate the relative error due to the rounding of the argument by

$$\frac{\widetilde{y}-y}{y} \approx \frac{\xi}{x}\,\frac{x\,f'(x)}{f(x)} \quad \text{and approximately} \quad \left|\frac{\widetilde{y}-y}{y}\right| \le C\,\eta \quad \text{where} \quad C := \left|\frac{x\,f'(x)}{f(x)}\right|\,. \tag{3.7}$$

The error in the argument is multiplied by the factor C. Generally it is called the "**condition number**" of the problem.
Because we want to read the result by our human eye, we want to convert it back to decimal $\widehat{y} = \widetilde{y}\,(1+\vartheta)$ making another relative error $|\,\vartheta\,| \le \eta$ in the result. We conclude, that in any case a relative error bounded by $(y - \widehat{y})/y \le C\eta + \eta$ may be expected not depending on the way f is computed. We call this the "**unavoidable error"**.

3.d Examples of round-off error analysis

Task: given a (real) function φ, compute the value $x = \varphi(a)$.

Using an *algorithm* for the computation $\varphi(a)$ we find a *computed* value $\mathbf{fl}(x)$ possibly corrupted by round-off errors.

In an error analysis we try to find (or at least, estimate) errors δ_x, δ_a or ε_a and ε_x such that

$$\begin{aligned} \mathbf{fl}(x) &= x + \delta_x && \text{forward error analysis} \\ &= \varphi(a + \delta_a) && \text{backward error analysis} \\ &= \varphi(a + \varepsilon_a) + \varepsilon_x && \text{mixed error analysis} \end{aligned}$$

Definition: An *algorithm* is called **numerically stable** if we can prove:
δ_x or ε_x is of the same order of magnitude as the unavoidable error is,
δ_a of ε_a is of the same order of magnitude as the machine precision is.

Example 1: For given reals a and b there is an ε satisfying $|\,\varepsilon\,| \le \eta\,(=\text{machine precision})$ such that

$$\mathbf{fl}(a + b) = \begin{cases} a + b + \varepsilon\,(a+b) & \text{forward} \\ \widetilde{a} + \widetilde{b} \quad \text{with} \quad \widetilde{a} := a(1+\varepsilon) \quad \text{and} \quad \widetilde{b} := b(1+\varepsilon) & \text{backward} \end{cases}$$

In the forward line the error $\varepsilon\,(a+b)$ is considered as deviation from the result and in the backward line the errors $\varepsilon\,a$ and $\varepsilon\,b$ are considered as deviations of the arguments.

Example 2: There are numbers ε_1 and ε_2 (satisfying $|\,\varepsilon_i\,| \le \eta$) such that

$$\begin{aligned} \mathbf{fl}(1 - x^2) &= (1 - x * x * (1+\varepsilon_1)) * (1+\varepsilon_2) \\ &= (1 - \widetilde{x}^2)\,(1+\varepsilon_2) \quad \text{with} \quad \widetilde{x} := x\sqrt{1+\varepsilon_1} \qquad \text{mixed.} \end{aligned}$$

The round-off error is in part attributed to the argument $\widetilde{x}$ and in part to the result.

Example 3: Estimate the round-off error in the computed value of the positive root of the quadratic equation

$$a - 2x - c\,x^2 \quad \text{with} \quad a \geq 0 \quad \text{and} \quad c \geq 0$$

using the formula

$$x := \frac{-1 + \sqrt{1 + a\,c}}{c}$$

and assuming that the round-off error in the computed value of a square root satisfies the estimate

$$\mathbf{fl}(\sqrt{x}) = \sqrt{x}\,(1 + \varepsilon_x) \quad \text{with} \quad |\,\varepsilon_x\,| \leq \eta \quad \text{for all } x\,.$$

Answer: There exist numbers ε_1 , ε_2 and ε_3 with $|\,\varepsilon_i\,| \leq \eta$, such that

$$\begin{aligned}
\mathbf{fl}(\sqrt{1 + a\,c}) &= \sqrt{(1 + a\,c\,(1 + \varepsilon_1))\,(1 + \varepsilon_2)}\,(1 + \varepsilon_3) \\
&= \sqrt{1 + \widetilde{a}\,c}\,(1 + \xi_1) \quad \text{with} \quad \xi_1 := \sqrt{1 + \varepsilon_2}\,(1 + \varepsilon_3) - 1 \quad \text{and} \quad \widetilde{a} := a\,(1 + \varepsilon_1)
\end{aligned}$$

As a consequence, there are numbers ξ_2 en ξ_3, $(\,|\,\xi_i\,| \leq \eta\,)$ such that:

$$\begin{aligned}
\mathbf{fl}(x) &= \frac{-1 + \sqrt{1 + \widetilde{a}\,c}\,(1 + \xi_1)}{c}(1 + \xi_2)\,(1 + \xi_3) \\
&= \frac{-1 + \sqrt{1 + \widetilde{a}\,c}}{c}\,(1 + \xi_2)\,(1 + \xi_3) + \frac{\sqrt{1 + \widetilde{a}\,c}}{c}\,\xi_1\,(1 + \xi_2)\,(1 + \xi_3)
\end{aligned}$$

The second term may be large in comparison to x if $|\,ac\,| \ll 1$ and in that case the formula is not numerically stable and should be avoided.

An *alternative (numerically stable) algorithm* for this root is:

$$x := \frac{a}{1 + \sqrt{1 + a\,c}}\,.$$

Example 4: Round-off error in the computed value of an inner product

$S := \sum_{i=1}^{n} x_i\,y_i$ to be computed by the algorithm:

$$\left[\begin{array}{l} S := 0; \\ \textbf{for } i := 1 \textbf{ to } n \textbf{ do } S := S + x_i * y_i \end{array}\right.$$

For the *computed* value of S numbers ξ_i and η_i exist with $|\,\xi_i\,|, |\,\varepsilon_i\,| \leq \eta$, $i = 1 \cdots n$ such that:

$$\begin{aligned}
\mathbf{fl}(S) = \; & x_1\,y_1\,(1 + \xi_1)\,(1 + \varepsilon_2) \cdots (1 + \varepsilon_n) \\
& + x_2\,y_2\,(1 + \xi_2)\,(1 + \varepsilon_2) \cdots (1 + \varepsilon_n) \\
& + \cdots \\
& + x_{n-2}\,y_{n-2}\,(1 + \xi_{n-2})\,(1 + \varepsilon_{n-2}) \cdots (1 + \varepsilon_n) \\
& + x_{n-1}\,y_{n-1}\,(1 + \xi_{n-1})\,(1 + \varepsilon_{n-1})\,(1 + \varepsilon_n) \\
& + x_n\,y_n\,(1 + \xi_n)\,(1 + \varepsilon_n)\,.
\end{aligned}$$

Hence,

$$S - \mathbf{fl}(S) = \sum_{i=1}^{n} x_i\,y_i\,\zeta_i$$

where

$$\zeta_i := 1 - (1 + \xi_i)\,(1 + \varepsilon_i) \cdots (1 + \varepsilon_n) \quad \text{and} \quad |\,\zeta_i\,| \leq (n - i + 2)\,\eta \quad \text{provided} \quad n\eta \leq 0.1\,.$$

As a consequence, the *forward* error satisfies:

$$\left|\frac{S - \mathbf{fl}(S)}{S}\right| \le \frac{(n+1)\,\eta}{|\,S\,|}\sum_{i=1}^{n}|\,x_i\,y_i\,| \le (n+1)\,\eta\,\frac{\|\,\mathbf{x}\,\|_2\,\|\,\mathbf{y}\,\|_2}{|\,\mathbf{x}^T\mathbf{y}\,|} \tag{3.8}$$

Example 5: Compute x_n from the equation

$$a = \sum_{i=1}^{n} x_i\,y_i\,, \qquad a,\, x_1\ \cdots\ x_{n-1}\,,\, y_1\ \cdots\ y_n \quad \text{given,}$$

and estimate the round-off error in the computed value of x_n.

algorithm:
- $S := a$;
- **for** $i := 1$ **to** $n-1$ *DO* $S := S - x_i * y_i$;
- $x_n := S\,/\,y_n$

For the **computed** value of S and x_n numbers ξ_i en η_i exist, satisfying $|\,\xi_i\,|$, $|\,\varepsilon_i\,| \le \eta$, such that:

$$\begin{aligned}
\mathbf{fl}(S) \ &=\ a\,(1+\varepsilon_1)\ \cdots\ (1+\varepsilon_{n-1})\\
&\quad -\ x_1\,y_1\,(1+\xi_1)\,(1+\varepsilon_1)\ \cdots\ (1+\varepsilon_{n-1})\\
&\quad -\ x_2\,y_2\,(1+\xi_2)\,(1+\varepsilon_2)\ \cdots\ (1+\varepsilon_{n-1})\\
&\quad -\ \cdots\\
&\quad -\ x_{n-2}\,y_{n-2}\,(1+\xi_{n-2})\,(1+\varepsilon_{n-2})\,(1+\varepsilon_{n-1})\\
&\quad -\ x_{n-1}\,y_{n-1}\,(1+\xi_{n-1})\,(1+\varepsilon_{n-1})
\end{aligned}$$

and

$$\widetilde{x}_n := \mathbf{fl}(x_n) = \mathbf{fl}(S)\,/\,(\,y_n\,(1+\xi_n)\,)$$

Division by $(1+\varepsilon_1)\ \cdots\ (1+\varepsilon_{n-1})$ yields the *backward* error estimate:

$$\begin{aligned}
a \ &=\ x_1\,y_1\,(1+\xi_1) + x_2\,y_2\,\frac{1+\xi_2}{1+\varepsilon_1} + \cdots\\
&\quad +\ x_{n-1}\,y_{n-1}\,\frac{1+\xi_{n-1}}{(1+\varepsilon_1)\ \cdots\ (1+\varepsilon_{n-2})}\\
&\quad +\ \widetilde{x}_n\,y_n\,\frac{1+\xi_n}{(1+\varepsilon_1)\ \cdots\ (1+\varepsilon_{n-1})}\\
&=\ \sum_{i=1}^{n-1} x_i\,y_i\,(1+\delta_i) + \widetilde{x}_n\,y_n\,(1+\delta_n)
\end{aligned}$$

where

$$\delta_i := \frac{1+\xi_i}{(1+\varepsilon_1)\ \cdots\ (1+\varepsilon_{i-1})} - 1\,, \quad \text{satisfying} \quad |\,\delta_i\,| \le (i+1)\,\eta \quad \text{if} \quad n\,\eta < 0.1\,.$$

Conclusion: The *computed* value $\widetilde{x}_n$ is the solution of the *neighbouring* equation

$$a = \sum_{j=1}^{n} x_j\,\widetilde{y}_j\,, \qquad \widetilde{y}_j := y_j\,(1+\delta_j)\,. \tag{3.9}$$

Example 6, an error estimate for E_n. In section 2a we considered the recursion:

$$E_n = 1 - n\,E_{n-1}\,. \tag{3.10}$$

Let $\widetilde{E}_n := \mathbf{fl}(E_n)$ be the computed value of E_n, then there exist numbers ξ_n en ζ_n satisfying:

$$\widetilde{E}_n = \mathbf{fl}(1 - \mathbf{fl}(n\,\widetilde{E}_{n-1})) = (1 - n\,\widetilde{E}_{n-1}(1+\xi_n))/(1+\zeta_n)\,, \qquad |\,\xi_n\,| \le \eta \ \text{ en } \ |\,\zeta_n\,| \le \eta\,, \tag{3.11}$$

written in a different way,

$$\widetilde{E}_n + \zeta_n \widetilde{E}_n = 1 - n\,\widetilde{E}_{n-1} - n\,\xi_n\,\widetilde{E}_{n-1}\,. \tag{3.12}$$

Subtracting (3.10) we find a recursion for the errors:

$$\widetilde{E}_n - E_n = -n(\widetilde{E}_{n-1} - E_{n-1}) - \zeta_n \widetilde{E}_n - n\,\xi_n\,\widetilde{E}_{n-1}\,. \tag{3.13}$$

Defining $F_n := \widetilde{E}_n - E_n$ and $\delta_n := -\zeta_n \widetilde{E}_n - n\,\xi_n\,\widetilde{E}_{n-1}$ we find the recursion

$$F_n = -n\,F_{n-1} + \delta_n\,, \quad F_0 = \mathbf{fl}(E_0) - E_0\,, \quad |\,F_0\,| \le \eta\,E_0 \le \eta. \tag{3.14}$$

Since $E_n > 0$ for all n, equation (3.10) implies, that $E_{n-1} \le 1/n$; this should be true also for $\widetilde{E}_{n-1}$ as long as it is a reasonable approximation of E_n is. Under this condition we have $|\,\delta_n\,| \le 2\eta$ and F_n satisfies in that case the inequality

$$|\,F_n\,| \le n\,|\,F_{n-1}\,| + 2\eta\,. \tag{3.15}$$

Hence, there is a majorizing sequence $\{\widehat{F}_n\}$ such that

$$|\,F_n\,| \le \widehat{F}_n \quad \text{with } \widehat{F}_n = n\,\widehat{F}_{n-1} + 2\eta\,, \quad \widehat{F}_0 = \eta\,. \tag{3.16}$$

The recursion for $\widehat{F}_n$ gives an **a priori** upper bound for the error in the computed value of E_n:

$$|\,\widetilde{E}_n - E_n\,| = F_n \le \widehat{F}_n = n!\,\eta\left(1 + \frac{2}{1!} + \frac{2}{2!} + \cdots + \frac{2}{n!}\right) \le n!\,\eta\,(2\,e^1 - 1)\,. \tag{3.17}$$

If $n = 18$, this upper bound is already much larger than 1, such that it is likely that the condition $|\,\widetilde{E}_n\,| \le 1$ is no longer satisfied.

A better upper bound can be obtained by computing together with $\widetilde{E}_n$ a (tight) upper bound for the round-off error in it. Since (3.13) implies

$$|\,F_n\,| \le n\,|\,F_{n-1}\,| + \eta\,|\,\widetilde{E}_n\,| + n\,\eta\,|\,\widetilde{E}_{n-1}\,|\,, \tag{3.18}$$

we can compute a **running error estimate** $\widetilde{F}_n$ in the recursion together with $\widetilde{E}_n$

$$\widetilde{F}_n = n\,\widetilde{F}_n + \eta\,|\,\widetilde{E}_n\,| + n\,\eta\,|\,\widetilde{E}_{n-1}\,|\,. \tag{3.19}$$

This provides for each n a relatively good **a posteriori** upper bound for the absolute error in the computed value of E_n in algorithm (3.10). We remark that this (smaller) upper bound can be obtained only after the actual computations, because it takes into account the actual round-off errors.

3.e Exercises

1. Rewrite the following expressions in a numerically stable form

$$\frac{1}{1+2x} - \frac{1-x}{1+x} \quad \text{voor} \quad |\,x\,| \ll 1 \tag{3.20}$$

$$\sqrt{x + \frac{1}{x}} - \sqrt{x - \frac{1}{x}} \quad \text{voor} \quad |\,x\,| \gg 1 \tag{3.21}$$

$$\frac{1 - \cos x}{x} \quad \text{voor} \quad |\,x\,| \ll 1 \tag{3.22}$$

2. If a routine is available for computing the inverse sine function $x \mapsto \arcsin(x)$ in a numerically stable way, we can evaluate the arctan (inverse tangent) function using the relation

$$\arctan x = \arcsin \frac{x}{\sqrt{1+x^2}}\,. \tag{3.23}$$

Estimate the relative error in the result, assuming that the sqrt and arcsin functions return an approximation with good relative accuracy. For what values of x this method is reliable?

3. Let f be a sufficiently smooth function (e.g. $f(x) = sin(x)$) satisfying

$$\max_x |f'''(x)| \le M\,.$$

The derivative of f in x can be approximated by the central difference

$$D_h f(x) := \frac{f(x+h) - f(x-h))}{2h}\,.$$

a. Show, that the cut-off error in $D_h f$ satiesfies:

$$\frac{f(x+h) - f(x-h)}{2h} = f'(x) + \frac{h^2}{6} f'''(x+\vartheta h) \quad \text{for some} \quad |\vartheta| \le 1\,. \tag{3.24}$$

b. Assume that a routine for the computation of f is available, that returns for every x a result with a relative error smaller than 2η. Find a (good) upper bound for the relative error in the computed value of $D_h f$ as a function of h and sketch the graph of the total error (cut-off plus round-off errors) in the computed approximation of the derivative $f'(x)$ as a function of h (i.e. sketch a graph for an upper bound of $|\{f'(x) - \mathbf{fl}(D_h f(x))\}/f'(x)|$ as a function of h).

4. We may represent a polynomial P of degree n by a sum or a product,

$$P(x) := \sum_{k=0}^{n} a_k\, x^{n-k} \qquad \text{or} \qquad P(x) := a_0 \prod_{k=1}^{n} (x - x_k) \quad (\text{with} \;\; a_0 \ne 0\,),$$

with coefficients $a_0\,,\; a_1\,,\; \cdots\,,\; a_n$ or (complex) zeros $x_1\,,\; x_2\,,\; \cdots\,,\; x_n$ respectively and with non-zero leading coefficient $a_0 \ne 0$. In the first case, the best way to compute the value of the polynomial for a given argument ξ is the algorithm of Horner:

$$b_0 := a_0\,; \;\; \textbf{for} \;\; k := 1 \;\; \textbf{to} \;\; n \;\; \textbf{do} \;\; b_k := b_{k-1} * \xi + a_k \;\; \textbf{end}\,, \tag{3.25}$$

resulting in $P(\xi) = b_n$. The value D of the derivative $P'(\xi)$ can be computed in the same loop,

$$D := 0\,; \;\; P := a_0\,; \;\; \textbf{for} \;\; k := 1 \;\; \textbf{to} \;\; n \;\; \textbf{do} \;\; D := D * \xi + P\,; \;\; P := P * \xi + a_k \;\; \textbf{end}\,.$$

a. Prove correctness of algorithm (3.25).

b. Show that the coefficients $b_0\,,\; \cdots\,,\; b_{n-1}$ computed in (3.25) satisfy

$$P(x) := b_n + (x - \xi) \sum_{k=0}^{n-1} b_k\, x^{n-k} \tag{3.26}$$

such that $b_n = 0$ implies that ξ is a zero of the polynomial and vice versa. This implies that Horner's scheme computes the coefficients of the deflated polynomial $P(x)/(x - \xi)$ of degree $n-1$ if ξ is a zero of P (synthetic division).

c. Show that numbers δ_k exist, such that the value $\mathbf{fl}(P(x))$ computed by Horner's scheme is equal to the exact value of a neighbouring polynomial,

$$\mathbf{fl}(P(x)) = \sum_{k=0}^{n} \tilde{a}_k\, x^{n-k} \quad \text{met} \quad \tilde{a}_k := a_k\,(1+\delta_k) \quad \text{and} \quad |\,\delta_k\,| \le (2n-2k+1)\eta + O(\eta^2)\,.$$

d. Show that the following algorithm,

$P := a_0\,;\ d := 0;$ **for** $k := 1$ **to** n **do** $d := d + |\,P\,|\,;\ P := P * x + a_k\,;\ d := d * |\,x\,| + |\,P\,|$ **end**

computes together with the value of the polynomial a "running error estimate" d that satisfies *after termination*:

$$|\,\mathbf{fl}(P(x)) - P(x)\,| \le d\,\eta\,.$$

5. The standard deviation S of a set of measurements $\{x_1 \cdots x_n\}$ can be computed in two mathematically equivalent ways:

$$S^2 = \frac{1}{n-1}\,(\sum_{i=1}^{n} x_i^2 \; - n\,g^2\;) \qquad \text{and} \qquad S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - g)^2$$

where g is the mean:

$$g := \frac{1}{n}\sum_{i=1}^{n} x_i\,.$$

Which of both formulae should be preferred in numerical computations and why?
Apply to the computations of g and S^2 an error analysis analogous to those in the previous section.

4 Linear Algebra

4.a notations

The theory in linear algebra and their proofs can be be formulated quite elegantly in terms of an abstract vector space E of dimension n over the field of real or complex numbers. However, for actual computations we always have to choose a basis and we have to represent vectors and matrices as a set of numbers with respect to this basis. So, we will work always with the vector spaces $\mathbb{R}^n$ or $\mathbb{C}^n$ in which a vector is a column of n numbers and where a (linear) transformation is a matrix, an array of $m \times n$ numbers in $\mathbb{R}^{m\times n}$ or $\mathbb{C}^{m\times n}$.

• A <u>vector</u> $\mathbf{x} \in \mathbb{R}^n$ is a column of n real (or complex) numbers,

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad \text{with components} \quad x_1\,,\,\cdots\,,\,x_n\,. \tag{4.1}$$

In print we use the boldface type $\mathbf{x}$ and in manuscript we use underlining $\underline{x}$; we denote its components by the italic type of the same letter plus a subscript x_k.

• For a <u>matrix</u> $A \in \mathbb{R}^{m\times n}$ we always use a (italic) capital letter. The matrix elements a_{ij} are denoted by the corresponding minuscule with two indices. The columns of a matrix are vectors in $\mathbb{R}^m$, denoted by boldface minuscules with one index; their span is the image (sub)space $Im(A)$:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} = (\mathbf{a}_1\,|\,\cdots\,|\,\mathbf{a}_n)\;, \quad \text{such that} \quad \mathbf{a}_k = \begin{pmatrix} a_{1k} \\ \vdots \\ a_{mk} \end{pmatrix}\,. \tag{4.2}$$

The corresponding notation in matlab is: if `A` is a matrix, then the vector `A(:,k)` is its k–th column.
As is usual in Matlab, a vector is identified with a matrix consisting of **one** column.

• A matrix $A \in \mathbb{R}^{m\times n}$ and a vector $\mathbf{x} \in \mathbb{R}^n$ can be partitioned as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad \mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} \quad \text{such that} \quad A\,\mathbf{x} = \begin{pmatrix} A_{11}\mathbf{x}_1 + A_{12}\mathbf{x}_2 \\ A_{21}\mathbf{x}_1 + A_{22}\mathbf{x}_2 \end{pmatrix} \tag{4.3}$$

provided the dimensions match:

$$A_{11} \in \mathbb{R}^{p\times r}\,,\ A_{12} \in \mathbb{R}^{q\times r}\,, A_{21} \in \mathbb{R}^{p\times s}\,,\ A_{22} \in \mathbb{R}^{q\times s}$$

$$\mathbf{u} \in \mathbb{R}^p\,,\ \mathbf{v} \in \mathbb{R}^q\,,\ p+q=n \ \text{ and } \ r+s=m$$

The corresponding notation in matlab works as follows: if A is a matrix, then the part A_{22} is selected by the statement `B=A(p+1:n , r+1:m)` . Remember that the indices in B are shifted, such that `B(1,1)=A(p+1,r+1)`, etc.

• The transposed of a matrix A is denoted by A^T; for complex matrices we have ordinary transposition (denoted by A^T) and complex or Hermitian transposition (denoted by A^H). In the latter all elements are transposed and complex conjugated. In matlab the accent `A'` means Hermitian transposition.

• The norm of a vector $\mathbf{x} \in \mathbb{R}^n$ is denoted by $\|\,\mathbf{x}\,\|$. As is well known, all norms in a vector space of finite dimension are equivalent (*why?*). In this course we shall use only three vector norms, the Euclidian norm $\|\cdot\|_2$ (or ℓ^2-norm), the max-norm $\|\cdot\|_\infty$ (or ℓ^∞-norm) and the 1–norm $\|\cdot\|_1$ (ℓ^1–norm or dual of the max-norm):

$$\|\,\mathbf{x}\,\|_1 := \sum_{j=1}^{n} |\,x_j\,|\ , \quad \|\,\mathbf{x}\,\|_2^2 := \sum_{j=1}^{n} |\,x_j\,|^2 \quad \text{and} \quad \|\,\mathbf{x}\,\|_\infty := \max_j\ |\,x_j\,| \tag{4.4}$$

The Euclidian norm is derived from an inner product:

$$\text{If} \ \ \mathbf{u}\,,\ \mathbf{v} \in \mathbb{C}^m\,, \ \ \text{then} \ \ \langle \mathbf{u}\,,\,\mathbf{v}\rangle := \mathbf{u}^H\mathbf{v} = \sum_{j=1}^{n} \overline{u}_j\, v_j\,. \tag{4.5}$$

Since $\mathbf{u}^H$ is a row-vector and is identified with an $1\times n$–matrix, we may identify the inner product (4.5) with the matrix–matrix product $\mathbf{u}^H\mathbf{v}$ (or $\mathbf{u}^T\mathbf{v}$ for real vectors) and with `u'*v` in matlab.
The vectors $\mathbf{u}\,,\ \mathbf{v}$ are called orthogonal if $\mathbf{u}^T\,\mathbf{v} = 0$.

• For a matrix $A \in \mathbb{R}^{m\times n}$ the matrixnorm $A \mapsto \|\,A\,\|$, associated (subordinate) to the vector norm $\mathbf{x} \mapsto \|\,\mathbf{x}\,\|$, is defined by

$$\|\,A\,\| := \max_{\mathbf{x}\,\in\,\mathbb{R}^n,\ \|\mathbf{x}\| \neq 0} \frac{\|\,A\,\mathbf{x}\,\|}{\|\mathbf{x}\|} = \max_{\mathbf{x}\,\in\,\mathbb{R}^n,\ \|\mathbf{x}\| = 1} \|\,A\,\mathbf{x}\,\| \tag{4.6}$$

In the numerator we see a vectornorm in $\mathbb{R}^m$ and in the denominator a vectornorm in $\mathbb{R}^n$. A matrix norm defined in this way is often called a *lub*–norm (*lub* is derived from 'least upper bound'). Check, that a *lub*–norm not only satisfies all requirements for a norm, but also satisfies the product (or algebra) property $\|\,A\,B\,\| \le \|\,A\,\|\,\|\,B\,\|$.
A *lub*–norm generally is denoted by the same symbol $\|\cdot\|_1$, $\|\cdot\|_2$ or $\|\cdot\|_\infty$ as is the vector norm to which it is subordinate.

• The Frobenius–norm of a matrix $A \in \mathbb{R}^{m\times n}$ is defined by

$$\|\,A\,\|_F^2 := \sum_{i=1}^{m}\sum_{j=1}^{n} |\,a_{ij}\,|^2\,. \tag{4.7}$$

This norm satisfies the product property $\| A\,B \|_F \le \| A \|_F \, \| B \|_F$ but it is not a *lub*–norm. In fact it is the Euclidian norm of the matrix considered as an element of an $m \times n$–dimensional vector space.

• In Matlab those vector and matrix norms of an object `a` are computed by the function `norm(a,p)`, where `p` stands for one of the symbols "1", "2", "`inf`" or "`'fro'`" (in the last one the quotes are mandatory!).

• A square (real) matrix $A \in I\!R^{n\times n}$ is <u>orthogonal</u> if $A^T A = I$, the identity in $I\!R^n$; a (complex) matrix $A \in \mathbb{C}^{n\times n}$ is <u>unitary</u> if $A^H A = I$. Check that these definitions imply: $A\,A^T = I$ and $A\,A^H = I$ respectively.
If $A \in I\!R^{m\times n}$ with $m > n$ and $A^T A = I$, then the columns of A are <u>orthonormal</u> and A is called a <u>partial</u> <u>isometry</u>.

• A <u>diagonal</u> <u>matrix</u> $D \in I\!R^{m\times n}$ is a matrix whose elements outside the main diagonal are zero, i.e. $D = (d_{ij})$ with $d_{ij} = 0$ if $i \ne j$.
For a vector $\mathbf{a} \in I\!R^n$ we define the diagonal matrix $D := diag(\mathbf{a}) \in I\!R^{m\times n}$ with $m \ge n$ by $d_{ii} = a_i$ and $d_{ij} = 0$ if $i \ne j$; we assume $m = n$, unless it is clear from the context that m should be larger.
The matlab function `diag` constructs from a vector a square matrix with the elements of this vector on the main diagonal. The application of this function to an $m\times n$ matrix ($m > 1$ and $n > 1$) extracts the main diagonal and delivers it as a vector of length `min(m,n)`.

4.b Exercises

1. Prove the following identities for the subordinate matrix norms:
$$\|A\|_1 = \max_j \sum_{i=1}^n |\, a_{ij} \,| \,, \qquad \|A\|_\infty = \max_i \sum_{j=1}^n |\, a_{ij} \,| \qquad \text{and} \qquad \|A\|_2 = \max_{\mathbf{x}\ne 0, \mathbf{y}\ne 0} \frac{|\,(A\mathbf{x}, \mathbf{y})\,|}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2} \,,$$
where $(\mathbf{x}, \mathbf{y}) := \sum_{i=1}^n x_i y_i$.

2. Prove the following inequalities for any $\mathbf{x} \in I\!R^n$ and any $A \in I\!R^{n\times n}$:
$$1)\quad \|\mathbf{x}\|_2 \le \|\mathbf{x}\|_1 \le \sqrt{n}\|\mathbf{x}\|_2 \qquad\qquad 2)\quad \|\mathbf{x}\|_\infty \le \|\mathbf{x}\|_2 \le \sqrt{n}\|\mathbf{x}\|_\infty$$
$$3)\quad \frac{1}{\sqrt{n}}\|A\|_2 \le \|A\|_1 \le \sqrt{n}\|A\|_2 \qquad\qquad 4)\quad \frac{1}{\sqrt{n}}\|A\|_\infty \le \|A\|_2 \le \sqrt{n}\|A\|_\infty$$
Show, that the inequalies are "sharp", i.e. find for any of the above inequalities a vector or matrix for which equality holds.

3. Show, that the 2-norm of a matrix is unitarily invariant (i.e. $\|U\,A\|_2 = \|A\|_2$ for any unitary transformation U).

4. Show that the "Frobenius" norm cannot be subordinate to a vector norm. Show also that it satisfies the product property ($\|B\,A\|_F \le \|A\|_F \, \|B\|_F$) and that it is unitarily invariant ($\|U\,A\|_F = \|A\|_F$ for every unitary transformation U).

5. Prove:
$$\| A \|_F^2 = \text{trace}\,(\, A^T A\,) = \text{sum of all eigenvalues of } A^T A$$
$$\| A \|_2^2 = \text{largest eigenvalue of } A^T A$$
$$\frac{1}{\sqrt{n}} \| A \|_F \le \| A \|_2 \le \| A \|_F$$
Remark. The square roots of the eigenvalues of $A^T A$ are the "singular values" of A.

6. For given vector $\mathbf{a} \in I\!R^n$ the mapping $f_{\mathbf{a}} := \mathbf{x} \mapsto \mathbf{a}^T \mathbf{x}$ is a linear transformation from $I\!R^n$ to $I\!R$. Show that $\| f_{\mathbf{a}} \|_1 = \| \mathbf{a} \|_\infty$, $\| f_{\mathbf{a}} \|_\infty = \| \mathbf{a} \|_1$ and $\| f_{\mathbf{a}} \|_2 = \| \mathbf{a} \|_2$.

4.c The singular value decomposition

Theorem: For any (real) matrix $A \in I\!R^{m\times n}$ there exist orthogonal matrices $U \in I\!R^{m\times m}$ and $V \in I\!R^{n\times n}$ and $p := \min\{m,n\}$ non-negative numbers $\sigma_1, \cdots, \sigma_p$ such that

$$A = U\,\Sigma\,V^T, \qquad \Sigma := diag(\sigma_1, \cdots, \sigma_p) \in I\!R^{m\times n}. \tag{4.8}$$

Notes. – The numbers $\sigma_1, \cdots, \sigma_p$ are called the singular values of A.
– It is common practice to order the singular values in decreasing sense $\sigma_k \ge \sigma_{k+1}$.
– In Matlab the singular value decomposition is computed by the function `svd`:
`s=svd(A)` returns the singular values in the vector `s`.
`[U,S,V]=svd(A)` returns in `U, S` and `V` the three matrices of the decomposition (4.8).

Proof. We give two proofs, one using the eigenvalue decomposition of A^TA, the other is more elementary . For simplicity we choose $m \ge n$. The norms in this proof are the Euclidean vector norm and its subordinate matrix norm.

1. The matrix $A^TA \in I\!R^{n\times n}$ is symmetric and non-negative definite. Hence, it has n non-negative eigenvalues; we order them in decreasing sense $\lambda_1 \ge \lambda_2 \ge \cdots \ge \lambda_n \ge 0$. Associated to these eigenvalues is an orthonormal basis of eigenvectors $\mathbf{v}_1, \cdots, \mathbf{v}_n$ such that $A^TA\mathbf{v}_k = \lambda_k\mathbf{v}_k$. Define

$$\Sigma := diag(\sqrt{\lambda_1}, \cdots, \sqrt{\lambda_n}), \quad V := (\mathbf{v}_1 \,|\, \cdots \,|\, \mathbf{v}_n) \quad \text{and} \quad \widetilde{U} := \left(\frac{A\mathbf{v}_1}{\sqrt{\lambda_1}} \,\middle|\, \cdots \,\middle|\, \frac{A\mathbf{v}_n}{\sqrt{\lambda_n}}\right),$$

then V is an orthogonal matrix and the matrix $\widetilde{U}$ has orthonormal columns. We supplement this matrix with $m-n$ columns to an orthogonal matrix (*how?*). The result satisfies eq. (4.8).

2. An elegant elementary proof by induction runs as follows. Define $\sigma_1 := \|A\|$. The function $\mathbf{x} \mapsto \|A\mathbf{x}\|$ is continuous and has a maximum on the unit ball $\{\|\,\mathbf{x}\,\| = 1\}$. Hence, there is a vector $\mathbf{v}_1$ with norm $\|\mathbf{v}_1\| = 1$ such that $\|A\mathbf{v}_1\| = \|A\| = \sigma_1$. Define $\mathbf{u}_1 := A\mathbf{v}_1/\sigma_1$ and construct orthogonal matrices $U := (\mathbf{u}_1 \,|\, \widehat{U})$ and $V := (\mathbf{v}_1 \,|\, \widehat{V})$ containing these vectors as their first columns by supplementing the sets $\{\mathbf{u}_1\}$ en $\{\mathbf{v}_1\}$ to orthonormal bases in $I\!R^m$ and $I\!R^n$ respectively (e.g. using the Gram-Schmidt process). So we find:

$$V := (\mathbf{v}_1 \,|\, \widehat{V}) \quad \text{such that} \quad A\,V = A\,(\mathbf{v}_1 \,|\, \widehat{V}) = (A\mathbf{v}_1 \,|\, A\widehat{V}) = (\sigma_1\mathbf{u}_1 \,|\, A\widehat{V})$$

and

$$\widetilde{A} := U^TA\,V = \begin{pmatrix} \mathbf{u}_1^T \\ \widehat{U} \end{pmatrix} (\sigma_1\mathbf{u}_1 \,|\, A\widehat{V}) = \begin{pmatrix} \sigma_1\mathbf{u}_1^T\mathbf{u}_1 & \mathbf{u}_1^TA\widehat{V} \\ \sigma_1\widehat{U}^T\mathbf{u}_1 & \widehat{U}^TA\widehat{V} \end{pmatrix} = \begin{pmatrix} \sigma_1 & \mathbf{w}^T \\ \mathbf{0} & \widehat{A} \end{pmatrix}. \tag{4.9}$$

Here $\mathbf{0}$ is the zero vector; its components are zero because the columns of $\widehat{U}$ are orthogonal to $\mathbf{u}_1$ by definition.
The (row) vector $\mathbf{w}^T := \mathbf{u}_1^TA\widehat{V}$ is the first row of $\widetilde{A}$ but for the first element; we shall prove that this vector is zero too. The remaining part of the matrix $\widehat{A} := \widehat{U}^TA\widehat{V} \in I\!R^{(m-1)\times(n-1)}$ is of smaller dimension.

In order to prove that $\mathbf{w}$ is the null vector, we estimate the following norm in two ways. First we estimate the norm of the image of the vector $(\sigma_1, \mathbf{w}^T)^T$ from below by its first component,

$$\left\|\widetilde{A}\begin{pmatrix} \sigma_1 \\ \mathbf{w} \end{pmatrix}\right\|^2 = \left\|\begin{pmatrix} \sigma_1^2 + \mathbf{w}^T\mathbf{w} \\ \widehat{A}\mathbf{w} \end{pmatrix}\right\|^2 \ge (\sigma_1^2 + \mathbf{w}^T\mathbf{w})^2;$$

however, since the matrix norm is invariant under orthogonal transformations, we also have the estimate from above

$$\left\|\widetilde{A}\begin{pmatrix} \sigma_1 \\ \mathbf{w} \end{pmatrix}\right\|^2 \le \sigma_1^2 \left\|\begin{pmatrix} \sigma_1 \\ \mathbf{w} \end{pmatrix}\right\|^2 = \sigma_1^2\,(\sigma_1^2 + \mathbf{w}^T\mathbf{w}).$$

As a consequence we have $\sigma_1^2 + \mathbf{w}^T\mathbf{w} \le \sigma_1^2$. This squeezes the vector $\mathbf{w} \in \mathbb{R}^{n-1}$ to zero length. So we find

$$U^T A\, V = \begin{pmatrix} \sigma_1 & \mathbf{0}^T \\ \mathbf{0} & \widehat{A} \end{pmatrix} . \tag{4.10}$$

We can now apply the same argument to the smaller matrix $\widehat{A}$ and go on until it has size 1. □

The singular value decomposition, abbreviated SVD, looks as a very simple tool to compute the rank of a matrix. In practice round-off errors disturb the picture. Any reliable algorithm to compute the SVD will at best compute the singular value decomposition of a neighbouring matrix. Since the invertible matrices form a dense subset in the set of all $n \times n$ matrices (and the subset of all $n \times n$ matrices of rank $k < n$ is a dense subset in subset of all $n \times n$ matrices of rank k+1) a neighbouring matrix almost surely is of full rank. Hence it is impossible to find the true rank of a rank-deficient matrix (by numerical means). Therefore the numerical rank $rank(A, \varepsilon)$ is defined as the minimal rank of all matrices in a ball around A with radius ε:

$$rank(A, \varepsilon) := \min \{ rank(A + E) \,|\, E \in \mathbb{R}^{m\times n},\ \|E\| \le \varepsilon \} \tag{4.11}$$

In this setting most often the 2-norm is used, because this norm is unitarily invarant and the 2-norm of a matrix is equal to the largest singular value. Let $A \in \mathbb{R}^{m\times n}$ have the singular values $\sigma_1\,, \cdots, \sigma_p$ with $p = \min\{m, n\}$. If these satisfy

$$\sigma_1 \ge \cdots \ge \sigma_r > \varepsilon \ge \sigma_{r+1} \ge \cdots \ge \sigma_p\,, \qquad \text{then} \qquad rank(A, \varepsilon) := r\,; \tag{4.12}$$

if $A = U\Sigma\, V^T$, then the matrix $E := U\, diag(0\,, \cdots, 0\,, \sigma_{r+1}\,, \cdots, \sigma_p)\, V^T$ satisfies the condition $\|E\|_2 \le \varepsilon$ and $rank(A - E) = r$.

The study of reliable algorithms for computing the SVD is outside the scope of this course. The function `svd(A)` in Matlab computes factors U, Σ and V that satisfy $\|\, A - U\, \Sigma\, V^T \,\|_2 \le \eta \,\|\, A \,\|_2$, i.e. factors that form the exact SVD of a neighbouring matrix.

Example of the use of SVD for data reduction. The magic square from the woodcut "Melencolia" of Dürer is scanned to a matrix of 359×371 gray values. The SVD of this matrix is computed and the singular values are plotted in the left plot of fig.2. The right picture shows the matrix that results if all but the largest singular value is set to zero and the middle if all but the 36 largest are set to zero. The grid is the most dominant feature in the picture. The reconstruction using only the 36 dominant singular values provides already a very good approximation. This

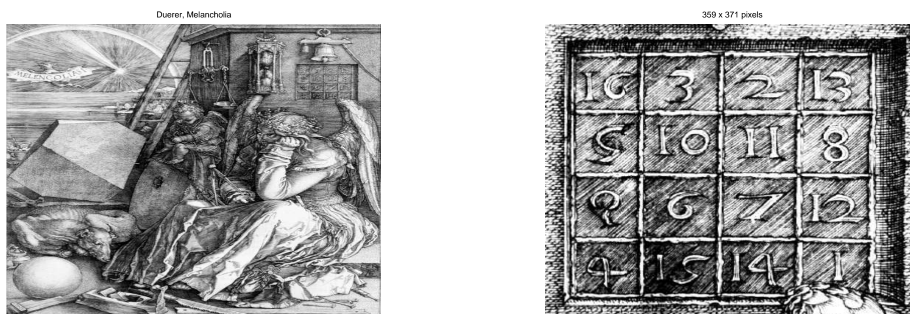

Figure 1: Dürer's woodcut "melencolia" (left) and the detail "the magic square" in the upper right corner (right)

analysis technique is not very common for pictures. In statistics this type of data reduction is very popular under the name "principal component analysis". The grid in the right picture of fig. 2 is the first principal component of the picture of the magic square; the left picture of the singular values is the "scree plot".

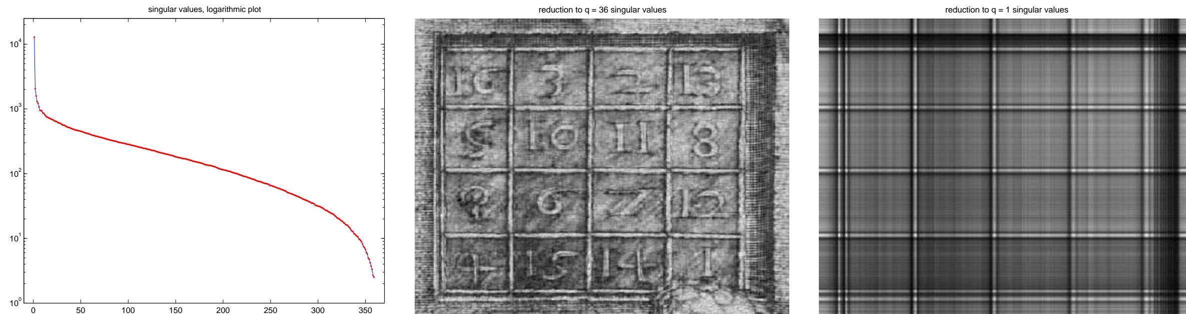

Figure 2: Logarithmic plot of the 359 singular values of the matrix of gray values of the pixels of the detail (left) and reconstructions of the detail using 36 (middle) and only 1 singular value (right). Apparently, the grid is the most dominant feature in the picture.

4.d The Condition Number of a Matrix

Given an invertible matrix $A \in \mathbb{R}^{n\times n}$ and a vector $\mathbf{b} \in \mathbb{R}^n$ we want to solve the set of n linear equations

$$A\,\mathbf{x} = \mathbf{b}\,. \tag{4.13}$$

Before we study algorithms, it is advantageous to study the sensitivity of the problem with respect to small perturbations of A and $\mathbf{b}$ (as we did in section 3.c for a univariate function). Let E and $\mathbf{d}$ be (small) perturbations of A and $\mathbf{b}$. We consider the "perturbed" problem

$$(A+E)\,(\mathbf{x}+\mathbf{w}) = \mathbf{b}+\mathbf{d}\,, \qquad \text{where } E \in \mathbb{R}^{n\times n} \quad \text{and} \quad \mathbf{d} \in \mathbb{R}^n \quad \text{small.} \tag{4.14}$$

We try to estimate the resulting deviation $\mathbf{w}$ of from the solution $\mathbf{x}$ of (4.13). The perturbed equation can be solved (uniquely) only if $A+E$ is invertible:

Lemma. If $A \in \mathbb{R}^{n\times n}$ is invertible and if $E \in \mathbb{R}^{n\times n}$ is so small that $\|A^{-1}E\| < 1$, then $A+E$ invertible and satisfies the estimate

$$\|(A+E)^{-1}\| \leq \frac{\|A^{-1}\|}{1-\|A^{-1}E\|} \tag{4.15}$$

Proof. Let I be the identity in $\mathbb{R}^n$ and let $F \in \mathbb{R}^{n\times n}$ satisfy $\|F\| < 1$, then all $\mathbf{x} \in \mathbb{R}^n$ satisfy the inequality

$$\|I\mathbf{x}+F\mathbf{x}\| \geq \|\mathbf{x}\| - \|F\mathbf{x}\| \geq (1-\|F\|)\|\mathbf{x}\| > 0\,.$$

Hence, no non-zero vector is mapped to the zero vector by $I+F$ and so it is invertible. Replacing $\mathbf{x}$ in this formula by $(I+F)^{-1}\mathbf{y}$ we find

$$\|\mathbf{y}\| = \|(I+F)(I+F)^{-1}\mathbf{y}\| \geq (1-\|F\|)\|(I+F)^{-1}\mathbf{y}\| \qquad \text{for all } \mathbf{y} \in \mathbb{R}^n\,.$$

Taking the maximum over all $\mathbf{y}$ in the unit ball we find

$$\|(I+F)^{-1}\| \leq \frac{1}{1-\|F\|}\,.$$

Since $A+E = A(I+A^{-1}E)$ this implies the inequality (4.15). □

We now return to problem (4.14), to find an upper bound for $\|\mathbf{w}\|$. We subtract (4.13) from (4.14) and find

$$(A+E)\,\mathbf{w} = \mathbf{d} - E\mathbf{x}\,.$$

Using the lemma we estimate:

$$\|\mathbf{w}\| \leq \|(A+E)^{-1}\| \left(\|\mathbf{d}\| + \|E\|\,\|\mathbf{x}\|\right) \leq \frac{\|A^{-1}\|(\|\mathbf{d}\| + \|E\|\,\|\mathbf{x}\|)}{1 - \|A^{-1}E\|}\,. \tag{4.16}$$

Dividing by $\|\mathbf{x}\|$ and using the inequality $\|A\|\,\|\mathbf{x}\| \geq \|A\,\mathbf{x}\| = \|\mathbf{b}\|$ we find an estimate for the **relative** perturbation:

$$\frac{\|\mathbf{w}\|}{\|\mathbf{x}\|} \leq \frac{\|A^{-1}\|\,\|A\|}{1 - \|A^{-1}E\|}\left(\frac{\|\mathbf{d}\|}{\|\mathbf{b}\|} + \frac{\|E\|}{\|A\|}\right)\,. \tag{4.17}$$

We see in this formula that the relative magnitudes of the perturbations of A and $\mathbf{b}$ are multiplied by the factor $\kappa(A) := \|A^{-1}\|\,\|A\|$ (provided that the term $\|A^{-1}E\|$ in the numerator is negligible). This factor κ is called the Condition number of the matrix A. This condition number depends on the matrix and vector norms used in the analysis. Most often we use the condition numbers

$$\kappa_1 := \|A^{-1}\|_1\,\|A\|_1 \qquad \kappa_2 := \|A^{-1}\|_2\,\|A\|_2 \qquad \text{and} \qquad \kappa_\infty := \|A^{-1}\|_\infty\,\|A\|_\infty \tag{4.18}$$

with respect to the usual *lub* matrix norms. The computation of these condition numbers requires the computation of the inverse (for κ_1 and κ_∞) or the SVD (for κ_2). Condition number estimators exist, that require much less computational effort.

4.e Exercises

1. Compute the singular value decomposition of the $n\times 1$–matrix $A := \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}$.

2. Compute the singular value decomposition of the $n\times 2$–matrix $A := (\,\mathbf{u}\,|\,\mathbf{v}\,)$, where the vectors $\mathbf{u} \in I\!R^n$ en $\mathbf{v} \in I\!R^n$ are perpendicular ($\mathbf{u}^T\mathbf{v} = 0$).

3. For the following matrices B^{-1} compute the inverse and the condition number $\kappa_\infty(B) := \|B\|_\infty\,\|B^{-1}\|_\infty$

   a.
$$B := \begin{pmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \in I\!R^{n\times n} \quad \text{where} \quad B_{ij} = \begin{cases} 1 & \text{if} \quad j = i\,, \\ -1 & \text{if} \quad j > i\,, \\ 0 & \text{if} \quad j < i\,. \end{cases} \tag{4.19}$$

   b.
$$B := \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \in I\!R^{n\times n} \quad \text{where} \quad B_{ij} = \begin{cases} 1 & \text{if} \quad j \geq i\,, \\ 0 & \text{if} \quad j < i\,. \end{cases} \tag{4.20}$$

   c.
$$B := \begin{pmatrix} 1 & 2 & \cdots & n \\ 0 & 1 & \cdots & n-1 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \in I\!R^{n\times n} \quad \text{where} \quad B_{ij} = \begin{cases} j-i+1 & \text{if} \quad j \geq i\,, \\ 0 & \text{if} \quad j < i\,. \end{cases} \tag{4.21}$$

   Hint: Reduce B by Gauss-Jordan algorithm to the form (4.20).

   d.
$$B := \begin{pmatrix} 1 & -1 & \cdots & (-1)^{n-1} \\ 0 & 1 & \cdots & (-1)^{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix} \in I\!R^{n\times n} \quad \text{where} \quad B_{ij} = \begin{cases} (-1)^{j-i} & \text{if} \quad j \geq i\,, \\ 0 & \text{if} \quad j < i\,. \end{cases} \tag{4.22}$$

4.f Gaussian Elimination

A triangular sytem of equations $L\mathbf{x} = \mathbf{b}$ or $U\mathbf{y} = \mathbf{c}$ where L is a lower (left) triangular matrix with $L_{ij} = 0$ if $j > i$ and where U is an upper (right) triangular matrix with $U_{ij} = 0$ if $j < i$,

$$L = \begin{pmatrix} L_{11} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ L_{1,n-1} & \cdots & L_{n-1,n-1} & 0 \\ L_{n1} & \cdots & \cdot & L_{nn} \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} U_{11} & \cdots & \cdots & U_{1n} \\ 0 & U_{22} & \cdots & U_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & U_{nn} \end{pmatrix} \tag{4.23}$$

can be solved easily top down or bottom up. In Matlab this is coded in the following way:

```
x(1)=b(1)/L(1,1);
for k=2:n,
    x(k)=(b(k)-L(k,1:k-1)*x(1:k-1))/L(k,k);
end
```
(4.24)

```
y(n)=c(n)/U(n,n);
for k=n-1:-1:1,
    y(k)=(c(k)-U(k,k+1:n)*y(k+1:n))/U(k,k);
end
```
(4.25)

Exercise: Check that the following (columnwise) algorithm computes the same result as (4.24) does:

```
for k=1:n-1,
    x(k)=b(k)/L(k,k);
    b(k+1:n)=b(k:1:n)-L(k+1:n,k)*x(k);
end,
x(n)=b(n)/L(n,n) .
```
(4.26)

Write down the analogous columnwise algorithm for the solution of $\mathbf{y}$ in eq. (4.25). Determine the number of flops used for the computations of $\mathbf{x}$ and $\mathbf{y}$ in (4.24), (4.25) and (4.26).

Gauss elimination is an algorithm that reduces a general linear system of equations

$$A\mathbf{x} = \mathbf{b} \qquad \text{or} \qquad \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} \tag{4.27}$$

to upper triangular form $U\mathbf{x} = \mathbf{c}$. The basic idea is that any linear combination of equations of system (4.27) is again an equation that is satisfied by the solution. If $A_{11} \neq 0$, we can subtract from the second up to the n–th equation a multiple of the first equation in such a way that the coefficient of the first unknown x_1 in these equations vanishes. This is accomplished by replacing row(k) in the matrix by row$(k) - A_{k1}/A_{11} \times$ row(1).

If $A_{22} \neq 0$ in the resulting matrix, we can eliminate similarly the dependence on x_2 from the third up to the n–th equations, etc. After $n-1$ steps an (equivalent) triangular set of equations remains, that can be solved easily by algorithm (4.25). We can write down the algorithm a little more formally as:

for $k = 1 : n-1$
  If $A_{kk} \neq 0$,
    for $j = k+1 : n$
      replace row(j) of A by row$(j) - A_{jk}/A_{kk} \times$ row(k)
      and replace the j-th element b_j of the r.h.s. by $b_j - A_{jk}/A_{kk} \times b_k$
    end
end

(4.28)

In matlab this is coded compactly as

```
for k=1:n-1,
    for j=k+1:n,
        A(j,k+1:n) = A(j,k+1:n) - A(j,k) / A(k,k) * A(k,k+1:n);
        b(j) = b(j) - A(j,k) / A(k,k) * b(k);
    end
end
```
(4.29)

After termination all matrix elements `A(i,j)` with `i>j` should be zero. However, we did not take the trouble (and the additional work) to set all those elements to zero explicitly, because they are not used any more in the final solution step (4.25). Moreover, since the lower triangular part of the matrix has become irrelevant, we can use the memory space to store the multipliers `A(j,k) / A(k,k)`:

```
for k=1:n-1,
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n);
    b(k+1:n) = b(k+1:n) - A(k+1:n,k)*b(k);
end
```
(4.30)

The benefit of this is that we can split the algorithm in a part that works on the matrix alone and a part that works on the right-hand side **b**:

```
for k=1:n-1,
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n);
end
for k=1:n-1,
    b(k+1:n) = b(k+1:n) - A(k+1:n,k)*b(k);
end
```
(4.31)

After termination of algorithm (4.31) we can assign the elements of `A` partially to the lower triangular matrix L and partially to the upper triangular U:

$$L_{ij} = \begin{cases} 1 & \text{if} \quad i = j \\ \texttt{A(i,j)} & \text{if} \quad i > j \\ 0 & \text{if} \quad i < j \end{cases} \qquad \text{and} \qquad U_{ij} = \begin{cases} \texttt{A(i,j)} & \text{if} \quad i \leq j \\ 0 & \text{if} \quad i > j \end{cases} .$$

The matrices L and U constructed in this way satisfy the property $A_{\text{original}} = L\,U$, since every pair consisting of a solution $\mathbf{x}$ and a right-hand side $\mathbf{b} = A\mathbf{x}$ satisfies by construction $U\mathbf{x} = \mathbf{y}$ and $L\mathbf{y} = b$.

Row pivoting. An essential point in (4.28) is the fact that the **pivot** A_{kk} in de k–th step should be non-zero. However, this is not true in general as can be inferred from the following example

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} .$$

This system is perfectly solvable, but algorithm (4.30) does not work, because $A_{11} = 0$. The remedy is to interchange the order of both equations,

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_2 \\ b_1 \end{pmatrix} ,$$

and, hence, the interchange of both rows of the matrix. It is clear that this does not change the order of the unknowns x_1 and x_2.

Consider now the general case. Let us assume that algorithm (4.28) has executed $k-1$ elimination stages in which the elements below the diagonal in the first up to the $k-1$–st column have been zeroed. Hence, the system $A\mathbf{x} = \mathbf{b}$ has been reduced to the equivalent system

$$\begin{pmatrix} \widehat{A}_{11} & \cdots & \cdots & \widehat{A}_{1k} & \cdots & \widehat{A}_{1n} \\ 0 & \ddots & & \vdots & & \vdots \\ \vdots & \ddots & \ddots & \vdots & & \vdots \\ 0 & \cdots & 0 & \widehat{A}_{kk} & \cdots & \widehat{A}_{kn} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & \widehat{A}_{nk} & \cdots & \widehat{A}_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ \vdots \\ x_k \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \widehat{b}_1 \\ \vdots \\ \vdots \\ \widehat{b}_k \\ \vdots \\ \widehat{b}_n \end{pmatrix}. \tag{4.32}$$

In the next stage of algorithm (4.28) it is required, that $\widehat{A}_{kk}$ is non-zero. If the original matrix A is invertible, then the equivalent matrix $\widehat{A}$ is invertible too. Since all elements in the first $k-1$ places of the k–th up to n–th rows are zero, at least one element at the k–th place of these rows has to be non-zero, otherwise $\widehat{A}$ is singular. Hence, if $\widehat{A}_{kk} = 0$, we can find a row with index $p > k$ such that $\widehat{A}_{pk} \neq 0$ and we can interchange the corresponding p–th and k–th rows (and also the elements $\widehat{b}_p$ and $\widehat{b}_k$ of the right-hand side) and continue the elimination of the k–th column. The element $\widehat{A}_{pk}$ is called the pivot of stage k.

With this addition Gaussian elimination reduces every (uniquely solvable) set of equations to an equivalent upper triangular system, at least in theory. In practice, round-off errors may disrupt this picture. In theory, every non-zero pivot $\widehat{A}_{pk}$ will do. In practice, if it is very small (in absolute value) in comparison to other elements of the column, a multiplier $\widehat{A}_{jk}/\widehat{A}_{pk}$ may become very large, such that the original j-th row is drowned in the round-off errors by the operation $\text{row}(j) \leftarrow \text{row}(j) - \widehat{A}_{jk}/\widehat{A}_{pk} \times \text{row}(p)$; as a consequence this new j–th row is (almost) dependent on the p–th and the new matrix becomes (nearly) singular. This problem can be avoided by choosing the largest (in absolute value) from the column $\widehat{A}_{kk}, \cdots, \widehat{A}_{nk}$ as the pivot. This strategy implies that no multiplier has an absolute value larger than one.

for $k = 1 : n-1$
  Search among the elements $A_{kk}, \cdots, A_{nk}$
    the element largest in absolute value. Assume it has row-index p.
  Interchange row(k) and row(p) of A and the elements b_k and b_p of the r.h.s.
  for $j = k+1 : n$ (4.33)
    replace row(j) of A by row(j) $- A_{jk}/A_{kk} \times$ row(k)
    and replace the j-th element b_j of the r.h.s. by $b_j - A_{jk}/A_{kk} \times b_k$
  end
end

In Matlab this can be coded in the following way[2]:

```
for k=1:n-1,
    [m,p] = max(abs(A(k:n,k))); p = p+k-1;
    hulp = A(k,k:n); A(k,k:n) = A(p,k:n); A(p,k:n)=hulp;
    hulp = b(k); b(k) = b(p); b(p)=hulp;
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    for j=k+1:n, A(j,k+1:n) = A(j,k+1:n) - A(j,k)*A(k,k+1:n); end
    b(k+1:n) = b(k+1:n) - A(k+1:n,k)*b(k);
end
```
(4.34)

Finally, we can store all elimination information; the multipliers `A(j,k) / A(k,k)` go into the free locations `A(j,k)` ($j > k$) and the row index of the pivot in the k-th stage is stored in the k-th

[2]Because the Matlab standard function `max` computes the maximum `m` and its row index `p` in the vector `z(1:n-k+1)=abs(A(k:n,k))`, which has `n-k+1` elements numbered from 1 up to $n-k+1$, we have to correct the offset of this index by adding `k-1` in order to find the correct position in the matrix.

place of an additional array **p** of permutation indices. As before, we can now split the algorithm in an elimination phase and a solution phase, as in (4.31):

```
for k=1:n-1,
    [m,q] = max(abs(A(k:n,k))); p(k) = q+k-1;
    if p(k)>k, hulp = A(k,1:n); A(k,1:n) = A(p(k),1:n); A(p(k),1:n)=hulp; end
    A(k+1:n,k) = A(k+1:n,k)/A(k,k);
    A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n);
end
for k=1:n-1,
    if p(k)>k, hulp = b(k); b(k) = b(p(k)); b(p(k))=hulp; end
    b(k+1:n) = b(k+1:n) - A(k+1:n,k)*b(k);
end
```
(4.35)

We remark that we changed more in the transition from (4.34) to (4.35); we interchanged in the k-th stage not only the the elements $A(k, k : n)$ and $A(p, k : n)$, but also the multipliers $A(k, 1 : k-1)$ and $A(p, 1 : k-1)$, that have been formed in the previous stages. This is motivated by the following observation. At the start of the k–th stage of the basic algorithm (4.32) we have $A^{(1)} := A_{\text{original}} = L^{(k)}A^{(k)}$,

$$A^{(1)} := A_{\text{original}} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \ddots & & \vdots \\ L_{k1} & \cdots & 1 & \ddots & \vdots \\ \vdots & \cdots & 0 & \ddots & 0 \\ L_{n1} & \cdots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} A^{(k)}_{11} & \cdots & A^{(k)}_{1k} & \cdots & A^{(k)}_{1n} \\ 0 & \ddots & \vdots & & \vdots \\ 0 & \cdots & A^{(k)}_{kk} & \cdots & A^{(k)}_{kn} \\ 0 & & \vdots & & \vdots \\ 0 & \cdots & A^{(k)}_{nk} & \cdots & A^{(k)}_{nn} \end{pmatrix}. \tag{4.36}$$

where the non-trivial elements of $A^{(k)}$ are stored in the matrix elements `A(i,j)` with $i \leq j$ or $j \geq k$ and where the non-trivial elements of $L^{(k)}$ are contained in `A(i,j)` with $i > j$ and $j < k$. In the k–th stage we multiply $A^{(k)}$ from the left by the matrix G_k^{-1} (dubbed “Gauss Transformation” by Golub & Van Loan)

$$G_k^{-1} := \begin{pmatrix} 1 & & 0 & \cdots & 0 \\ \vdots & \ddots & & & \vdots \\ 0 & \cdots & 1 & \ddots & \vdots \\ \vdots & \cdots & -L_{k+1,k} & & 0 \\ \vdots & \cdots & \vdots & & \\ 0 & \cdots & -L_{nk} & \cdots & 1 \end{pmatrix} \quad \text{and} \quad G_k := \begin{pmatrix} 1 & & 0 & \cdots & 0 \\ \vdots & \ddots & & & \vdots \\ 0 & \cdots & 1 & \ddots & \vdots \\ \vdots & \cdots & L_{k+1,k} & & 0 \\ \vdots & \cdots & \vdots & & \\ 0 & \cdots & L_{nk} & \cdots & 1 \end{pmatrix}$$

where $L_{jk} := A^{(k)}_{jk}/A^{(k)}_{kk}$ for $j = k+1 \cdots n$. This results in $A^{(k+1)} = G_k^{-1}A^{(k)}$. In order to maintain the identity

$$A_{\text{original}} = L^{(k)}A^{(k)} \tag{4.37}$$

at the start of the next stage, we have to multiply $L^{(k)}$ from the right by G_k. This has precisely the effect that the k–th column of G_k is inserted as the k–th column of $L^{(k)}$ (check!).

The interchange of row(k) and row(p) in $A^{(k)}$ can be described as the multiplication of $A^{(k)}$ from the left by the permutation matrix P_k, which has the form

$$P_k := \begin{pmatrix} 1 & & & \cdots & & 0 \\ & \ddots & & & & \\ & & 0 & \cdots & 1 & \\ \vdots & & \vdots & & \vdots & \vdots \\ & & 1 & \cdots & 0 & \\ & & & & \ddots & \\ 0 & & & \cdots & & 1 \end{pmatrix} \begin{matrix} \\ \\ \text{row}(k) \\ \\ \text{row}(p) \\ \\ \\ \end{matrix} \tag{4.38}$$

In order to maintain the identity (4.37), we have to multiply $L^{(k)}$ from the right by the same permutation matrix P_k; this means the interchange of the **columns** of $L^{(k)}$ with indices k and $p(k)$. Because this product is not a lower triangular matrix, we also multiply $L^{(k)}$ from the left by P_k and we add the permutation matrix to the invariant

$$A_{\text{original}} = P_1 \cdots P_{k-1} P_k P_k L^{(k)} P_k P_k A^{(k)}. \tag{4.39}$$

To the matrix $P_k A^{(k)}$ (whose rows k and $p(k)$ are interchanged) we apply the Gauss transformation G_k^{-1} from the left and to $P_k L^{(k)} P_k$ we apply G_k from the right, such that

$$A_{\text{original}} = P_1 \cdots P_{k-1} P_k P_k L^{(k)} P_k G_k G_k^{-1} P_k A^{(k)} = P_1 \cdots P_{k-1} P_k L^{(k+1)} A^{(k+1)} \tag{4.40}$$
$$\text{where} \qquad L^{(k+1)} := P_k L^{(k)} P_k G_k \qquad \text{and} \qquad A^{(k+1)} := G_k^{-1} A^{(k)}.$$

In this way we construct the decomposition $A = PLU$ in a product consisting of a lower triangular matrix L with $L_{jj} = 1$ and $|\, L_{ij} \,| \leq 1$ if $i > j$ and and upper triangular matrix U and a product $P := P_1 \cdots P_n$ of permutation matrices. This proves the correctness of algorithm (4.35) and it proves the existence of a decomposition of the form $A = PLU$ for every invertible matrix A.

4.g The algorithm of Crout

Using the knowledge that the decomposition $A = PLU$ exists for every invertible matrix, we can derive the existence and the construction of it in a different way, that leads to variants of the algorithm. We assume for the sequel that the row interchanges already have been applied to A (or are not necessary) and that we have $A = LU$, or componentwise:

$$A_{ik} = \sum_{j=1}^{\min\{i,k\}} L_{ij} U_{jk} \quad \text{or} \quad \begin{cases} A_{kk} = & \sum_{j=1}^{k-1} L_{kj} U_{jk} + U_{kk} & \text{if} \quad i = k \quad (a) \\ A_{ki} = & \sum_{j=1}^{k-1} L_{kj} U_{ji} + U_{ki} & \text{if} \quad i > k \quad (b) \\ A_{ik} = & \sum_{j=1}^{k-1} L_{ij} U_{jk} + L_{ik} U_{kk} & \text{if} \quad i > k \quad (c) \end{cases} \tag{4.41}$$

where the indices i and k in (b) are interchanged to get $i \geq k$ in all equations. If $k = 1$, the sums are empty and we see that the first row of U is equal to the first row of A and that the first column of L is equal to the first column of A divided by $U_{11} = A_{11}$. If the first $k-1$ columns of L and the first $k-1$ rows of U have been computed (if L_{ij} and U_{ji} with $j < k$ are known for a given k), then U_{kk} can be computed from equation (a) and the remaining part of the k-th row of U can be computed from equation (b). Finally, with U_{kk} known, the elements of the k–th column of L can be computed from equation (c). Thus we find the algorithm of Crout for the LU-decomposition of A (without row interchanges):

$$\begin{array}{l} \textbf{for} \;\; k = 1 : n, \\ \quad U_{kk} = A_{kk} - \sum_{j=1}^{k-1} L_{kj} U_{jk}\,; \\ \quad U_{ki} = A_{ki} - \sum_{j=1}^{k-1} L_{kj} U_{ji}\,; \quad (i = k+1 \cdots n) \\ \quad L_{ik} = (A_{ik} - \sum_{j=1}^{k-1} L_{ij} U_{jk}) / U_{kk}\,; \quad (i = k+1 \cdots n) \\ \textbf{end} \end{array} \tag{4.42}$$

Since an element A_{pq} of A is only addressed once for the computation of the corresponding U_{pq} or L_{pq} and is no more used thereafter, we may overwrite the memory location by this element of L or U, as we did in (4.31) (Gauss elimination). So we derive the algorithm, known as the "Crout's LU-decomposition"

```
for k=1:n,
    A(k,k) = A(k,k) - A(k,1:k-1)*A(1:k-1,k);
    A(k,k+1:n) = A(k,k+1:n) - A(k,1:k-1)*A(1:k-1,k+1:n);
    A(k+1:n,k) = (A(k+1:n,k) - A(k+1:n,1:k-1)*A(1:k-1,k))/A(k,k);
end
```

(4.43)

This algorithm is only a reordering of the computations in comparison to Gauss elimination. Let us consider a fixed element (memory location) A_{pq}. During Gauss elimination (4.31) it is accessed in every stage $k < \min(p,q)$ for one subtraction $A_{pq} \leftarrow A_{pq} - A_{pk}A_{kq}$, whereas all subtractions are done at once in the $k = \min(p,q)$–th stage of (4.43) [`A(p,q) = A(p,q) - A(p,1:k-1)*A(1:k-1,q)`]. This implies that the round-off errors made during the computations are exactly the same[3] for both algorithms. This observation also provides the clue, how to incorporate the row-exchange strategy of Gaussian elimination in Crout's variant.

Exercise. Incorporate the row-exchange strategy of Gaussian elimination in Crout's LU-decomposition and write down the algorithm in Matlab.

4.h Round-off error analysis

From Crout's algorithm we see that every element of L and U is calculated from an equation of the form

$$A_{ik} = \sum_{j=1}^{\min\{i,k\}} L_{ij} U_{jk} \, .$$

This is exactly the form of example 5 in section 3. Hence, the computed values $\widehat{L}_{ij}$ and $\widehat{U}_{jk}$ of the elements of L and U satisfy exactly the equations

$$A_{ik} = \sum_{j=1}^{\min\{i,k\}} \widehat{L}_{ij} \widehat{U}_{jk} (1 + \varepsilon_{ijk}) \quad \text{where} \quad |\, \varepsilon_{ijk} \,| \leq (j{+}1)\eta \, .$$

As a consequence, the computed matrices $\widehat{L}$ and $\widehat{U}$ satisfy exactly a neighbouring equation

$$A = \widehat{L}\,\widehat{U} + E \qquad \text{where} \quad |\, E_{ik} \,| \leq (n+1)\,\eta \sum_{j=1}^{\min\{i,k\}} |\, \widehat{L}_{ij} \widehat{U}_{jk} \, .\,| \tag{4.44}$$

So, the perturbation matrix E satisfies

$$\|E\|_\infty \leq (n+1)\,\eta\, \|\widehat{L}\|_\infty \, \|\widehat{U}\|_\infty \tag{4.45}$$

Since all elements of $\widehat{L}$ are smaller than or equal to 1 in absolute value because of the row interchanges, the max-norm of this matrix is bounded by n. The most important factor in the bound for E is the magnitude of $\|\widehat{U}\|_\infty$. Although in practice this norm is comparable to that of A, examples exist in which this norm is 2^{n-1} times as large (see exercise 5 below). We conclude that Gaussian elimination with row interchanges (pivoting) in general is a very reliable and robust method for the solution of a system of linear equations.

4.i Exercises

1. There is a method to minimize the potential ill-conditioning of the factor U in Gaussian elimination by incorporating both row and column interchanges. In Gaussian elimination with pivoting, we search in the k–th stage for the maximal element of `abs(A(k:n,k))` and interchange the corresponding rows. We can do better when we search for the largest element of the whole sub-matrix `abs(A(k:n,k:n))` and bring this element in the `(k,k)`-position by interchanging both the corresponding row and column with the k–th row and column respectively. Columns can be interchanged by multiplying the matrix by a permutation matrix (4.38) from the right. All permutations from the right can be aggregated in one matrix Q

[3]Provided, no additional rounding occurs, when a result of one or of a series of arithmetical operations between registers in the processor is written back from a register to the memory.

resulting in a decomposition $A = P\,L\,U\,Q$ where L and U are lower and upper triangular matrices, pre- and post-multiplied by permutation matrices P and Q Write down the algorithm in Matlab.

2. When the decomposition $A = LU$ is given, we compute the solution $\mathbf{x}$ of $A\mathbf{x} = \mathbf{b}$ by solving first $\mathbf{y}$ from $L\mathbf{y} = \mathbf{b}$ and subsequently $\mathbf{x}$ from $U\mathbf{x} = \mathbf{y}$. Show that the computed vectors $\widehat{\mathbf{y}}$ and $\widehat{\mathbf{x}}$ are the exact solutions of the neighbouring equations $(L+F)\mathbf{y} = \mathbf{b}$ and $(U+G)\mathbf{x} = \widehat{\mathbf{y}}$ for some perturbation matrices $|\,F_{i,j}\,| \le (i+1)\eta\,|\,L_{i,j}\,|$ and $|\,G_{i,j}\,| \le (i+1)\eta\,|\,U_{i,j}\,|$.

3. Let $A \in I\!R^{n\times n}$ be a regular matrix and let $\mathbf{u}$, $\mathbf{v} \in I\!R^{n}$ be column vectors. Assume $\mathbf{v}^T A^{-1}\mathbf{u} \neq -1$. Prove the Sherman-Morrison formula :

$$(A + \mathbf{u}\mathbf{v}^T)^{-1} = A^{-1} - \frac{A^{-1}\mathbf{u}\mathbf{v}^T A^{-1}}{1 + \mathbf{v}^T A^{-1}\mathbf{u}} \,. \qquad (4.46)$$

4. For given vector $\mathbf{y} \in I\!R^{n}$ and index $k \in I\!N$, a matrix of the form

$$N(\,\mathbf{y}\,,\,k\,) \;:= I + \mathbf{y}\,\mathbf{e}_k^T \;\in I\!R^{n\times n}$$

is called a Gauss-Jordan transformation.

a. Inder what condition on $\mathbf{y}$ the matrix $N(\mathbf{y}\,,\,k)$ is invertible; find a formula for its inverse.

b. Given a (fixed) vector $\mathbf{x} \in I\!R^{n}$, under what conditions a vector $\mathbf{y} \in I\!R^{n}$ exists such that

$$N(\mathbf{y}\,,\,k)\;(\mathbf{x}) = \mathbf{e}_k.$$

Deduce a formula for it.

c. Deduce an algorithm that overwrites the matrix A by its inverse A^{-1} using n Gauss-Jordan transformations.

d. What conditions on A ensure that the algorithm is succesful?

5. Let $A \in I\!R^{n\times n}$ be a matrix with elements

$$A_{kk} = 1\,, \quad A_{ki} = -1 \quad \text{if} \quad i < k\,, \quad A_{kn} = 1 \quad \text{and} \quad A_{ki} = 0 \quad \text{if} \quad k < i < n\,.$$

Calculate the LU–decomposition of A and the norms $\|A\|_\infty$ and $\|U\|_\infty$.
This is an example in which the norm of U is much larger than the norm of A is.

6. Let $A \in I\!R^{n\times n}$ be a rowwise *diagonally dominant* matrix, i.e.

$$\text{if} \quad A = (a_{ij})_{i,j=1}^{n} \qquad \text{then} \qquad |\,a_{jj}\,| > \sum_{i=1,\,i\neq j}^{n} |\,a_{ji}\,|\,, \quad \forall\, j,$$

Prove that A has an LU–decomposition without row permutations with a U–factor satisfying $\|\,U\,\|_\infty \le 2\max_k |U_{kk}|$.
Hint: Show that the submatrix $A(2\!:\!n, 2\!:\!n)$ after the first stage of the Gaussian elimination is again diagonally dominant.
Remark: The norm of the factor L may become very large in this case, as can be seen from the following example. We display the matrix, the result after the first stage of the elimination and the final result after the second stage :

$$A := \begin{pmatrix} 1 & \sqrt{\alpha} & 0 \\ \sqrt{\alpha} & 1 & 0 \\ 0 & \alpha & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \sqrt{\alpha} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \sqrt{\alpha} & 0 \\ 0 & 1-\alpha & 0 \\ 0 & \alpha & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ \sqrt{\alpha} & 1 & 0 \\ 0 & \frac{\alpha}{1-\alpha} & 1 \end{pmatrix} \begin{pmatrix} 1 & \sqrt{\alpha} & 0 \\ 0 & 1-\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

For every $\alpha \in [0, 1)$ the matrix A is diagonally dominant, but $\|L\|_\infty \nearrow \infty$ as $\alpha \nearrow 1$. When the usual row interchange strategy is used, we have to interchange the second and third rows in the second stage if $\alpha \in (\frac{1}{2}, 1)$. In that case we find

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \sqrt{\alpha} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \sqrt{\alpha} & 0 \\ 0 & \alpha & 1 \\ 0 & 1-\alpha & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \sqrt{\alpha} & \frac{1-\alpha}{\alpha} & 1 \end{pmatrix} \begin{pmatrix} 1 & \sqrt{\alpha} & 0 \\ 0 & \alpha & 1 \\ 0 & 0 & \frac{\alpha-1}{\alpha} \end{pmatrix}.$$

However, we observe the conservation of trouble; instead of an L–factor with a large norm we find a badly conditioned U-factor as $\alpha \approx 1$.
Analogously: If A is *columnwise diagonally dominant*, then A has an LU–decomposition without pivoting with a bound on the L-factor $\| L \|_1 \leq 2$.

5 Linear Least Squares Problems

A standard example of the origin of linear least squares problems is the (statistical) question to find the best fitting line to a number of datapoints $\{(x_1, y_1) \cdots (x_n, y_n)\}$ in the plane (regression).

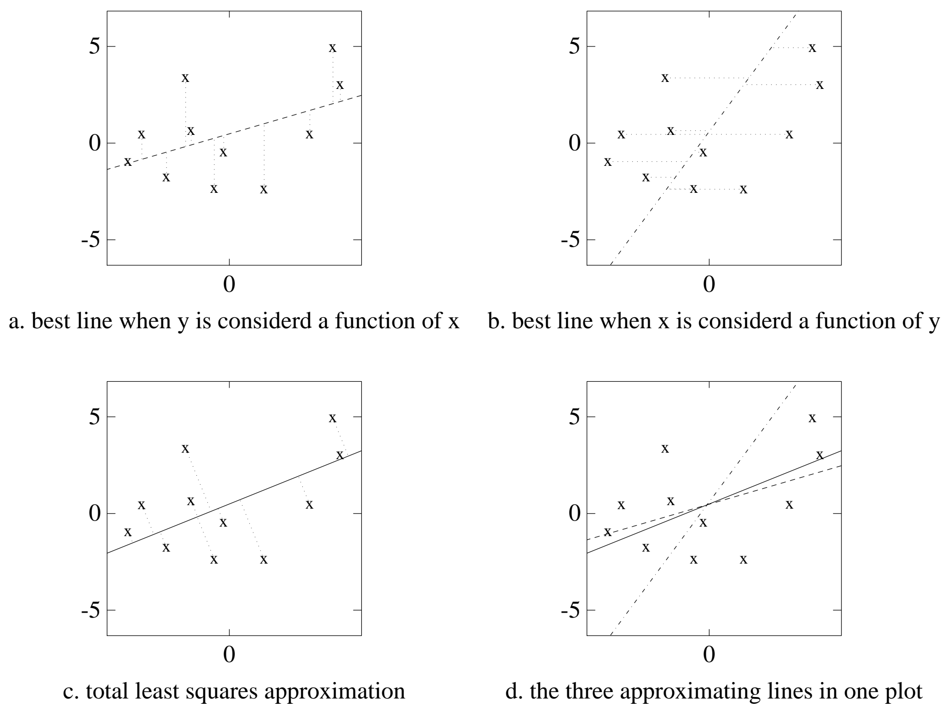

Figure 3: Datapoints in the plane with "best fitting" lines. In each of the pictures (a), (b) and (c) this is the line that minimizes the sum of squares of the length of the dotted lines, the distances along the x-axis (a), the distance along the y-axis (b), the Euclidean distance (c). In (d) the three regression lines are plotted for comparison.

The question is to find a line $y = a + b\,x$ such that the sum of squares of the deviations is minimal:

$$\text{find } (a, b) \text{ such that } \quad J(a,b) := \sum_{k=1}^{n} (y_k - a - bx_k)^2 \text{ is minimal.} \tag{5.1}$$

Among all optimality criteria the minimisation of a sum of squares is by far the easiest, because this functional is quadratic in the unknowns a and b, implying a unique minimum, that is the solution of a set of linear equations.

The functional J in (5.1) can be viewed as the square of a norm in $\mathbb{R}^n$; define the vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{e} \in \mathbb{R}^n$ and the matrix $A \in \mathbb{R}^{n\times 2}$,

$$\mathbf{x} := \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \mathbf{y} := \begin{pmatrix} x_1 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{e} := \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \quad \text{and} \quad A := (\mathbf{e}\,|\,\mathbf{x}) ,$$

then J can be written as

$$J(a,b) = \|A \begin{pmatrix} a \\ b \end{pmatrix} - \mathbf{y}\,\|^2$$

and we can interprete problem (5.1) as the search for the point in the image of A, nearest to $\mathbf{y}$ (in Euclidean norm).

This can be generalised as follows. Given a matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and a vector $\mathbf{b} \in \mathbb{R}^m$

$$\text{find } \mathbf{x} \in \mathbb{R}^n \text{ such that } \; J(\mathbf{x}) := \|\, A\mathbf{x} - \mathbf{b}\,\|^2 \; \text{ is minimal.} \tag{5.2}$$

Stated otherwise, find the point $\mathbf{x}$ in the image of A ($= Im(A)$) that is nearest to $\mathbf{b}$ (in Euclidean norm) and find its pre-image.
In the sequel we shall assume, that matrix A is of full rank, such that A, considerd as a transformation from $\mathbb{R}^n$ onto $Im(A)$ is one-to-one and invertible.

5.a The normal equations

A simple method for solving problem (5.2) is, to use the geometric argument that the distance

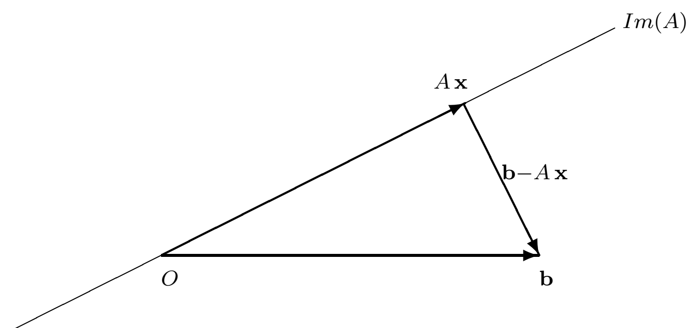

Figure 4: The vector $\mathbf{b}$, its orthogonal projection $A\mathbf{x}$ on $Im(A)$ and the residual $\mathbf{b} - A\mathbf{x}$. The residual is minimal if it is perpendicular to $Im(A)$.

between the vector $\mathbf{b}$ and a vector $\mathbf{y} \in Im(A)$ is minimal if their difference is perpendicular to $Im(A)$. Hence,

$$\|\, A\mathbf{x} - \mathbf{b}\,\|^2 \;\text{ is minimal} \quad \Leftrightarrow \quad A\mathbf{x} - \mathbf{b} \perp Im(A) \quad \Leftrightarrow \quad \mathbf{z}^T A^T (A\mathbf{x} - \mathbf{b}) = 0 \quad \text{for all } \mathbf{z} \in \mathbb{R}^n.$$

As a consequence, the search for the minimum of (5.2) is equivalent with the solution of the linear system

$$A^T A\mathbf{x} = A^T\mathbf{b}\,. \tag{5.3}$$

These equations are called the <u>normal eqations</u> corresponding to the least squares problem (5.2).

If $A \in \mathbb{R}^{m\times n}$ ($m>n$) is of full column rank, $rank(A) = n$, then $A^T A$ is symmetric and positive definite and (5.2) has a unique solution that can be computed by calculating first $A^T A$ and $A^T\mathbf{b}$ and subsequently solving system (5.3) using a Cholesky decomposition (a variant of Gauss or Crout elimination for a positive definite symmetric matrix, that requires only half of the number flops).

This solution method via the normal equations will work also if $m = n$ and, hence, if A is square and invertible (because we assume it is of full rank). The least squares problem (5.2) and

its solution method via the normal equations (5.3) is then equivalent to the solution of problem $A\mathbf{x} = \mathbf{b}$. However, the condition numbers of both problems (w.r.t. the $\|\cdot\|_2$–norm)

$$\kappa_2(A) = \frac{\sigma_1}{\sigma_n} \qquad \text{and} \qquad \kappa_2(A^T A) = \frac{\sigma_1^2}{\sigma_n^2} \tag{5.4}$$

where σ_1 and σ_n are the largest and smallest singular vaues of A, may be quite different. The condition number of the problem, we have to solve using the normal equations, is the square of the condition number of the original problem. If the condition number of A is already large, its square may be huge, making the solution of the normal equations completely unreliable.

We also can define a condition number for problem (5.2) in case A is of full rank but not square. In that case A is a one-to-one mapping onto $Im(A)$; its restriction to a transformation from $\mathbb{R}^n$ to $Im(A)$ has a well-defined inverse. So we can solve the least squares problem (5.2) (in theory) computing first the orthogonal projection $\mathbf{y}$ of $\mathbf{b}$ on $Im(A)$ and solving subsequently the consistent system of equations $A\mathbf{x} = \mathbf{y}$. De sensitivity of this problem is then characterised by the condition number $\kappa_2(A) = \sigma_1/\sigma_n$, where σ_1 and σ_n are (again) the largest and smallest singular values of the restriction of A. Clearly we should define the condition number of the least squares problem (5.2) (w.r.t. the $\|\cdot\|_2$–norm) as the condition number of this restriction. In this view, the solution of the least squares problem (5.2) from the normal equations **always** implies squaring of the condition number. If A is well conditioned, this is no problem. However, if A is badly conditioned, this may produce a "solution" that is completely unreliable. There are several methods to circumvent this squaring by exploiting the idea of orthogonality.

5.b The method of Gram-Schmidt

The method of normal equations (5.3) is derived from the observation, that the residual $\mathbf{b} - A\mathbf{x}$ is perpendicular to $Im(A)$. The construction of an orthogonal basis in $Im(A)$ provides an easy method for the computation of the orthogonal projection $\mathbf{y}$ of $\mathbf{b}$ on $Im(A)$ and the computation of the solution of the compatible system of equation $A\mathbf{x} = \mathbf{y}$. This can be accomplished by the method of Gram-Schmidt. The columns of the matrix $A = (\mathbf{a}_1 \,|\, \cdots \,|\, \mathbf{a}_n)$ form a basis in $Im(A)$; the method of Gram-Schmidt computes from this an orthogonal basis as follows:

Normalise the first column of A and denote it by $\mathbf{q}_1$,
$\mathbf{q}_1 := \mathbf{a}_1/\|\mathbf{a}_1\|$;
for $k = 2 : n$, **do**
  Orthogonalise the k–th column of A w.r.t. all previous ones (i.e. $\perp$ to $\{\mathbf{q}_1 \cdots \mathbf{q}_{k-1}\}$),
  $\widehat{\mathbf{a}}_k := \mathbf{a}_k - \sum_{j=1}^{k-1} \mathbf{q}_j^T \mathbf{a}_k \, \mathbf{q}_j$; (5.5)
  Normalise the result and denote it by $\mathbf{q}_k$,
  $\mathbf{q}_k := \widehat{\mathbf{a}}_k/\|\widehat{\mathbf{a}}_k\|$;
end

This produces the orthonormal basis $\{\mathbf{q}_1 \cdots \mathbf{q}_n\}$ for $Im(A)$. The relation between the vectors of the original basis $\{\mathbf{a}_1 \cdots \mathbf{a}_n\}$ and those of the new one is

$$\mathbf{a}_k = \sum_{j=1}^{k-1} \mathbf{q}_j^T \mathbf{a}_k \, \mathbf{q}_j + \|\widehat{\mathbf{a}}_k\| \, \mathbf{q}_k \,, \tag{5.6}$$

or in matrix notation,

$$A = Q\,R \quad \textit{with} \quad Q := \Big(\mathbf{q}_1 \,|\, \cdots \,|\, \mathbf{q}_n\Big), \quad R = \Big(r_{jk}\Big), \quad r_{jk} = \begin{cases} \mathbf{q}_j^T \mathbf{a}_k & \text{if} \quad j < k\,, \\ \|\widehat{\mathbf{a}}_k\| & \text{if} \quad j = k\,, \\ 0 & \text{if} \quad j > k\,. \end{cases} \tag{5.7}$$

Using this new basis the projection of $\mathbf{b}$ in $Im(A)$ is given by $\mathbf{y} = \sum_{k=1}^{n}(\mathbf{q}_k^T\mathbf{b})\,\mathbf{q}_k$. With this projection the system $A\mathbf{x} = \mathbf{y}$ is compatible because $\mathbf{y} \in Im(A)$, but this system cannot be solved in practice because round-off errors may drive the computed projection outside $Im(A)$. The practical solution method comes from the fact that R is the matrix of the (abstract) transformation A with respect to this new basis of $Im(A)$ and that the coefficients of $\mathbf{y}$ in this basis are $\{\mathbf{q}_1^T\mathbf{b}\,,\,\cdots\,,\,\mathbf{q}_n^T\mathbf{b}\}$. So we only have to solve $R\mathbf{x} = Q^T\mathbf{b}$, which is easy because R is upper triangular.

Summarizing, we find the (GS) algorithm: Compute by the Gram-Schmidt method (5.5) a decomposition of the matrix $A = QR$ into factors $Q \in \mathbb{R}^{m\times n}$ with orthonormal columns and $R \in \mathbb{R}^{n\times n}$ upper triangular and solve the upper triangular system $R\mathbf{x} = Q^T\mathbf{b}$ by (4.25). We can easily prove the correctness. Since R is invertible we have

$$\min_{\mathbf{x}\in\mathbb{R}^n} \|\,A\mathbf{x} - \mathbf{b}\| = \min_{\mathbf{x}\in\mathbb{R}^n} \|\,QR\mathbf{x} - \mathbf{b}\| = \min_{\mathbf{x}\in\mathbb{R}^n} \|\,Q\mathbf{z} - \mathbf{b}\|\,. \tag{5.8}$$

The minimum of the right-hand side is given by the normal equations $Q^TQ\mathbf{z} = \mathbf{z} = Q^T\mathbf{b}$ (which are perfectly conditioned with condition number 1). Hence, the minimum of the original problem is given by the solution of $R\mathbf{x} = \mathbf{z} = Q^T\mathbf{b}$.

In practice it is observed that this Gram-Schmidt method (5.5) may be quite sensitive to rounding errors, in particular if the angles between the columns of A are small. We may strongly improve on this by the following heuristics. In the algorithm we have to orthogonalise $\mathbf{a}_k$ with respect to all predecessors $\{\mathbf{q}_1\ \cdots\ \mathbf{q}_{k-1}\}$ and, hence to compute the inner products of $\mathbf{a}_k$ with all those vectors. In example 4 of section 3.d we have derived the estimate (3.8) for the absolute rounding error in the computed value of an inner product

$$|\,\mathbf{fl}(\mathbf{q}_j^T\mathbf{a}_k) - \mathbf{q}_j^T\mathbf{a}_k\,| \ \le n\eta\,\|\mathbf{q}_j\|_2\,\|\mathbf{a}_k\|_2 = n\eta\,\|\mathbf{a}_k\|_2 \quad \text{because} \quad \|\mathbf{q}_j\|_2 = 1\,.$$

Let us consider the sequential computation of $\mathbf{q}_1^T\mathbf{a}_k$, $\mathbf{q}_2^T\mathbf{a}_k$, etc. for some $k > 2$. When we have finished the computation of the inner product $\mathbf{q}_1^T\mathbf{a}_k$ and we are to begin the calculation of the next inner product $\mathbf{q}_2^T\mathbf{a}_k$, we may also use the formula $\mathbf{q}_2^T(\mathbf{a}_k - \alpha\mathbf{q}_1)$, which should be independent of α, because $\mathbf{q}_2^T\mathbf{q}_1 = 0$ by definition. In a rounding environment this independence is likely to be lost and a good choice of α may make the difference. For the absolute error in the computed value we have the α–dependent upper bound

$$|\,\mathbf{fl}(\mathbf{q}_2^T(\mathbf{a}_k - \alpha\mathbf{q}_1)) - \mathbf{q}_2^T(\mathbf{a}_k - \alpha\mathbf{q}_1)\,| \ \le n\eta\,\|\mathbf{a}_k - \alpha\mathbf{q}_1\|_2\,, \tag{5.9}$$

which is minimal if $\mathbf{a}_k - \alpha\mathbf{q}_1 \ \perp\ \mathbf{q}_1$. So we can minimize the upper bound (5.9) when we orthogonalise $\mathbf{a}_k$ w.r.t. $\mathbf{q}_1$ *before* we start the computation of the inner product with $\mathbf{q}_2$. Next we orthogonalise $\mathbf{a}_k$ w.r.t. $\mathbf{q}_2$ before we compute the inner product with $\mathbf{q}_2$, and so on. Minimisation of an upper bound for the error obviously does not imply that the actual error is minimal, but this strategy is the best feasible. This strategy results in a numerically stable variant of (5.9), which has been dubbed MGS or “Modified Gram Schmidt” in literature:

$$\begin{array}{l}
r_{11} := \|\mathbf{a}_1\|\,;\quad \mathbf{q}_1 := \mathbf{a}_1/r_{11}\,;\\
\textbf{for}\ \ k = 2:n\,,\ \ \textbf{do}\\
\quad \textbf{for}\ \ j = 1:k-1\,,\ \ \textbf{do}\\
\qquad r_{jk} = \mathbf{q}_j^T\mathbf{a}_k\,;\quad \mathbf{a}_k := \mathbf{a}_k - r_{jk}\,\mathbf{q}_j\,;\\
\quad \textbf{end}\\
\quad r_{kk} := \|\,\mathbf{a}_k\,\|\,;\quad \mathbf{q}_k := \mathbf{a}_k/r_{kk}\,;\\
\textbf{end}
\end{array} \tag{5.10}$$

In the computation of $\mathbf{c} := Q^T\mathbf{b}$ we obviously have to apply the same idea:

$$\textbf{for}\ \ k = 1:n\,,\ \ \textbf{do}\ \ c_k := \mathbf{q}_k^T\mathbf{b}\,;\ \ \mathbf{b} := \mathbf{b} - c_k\mathbf{q}_k\,;\ \textbf{end}\,, \tag{5.11}$$

and also here we have to orthogonalise immediately after the computation of each inner product. It can be shown, that MGS is a numerically stable algorithm that produces a (computed) QR-decomposition that is the exact decomposition of a neighbouring matrix. This implies that the rounding errors in the solution $\mathbf{x}$ are dominated by those produced by the solution of the triangular system $R\mathbf{x} = Q^T\mathbf{b}$, which are bounded by the condition number of R or, equivalently, the condition number of A. We conclude that MGS should always be preferred to normal equations, although it is somewhat more expensive in flop count.

Exercises.

1. Show that the solution of (5.2) by MGS requires $2mn^2 + O(n^2)$ flops and that its solution by normal equations requires "only" $mn(n+1) + n^3/3 + O(n^2)$ flops.
2. We can do the MGS computations in a different order,

$$
\begin{array}{l}
\textbf{for}\ \ k = 1:n\,,\ \ \textbf{do} \\
\quad r_{kk} := \|\,\mathbf{a}_k\,\|\,;\quad \mathbf{q}_k := \mathbf{a}_k / r_{kk}\,; \\
\quad \textbf{for}\ \ j = k+1:n\,,\ \ \textbf{do} \\
\quad\quad r_{kj} = \mathbf{q}_k^T \mathbf{a}_j\,;\quad \mathbf{a}_j := \mathbf{a}_j - r_{kj}\,\mathbf{q}_k\,; \\
\quad \textbf{end} \\
\textbf{end}\,.
\end{array}
\tag{5.12}
$$

   In the k–th stage we first normalise $\mathbf{a}_k$ into the vector $\mathbf{q}_k$ and we subsequently normalise the remaining columns $\mathbf{a}_{k+1} \cdots \mathbf{a}_n$ with respect to this vector $\mathbf{q}_k$. Obviously, we may overwrite the columns of Q by those of A in practice.
   Show that this order is equivalent to (5.10), inclusive the MGS-idea that immediately after each inner product computation the vector is orthogonalised w.r.t. this vector.
3. We can obtain a further improvement of the accuracy (mainly useful if the matrix is rank-deficient in order to produce a "*rank revealing decomposition*"), when in each stage k we assign to $\mathbf{q}_k$ the largest of the remaining columns $\mathbf{a}_k \cdots \mathbf{a}_n$ and orthogonalise the other columns with respect to this one. The usual strategy is to interchange the largest column with the k–th and store its index for later use. This implies that we have to know the length of all remaining vectors at the begin of each stage. At first sight the computation of all those norms in each stage requires $2m(n-k+1)$ flops in the k-th stage. However, using the Pythagoras theorem we can derive the square of the the norm of a vector in stage $k+1$ from its value in stage k using only 2 flops. This results in a decomposition of the form $A = QRP$ where P is a permutation matrix.

   In order to derive a correct implementation of those interchanges, we have a more precise look at algorithm (5.12). In the k-th stage we add the upper index (k) to the update of the j-th column, denoting this update by $\mathbf{a}_j^{(k)}$; hence, $\mathbf{a}_j^{(0)}$ is equal to the j-th column of the original matrix A. We define for $k = 0\,,\ \cdots\,,\ n$ intermediate results

$$
\begin{aligned}
Q^{(k)} &:= (\,\mathbf{q}_1\,|\,\cdots\,|\,\mathbf{q}_k\,|\,\mathbf{0}\,|\,\cdots\,|\,\mathbf{0}\,) \in I\!R^{m\times n}\,, \\
A^{(k)} &:= \left(\mathbf{0}\,|\,\cdots\,|\,\mathbf{0}\,|\,\mathbf{a}_{k+1}^{(k)}\,|\,\cdots\,|\,\mathbf{a}_n^{(k)}\right) \in I\!R^{m\times n} \quad \text{and} \\
R^{(k)} &:= \begin{pmatrix}
r_{11} & \cdots & \cdots & \cdots & \cdots & r_{1n} \\
0 & \ddots & & & & \vdots \\
\vdots & \ddots & r_{kk} & \cdots & \cdots & r_{kn} \\
\vdots & & \ddots & 0 & \cdots & 0 \\
\vdots & & & \ddots & \ddots & \vdots \\
0 & \cdots & \cdots & \cdots & 0 & 0
\end{pmatrix} \in I\!R^{n\times n}\,.
\end{aligned}
\tag{5.13}
$$

   This implies, that $Q^{(0)}$, $R^{(0)}$ en $A^{(n)}$ are null matrices and that $A^{(0)} = A$ is the original

matrix. Hence, $A = Q^{(0)}\, R^{(0)} + A^{(0)}$ is true at the beginning of the algorithm:

$$
\begin{array}{l}
\textbf{for}\ \ k = 1:n\,,\ \ \textbf{do} \\
\quad r_{kk} := \|\,\mathbf{a}_k^{(k-1)}\,\|\,;\quad \mathbf{q}_k := \mathbf{a}_k^{(k-1)}/r_{kk}\,; \\
\quad \textbf{for}\ \ j = k+1:n\,,\ \ \textbf{do} \\
\qquad r_{kj} = \mathbf{q}_k^T \mathbf{a}_j^{(k-1)}\,;\quad \mathbf{a}_j^{(k)} := \mathbf{a}_j^{(k-1)} - r_{kj}\,\mathbf{q}_k\,; \\
\quad \textbf{end} \\
\quad \left\{ \mathbf{a}_j^{(k-1)} = \mathbf{q}_k\, r_{kj} + \mathbf{a}_j^{(k)}\ \ \text{voor}\ j = k{+}1 \cdots n \text{ en dus geldt }\ A = Q^{(k)}\, R^{(k)} + A^{(k)} \right\} \\
\textbf{end}
\end{array}
\tag{5.14}
$$

In each stage we add a column to Q and a row to R, we remove a column from A and we update it such that the equality (the *invariant*) $A = Q^{(k)}\, R^{(k)} + A^{(k)}$ remains true. So we have the equality $A = QR$ at the end.
Applying the column interchange in the k-th stage, we have to bring the longest column of $A^{(k-1)}$ on the k-th position. This accomplished by multiplying it (from the right) by the permutation matrix (4.38). In order to conserve the invariant $A = Q^{(k-1)}\, R^{(k-1)} + A^{(k-1)}$, we have to apply this to every term in it.
Write down the Matlab code for this variant of MGS with column interchanges.

5.c Householder Transformations

Around 1950 A.S. Householder proposed an elegant and numerically stable method for the computation of the solution of the least squares problem (5.2) by orthogonalisation. For a given non-trivial vector $\mathbf{u} \in I\!R^m$ we define the Householder transformation $H_{\mathbf{u}} \in I\!R^{m\times m}$ by

$$
H_{\mathbf{u}} := I - \frac{2\mathbf{u}\,\mathbf{u}^T}{\mathbf{u}^T\mathbf{u}}\,, \qquad I \text{ is the identity matrix.} \tag{5.15}
$$

This transformation satisfies the following properties:

a. $H_{\mathbf{u}}$ is symmetric and orthogonal,

$$
H_{\mathbf{u}} = H_{\mathbf{u}}^T \quad \text{and} \quad H_{\mathbf{u}}^T H_{\mathbf{u}} = I - \frac{4\mathbf{u}\,\mathbf{u}^T}{\mathbf{u}^T\mathbf{u}} + \frac{4\mathbf{u}\,\mathbf{u}^T\,\mathbf{u}\,\mathbf{u}^T}{(\mathbf{u}^T\mathbf{u})^2} = I
$$

b. $H_{\mathbf{u}}$ maps $\mathbf{u}$ onto $-\mathbf{u}$ and leaves all vectors in $\mathbf{u}^\perp$ invariant, $H_{\mathbf{u}}\mathbf{v} = \mathbf{v}$ if $\mathbf{v} \perp \mathbf{u}$. A vector $\mathbf{w} \in I\!R^m$ can be decomposed into a component parallel to $\mathbf{u}$ and a component perpendicular to $\mathbf{u}$. The first component is mapped onto minus itself and the second part remains invariant. Hence, $H_{\mathbf{u}}$ is a reflection of the space with respect to the plane perpendicular to $\mathbf{u}$, see figure 5.

The vector $\mathbf{u}$ is called the Householder vector or the reflection vector. For a given reflection vector $\mathbf{u}$ we can compute the image $H_{\mathbf{u}}\mathbf{w}$ of a vector $\mathbf{w} \in I\!R^m$.

We can go the other way around an ask for a reflection vector $\mathbf{u}$ such that the corresponding Householder transformation maps a given vector $\mathbf{w}$ onto the mirror image $\mathbf{v}$. Obviously, those vectors should have the same length $\|\,\mathbf{v}\,\| = \|\,\mathbf{w}\,\|$ because the reflection is orthogonal. If this is true, we see from figure 5, that $\mathbf{v}$ is the mirror image of $\mathbf{w}$ if the "mirror" is the bisecting (hyper)plane of $\mathbf{v}$ and $\mathbf{w}$. This plane is $(\mathbf{v}-\mathbf{w})^\perp$, the subspace of all vectors perpendicular to the difference $\mathbf{v}-\mathbf{w}$. Indeed, if $\|\mathbf{v}\| = \|\mathbf{w}\|$ and $\mathbf{z}^T(\mathbf{v}-\mathbf{w}) = 0$, the difference of the squares of the distances from $\mathbf{z}$ to $\mathbf{v}$ and $\mathbf{w}$ satisfies:

$$
\|\mathbf{v}-\mathbf{z}\|^2 - \|\mathbf{w}-\mathbf{z}\|^2 = \|\mathbf{v}\|^2 - 2\mathbf{z}^T\mathbf{v} + \|\mathbf{z}\|^2 - \|\mathbf{w}\|^2 + 2\mathbf{z}^T\mathbf{w} - \|\mathbf{z}\|^2 = 0\,.
$$

With the help of a sequence of such Householder transformations or reflections we can transform a matrix $A \in I\!R^{m\times n}$ into upper triangular form in a way analogous to Gaussian elimination.

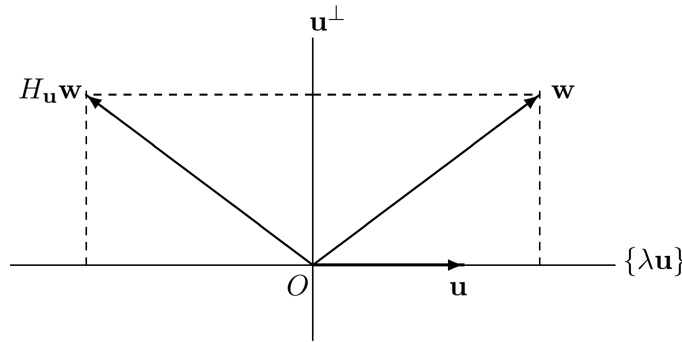

Figure 5: A vector $\mathbf{w}$, its decomposition in a component parallel to the reflection vector $\mathbf{u}$ and a component perpendicular to it and the mirror image $H_\mathbf{u}\mathbf{w}$ with respect to the (hyper)plane perpendicular to $\mathbf{u}$.

In the k–th stage of Gaussian elimination the matrix is multiplied from the left by a Gaussian transformation G_k, that sets to zero all elements of the k–th column below the main diagonal, see (4.36). The same can be done by a suitable Householder transformation.

Let us consider the first stage; we want to find a reflection vector $\mathbf{u}_1$ such that $H_{\mathbf{u}_1}$ maps the first column $\mathbf{a}_1$ of A on $(\alpha,\, 0,\, \cdots\, 0)^T = \alpha\, \mathbf{e}_1$, a vector whose components are all zero except for the first. Because the length is invariant, we may choose α in two ways, $\alpha = \pm\|\mathbf{a}_1\|$. Hence, the two reflection vectors are $\mathbf{a}_1 \mp \|\mathbf{a}_1\|\, \mathbf{e}_1$. Those vectors differ from $\mathbf{a}_1$ in their first components only. So we are able to choose the sign in the first component $a_{11} \mp \|\mathbf{a}_1\|$ in such a way, that no loss of significance is occurring. So we choose *plus* if both terms have the same sign and *minus* if the signs are opposite[4],

$$\mathbf{u}_1 = \mathbf{a}_1 + \operatorname{sign}(a_{11})\|\mathbf{a}_1\|\, \mathbf{e}_1\,, \tag{5.16}$$

such that

$$\begin{aligned}
\mathbf{u}_1^T\mathbf{u}_1 &= \mathbf{a}_1^T\mathbf{a}_1 + 2\operatorname{sign}(a_{11})\|\mathbf{a}_1\|\, \mathbf{e}_1^T\mathbf{a}_1 + \|\mathbf{a}_1\|^2\, \mathbf{e}_1^T\mathbf{e}_1 = 2\|\mathbf{a}_1\|(\|\mathbf{a}_1\| + |\, a_{11}\,|)\,,\\
\mathbf{u}_1^T\mathbf{a}_1 &= \mathbf{a}_1^T\mathbf{a}_1 + \operatorname{sign}(a_{11})\|\mathbf{a}_1\|\, \mathbf{e}_1^T\mathbf{a}_1 = \|\mathbf{a}_1\|(\|\mathbf{a}_1\| + |\, a_{11}\,|)\,,\\
H_{\mathbf{u}_1}\mathbf{a}_1 &= \mathbf{a}_1 - \frac{2\mathbf{u}_1^T\mathbf{a}_1}{\mathbf{u}_1^T\mathbf{u}_1}\,\mathbf{u}_1 = -\operatorname{sign}(a_{11})\|\mathbf{a}_1\|\, \mathbf{e}_1\,.
\end{aligned} \tag{5.17}$$

This implies:

$$H_{\mathbf{u}_1} A = \begin{pmatrix} \alpha_1 & \widetilde{a}_{12} & \cdots & \widetilde{a}_{1n} \\ 0 & \widetilde{a}_{22} & \cdots & \widetilde{a}_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & \widetilde{a}_{m2} & \cdots & \widetilde{a}_{mn} \end{pmatrix}, \qquad \alpha_1 = -\operatorname{sign}(a_{11})\,.$$

In an analogous way we can transform the elements $\{a_{32} \cdots a_{m2}\}$ of the second column to zero by a reflection that maps

$$\widehat{\mathbf{a}}_2 := \begin{pmatrix} 0 \\ \widetilde{a}_{22} \\ \vdots \\ \widetilde{a}_{m2} \end{pmatrix} \quad \text{onto} \quad \begin{pmatrix} 0 \\ \alpha_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \text{using the reflection vector } \mathbf{u}_2 = \widehat{\mathbf{a}}_2 - \alpha_2\|\widehat{\mathbf{a}}_2\|\, \mathbf{e}_2\,,\ \ \alpha_2 = -\operatorname{sign}(\widetilde{a}_{22}).$$

Here, it should be clear that the first component of $\mathbf{u}_2$ has to be zero, since the first row of $H_{\mathbf{u}_1} A$ should not change any more in the multiplication by $H_{\mathbf{u}_2}$.

[4]The function “sign” here has the value $+1$ if its argument is non-negative and the value -1 otherwise. The Matlab function `sign` cannot be used since it is zero if its argument happens to be zero.

Continuing in this way we find n reflection vectors $\mathbf{u}_1, \cdots, \mathbf{u}_n$ and the associated Householder transformations transform A into an upper triangular matrix R,

$$H_{\mathbf{u}_n} \cdots H_{\mathbf{u}_1} A = \begin{pmatrix} r_{11} & \cdots & r_{1n} \\ 0 & \ddots & \vdots \\ \vdots & \ddots & r_{nn} \\ \vdots & & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix} =: R \tag{5.18}$$

such that

$$A = Q\,R \quad \text{with} \quad Q := H_{\mathbf{u}_1} \cdots H_{\mathbf{u}_n} \in I\!R^{m\times m} \text{ an orthogonal matrix.} \tag{5.19}$$

Also in this way we find a QR–decomposition of A. However, it differs strongly from (5.6). In (5.19) Q is an orthogonal (and hence square) matrix and R has the same dimensions as A has; in contrast, MGS only computes a matrix Q with orthonormal columns and a square matrix R.

In order to find the solution of the original least squares problem (5.2) using this Householder decomposition, we split R in the *square* upper triangular matrix $R_1 \in I\!R^{n\times n}$ consisting of the first n (non-trivial) rows of R and an $(m-n)\times n$ submatrix consisting of zeros,

$$R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}.$$

If A is of full rank, R_1 is invertible. The LS problem (5.2) is solved in the following way. Since the norm is invariant under othogonal transformations, we have:

$$\|\, A\mathbf{x} - \mathbf{b} \,\|^2 = \|\, Q\,R\mathbf{x} - \mathbf{b} \,\|^2 = \|\, R\mathbf{x} - Q^T\mathbf{b} \,\|^2 .$$

Partitioning the vectors $R\mathbf{x} = \begin{pmatrix} R_1\mathbf{x} \\ 0 \end{pmatrix}$ and $Q^T\mathbf{b} = \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \end{pmatrix}$ in two vectors consisting of the first n components and the remaining $m-n$ components respectively, we find

$$\|\, R\mathbf{x} - Q^T\mathbf{b} \,\|^2 = \|\, \begin{pmatrix} R_1\mathbf{x} \\ 0 \end{pmatrix} - \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \end{pmatrix} \,\|^2 = \|\, R_1\mathbf{x} - \mathbf{c} \,\|^2 + \|\mathbf{d}\|^2 .$$

The right-hand side is minimized by the solution of $R_1\mathbf{x} = \mathbf{c}$ (provided A is of full rank); this solution solves the least squares problem (5.2). The residual is $\mathbf{d}$.

The factorisation algorithm is:

for $k = 1 : n$,
  {Transform the part $a_{k+1,k} \cdots a_{m,k}$ of the k-th column to zero using a suitable Householder transformation;}
  $\alpha := \sqrt{\sum_{j=k}^{m} a_{jk}^2}\,;$ {norm of the relevant vector}
  $\mathbf{u}_k := (0 \cdots 0\,,\, a_{kk} + \alpha\,\mathrm{sign}(a_{kk})\,,\, a_{k+1,k} \cdots a_{mk})^T\,;$ {reflection vector}
  $a_{kk} := -\,\mathrm{sign}(a_{kk}) * \alpha\,,$
  $\gamma := \alpha\,|\,u_{kk}\,|\,,$ $\{= \frac{1}{2} \times$ norm of $\mathbf{u}_k\}$ (5.20)
  for $j = k+1 : n\,,$ {apply the Householder transformation}
    $\mathbf{a}_j = \mathbf{a}_j - \mathbf{u}_k\,\mathbf{u}_k^T\mathbf{a}_j/\gamma\,;$ {to the remaining columns of A}
  end
  $\mathbf{b} = \mathbf{b} - \mathbf{u}_k\,\mathbf{u}_k^T\mathbf{b}/\gamma\,;$ {apply the transformation to the right-hand side $\mathbf{b}$}
end

After termination of the algorithm, the upper triangle of A contains the relevant (non-zero) elements of R and $\mathbf{b}$ contains the transformed vector $Q^T\mathbf{b}_{\text{original}}$. It remains to solve the upper triangular system for the solution of the LS-problem. Clearly, it is not necessary to compute the orthogonal matrix Q explicitly.

Remark. In case it is useful to store Q for later use, the best strategy is to store rescaled reflection vectors; this uses less memory and much less flops than the explicit computation of Q does. After the application of the reflection to A in the k-th stage, the content of the part $a_{k+1,k} \cdots a_{m,k}$ of the k–th column of A has become irrelevant. Hence, those (memory) locations can be used to store the k+1–st up to m–th components of the k-th reflection vector divided by $u_{k,k}$. By this division the k–th component is set to 1 and needs not be stored. The first up to k−1–st element are zero anyway. This way of storing Q in the form of the reflection vectors is referred to as "in factored form".

Exercises.

1. Show that algorithm (5.20) uses $2n^2(m - n/3) + O(mn)$ flops; so it is more costly than the method of normal equations is but less than MGS.
   Find the additional number of flops that is needed to compute Q explicitly.
   Compare the number of flops used for the computation of $Q^T\mathbf{b}$ with explicit and factored Q.
2. Write the code for algorithm (5.20) in matlab.
3. In the same way as in MGS (exercise 3 in section 6c) we can improve numerical stability by column interchanges. In the k–th stage we interchange the k–th column with the longest of the remaining columns (with indices k up to n). Since the k–th reflection works only on the sub-vector consisting of the k–th up to m–th elements, we have to restrict the norms to those elements. The norms of the remaining columns need not be recomputed in each stage; since the number of relevant elements of the columns deminishes by one in each stage, we simply can update the (squares of) each column norm with two flops. This improvement results in a (*rank revealing*) factorisation $A = QRP$ of S in the product of an orthogonal matrix Q, an upper triangular matrix R and a permutation matrix P.
   Write the code of the algorithm in Matlab.
4. The pseudo-inverse or Moore Penrose inverse of a matrix $A \in I\!R^{m\times n}$, denoted by $A^\dagger$, is defined via the SVD. If

   $$A = U\Sigma V^T \quad \text{where} \quad \Sigma = diag(\sigma_1 \cdots \sigma_p) \in I\!R^{m\times n} \quad \text{and} \quad p := \min\{m, n\}$$

   is the singular value decomposition and if

   $$rank(A) = r\,, \quad \text{such that} \quad \sigma_1 \geq \cdots \geq \sigma_r > 0 \quad \text{and} \quad \sigma_{r+1} = \cdots = \sigma_p = 0\,,$$

   then the pseudo inverse is defined as

   $$A^\dagger := V\Sigma^\dagger U^T \quad \text{where} \quad \Sigma^\dagger := diag(\sigma_1^{-1} \cdots \sigma_r^{-1}\,, 0 \cdots 0) \in I\!R^{n\times m}\,. \tag{5.21}$$

   Prove the following properties:

   a. $A^\dagger$ is map from $I\!R^m$ to $I\!R^n$ with kernel $Ker(A^\dagger) = Im(A)^\perp$
      and image $Im(A^\dagger) = Ker(A)^\perp$.
   b. $A^\dagger A$ and $AA^\dagger$ are orthogonal projections on $Im(A)$ and $Ker(A)^\perp$ respectively.
   c. $AA^\dagger A = A$ and $A^\dagger AA^\dagger = A^\dagger$.
   d. If $m \geq n$ and $rank(A) = n$ then $A^\dagger\mathbf{b}$ is the solution of the least squares problem (5.2).
   e. If $rank(A) < n$ (or $m < n$), then $A^\dagger\mathbf{b}$ is the solution of the least squares problem (5.2) that has minimal norm, i.e. $A^\dagger\mathbf{b} = \text{argmin}_{\mathbf{x}\in I\!R^n} \{\, \|\mathbf{x}\| \,\big|\, \mathbf{x} = \text{argmin}_{\mathbf{y}\in I\!R^n} \|A\mathbf{y} - \mathbf{b}\|^2 \,\}$.

   Remark: Equivalently, the pseudo inverse $A^\dagger$ can be defined as the (unique) matrix satisfying (b) and (c) (the Penrose conditions) and (5.21) is then a consequence.

5.d Givens rotations

A third (often used) way to factorise a matrix A into a product of an orthogonal transformation and an upper triangular matrix works with Givens rotations. The idea is easily explained in $\mathbb{R}^2$.

The rotation of a vector $(x,y)^T \in \mathbb{R}^2$ over an angle φ is given by the matrix

$$J(\varphi) := \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} \quad \text{such that} \quad J(\varphi)\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\varphi + y\sin\varphi \\ y\cos\varphi - x\sin\varphi \end{pmatrix}. \tag{5.22}$$

For a given vector $(x,y)^T \in \mathbb{R}^2$ the angle φ can be chosen such that the second component of the image is zero, i.e.

$$J(\varphi)\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x\cos\varphi + y\sin\varphi \\ y\cos\varphi - x\sin\varphi \end{pmatrix} = \begin{pmatrix} z \\ 0 \end{pmatrix} \quad \text{such that} \quad y\cos\varphi - x\sin\varphi = 0\,. \tag{5.23}$$

The cosine and the sine can be computed in two ways,

$$\begin{cases} y^2(1-\sin^2\varphi) = x^2\sin^2\varphi \quad \text{such that} \quad \sin\varphi = \pm\dfrac{y}{\sqrt{x^2+y^2}} \quad \text{and} \quad \cos\varphi = \dfrac{x\sin\varphi}{y}\,, \\[2ex] y^2\cos^2\varphi = x^2(1-\cos^2\varphi) \quad \text{such that} \quad \cos\varphi = \pm\dfrac{x}{\sqrt{x^2+y^2}} \quad \text{and} \quad \sin\varphi = \dfrac{y\cos\varphi}{x}\,, \end{cases} \tag{5.24}$$

provided $y \neq 0$ or $x \neq 0$ respectively. We see that it is of no use to compute the angle φ explicitly in the computation of the rotation matrix. It suffices to compute $c := \cos\varphi$ and $s := \sin\varphi$ via one of both formulae in (5.24). Moreover, the choice of the sign is free. We may use this sign to make the first component z of the image positive. In order to minimize the number of flops and the rounding errors in the computation of c and s, this computation most often is implemented as follows (provided x and y are not both zero):

$$\begin{array}{l} \textbf{if} \quad |\,x\,| \geq |\,y\,| \quad \textbf{then} \\ \qquad t := \dfrac{y}{x}\,;\; c := \dfrac{\text{sign}(x)}{\sqrt{1+t^2}}\,;\; s := t * c\,;\; z = \dfrac{x}{c}\,; \\ \textbf{else} \\ \qquad t := \dfrac{x}{y}\,;\; s := \dfrac{\text{sign}(y)}{\sqrt{1+t^2}}\,;\; c := t * s\,;\; z = \dfrac{y}{s}\,; \\ \textbf{end} \end{array} \tag{5.25}$$

A Givens rotation in $\mathbb{R}^m$ is an $m\times m$–matrix of the form

$$J(k,\ell,\varphi) := \begin{pmatrix} 1 &&&&&&&&&& \\ & \ddots &&&&&&&&& \\ && 1 &&&&&&&& \\ &&& c && \cdots && s && \cdots & \\ &&&& 1 &&&&&& \\ &&& \vdots && \ddots && \vdots &&& \\ &&&&&& 1 &&&& \\ &&& -s && \cdots && c && \cdots & \\ &&&&&&&& 1 && \\ &&& \vdots &&&& \vdots && \ddots & \\ &&&&&&&&&& 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \leftarrow \text{row}(k) \\ \\ \\ \\ \leftarrow \text{row}(\ell) \\ \\ \\ \\ \end{matrix} \tag{5.26}$$

$$\qquad\qquad\qquad\qquad \text{column}(k) \qquad \text{column}(\ell)$$

In this matrix all diagonal elements are 1, except those with index (k,k) and (ℓ,ℓ), which are equal to $c := \cos\varphi$. All other matrix elements are zero, except those with index (k,ℓ)– and (ℓ,k),

which are equal to s and $-s$ respectively with $s := \sin\varphi$. This matrix acts as a rotation in in the plane spanned by the k–th and ℓ–th coordinate vectors $\mathbf{e}_k$ and $\mathbf{e}_\ell$.

Such a Givens rotation or "plane rotation" can be used to zero elements of a vector or a matrix selectively. The ℓ–th component of a vector $\mathbf{a} \in I\!R^m$ can be rotated to zero by a Givens rotation $J(k, \ell, \varphi)$, in which the sine and cosine of the rotation angle φ are computed by formula (5.25) with $x = a_k$ and $y = a_\ell$. Using a series of $m-1$ rotations of the form $J(k, k+1, \varphi_k)$ for $k = m-1 : -1 : 1$ we can transform the vector $\mathbf{a}$ to a multiple of the first coordinate vector $\mathbf{e}_1$. This can also be accomplished by a sequence of the form $J(1, k, \vartheta_k)$ in the order $k = 2 : m$ (or $k = m : -1 : 2$). We conclude, that there is a large freedom in the choice of subsequent rotation planes. We have to take care that zeros, once they are created, do not disappear again in subsequent rotations.

In the same way the matrix $A \in I\!R^{m\times n}$ can be transformed to upper trangular form using a series of Givens rotations; e.g. we may chose the order (working along diagonals)

> **for** $k = m-1 : -1 : 1$ **do**
>   **for** $j = 1 : \min(n, m-k)$ **do**
>   Make $a_{k+j,j}$ zero by multiplying A from the left by the rotation $J(k+j-1\,,\ k+j\,,\ \varphi_{kj})$ in the plane spanned by $\mathbf{e}_{k+j-1}$ and $\mathbf{e}_{k+j}$; c and s are to be computed from (5.25) with $x = a_{k+j-1,j}$ and $y = a_{k+j,j}$. (5.27)
>   **end**
> **end**

Here too, we may choose different orders in which we zero out the elements of the lower triangle. The computation of a QR factorisation using Givens rotations requires more flops than those with Householder transformations or MGS. The advantage of Givens rotations is that elements of a matrix can be zeroed selectively. Such an operation only affects the two rows involved and no other rows. This may be very important mainly for **sparse matrices** in which the large majority of matrix elements are zero (e.g. produced by the discretisation of a partial differential equations).

Exercise: 1. Write down the Matlab code for the solution of the least squares problem (5.2) using Givens rotations. Like in the Householder–QR we do not have to store the rotations, if we apply them immediately to the right-hand side.
Show, that this algorithm (5.27) uses $3n^2(m - n/3) + O(mn)$ flops.