Scientific Computing: An Introduction using Maple and MATLAB

Instructor: Prof. Xiaoming Yuan

October 9, 2017

Chapter 1

Finite Precision Arithmetic

Finite precision arithmetic underlies all the computations performed numerically, e.g. in Matlab; only symbolic computations, e.g. Maple, are largely independent of finite precision arithmetic. Historically, when the invention of computers allowed a large number of operations to be performed in very rapid succession, nobody knew what the influence of finite precision arithmetic would be on this many operations: would small rounding errors sum up rapidly and destroy results? Would they statistically cancel? The early days of numerical analysis were therefore dominated by the study of rounding errors, and made this rapidly developing field not very attractive. Fortunately, this view of numerical analysis has since changed, and nowadays the focus of numerical analysis is the study of algorithms for the problems of continuous mathematics . There are nonetheless a few pitfalls every person involved in scientific computing should know, and this chapter is precisely here for this reason. After an introductory example in Section 1.1, we present the difference between real numbers and machine numbers in Section 1.2 on a generic, abstract level, and give for the more computer science oriented reader the concrete IEEE arithmetic standard in Section 1.3. We then discuss the influence of rounding errors on operations in Section 1.4, and explain the predominant pitfall of catastrophic cancellation when computing differences. In Section 1.5, we explain in very general terms what the condition number of a problem is, and then show in Section 1.6 two properties of algorithms for a given problem, namely forward stability and backward stability. It is the understanding of condition numbers and stability that allowed numerical analysts to move away from the study of rounding errors, and to focus on algorithmic development. Sections 1.7 and 1.8 represent a treasure trove with advanced tips and tricks when computing in finite precision arithmetic.

1.1 Introductory Example

A very old problem already studied by ancient Greek mathematicians is the squaring of a circle. The problem consists of constructing a square that has the same area as the unit circle. Finding a method for transforming a circle into a square this way (quadrature of the circle) became a famous problem that remained unsolved until the 19th century, when it was proved using Galois theory that the problem cannot be solved with the straight edge and compass. We know today that the area of a circle is given

by $A = \pi r^2$, where r denotes the radius of the circle. An approximation is obtained by drawing a regular polygon inside the circle, and by computing the surface of the polygon. The approximation is improved by increasing the number of sides. Archimedes managed to produce a 96-sided polygon, and was able to bracket π in the interval $(3\frac{10}{71}, 3\frac{1}{7})$. The enclosing interval has length $1/497 = 0.00201207243-$ surely good enough for most practical applications in his time.

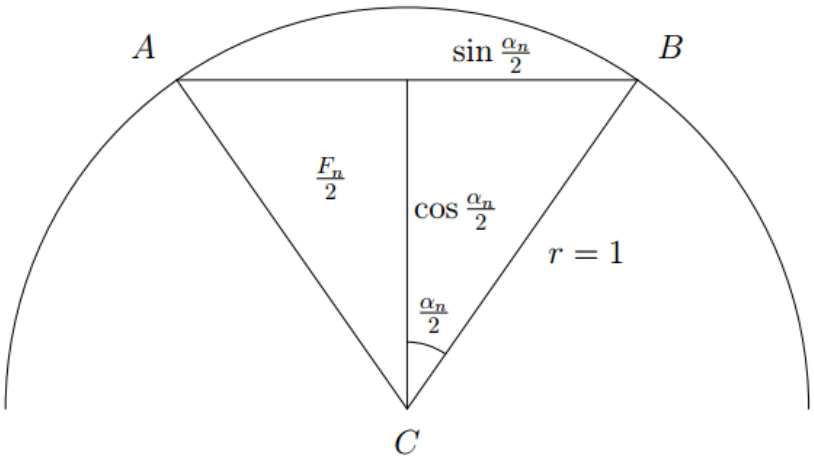


Figure 1.1: Squaring of a circle

To compute such a polygonal approximation of π, we consider Figure 1.1. Without loss of generality, we may assume that $r = 1$. Then the area F_n of the isosceles triangle ABC with center angle $\alpha_n := \frac{2\pi}{n}$ is

$$F_n = \cos\frac{\alpha_n}{2}\sin\frac{\alpha_n}{2},$$

and the area of the associated $n-$sided polygon becomes

$$A_n = nF_n = \frac{n}{2}(2\cos\frac{\alpha_n}{2}\sin\frac{\alpha_n}{2}) = \frac{n}{2}\sin\alpha_n = \frac{n}{2}\sin\frac{2\pi}{n}$$

Clearly, computing the approximation A_n using π would be rather contradictory. Fortunately, A_{2n} can be derived from A_n by simple algebraic transformations, i.e. by expressing $\sin\frac{\alpha_n}{2}$ in terms of $\sin\alpha_n$. This can be achieved by using identities for trigonometric functions:

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}} \tag{1.1}$$

Thus, we have obtained a recurrence for $\sin\frac{\alpha_n}{2}$ from $\sin\alpha_n$. To start the recurrence, we compute the area A_6 of the regular hexagon. The length of each side of the six equilateral triangles is 1 and the angle is $\alpha_6 = 60^o$, so that $\sin\alpha_6 = \frac{\sqrt{3}}{2}$. Therefore, the area of the triangle is $F_6 = \frac{\sqrt{3}}{4}$ and $A_6 = 3\frac{\sqrt{3}}{2}$. We obtain the following program for computing the sequence of approximations A_n:

Algorithm 1 Computation of π, Naive Version

```
s = sqrt(3)/2; A=3*s; n = 6 ;   %\text{initialization}
z =[ A - pi n A s];             % store the results
while s>1e-10                   %termination if s=sin(alpha) small
  s=sqrt((1-sqrt(1-s*s))/2);    % new sin(alpha/2) value
  n=2*n; A=n/2*s;               % A=new polygon area
  z=[z; A-pi n A s];
  end
m=length(z);
for i=1:m
  fprintf('%10d  %20.15f %20.15f %20.15f n', z(i,2), z(i,3), z(i,1),z(i,4))
end
```

The results, displayed in Table 1.1, are not what we would expect: initially, we observe convergence towards π, but for $n > 49152$, the error grows again and finally we obtain $A_n = 0$?! *Although the theory and the program are both correct, we still obtain incorrect answers.* We will explain in this chapter why this is the case.

n	A_n	$A_n - \pi$	$\sin(\alpha_n)$
6	2.598076211353316	−0.543516442236477	0.866025403784439
12	3.000000000000000	−0.141592653589794	0.500000000000000
24	3.105828541230250	−0.035764112359543	0.258819045102521
48	3.132628613281237	−0.008964040308556	0.130526192220052
96	3.139350203046872	−0.002242450542921	0.065403129230143
192	3.141031950890530	−0.000560702699263	0.032719082821776
384	3.141452472285344	−0.000140181304449	0.016361731626486
768	3.141557607911622	−0.000035045678171	0.008181139603937
1536	3.141583892148936	−0.000008761440857	0.004090604026236
3072	3.141590463236762	−0.000002190353031	0.002045306291170
6144	3.141592106043048	−0.000000547546745	0.001022653680353
12288	3.141592516588155	−0.000000137001638	0.000511326906997
24576	3.141592618640789	−0.000000034949004	0.000255663461803
49152	3.141592645321216	−0.000000008268577	0.000127831731987
98304	3.141592645321216	−0.000000008268577	0.000063915865994
196608	3.141592645321216	−0.000000008268577	0.000031957932997
393216	3.141592645321216	−0.000000008268577	0.000015978966498
786432	3.141592303811738	−0.000000349778055	0.000007989482381
1572864	3.141592303811738	−0.000000349778055	0.000003994741190
3145728	3.141586839655041	−0.000005813934752	0.000001997367121
6291456	3.141586839655041	−0.000005813934752	0.000000998683561
12582912	3.141674265021758	0.000081611431964	0.000000499355676
25165824	3.141674265021758	0.000081611431964	0.000000249677838
50331648	3.143072740170040	0.001480086580246	0.000000124894489
100663296	3.137475099502783	−0.004117554087010	0.000000062336030
201326592	3.181980515339464	0.040387861749671	0.000000031610136
402653184	3.000000000000000	−0.141592653589793	0.000000014901161
805306368	3.000000000000000	−0.141592653589793	0.000000007450581
1610612736	0.000000000000000	−3.141592653589793	0.000000000000000

Table 1.1: Unstable computation of π

1.2 Real Numbers and Machine Numbers

Every computer is a finite automaton. This implies that a computer can only store a finite set of numbers and perform only a finite number of operations. In mathematics, we are used to calculating with real numbers $\mathbb{R}$ covering the continuous interval $(-\infty, \infty)$, but on the computer, we must contend with a discrete, finite set of *machine numbers* $\mathbb{M} = \{-\tilde{a}_{min}, \ldots, \tilde{a}_{max}\}$. Hence each real number a has to be mapped onto a machine number $\tilde{a}$ to be used on a computer. In fact a whole interval of real numbers is mapped onto one machine number, as shown in Figure 1.2.

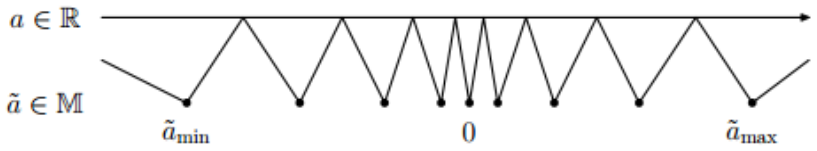


Figure 1.2: Mapping of real numbers $\mathbb{R}$ onto machine numbers $\mathbb{M}$

Nowadays, machine numbers are often represented in the *binary system.* In general, any *base (or radix)* B could be used to represent numbers. A real machine number or *floating point number* consists of two parts, a *mantissa (or significant)* m and an *exponent* e

$$\begin{aligned} \tilde{a} &= \pm m \times B^e \\ m &= D.D\ldots D \qquad \text{mantissa} \\ e &= D\ldots D \qquad \text{exponent} \end{aligned}$$

where $D \in \{0, 1, \ldots, B-1\}$ stands for one *digit.* To make the representation of machine numbers unique (note that e.g. $1.2345 \times 10^3 = 0.0012345 \times 10^6$), we require for a machine number $\tilde{a} \neq 0$ that the first digit before the decimal point in the mantissa be nonzero; such numbers are called *normalized.* One defining characteristic for any *finite precision arithmetic* is the number of digits used for the mantissa and the exponent: the number of digits in the exponent defines the *range of the machine numbers*, whereas the numbers of digits in the mantissa defines the *precision.*

More specifically, a finite precision arithmetic is defined by four integer parameters: B, the base or radix, p, the number of digits in the mantissa, and l and u defining the exponent range: $l \leq e \leq u$.

The *precision* of the *machine* is described by the real machine number *eps.* Historically, *eps is defined to be the smallest positive* $\tilde{a} \in \mathbb{M}$ *such that* $\tilde{a} + 1 \neq 1$ *when the addition is carried out on the computer.* Because this definition involves details about the behavior of floating point addition, which are not easily accessible, a newer definition of *eps* is simply *the spacing of the floating point numbers between 1 and* B (usually $B = 2$). The current definition only relies on how the numbers are represented.

Simple calculators often use the familiar decimal system ($B = 10$). Typically there are $p = 10$ digits for the mantissa and 2 for the exponent ($l = -99$ and $u = 99$). In this finite precision arithmetic, we have

- eps=0.000000001 = 1.000000000 $\times 10^{-9}$,

- the largest machine number

$$\tilde{a}_{max} = 9.999999999 \times 10^{+99},$$

- the smallest machine number

$$\tilde{a}_{min} = -9.999999999 \times 10^{+99},$$

- the smallest(normalized) positive machine number

$$\tilde{a}_{+} = 1.000000000 \times 10^{-99},$$

Early computers, for example the MARK 1 designed by Howard Aiken and Grace Hopper at Harvard and built in 1944, or the ERMETH (Elektronische Rechenmaschine der ETH) constructed by Heinz Rutishauser, Ambros Speiser and Eduard Stiefel, were also decimal machines. The ERMETH, built in 1956, was operational at ETH Zurich from 1956-1963. The representation of a real number used 16 decimal digits: The first digit, the $q-$digit, stored the sum of the digits modulo 3. This was used as a check to see if the machine word had been transmitted correctly from memory to the registers. The next three digits contained the exponent. Then the next 11 digits represented the mantissa, and finally, the last digit held the sign. The range of positive machine numbers was $1.0000000000 \times 10^{-200} \leq \tilde{a} \leq 9.9999999999 \times 10^{199}$. The possibly larger exponent range in this setting from -999 to 999 was not fully used.

In contrast, the very first programmable computer, the Z3, which was built by the German civil engineer Konrad Zuse and presented in 1941 to a group of experts only, was already using the binary system. The Z3 worked with an exponent of 7 bits and a mantissa of 14 bits (actually 15, since the numbers were normalized). The range of positive machine numbers was the interval

$$[2^{-63}, 1.11111111111111 \times 2^{62}] \approx [1.08 \times 10^{-19}, 9.22 \times 10^{18}].$$

In Maple (a computer algebra system), numerical computations are performed in base 10. The number of digits of the mantissa is defined by the variable Digits, which can be freely chosen. The number of digits of the exponent is given by the word length of the computer for $32-$bit machines, we have a huge maximal exponent of $u = 2^{31} = 2147483648$.

1.3 The IEEE Standard

Since 1985 we have for computer hardware the *ANSI/IEEE Standard 754 for Floating Point Numbers*. It has been adopted by almost all computer manufacturers. The base is $B = 2$.

1.3.1 Single Precision

The IEEE single precision floating point standard representation uses a $32-$ bit word with bits numbered from 0 to 31 from left to right. The first bit S is the sign bit, the

next eight bits E are the exponent bits, $e = EEEEEEEE$, and the final 23 bits are the bits F of the mantissa m:

The value $\tilde{a}$ represented by the 32 bit word is defined as follows:

- **normal numbers**: If $0 < e < 255$, then $\tilde{a} = (-1)^S \times 2^{e-127} \times 1.m$, where $1.m$ is the binary number created by prefixing m with an implicit leading 1 and a binary point.
- **subnormal numbers**: If $e = 0$ and $m \neq 0$, then $\tilde{a} = (-1)^S \times 2^{-126} \times 0.m$.
  These are known as *denormalized (or subnormal) numbers.*
  If $e = 0$ and $m = 0$ and $S = 1$, then $\tilde{a} = -0$.
  If $e = 0$ and $m = 0$ and $S = 0$, then $\tilde{a} = 0$
- **exceptions**: If $e = 255$ and $m \neq 0$, then $\tilde{a} = NaN$ (*Not a number*)
  If $e = 255$ and $m = 0$ and $S = 1$, then $\tilde{a} = -Inf$.
  If $e = 255$ and $m = 0$ and $S = 0$, then $\tilde{a} = Inf$.

Some examples:

```
0  10000000 00000000000000000000000 = +1 x 2^(128-127) x 1.0   =  2
0  10000001 10100000000000000000000 = +1 x 2^(129-127) x 1.101 =  6.5
1  10000001 10100000000000000000000 = -1 x 2^(129-127) x 1.101 = -6.5

0  00000000 00000000000000000000000 =  0
1  00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 =  Inf
1 11111111 00000000000000000000000 = -Inf

0  11111111  00000100000000000000000  =  NaN
1  11111111  00100010001001010101010  =  NaN

0  00000001 00000000000000000000000 =  +1 x 2^(1-127) x 1.0 =   2^(-126)
0  00000000 10000000000000000000000 =  +1 x 2^( -126) x 0.1 =   2^(-127)

0  00000000  00000000000000000000001
=    +1  x  2^(-126) x  0.00000000000000000000001 =  2^(-149)
=    smallest  positive  denormalized machine  number
```

In Matlab, real numbers are usually represented in *double precision.* The function **single** can however be used to convert numbers to single precision. Matlab can also print real numbers using the hexadecimal format, which is convenient for examining their internal representations:

```
>> format hex
>>  x=single(2)
x =
40000000
>> 2
ans =
4000000000000000
>>        s=realmin('single')*eps('single')s =
00000001
>>  format  long
>> s s =
1.4012985e-45
>>  s/2 ans =
0
% Exceptions
>>   z=sin(0)/sqrt(0)
Warning: Divide by zero. z =
NaN
>> y=log(0)
Warning: Log of zero. y =
-Inf
>>  t=cot(0)
Warning: Divide by zero.
> In cot at 13 t =
Inf
```

We can see that `x` represents the number 2 in single precision. The functions `realmin` and `eps` with parameter `'single'` compute the machine constants for single precision. This means that `s` is the smallest denormalized number in single precision. Dividing `s` by 2 gives zero because of underflow. The computation of `z` yields an undefined expression which results in `NaN` even though the limit is defined. The final two computations for `y` and `t` show the exceptions `Inf` and `-Inf`.

1.3.2 Double Precision

The IEEE *double precision* floating point standard representation uses a $64-$ bit word with bits numbered from 0 to 63 from left to right. The first bit S is the sign bit, the next eleven bits E are the exponent bits for e and the final 52 bits F represent the mantissa m:

$$\underbrace{S}_{0}\ \overbrace{\underbrace{EEEEEEEEEEE}_{1\ \ \ \ \ \ \ \ \ \ 11}}^{e}\ \overbrace{\underbrace{FFFFF\ldots FFFFF}_{12\ \ \ \ \ \ \ \ \ \ 63}}^{m}$$

The value $\tilde{a}$ represented by the $64-$bit word is defined as follows:

- **normal numbers**: If $0 < e < 2047$, then $\tilde{a} = (-1)^S \times 2^{e-1023} \times 1.m$, where $1.m$ is the binary number created by prefixing m with an implicit leading 1 and a binary point.

- **subnormal numbers**: If $e = 0$ and $m \neq 0$, then $\tilde{a} = (-1)^S \times 2^{-1022} \times 0.m$.
  These are known as *denormalized (or subnormal) numbers.*
  If $e = 0$ and $m = 0$ and $S = 1$, then $\tilde{a} = -0$.
  If $e = 0$ and $m = 0$ and $S = 0$, then $\tilde{a} = 0$

- **exceptions**: If $e = 2047$ and $m \neq 0$, then $\tilde{a} = NaN$ (*Not a number*)
  If $e = 2047$ and $m = 0$ and $S = 1$, then $\tilde{a} = -Inf$.
  If $e = 2047$ and $m = 0$ and $S = 0$, then $\tilde{a} = Inf$.

In Matlab, real computations are performed in IEEE double precision by default. Using again the hexadecimal format in Matlab to see the internal representation, we obtain for example

```
>> format hex
>> 2
ans = 4000000000000000
```

If we expand each hexadecimal digit to 4 binary digits we obtain for the number 2:

```
0100 0000 0000 0000 0000 0000 .... 0000 0000 0000
```

We skipped with ... seven groups of four zero binary digits. The interpretation is: $+1 \times 2^{1024-1023} \times 1.0 = 2$.

```
>> 6.5
ans = 401a000000000000
```

This means

```
0100 0000 0001 1010 0000 0000 .... 0000 0000 0000
```

Again we skipped with ... seven groups of four zeros. The resulting number is $+1 \times 2^{1025-1023} \times (1 + \frac{1}{2} + \frac{1}{8}) = 6.5$.

From now on, our discussion will focus on double precision arithmetic, since this is the usual mode of computation for real numbers in the IEEE Standard. Furthermore, we will stick to the IEEE Standard as used in Matlab. In other, more low−level programming languages, the behavior of the IEEE arithmetic can be adapted, e.g. the exception handling can be explicitly specified.

- The *machine precision* is $eps = 2^{-52}$.

- The largest machine number $\tilde{a}_{max}$ is denoted by `realmax`. Note that

```
>> realmax
ans = 1.7977e+308
>> log2(ans)
ans = 1024
>> 2^1024
ans = Inf
```

This looks like a contradiction at first glance, since the largest exponent should be $2^{2046-1023} = 2^{1023}$ according the IEEE conventions. But `realmax` is the number with the largest possible exponent and with the mantissa F consisting of all ones:

```
>> format hex
>> realmax
ans = 7fefffffffffffff
```

This is

$$\begin{aligned} V &= +1 \times 2^{2046-1023} \times 1.\overbrace{11\ldots1}^{52Bits} \\ &= 2^{1023} \times \left(1 + (\frac{1}{2})^1 + (\frac{1}{2})^2 + \cdots + (\frac{1}{2})^{52}\right) \\ &= 2^{1023} \times \frac{1-(\frac{1}{2})^{53}}{1-(\frac{1}{2})} = 2^{1023} \times (2 - eps) \end{aligned}$$

Even though Matlab reports `log2(realmax)=1024`, `realmax` does not equal 2^{1024}, but rather $(2 - eps) \times 2^{1023}$; taking the logarithm of `realmax` yields 1024 only because of rounding. Similar rounding effects would also occur for machine numbers that are a bit smaller than `realmax`.

- The computation range is the interval `[-realmax, realmax]`. If an operation produces a result outside this interval, then it is said to *overflow*. Before the IEEE Standard, computation would halt with an error message in such a case. Now the result of an overflow operation is assigned the number $\pm$`Inf`.
- The smallest positive normalized number is `realmin` $= 2^{-1022}$.
- IEEE allows computations with *denormalized numbers*. The positive denormalized numbers are in the interval [`realmin * eps, realmin`]. If an operation produces a strictly positive number that is smaller than `realmin * eps`, then this result is said to be in the *underflow range.* Since such a result cannot be represented, zero is assigned instead.
- When computing with denormalized numbers, we may suffer a loss of precision. Consider the following Matlab program:

```
>> format long
>> res=pi*realmin/123456789101112
res = 5.681754927174335e-322
>> res2=res*123456789101112/realmin
res2 = 3.15248510554597
>> pi = 3.14159265358979
```

The first result `res` is a denormalized number, and thus can no longer be represented with full accuracy. So when we reverse the operations and compute `res2`, we obtain a result which only contains 2 correct decimal digits. We therefore recommend avoiding the use of denormalized numbers whenever possible.

1.4 Rounding Errors

1.4.1 Standard Model of Arithmetic

Let $\tilde{a}$ and $\tilde{b}$ be two machine numbers. Then $c = \tilde{a} \times \tilde{b}$ will in general not be a machine number anymore, since the product of two numbers contains twice as many digits. The computed result will therefore be rounded to a machine number $\tilde{c}$ which is closest to c.

As an example, consider the 8−digit decimal numbers

$$\tilde{a} = 1.2345678 \quad \text{and} \quad \tilde{b} = 1.1111111,$$

whose product is

$$c = 1.37174198628258 \quad \text{and} \quad \tilde{c} = 1.3717420.$$

The *absolute rounding error* is the difference $r_a = \tilde{c} - c = 1.371742e - 8$, and

$$r = \frac{r_a}{c} = 1e - 8$$

is called the *relative rounding error.*

On today's computers, basic arithmetic operations obey the *standard model of arithmetic*: for $a, b \in \mathbb{M}$, we have

$$a \tilde{\oplus} b = (a \oplus b)(1 + r) \tag{1.2}$$

where r is the relative rounding error with $|r| < eps$, the machine precision. We denote with $\oplus \in \{+, -, \times, /\}$ the exact basic operation and with $\tilde{\oplus}$ the equivalent computer operation.

Another interpretation of the standard model of arithmetic is due to Wilkinson. In what follows, we will no longer use the multiplication symbol $\times$ for the exact operation; it is common practice in algebra to denote multiplication without any symbol: $ab \Longleftrightarrow a \times b$. Consider the operations

$$\begin{aligned}
&Addition : a\tilde{+}b = (a + b)(1 + r) = (a + ar) + (b + br) = \tilde{a} + \tilde{b} \\
&Subtraction : a\tilde{-}b = (a - b)(1 + r) = (a + ar) - (b + br) = \tilde{a} - \tilde{b} \\
&Multiplication : a\tilde{\times}b = ab(1 + r) = a(b + br) = a\tilde{b} \\
&Division : a\tilde{/}b = (a/b)(1 + r) = (a + ar)/b = \tilde{a}/b
\end{aligned}$$

In each of the above, the operation satisfies

Wilkinson's Principle

The result of a numerical computation on the computer is the result of an exact computation with slightly perturbed initial data.

For example, the numerical result of the multiplication $a\tilde{\times}b$ is the exact result $a\tilde{b}$ with a slightly perturbed operand $\tilde{b} = b + br$. As a consequence of Wilkinson's Principle, we need to study the effect that slightly perturbed data have on the result of a computation. This is done in Section 1.6.

1.4.2 Cancellation

A special rounding error is called *cancellation*. If we subtract two almost equal numbers, leading digits will cancel. Consider the following two numbers with 5 decimal digits:

$$\begin{array}{ll} 1.2345\text{e}0 & \\ -1.2344\text{e}0 & \\ \hline 0.0001\text{e}0 & = 1.0000\text{e-}4 \end{array}$$

If the two numbers were exact, the result delivered by the computer would also be exact. But if the first two numbers had been obtained by previous calculations and were affected by rounding errors, then the result would at best be $1.XXXXe-4$, where the digits denoted by X are unknown.

This is exactly what happened in our example at the beginning of this chapter. To compute $\sin(\alpha/2)$ from $\sin\alpha$, we used the Algorithm (1):

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

Since $\sin\alpha_n \to 0$, the numerator on the right hand side is

$$1-\sqrt{1-\varepsilon^2}, \quad \text{with small} \quad \varepsilon = \sin\alpha_n,$$

and suffers from severe cancellation. This is the reason why the algorithm performed so badly, even though the theory and program are both correct.

It is possible in this case to rearrange the computation and avoid cancellation:

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

This last expression no longer suffers from cancellation, and we obtain the new program listed in Algorithm 2

This time we do converge to the correct value of π (see Table 1.2). Notice also the elegant termination criterion: since the surface of the next polygon grows, we theoretically have

$$A_6 < \cdots < A_n < A_{2n} < \pi.$$

However, this cannot be true forever in finite precision arithmetic, since there is only a finite set of machine numbers. Thus, the situation $A_n \geq A_{2n}$ must occur at some stage, and this is the condition to stop the iteration. Note that this condition is independent of the machine, in the sense that the iteration will always terminate as long as we have finite precision arithmetic, and when it does terminate, it always gives the best possible approximation for the precision of the machine. More examples of machine-independent algorithms can be found in Section 1.8.1. A second example in which cancellation occurs is the problem of numerical differentiation. Given a twice continuously differentiable

Algorithm 2 Computation of π, Stable Version

```
oldA=0;s=sqrt(3)/2;  newA=3*s;  n=6;      %  initialization
z=[newA-pi n newA  s];                    %  store the results
while newA>oldA                           %  quit if area does not increase
  oldA=newA;
  s=s/sqrt(2*(1+sqrt((1+s)*(1-s))));      %  new  sine  value n=2*n; newA=n/2*s;
  z=[z; newA-pi n newA s];
end m=length(z);
for  i=1:m
  fprintf('%10d  %20.15f %20.15f\n',z(i,2),z(i,3),z(i,1))
end
```

n	A_n	$A_n - \pi$
6	2.598076211353316	−0.543516442236477
12	3.000000000000000	−0.141592653589793
24	3.105828541230249	−0.035764112359544
48	3.132628613281238	−0.008964040308555
96	3.139350203046867	−0.002242450542926
192	3.141031950890509	−0.000560702699284
384	3.141452472285462	−0.000140181304332
768	3.141557607911857	−0.000035045677936
1536	3.141583892148318	−0.000008761441475
3072	3.141590463228050	−0.000002190361744
6144	3.141592105999271	−0.000000547590522
12288	3.141592516692156	−0.000000136897637
24576	3.141592619365383	−0.000000034224410
49152	3.141592645033690	−0.000000008556103
98304	3.141592651450766	−0.000000002139027
196608	3.141592653055036	−0.000000000534757
393216	3.141592653456104	−0.000000000133690
786432	3.141592653556371	−0.000000000033422
1572864	3.141592653581438	−0.000000000008355
3145728	3.141592653587705	−0.000000000002089
6291456	3.141592653589271	−0.000000000000522
12582912	3.141592653589663	−0.000000000000130
25165824	3.141592653589761	−0.000000000000032
50331648	3.141592653589786	−0.000000000000008
100663296	3.141592653589791	−0.000000000000002
201326592	3.141592653589794	0.000000000000000
402653184	3.141592653589794	0.000000000000001
805306368	3.141592653589794	0.000000000000001

Table 1.2: Stable Computation of π

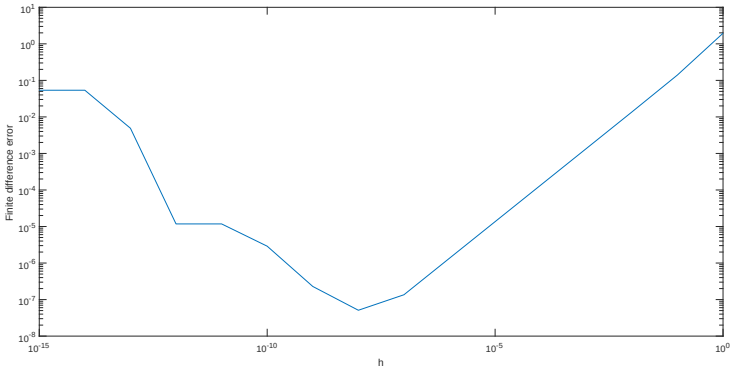


Figure 1.3: Results of numerical differentiation for $f(x) = e^x, x_0 = 1$

function $f : \mathbb{R} \to \mathbb{R}$, suppose we wish to calculate the derivative $f'(x_0)$ at some point x_0 using the approximation

$$f'(x_0) \approx D_{x_0,h}(f) = \frac{f(x_0 + h) - f(x_0)}{h}$$

This approximation is useful if, for instance, $f(x)$ is the result of a complex simulation, for which an analytic formula is not readily available. If we expand $f(x)$ by a Taylor series around $x = x_0$, we see that

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(\xi) \tag{1.3}$$

Thus, we expect the error to decrease linearly with h as we let h tend to zero. As an example, consider the problem of evaluating $f'(x_0)$ for $f(x) = e^x$ with $x_0 = 1$. We use the following code to generate a plot of the approximation error:

```
>> h=10.^(-15:0);
>> f=@(x) exp(x);
>> x0=1;
>> fp=(f(x0+h)-f(x0))./h;
>> loglog(h,abs(fp-exp(x0)));
```

Figure 1.3 shows the resulting plot. For relatively large h, i.e., for $h > 1e-8$, the error is indeed proportional to h, as suggested by (1.3). However, the plot clearly shows that the error is minimal for $h \approx 1e-8$, and then the error increases again as h decreases further. This is again due to severe cancellation: when h is small, we have $f(x_0 + h) \approx f(x_0)$. In particular, since $f(x_0) = f'(x_0) = 2.71828...$ is of moderate size, we expect for $h = 10^{-t}$ that $f(x_0 + h)$ differs from $f(x_0)$ by only $|\log_{10}(eps)| - t$ digits, i.e., t digits are lost due to finite precision arithmetic. Thus, when $h < 10^{-8}$, we lose more digits due to roundoff error than the accuracy gained by a better Taylor approximation. In general, the highest relative accuracy that can be expected by this approximation is about $\sqrt{eps}$, which is a far cry from the *eps* precision promised by the machine. We observe that in the first example, we obtain bad results due to an

unstable formula, but a better implementation can be devised to remove the instability and obtain good results. In the second example, however, it is unclear how to rearrange the computation without knowing the exact formula for $f(x)$; one might suspect that the problem is inherently harder. In order to quantify what we mean by easy or hard problems, we need to introduce the notion of *conditioning*.

1.5 Condition of a Problem

Intuitively, the *conditioning* of a problem measures how sensitive it is to small changes in the data; if the problem is very sensitive, it is inherently more difficult to solve it using finite precision arithmetic. In order to properly define "small" changes, however, we need define the notion of distance for $\mathbb{R}^n$.

1.5.1 Norms

A natural way to measure distance in higher dimensions is the *Euclidean norm*, which represents the distance of two points we are used to in daily life. There are however many other ways of measuring distance, also between matrices, and these distance characterizations are called norms.

Definition 1.5.1. ***(VECTOR NORM)*** *A vector norm is a function $\|x\| : \mathbb{R}^n \to \mathbb{R}$ such that*

1. $\|\boldsymbol{x}\| > 0$ *whenever* $\boldsymbol{x} \neq 0$.
2. $\|\alpha\boldsymbol{x}\| = |\alpha|\|\boldsymbol{x}\|$ *for all* $\alpha \in \mathbb{R}$ *and* $\boldsymbol{x} \in \mathbb{R}^n$.
3. $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ *for all* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$ *(triangle inequality).*

Note that vector norms can also be defined for vectors in $\mathbb{C}^n$, but we will mostly concentrate on real vector spaces. Frequently used norms are
The spectral norm or Euclidean norm or 2-norm: It measures the Euclidean length of a vector and is defined by

$$\|\boldsymbol{x}\|^2 := \sqrt{\sum_{i=1}^{n} |x_i|^2}.$$

The infinity norm or maximum norm: It measures the largest element in modulus and is defined by

$$\|\boldsymbol{x}\|_\infty := \max_{1 \leq i \leq n} |x_i|.$$

The 1−norm: It measures the sum of all the elements in modulus and is defined by

$$\|\boldsymbol{x}\|_1 := \sum_{i=1}^{n} |x_i|.$$

Definition 1.5.2. ***(MATRIX NORM)*** *A matrix norm is a function $\|A\| : \mathbb{R}^{m \times n} \to \mathbb{R}$ such that*

1. $\|A\| > 0$ *whenever* $A \neq 0$.
2. $\|\alpha A\| = |\alpha|\|A\|$ *for all* $\alpha \in \mathbb{R}$ *and* $A \in \mathbb{R}^{m\times n}$.
3. $\|A + B\| \leq \|A\| + \|B\|$ *for all* $A, B \in \mathbb{R}^{m\times n}$ *(triangle inequality).*

When the matrix $A \in \mathbb{R}^{m\times n}$ represents a linear map between the *normed linear spaces* $\mathbb{R}^n$ and $\mathbb{R}^m$, it is customary to define the induced matrix norm by

$$\|A\| = \sup_{\|\boldsymbol{x}\|=1} \|A\boldsymbol{x}\| \tag{1.4}$$

where the norms $\|\boldsymbol{x}\|$ and $\|A\boldsymbol{x}\|$ correspond to the norms used for $\mathbb{R}^n$ and $\mathbb{R}^m$ respectively. Induced matrix norms satisfy the *submultiplicative property*,

$$\|AB\| \leq \|A\|\|B\|. \tag{1.5}$$

However, there are applications in which A does not represent a linear map, for example in data analysis, where the matrix is simply an array of data values. In such cases, we may choose to use a norm that is not an induced matrix norm, such as the norm

$$\|A\|_\Delta = \max_{i,j} |a_{i,j}|,$$

or the Frobenius norm (see below). Such norms may or may not be submultiplicative: for instance, the Frobenius norm is submultiplicative, but $\|\cdot\|_\Delta$ is not.

For the vector norms we have introduced before, the corresponding induced matrix norms are

The spectral norm or 2-norm:

$$\|A\|_2 = \sup_{\|\boldsymbol{x}\|_2=1} \|A\boldsymbol{x}\|_2$$

It can be shown that $\|A\|_2^2$ is equal to the largest eigenvalue of $A^\top A$ (or, equivalently, to the square of the largest singular value of A, see Chapter 5). It follows that the $2-$norm is invariant under orthogonal transformations, i.e., we have $\|QA\|_2 = \|AQ\|_2 = \|A\|_2$ whenever $Q^\top Q = I$.

The infinity norm or maximum row sum norm:

$$\|A\|_\infty = \sup_{\|\boldsymbol{x}\|_\infty=1} \|A\boldsymbol{x}\|_\infty \equiv \max_{1\leq i\leq n} \sum_{j=1}^{n} |a_{ij}|$$

The last identity comes from the observation that the vector $\boldsymbol{x}$ which maximizes the supremum is given by $\boldsymbol{x} = (\pm 1, \pm 1, \ldots, \pm 1)^\top$ with the sign of the entries chosen according to the sign of the entries in the row of A with the largest row sum.

The 1-norm or maximum column sum norm:

$$\|A\|_1 = \sup_{\|\boldsymbol{x}\|_1=1} \|A\boldsymbol{x}\|_1 \equiv \max_{1\leq j\leq n} \sum_{i=1}^{n} |a_{ij}|$$

The last identity holds because the supremum is attained for the value $\boldsymbol{x} = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ with 1 at the column position of A with the largest column sum.

Note that all induced norms, including the ones above, satisfy $\|I\| = 1$, where I is the identity matrix. There is another commonly used matrix norm, the *Frobenius norm*, which does not arise from vector norms; it is defined by

$$\|A\|_F := \sqrt{\sum_{i,j=1}^{n} |a_{ij}|^2} \tag{1.6}$$

The square of the Frobenius norm is also equal to the sum of squares of the the singular values of A, see Chapter 5.

In the finite dimensional case considered here, all norms are equivalent, which means for any pair of norms $\|\cdot\|_a$ and $\|\cdot\|_b$, there exist constants C_1 and C_2 such that

$$C_1\|\boldsymbol{x}_a\| \leq \|\boldsymbol{x}_b\| \leq C_2\|\boldsymbol{x}_a\|, \quad \forall \mathrm{x} \in \mathbb{R}^n. \tag{1.7}$$

We are therefore free to choose the norm in which we want to measure distances; a good choice often simplifies the argument when proving a result, even though the result then holds in any norm (except possibly with a different constant). Note, however, that the constants may depend on the dimension n of the vector space, which may be large.

1.5.2 Big- and Little-O Notation

When analyzing roundoff errors, we would often like to keep track of terms that are *very* small, e.g., terms that are proportional to eps^2, without explicitly calculating with them. The following notations, due to Landau, allows us to do just that.

Definition 1.5.3. ***(Big-O, Little-O)*** *Let $f(x)$ and $g(x)$ be two functions. For a fixed $L \in \mathbb{R} \cup \{\pm\infty\}$, we write*

1. *$f(x) = O_{x\to L}(g(x))$ if there is a constant C such that $|f(x)| \leq C|g(x)|$ for all x in a neighborhood of L. This is equivalent to*

$$\limsup_{x\to L} \Big|\frac{f(x)}{g(x)}\Big| < \infty.$$

2. *$f(x) = o_{x\to L}(g(x))$ if $\lim\limits_{x\to L} \frac{|f(x)|}{|g(x)|} = 0$*

When the limit point L is clear from the context, we omit the subscript $x \to L$ and simply write $O(g(x))$ or $o(g(x))$.

The following properties of $O(\cdot)$ and $o(\cdot)$ are immediate consequences of the definition, but are used frequently in calculations.

Lemma 1.5.4. *For a given limit point $L \in \mathbb{R} \cup \{\pm\infty\}$, we have*

1. *$O(g_1) \pm O(g_2) = O(|g_1| + |g_2|)$.*
2. *$O(g_1) \cdot O(g_2) = O(g_1 g_2)$.*
3. *For any constant C, $C \cdot O(g) = O(g)$.*

4. For a fixed function f, $O(g)/f = O(g/f)$.

The same properties hold when $O(\cdot)$ is replaced by $o(\cdot)$.

Note carefully that we do not have an estimate for $O(g_1)/O(g_2)$: if $f_1 = O(g_1)$ and $f_2 = O(g_2)$, it is possible that f_2 is much smaller than g_2, so it is not possible to bound the quotient f_1/f_2 by g_1 and g_2.

Example 1.5.1. *Let $p(x) = c_d(x-a)^d + c_{d+1}(x-a)^{d+1} + \ldots + c_D(x-a)^D$ be a polynomial with $d < D$. If c_d and c_D are both nonzero, then we have*

$$p(x) = O(x^D), \quad as \quad x \to \pm\infty,$$
$$p(x) = O((x-a)^d), \quad as \quad x \to a.$$

Thus, it is essential to know which limit point L is implied by the the big–O notation.

Example 1.5.2. *Let $f : U \in \mathbb{R} \to \mathbb{R}$ be n times continuously differentiable on an open interval U. Then for any $a \in U$ and $k \leq n$, Taylor's theorem with the remainder term tells us that*

$$f(x) = f(a) + f'(a)(x-a) + \ldots + \frac{f^{(k-1)}(a)}{(k-1)!}(x-a)^{k-1} + \frac{f^{(k)}(\xi)}{k!}(x-a)^k$$

with $|\xi - a| \leq |x - a|$. Since $f^{(k)}$ is continuous (and hence bounded), we can write

$$f(x) = f(a) + f'(a)(x-a) + \ldots + \frac{f^{(k-1)}(a)}{(k-1)!}(x-a)^{k-1} + O((x-a)^k)$$

where the implied limit point is $L = a$. For $|h| \ll 1$, this allows us to write

$$f(a+h) = f(a) + f'(a)h + f''(a)\frac{h^2}{2} + O(h^3),$$
$$f(a-h) = f(a) - f'(a)h + f''(a)\frac{h^2}{2} + O(h^3).$$

Then it follows from Lemma 1.5.4 that

$$\frac{f(a+h) - f(a-h)}{2h} = f'(a) + O(h^2)$$

whenever f is at least three times continuously differentiable.

In Maple, Taylor series expansions can be obtained using the command `taylor`*, both for generic functions f and for built-in functions such as* `sin`*:*

```
>p1:=taylor(f(x),x=a,3);
```

$$p1 := f(a) + D(f)(a)(x-a) + \frac{1}{2}(D^{(2)})(f)(a)(x-a)^2 + O((x-a)^3)$$

```
> p2:=taylor(sin(x),x=0,8);
```

$$p2 := x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + O(x^8)$$

```
> p3:=taylor((f(x+h)-f(x-h))/2/h, h=0, 4);
```

$$p3 := D(f)(x) + \frac{1}{6}(D^{(3)})(f)(x)h^2 + O(h^3)$$

```
> subs(f=sin,x=0,p3);
```

$$D(sin)(0) + \frac{1}{6}(D^{(3)})(sin)(0)h^2 + O(h^3)$$

```
> simplify(%);
```

$$1 - \frac{1}{6}h^2 + O(h^3)$$

Here, the $O(\cdot)$ in Maple should be interpreted the same way as in Definition1.5.3, with the limit point L given by the argument to `taylor`*, i.e., $x \to a$ in the first command, $x \to 0$ in the second and $h \to 0$ for the remaining commands.*

1.5.3 Condition Number

Definition 1.5.5. *(Condition Number) The condition number κ of a problem $\mathcal{P} : \mathbb{R}^n \to \mathbb{R}^m$ is the smallest number such that*

$$\frac{|\hat{x}_i - x_i|}{|x_i|} \leq \varepsilon \quad \textit{for} \quad 1 \leq i \leq n \Longrightarrow \frac{\|\mathcal{P}(\hat{\boldsymbol{x}}) - \mathcal{P}(\boldsymbol{x})\|}{\|\mathcal{P}(\boldsymbol{x})\|} \leq \kappa\varepsilon + o(\varepsilon) \tag{1.8}$$

where $o(\varepsilon)$ represents terms that are asymptotically smaller than ε.

A problem is well conditioned if κ is not too large; otherwise the problem is *ill conditioned*. Well-conditioned means that the solution of the problem with slightly perturbed data does not differ much from the solution of the problem with the original data. Ill-conditioned problems are problems for which the solution is very sensitive to small changes in the data.

Example 1.5.3. *We consider the problem of multiplying two real numbers, $\mathcal{P}(x_1, x_2) := x_1x_2$. If we perturb the data slightly, say*

$$\hat{x}_1 := x_1(1 + \varepsilon_1), \quad \hat{x}_2 := x_2(1 + \varepsilon_2), \quad |\varepsilon_i| \leq \varepsilon,\ i = 1, 2,$$

we obtain

$$\frac{\hat{x}_1\hat{x}_2 - x_1x_2}{x_1x_2} = (1 + \varepsilon_1)(1 + \varepsilon_2) - 1 = \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2,$$

and since we assumed that the perturbations are small, $\varepsilon \ll 1$, we can neglect the product $\varepsilon_1\varepsilon_2$ compared to the sum $\varepsilon_1 + \varepsilon_2$, and we obtain

$$\frac{|\hat{x}_1\hat{x}_2 - x_1x_2|}{|x_1x_2|} \leq 2\varepsilon.$$

Hence the condition number of multiplication is $\kappa = 2$, and the problem of multiplying two real numbers is well conditioned.

Example 1.5.4. *Let A be a fixed, non-singular $n \times n$ matrix. Consider the problem of evaluating the matrix-vector product $\mathcal{P}(\boldsymbol{x}) = A\boldsymbol{x}$ for $\boldsymbol{x} = (x_1, \ldots, x_n)^\top$. Suppose the perturbed vector $\hat{\boldsymbol{x}}$ satisfies $\hat{x}_i = x_i(1+\varepsilon_i)$, $|\varepsilon_i| < \varepsilon$. Then considering the infinity norm, we have $\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|_\infty \leq \varepsilon \|\boldsymbol{x}\|_\infty$, so that*

$$\frac{\|A\hat{\boldsymbol{x}} - A\boldsymbol{x}\|_\infty}{\|A\boldsymbol{x}\|_\infty} \leq \frac{\|A\|_\infty \|\hat{\boldsymbol{x}} - \boldsymbol{x}\|_\infty}{\|A\boldsymbol{x}\|_\infty} \leq \frac{\varepsilon \|A\|_\infty \|\boldsymbol{x}\|_\infty}{\|A\boldsymbol{x}\|_\infty}.$$

Since $\|\boldsymbol{x}\| = \|A^{-1}A\boldsymbol{x}\| \leq \|A^{-1}\|\|A\boldsymbol{x}\|$, we in fact have

$$\frac{\|A\hat{\boldsymbol{x}} - A\boldsymbol{x}\|_\infty}{\|A\boldsymbol{x}\|_\infty} \leq \varepsilon \|A\|_\infty \|A^{-1}\|_\infty,$$

so the condition number is $\kappa_\infty = \|A\|_\infty \|A^{-1}\|_\infty$. We will see in Chapter 2 that κ_∞ also plays an important role in the solution of linear systems of equations.

Example 1.5.5. *Let us look at the condition number of subtracting two real numbers, $\mathcal{P}(x_1, x_2) := x_1 - x_2$. Perturbing again the data slightly as in the previous example, we obtain*

$$\frac{|(\hat{x}_1 - \hat{x}_2) - (x_1 - x_2)|}{|x_1 - x_2|} = \frac{|x_1\varepsilon_1 - x_2\varepsilon_2|}{|x_1 - x_2|} \leq \frac{|x_1| + |x_2|}{|x_1 - x_2|}\varepsilon.$$

We see that if $sign(x_1) = -sign(x_2)$, which means the operation is an addition, then the condition number is $\kappa = 1$, meaning that the addition of two numbers is well conditioned. If, however, the signs are the same and $x_1 \approx x_2$, then $\kappa = \frac{|x_1|+|x_2|}{|x_1-x_2|}$ becomes very large, and hence subtraction is ill conditioned in this case.

As a numerical example, taking $x_1 = 1/51$ and $x_2 = 1/52$, we obtain for the condition number $\kappa = \frac{1/51+1/52}{1/51-1/52} = 103$, and if we compute with three significant digits, we obtain $\hat{x}_1 = 0.196e-1$, $\hat{x}_2 = 0.192e-1$, and $\hat{x}_1 - \hat{x}_2 = 0.400e-3$, which is very different from the exact result $x_1 - x_2 = 0.377e-3$; thus, the large condition number reflects the fact that the solution is prone to large errors due to cancellation.

In our decimal example above, the loss of accuracy is easily noticeable, since the lost digits appear as zeros. Unfortunately, cancellation effects are rarely as obvious when working with most modern computers, which use binary arithmetic. As an example, let us first compute in Maple

```
> Digits:=30;
> a:=1/500000000000001;
```

$$\frac{1}{500000000000001}$$

```
> b:=1/500000000000002;
```

$$\frac{1}{500000000000002}$$

```
> ce:=a-b;
```

$$\frac{1}{250000000000001500000000000002}$$

```
> c:=evalf(a-b);
```

$$3.9999999999999760000000000011 \times 10^{-30} \tag{1.9}$$

This is the exact difference, rounded to thirty digits. We round the operands to 16-digit machine numbers:

```
> bm:=evalf[16](b);
```

$$1.999999999999992 \times 10^{-15}$$

```
> am:=evalf[16](a);
```

$$1.999999999999999 \times 10^{-15}$$

Now we calculate the difference to 30-digit precision

```
> c16:=am-bm;
```

$$4.010^{-30}$$

and also the difference to 16-digit precision to emulate floating point operations:

```
> cf16:=evalf[16](am-bm);
```

$$4.010^{-30}$$

which agrees with to the standard model $a\tilde{-}b = (a-b)(1+r)$ *with* $r = 0$*. In Maple, we see that the precision in this difference is reduced to two digits. Only two digits are displayed, which indicates a serious loss of precision due to cancellation.*

Performing the same operations in Matlab, we get:

```
format long
>> a=1/500000000000001
a =
1.999999999999996e-15
>> b=1/500000000000002
b =
1.999999999999992e-15
>> c=a-b
c =
3.944304526105059e-30
```

Comparing with the exact difference in Equation (1.9), we see again that only the first two digits are accurate. The apparently random digits appearing in the difference stem from binary-to-decimal conversion, and it would be hard to guess that they are meaningless. Had we looked at the same computation in hexadecimal format in Matlab,

```
format hex
>> a=1/500000000000001
a =
3ce203af9ee7560b
>> b=1/500000000000002
b =
```

```
3ce203af9ee75601
>> c=a-b
c =
39d4000000000000
```

we would have seen that Matlab also completes the result by zeros, albeit in binary.

Example 1.5.6. *Consider once again the problem of evaluating the derivative of f numerically via the formula*

$$f'(x_0) \approx D_{x_0,h}(f) = \frac{f(x_0+h) - f(x_0)}{h}.$$

Here, the function f acts as data to the problem. We consider perturbed data of the form

$$\hat{f}(x) = f(x)(1+\varepsilon g(x)), \quad |g(x)| \leq 1,$$

which models the effect of roundoff error on a machine with precision ε. The condition number then becomes

$$\begin{aligned} \Big|\frac{D_{x_0,h}(\hat{f}) - D_{x_0,h}(f)}{D_{x_0,h}(f)}\Big| &= \Big|\frac{f(x_0+h)(1+\varepsilon g(x_0+h)) - f(x_0)(1+\varepsilon g(x_0))}{f(x_0+h)-f(x_0)} - 1\Big| \\ &\leq \frac{\varepsilon(|f(x_0)| + |f(x_0+h)|)}{|f(x_0+h)-f(x_0)|} \approx \frac{2\varepsilon|f(x_0)|}{|hf'(x_0)|}. \end{aligned}$$

Thus, we have $\kappa \approx \frac{2|f(x_0)|}{|hf'(x_0)|}$, meaning that the problem becomes more and more ill-conditioned as $h \to 0$. In other words, if h is too small relative to the perturbation ε, it is impossible to evaluate $D_{x0,h}(f)$ accurately, no matter how the finite difference is implemented.

A related notion in mathematics is well- or ill-posed problems. Let $A : X \to Y$ be a mapping of some space X to Y . The problem $Ax = y$ is well posed if

1. For each $y \in Y$ there exists a solution $x \in X$.
2. The solution x is unique.
3. The solution x is a continuous function of the the data y.

If one of the conditions is not met, then the problem is said to be *ill posed*. For example, the problem of calculating the derivative f' based on the values of f alone is an ill-posed problem in the continuous setting, as can be seen from the fact that $\kappa \to \infty$ as $h \to 0$ in Example 1.5.6. A sensible way of choosing h to obtain maximum accuracy is discussed in the future.

If a problem is ill-posed because condition 3 is violated, then it is also ill conditioned. But we can also speak of an ill-conditioned problem if the problem is well posed but if the solution is very sensitive with respect to small changes in the data. A good example of an ill-conditioned problem is finding the roots of the Wilkinson polynomial, see Chapter 4. It is impossible to "cure" an ill-conditioned problem by a good algorithm, but one should avoid transforming a well-conditioned problem into an ill-conditioned one by using a bad algorithm, e.g. one that includes ill-conditioned subtractions that are not strictly necessary.

1.6 Stable and Unstable Algorithms

An algorithm for solving a given problem $\mathcal{P} : \mathbb{R}^n \to \mathbb{R}$ is a sequence of elementary operations,

$$\mathcal{P}(x) = f_n(f_{n-1}(\dots f_2(f_1(x))\dots)).$$

In general, there exist several different algorithms for a given problem.

1.6.1 Forward Stability

If the amplification of the error in the operation f_i is given by the corresponding condition number $\kappa(f_i)$, we naturally obtain

$$\kappa(\mathcal{P}) \le \kappa(f_1) \cdot \kappa(f_2) \cdot \ldots \cdot \kappa(f_n).$$

Definition 1.6.1. *(Forward Stability) A numerical algorithm for a given problem $\mathcal{P}$ is forward stable if*

$$\kappa(f_1) \cdot \kappa(f_2) \cdot \ldots \cdot \kappa(f_n) \le C\kappa(\mathcal{P}), \tag{1.10}$$

where C is a constant which is not too large, for example $C = O(n)$.

Example 1.6.1. *Consider the following two algorithms for the problem $\mathcal{P} := \frac{1}{x(1+x)}$:*

1. $x \nearrow x \searrow$, $1+x \nearrow$ $\quad x(1+x) \to \frac{1}{x(1+x)}$

2. $x \nearrow \frac{1}{x} \searrow$, $x \searrow 1+x \to \frac{1}{1+x} \nearrow$ $\quad \frac{1}{x} - \frac{1}{1+x} \to \frac{1}{x(1+x)}$

In the first algorithm, all operations are well conditioned, and hence the algorithm is forward stable. In the second algorithm, however, the last operation is a potentially very ill-conditioned subtraction, and thus this second algorithm is not forward stable.

Roughly speaking, an algorithm executed in finite precision arithmetic is called *stable if the effect of rounding errors is bounded;* if, on the other hand, an algorithm increases the condition number of a problem by a large amount, then we classify it as *unstable.*

Example 1.6.2. *As a second example, we consider the problem of calculating the values*

$$\cos(1), \cos(\frac{1}{2}), \cos(\frac{1}{4}), \dots, \cos(2^{-12}),$$

or more generally,

$$z_k = \cos(2^{-k}), \quad k = 0, 1, \dots, n.$$

By considering perturbations of the form $\hat{z}_k = \cos(2^{-k}(1+\varepsilon))$, we can calculate the condition number for the problem using Definition 1.5.5:

$$\left|\frac{\cos(2^{-k}(1+\varepsilon)) - \cos(2^{-k})}{\cos(2^{-k})}\right| \approx 2^{-k}\tan(2^{-k})\varepsilon \approx 4^{-k}\varepsilon \Rightarrow \kappa(\mathcal{P}) \approx 4^{-k}.$$

We consider two algorithms for recursively calculating z_k:

1. *double angle: we use the relation* $\cos 2\alpha = 2\cos^2\alpha - 1$ *to compute*

$$y_n = \cos(2^{-n}), \quad y_{k-1} = 2y_k^2 - 1, \quad k = n, n-1, \ldots, 1.$$

2. *half angle: we use* $\cos\frac{\alpha}{2} = \sqrt{\frac{1+\cos\alpha}{2}}$ *and compute*

$$x_0 = \cos(1), \quad x_{k+1} = \sqrt{\frac{1+x_k}{2}}, \quad k = 0, 1, \ldots, n-1$$

2^{-k}	$y_k - z_k$	$x_k - z_k$
1	-0.0000000005209282	0.0000000000000000
5.000000e-01	-0.0000000001483986	0.0000000000000000
2.500000e-01	-0.0000000000382899	0.0000000000000001
1.250000e-01	-0.0000000000096477	0.0000000000000001
6.250000e-02	-0.0000000000024166	0.0000000000000000
3.125000e-02	-0.0000000000006045	0.0000000000000000
1.562500e-02	-0.0000000000001511	0.0000000000000001
7.812500e-03	-0.0000000000000377	0.0000000000000001
3.906250e-03	-0.0000000000000094	0.0000000000000001
1.953125e-03	-0.0000000000000023	0.0000000000000001
9.765625e-04	-0.0000000000000006	0.0000000000000001
4.882812e-04	-0.0000000000000001	0.0000000000000001
2.441406e-04	0.0000000000000000	0.0000000000000001

Table 1.3: Stable and unstable recursions

The results are given in Table 1.3. We notice that the y_k *computed by Algorithm 1 are significantly affected by rounding errors while the computations of the* x_k *with Algorithm 2 do not seem to be affected. Let us analyze the condition of one step of Algorithm 1 and Algorithm 2. For Algorithm 1, one step is* $f_1(y) = 2y^2 - 1$*, and for the condition of the step, we obtain from*

$$\frac{f_1(y(1+\varepsilon) - f_1(y))}{f_1(y)} = \frac{f_1(y) + f_1'(y)\cdot y\varepsilon + O(\varepsilon^2) - f_1(y)}{f_1(y)} = \frac{4y^2\varepsilon + O(\varepsilon^2)}{2y^2 - 1} \tag{1.11}$$

and from the fact that ε *is small, that the condition number for one step is* $\kappa_1 = \frac{4y^2}{|2y^2-1|}$ *. Since all the* y_k *in this iteration are close to one, we have* $\kappa_1 \approx 4$*. Now to obtain* y_k*, we must perform* $n-k$ *iterations, so the condition number becomes approximately* 4^{n-k}*. Thus, the constant* C *in (1.10) of the definition of forward stability can be estimated by*

$$C \approx \frac{4^{n-k}}{\kappa(\mathcal{P})} \approx \frac{4^{n-k}}{4^{-k}} = 4^n.$$

This is clearly not a small constant, so the algorithm is not forward stable.

For Algorithm 2, one step is $f_2(x) = \sqrt{\frac{1+x}{2}}$*. We calculate the one-step condition number similarly:*

$$\frac{f_2(y+\varepsilon) - f_2(y)}{f_2(y)} = \frac{1}{2(1+x)}\varepsilon + O(\varepsilon^2).$$

Thus, for ε *small, the condition number of one step is* $\kappa_2 = \frac{1}{2|1+x|}$*; since all* x_k *in this iteration are also close to one, we obtain* $\kappa_2 \approx \frac{1}{4}$ *. To compute* x_k*, we need* k *iterations,*

so the overall condition number is $4-k$. Hence the stability constant C in (1.10) is approximately

$$C \approx \frac{4^{-k}}{\kappa(\mathcal{P})} \approx 1$$

meaning Algorithm 2 is stable.

We note that while Algorithm 1 runs through the iteration in an unstable manner, Algorithm 2 performs the same iteration, but in reverse. Thus, if the approach in Algorithm 1 is unstable, inverting the iteration leads to the stable Algorithm 2. This is also reflected in the one-step condition number estimate (4 and 1/4 respectively), which are the inverses of each other.

Finally, using for $n = 12$ a perturbation of the size of the machine precision $\varepsilon = 2.2204e-16$, we obtain for Algorithm 1 that $4^{12}\varepsilon = 3.7e-9$, which is a good estimate of the error $5e-10$ of y_0 we measured in the numerical experiment.

Example 1.6.3. *An important example of an unstable algorithm, which motivated the careful study of condition and stability, is Gaussian elimination with no pivoting (see Example 2.2.4). When solving linear systems using Gaussian elimination, it might happen that we eliminate an unknown using a very small pivot on the diagonal. By dividing the other entries by the small pivot, we could be introducing artificially large coefficients in the transformed matrix, thereby increasing the condition number and transforming a possibly well-conditioned problem into an ill-conditioned one. Thus, choosing small pivots makes Gaussian elimination unstable we need to apply a pivoting strategy to get a numerically satisfactory algorithm (cf. Section 2.2).*

Note, however, that if we solve linear equations using orthogonal transformations (Givens rotations, or Householder reflections, see Section 2.5), then the condition number of the transformed matrices remains constant. To see this, consider the transformation

$$A\boldsymbol{x} = \mathbf{b} \Rightarrow QA\boldsymbol{x} = Q\mathbf{b}.$$

where $Q^\top Q = I$. Then the $2-$norm condition number of QA (as defined in Theorem 2.3.1 satisfies $\kappa(QA) = \|(QA)^{-1}\|_2\|QA\|_2 = \|A^{-1}Q^\top\|_2\|QA\|_2 = \|A^{-1}\|_2\|A\|_2 = \kappa(A)$, since the $2-$norm is invariant under multiplication with orthogonal matrices.

Unfortunately, as we can see in Example 1.6.2, it is generally difficult and laborious to verify whether an algorithm is forward stable, since the condition numbers required by (1.10) are often hard to obtain for a given problem and algorithm. A different notion of stability, based on perturbations in the initial data rather than in the results of the algorithm, will often be more convenient to use in practice.

1.6.2 Backward Stability

Because of the difficulties in verifying forward stability, Wilkinson introduced a different notion of stability, which is based on the Wilkinson principle we have already seen in Section 1.4:

The result of a numerical computation on the computer is the result of an exact computation with slightly perturbed initial data.

Definition 1.6.2. *(Backward Stability) A numerical algorithm for a given problem $\mathcal{P}$ is backward stable if the result $\hat{y}$ obtained from the algorithm with data x can be interpreted as the exact result for slightly perturbed data $\hat{x}, \hat{y} = \mathcal{P}(\hat{x})$, with*

$$\frac{|\hat{x}_i - x_i|}{|x_i|} \leq Ceps \tag{1.12}$$

where C is a constant which is not too large, and eps is the precision of the machine.

Note that in order to study the backward stability of an algorithm, one does not need to calculate the condition number of the problem itself.

Also note that a backward stable algorithm does not guarantee that the error $\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|$ is small. However, if the condition number κ of the problem is known, then the relative forward error can be bounded by

$$\frac{\|\hat{\boldsymbol{y}} - \boldsymbol{y}\|}{\|\boldsymbol{y}\|} = \frac{\|\mathcal{P}(\hat{\boldsymbol{x}}) - \mathcal{P}(\boldsymbol{x})\|}{\mathcal{P}(\boldsymbol{x})} \leq \kappa(\mathcal{P}) \max_i \frac{|\hat{x}_i - x_i|}{|x_i|} \leq \kappa(\mathcal{P}) \cdot Ceps$$

Thus, a backward stable algorithm is automatically forward stable, but not vice versa.

Example 1.6.4. *We wish to investigate the backward stability of an algorithm for the scalar product $\boldsymbol{x}^\top \boldsymbol{y} := x_1 y_1 + x_2 y_2$. We propose for the algorithm the sequence of operations*

$$(x_1, x_2, y_1, y_2) \begin{array}{c} \nearrow x_1 y_1 \searrow \\ \searrow x_2 y_2 \nearrow \end{array} x_1 y_1 + x_2 y_2$$

Using the fact that storing a real number x on the computer leads to a rounded quantity $x(1+\varepsilon), |\varepsilon| \leq eps$, and that each multiplication and addition again leads to a roundoff error of the size of eps, we find for the numerical result of this algorithm

$$(x_1(1+\varepsilon_1)y_1(1+\varepsilon_2)(1+\eta_1) + x_2(1+\varepsilon_3)y_2(1+\varepsilon_4)(1+\eta_2))(1+\eta_3) = \hat{x}_1\hat{y}_1 + \hat{x}_2\hat{y}_2,$$

where $|\varepsilon_i|, |\eta_i| \leq eps$, and

$$\begin{aligned} \hat{x}_1 &= x_1(1+\varepsilon_1)(1+\eta_1), & ,\hat{y}_1 &= y_1(1+\varepsilon_2)(1+\eta_3), \\ \hat{x}_2 &= x_2(1+\varepsilon_3)(1+\eta_2), & ,\hat{y}_2 &= y_2(1+\varepsilon_4)(1+\eta_3), \end{aligned}$$

Hence the backward stability condition (1.12) is satisfied with the constant $C = 2$, and thus this algorithm is backward stable.

Example 1.6.5. *As a second example, we consider the product of two upper triangular matrices A and B,*

$$A = \begin{pmatrix} a_{11} & a_{12} \\ 0 & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{pmatrix}.$$

If we neglect for simplicity of presentation the roundoff error storing the entries of the matrix in finite precision arithmetic, we obtain using the standard algorithm for computing the product of A and B on the computer

$$\begin{pmatrix} a_{11}b_{11}(1+\varepsilon_1) & (a_{11}b_{12}(1+\varepsilon_2) + a_{12}b_{22}(1+\varepsilon_3))(1+\varepsilon_4) \\ 0 & a_{22}b_{22}(1+\varepsilon_5) \end{pmatrix},$$

where $|\varepsilon_j| \leq eps$ *for* $j = 1, \ldots, 5$. *If we define the slightly modified matrices*

$$\hat{A} = \begin{pmatrix} a_{11} & a_{12}(1+\varepsilon_3)(1+\varepsilon_4) \\ 0 & a_{22}(1+\varepsilon_5) \end{pmatrix},$$

$$\hat{B} = \begin{pmatrix} b_{11}(1+\varepsilon_1) & b_{12}(1+\varepsilon_2)(1+\varepsilon_4) \\ 0 & b_{22} \end{pmatrix},$$

their product is exactly the matrix we obtained by computing AB *numerically. Hence the computed product is the exact product of slightly perturbed* A *and* B*, and this algorithm is backward stable.*

The notion of backward stability will prove to be extremely useful in Chapter 2, where we use the same type of analysis to show that Gaussian elimination is in fact stable when combined with a pivoting strategy, such as complete pivoting.

1.7 Calculating with Machine Numbers: Tips and Tricks

1.7.1 Associative Law

Consider the associative law for exact arithmetic:

$$(a+b)+c = a+(b+c).$$

This law does not hold in finite precision arithmetic. As an example, take the three numbers

$$a = 1.23456e-3, \quad b = 1.00000e0, \quad c = -b.$$

Then it is easy to see that, in decimal arithmetic, we obtain $(a+b)+c = 1.23000e-3$, but $a+(b+c) = a = 1.23456e-3$. It is therefore important to use parentheses wisely, and also to consider the order of operations.

Assume for example that we have to compute a sum $\sum_{i=1}^{N} a_i$, where the terms $a_i > 0$ are monotonically decreasing, i.e., $a_1 > a_2 > \ldots > a_n$. More concretely, consider the harmonic series

$$S = \sum_{i=1}^{N} \frac{1}{i}.$$

For $N = 10^6$, we compute an "exact" reference value using Maple with sufficient accuracy `(Digits:=20):`

```
Digits:=20;
s:=0;
for i from 1 to 1000000 do
s:=s+1.0/i:
od:
s;
14.392726722865723804
```

Using Matlab with IEEE arithmetic, we get

```
N=1e6;
format long e
s1=0;
for i=1:N
s1=s1+1/i;
end
s1
ans = 1.439272672286478e+01
```

We observe that the last three digits are different from the Maple result. If we sum again with Matlab but in reverse order, we obtain

```
s2=0;
for i=N:-1:1
s2=s2+1/i;
end
s2
ans = 1.439272672286575e+01
```

a much better result, since it differs only in the last digit from the Maple result! We have already seen this effect in the associative law example: if we add a small number to a large one, then the least significant bits of the smaller machine number are lost. Thus, it is better to start with the smallest elements in the sum and add the largest elements last. However, sorting the terms of the sum would mean more computational work than strictly required.

1.7.2 Summation Algorithm by W. Kahan

An accurate algorithm that does not require sorting was given by W. Kahan. The idea here is to keep as carry the lower part of the small term, which would have been lost when added to the partial sum. The carry is then added to the next term, which is small enough that it would not be lost.

Algorithm 3 Kahan's Summation of $\sum_{j=1}^{N} \frac{1}{j}$

```
s=0;                    % partial sum
c=0;                    % carry
for j=1:N
  y=1/j+c;
  t=s+y;                % next partial sum, with roundoff
  c=(s-t)+y;            % recapture the error and store as carry
  s=t;
end
s=s+c
```

Doing so gives a remarkably good result, which agrees to the last digit with the Maple result:

```
s = 1.439272672286572e+01
```

1.7.3 Small Numbers

If $a + x = a$ holds in exact arithmetic, then we conclude that $x = 0$. This is no longer true in finite precision arithmetic. In IEEE arithmetic for instance, $1 + 1e - 20 = 1$ holds; in fact, we have $1 + w = 1$ not only for $1e - 20$, but for all positive machine numbers w with $w < eps$, where eps is the machine precision.

1.7.4 Monotonicity

Assume we are given a function f which is strictly monotonically increasing in $[a, b]$. Then for $x_1 < x_2$ with $x_i \in [a, b]$ we have $f(x_1) < f(x_2)$. Take for example $f(x) = \sin(x)$ and $0 < x_1 < x_2 < \frac{\pi}{2}$. Can we be sure that in finite precision arithmetic $\sin(x_1) < \sin(x_2)$ also holds? The answer in general is no. *For standard functions* however, special care was taken when they were implemented in IEEE arithmetic, so that at least monotonicity is maintained, only strict monotonicity is not guaranteed. In the example, IEEE arithmetic guarantees that $\sin(x_1) \leq \sin(x_2)$ holds.

As an example, let us consider the polynomial

$$f(x) = x^3 - 3.000001x^2 + 3x - 0.999999.$$

This function is very close to $(x - 1)^3$; it has the 3 isolated roots at

$$0.998586, \quad 1.00000, \quad 1.001414.$$

Let us plot the function f:

```
figure(1)
a=-1; b=3; h=0.1;
x=a:h:b; y=x.^3-3.000001*x.^2+3*x-0.999999;
plot(x,y)
line([a,b],[0,0])
legend('x^3-3.000001*x^2+3*x-0.999999')

figure(2)
a=0.998; b=1.002; h=0.0001;
x=a:h:b; y=x.^3-3.000001*x.^2+3*x-0.999999;
plot(x,y)
line([a,b],[0,0])
legend('x^3-3.000001*x^2+3*x-0.999999')

figure(3)
a=0.999999993; b=1.000000007; h=0.000000000005;
x=a:h:b; y=x.^3-3.000001*x.^2+3*x-0.999999;
axis([a b -1e-13 1e-13])
plot(x,y)
line([a,b],[0,0])
```

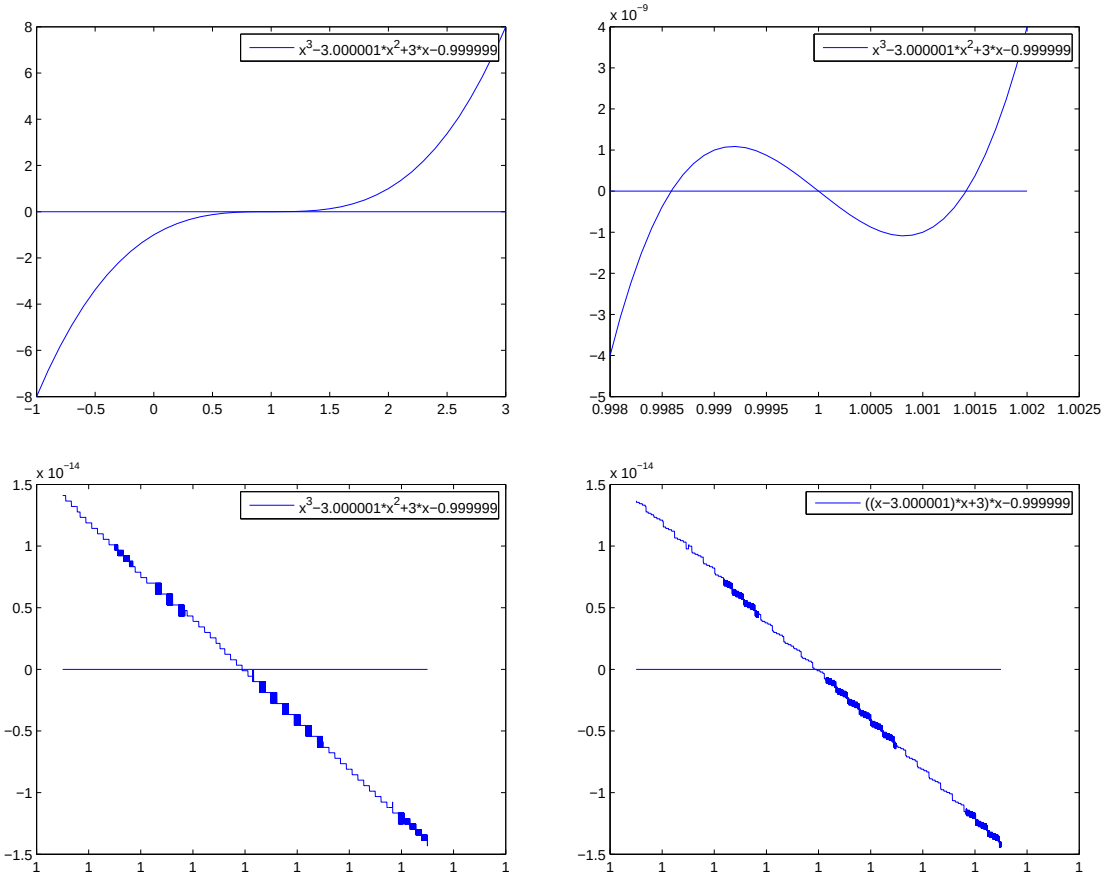


Figure 1.4: Close to the middle root, monotonicity is lost in finite precision arithmetic. Using Horner's rule in the last graph, we see that the result is slightly better.

```
legend('x^3-3.000001*x^2+3*x-0.999999')

figure(4)                                          % using Horner's rule
a=0.999999993; b=1.000000007; h=0.000000000005;
x=a:h:b; y=((x-3.000001).*x+3).*x-0.999999;
axis([a b -1e-13 1e-13])
plot(x,y)
line([a,b],[0,0])
legend('((x-3.000001)*x+3)*x-0.999999')
```

If we zoom in to the root at 1, we see in Figure 1.4 that f behaves like a step function and we cannot ensure monotonicity.

1.7.5 Avoiding Overflow

To avoid overflow, it is often necessary to modify the way quantities are computed. Assume for example that we wish to compute the polar coordinates of a given point

(x, y) in the plane. To compute the radius $r > 0$, the textbook approach is to use

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

However, if $|x|$ or $|y|$ is larger than $\sqrt{\texttt{realmax}}$, then x^2 or y^2 will overflow and produce the result Inf and hence also $r = Inf$. Consider for example $x = 1.5e200$ and $y = 3.6e195$. Then

$$r^2 = 2.25e400 + 12.96e390 = 2.250000001296e400 > \texttt{realmax},$$

but $r = 1.500000000432e200$ would be well within the range of the machine numbers. To compute r without overflowing, one remedy is to factor out the large quantities:

```
>> x=1.5e200
x = 1.500000000000000e+200
>> y=3.6e195
y = 3.600000000000000e+195
>> if abs(x)>abs(y),
r=abs(x)*sqrt(1+(y/x)^2)
elseif y==0,
r=0
else
r=abs(y)*sqrt((x/y)^2+1)
end
r = 1.500000000432000e+200
```

A simpler program (with more operations) is the following:

```
m=max(abs(x),abs(y));
if m==0,
r=0
else
r=m*sqrt((x/m)^2+(y/m)^2)
end
```

Note that with both solutions we also avoid possible underflow when computing r.

1.7.6 Testing for Overflow

Assume we want to compute x^2 but we need to know if it overflows. With the IEEE standard, it is simple to detect this:

```
if x^2==Inf
```

Without IEEE, the computationmight haltwith an error message. A machine-independent test that works in almost all cases for normalized numbers is

```
if (1/x)/x==0 % then x^2 will overflow
```

In the case we want to avoid working with denormalized numbers, the test should be

```
if (eps/x)/x==0 % then x^2 will overflow
```

It is however difficult to guarantee that such a test catches overflow for all machine numbers.

In the IEEE standard, `realmin` and `realmax` are not quite symmetric, since the equation

$$\texttt{realmax} \times \texttt{realmin} = c \approx 4$$

holds with some constant c which depends on the processor and/or the version of Matlab. In an ideal situation, we would have $c = 1$ in order to obtain perfect symmetry.

1.7.7 Avoiding Cancellation

We have already seen in Subsection 1.4.2 how to avoid cancellation when calculating the area of a circle. Consider as a second example for cancellation the computation of the exponential function using the Taylor series:

$$e^x = \sum_{j=0}^{\infty} \frac{x^j}{j!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

It is well known that the series converges for any x. A naive approach is therefore (in preparation of the better version later, we write the computation in the loop already in a particular form):

Algorithm 4 Computation of e^x, Naive Version

```
function s=ExpUnstable(x,tol);
% EXPUNSTABLE computation of the exponential function
% s=ExpUnstable(x,tol); computes an approximation s of exp(x)
% up to a given tolerance tol.
% WARNING: cancellation for large negative x.

s=1; term=1; k=1;
while abs(term)>tol*abs(s)
  so=s; term=term*x/k;
  s=so+term; k=k+1;
end
```

For positive x, and also small negative x, this program works quite well:

```
>> ExpUnstable(20,1e-8)
ans = 4.851651930670549e+08
>> exp(20)
ans = 4.851651954097903e+08
>> ExpUnstable(1,1e-8)
ans = 2.718281826198493e+00
>> exp(1)
```

```
ans = 2.718281828459045e+00
>> ExpUnstable(-1,1e-8)
ans = 3.678794413212817e-01
>> exp(-1)
ans = 3.678794411714423e-01
But for large negative x, e.g. for x = -20 and x = -50, we obtain
>> ExpUnstable(-20,1e-8)
ans = 5.621884467407823e-09
>> exp(-20)
ans = 2.061153622438558e-09
>> ExpUnstable(-50,1e-8)
ans = 1.107293340015503e+04
>> exp(-50)
ans = 1.928749847963918e-22
```

which are completely incorrect. The reason is that for $x = -20$, the terms in the series

$$1 - \frac{20}{1!} + \frac{20^2}{2!} - \cdots + \frac{20^{20}}{20!} - \frac{20^{21}}{21!} + \cdots$$

become large and have alternating signs. The largest terms are

$$\frac{20^{19}}{19!} = \frac{20^{20}}{20!} = 4.3e7.$$

The partial sums should converge to $e^{-20} = 2.06e - 9$. But because of the growth of the terms, the partial sums become large as well and oscillate as shown in Figure 1.5. Table 1.4 shows that the largest partial sum has about the same size as the largest term. Since the large partial sums have to be cancellation. Neither does it help to first sum up all positive and negative parts separately, because when the two sums are subtracted at the end, the result would again suffer from catastrophic cancellation. Indeed, since the result

$$e^{-20} \approx 10^{-17}\frac{20^{20}}{20!}$$

is about 17 orders of magnitude smaller than the largest intermediate partial sum and the IEEE Standard has only about 16 decimal digits of accuracy, we cannot expect to obtain even one correct digit!

number of terms summed	partial sum
20	$-2.182259377927747e+07$
40	$-9.033771892137873e+03$
60	$-1.042344520180466e-04$
80	$6.138258384586164e-09$
100	$6.138259738609464e-09$
120	$6.138259738609464e-09$
exact value	2.061153622438558e-09

Table 1.4: Numerically Computed Partial Sums of e^{-20}

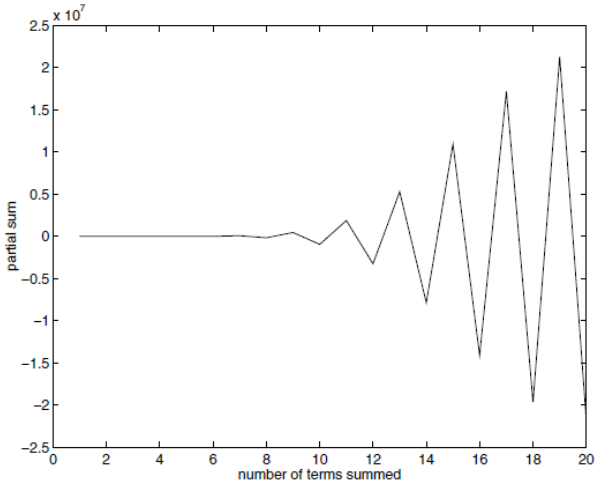


Figure 1.5: Partial sum of the Taylor expansion of e^{-20}

1.7.8 Computation of Mean and Standard Deviation

A third example for cancellation is the recursive computation of the mean and the *standard deviation* of a sequence of numbers. Given the real numbers $x_1, x_2, \ldots, x_n$, the mean is

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{1.13}$$

One definition of the variance is

$$\mathrm{var}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_n)^2. \tag{1.14}$$

The square-root of the variance is the standard deviation

$$\sigma_n = \sqrt{\mathrm{var}(\boldsymbol{x})}. \tag{1.15}$$

Computing the variance using (1.14) requires two runs through the data x_i. By manipulating the variance formula as follows, we can obtain a new expression allowing us to compute both quantities with only one run through the data. By expanding the square bracket we obtain from (1.14)

$$\mathrm{var}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2\mu_n x_i + \mu_n^2) = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\mu_n \frac{1}{n}\sum_{i=1}^{n} x_i + \mu_n^2 \frac{1}{n}\sum_{i=1}^{n} 1,$$

which simplifies to

$$\sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \mu_n^2 \tag{1.16}$$

This relation leads to the classical recursive computation of mean, variance and standard deviation. In the following test, we use the values

```
x=100*ones(100,1)+1e-5*(rand(100,1)-0.5)
```

and compare the results with the Matlab functions `mean`, `var` and `std`, which perform two runs through the data:

Algorithm 5 Mean, Standard Deviation – Classical Unstable Computation

```
format long
x=100*ones(100,1)+1e-5*(rand(100,1)-0.5);
s=0; sq=0; n=0;
while n<length(x),
  n=n+1;
  s=s+x(n);
  sq=sq+x(n)^2;
  mu=s/n;
end
means=[mu mean(x)]
sigma2=sq/n-mu^2;
variances=[sigma2 var(x,1)]
sigma=sqrt(sigma2);
standarddev=[sigma std(x,1)]
```

Each execution of these statements will be different since we use the function `rand` to generate the x_i. However, we typically get results like

```
means =
1.0e+02 *
1.00000000308131 1.00000000308131
variances =
1.0e-11 *
0.90949470177293 0.81380653750974
standarddev =
1.0e-05 *
0.30157829858478 0.28527294605513
```

which show that the classical formulas are numerically unstable. It may even happen that the standard deviation becomes complex because the variance becomes negative! Of course, this is a numerical effect due to severe cancellation, which can occur when using (1.16).

A better updating formula, which avoids cancellation, can be derived as follows:

$$\begin{aligned}
n\sigma_n^2 &= \sum_{i=1}^{n}(x_i-\mu_n)^2 \\
&= \sum_{i=1}^{n-1}(x_i-\mu_n)^2+(x_n-\mu_n)^2 \\
&= \sum_{i=1}^{n-1}((x_i-\mu_{n-1})-(\mu_n-\mu_{n-1}))^2+(x_n-\mu_n)^2 \\
&= \sum_{i=1}^{n-1}(x_i-\mu_{n-1})^2-2(\mu_n-\mu_{n-1})\sum_{i=1}^{n-1}(x_i-\mu_{n-1})+(n-1)(\mu_n-\mu_{n-1})^2+(x_n-\mu_n)^2 \\
&= (n-1)\sigma_{n-1}^2+0+(n-1)(\mu_n-\mu_{n-1})^2+(x_n-\mu_n)^2.
\end{aligned}$$

For the mean we have the relation

$$n\mu_n=(n-1)\mu_{n-1}+x_n,$$

which implies

$$\mu_{n-1}=\frac{n}{n-1}\mu_n-\frac{1}{n-1}x_n,$$

and therefore

$$(n-1)(\mu_n-\mu_{n-1})^2=(n-1)(\mu_n-\frac{n}{n-1}\mu_n+\frac{1}{n-1}x_n)^2=\frac{(x_n-\mu_n)^2}{n-1}.$$

Using this in the recursion for σ_n^2 , we obtain

$$n\sigma_n^2=(n-1)\sigma_{n-1}^2+\frac{n}{n-1}(x_n-\mu_n)^2,$$

and finally

$$\sigma_n^2=\frac{(n-1)}{n}\sigma_{n-1}^2+\frac{1}{n-1}(x_n-\mu_n)^2. \tag{1.17}$$

This leads to the new algorithm

With this new algorithm, we now obtain significantly better results. A typical run gives

```
means =
1.0e+02 *
1.00000000308131 1.00000000308131
variances =
1.0e-11 *
0.81380653819342 0.81380653750974
standarddev =
1.0e-05 *
0.28527294617496 0.28527294605513
```

Algorithm 6 Mean, Standard Deviation − Stable Computation

```
format long
x=100*ones(100,1)+1e-5*(rand(100,1)-0.5);
s=x(1);mu=s;sigma2=0;n=1;
while n<length(x),
  n=n+1;
  s=s+x(n);
  mu=s/n;
  sigma2=(n-1)*sigma2/n+(x(n)-mu)^2/(n-1);
end
means=[mu mean(x)]
variances=[sigma2 var(x,1)]
sigma=sqrt(sigma2);
standarddev=[sigma std(x,1)]
```

1.8 Stopping Criteria

An important problem when computing approximate solutions using a computer is to decide when the approximation is accurate enough. When computing in finite precision arithmetic, the properties discussed in the previous sections can often be exploited to design elegant algorithms that work *because of* (and not in spite of) rounding errors and the finiteness of the set of machine numbers.

1.8.1 Machine-independent Algorithms

Consider again as an example the computation of the exponential function using the Taylor series. We saw that we obtained good results for $x > 0$. Using the *Stirling Formula* $n! \sim \sqrt{2\pi}(\frac{n}{e})^n$, we see that for a given x, the $n-$th term satisfies

$$t_n = \frac{x^n}{n!} \sim \frac{1}{\sqrt{2\pi}}(\frac{xe}{n})^n \to 0, n \to \infty.$$

The largest term in the expansion is therefore around $n \approx |x|$, as one can see by differentiation. For larger n, the terms decrease and converge to zero. Numerically, the term t_n becomes so small that in finite precision arithmetic we have

$$s_n + t_n = s_n, \quad \text{with} \quad s_n = \sum_{i=0}^{n} \frac{x^i}{i!}.$$

This is an elegant termination criterion which does not depend on the details of the floating point arithmetic but makes use of the finite number of digits in the mantissa. This way the algorithm is *machine-independent*; it would not work in exact arithmetic, however, since it would never terminate.

In order to avoid cancellation when $x < 0$, we use a property of the exponential function, namely $e^x = 1/e^{-x}$: we first compute $e^{|x|}$, and then $e^x = 1/e^{|x|}$. We thus get the following stable algorithm for computing the exponential function for all x:

Algorithm 7 Stable Computation of e^x

```
function s=Exp(x);
% EXP stable computation of the exponential function
% s=Exp(x); computes an approximation s of exp(x) up to machine
% precision.

if x<0, v=-1; x=abs(x); else v=1; end
  so=0; s=1; term=1; k=1;
  while s~=so
    so=s; term=term*x/k;
    s=so+term; k=k+1;
  end
if v<0, s=1/s; end;
```

We now obtain very good results also for large negative x:

```
>> Exp(-20)
ans = 2.061153622438558e-09
>> exp(-20)
ans = 2.061153622438558e-09

>> Exp(-50)
ans = 1.928749847963917e-22
>> exp(-50)
ans = 1.928749847963918e-22
```

Note that we have to compute the terms recursively

$$t_k = t_{k-1}\frac{x}{k} \quad \text{and not explicitly} \quad t_k = \frac{x^k}{k!}$$

in order to avoid possible overflow in the numerator or denominator.

As a second example, consider the problem of designing an algorithm to compute the square root. Given $a > 0$, we wish to compute

$$x = \sqrt{a} \Longleftrightarrow f(x) = x^2 - a = 0.$$

Applying *Newton's iteration*, we obtain

$$x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x} = \frac{1}{2}(x + \frac{a}{x})$$

and the quadratically convergent iteration (also known as *Heron's formula*)

$$x_{k+1} = (x_k + a/x_k)/2. \tag{1.18}$$

When should we terminate the iteration? We could of course test to see if successive iterations are identical up to some relative tolerance. But here we can develop a much nicer termination criterion. The geometric interpretation of Newton's method shows us that if $\sqrt{a} < x_k$ then $\sqrt{a} < x_{k+1} < x_k$. Thus if we start the iteration with $\sqrt{a} < x_0$ then the sequence $\{x_k\}$ is *monotonically decreasing* toward $s = \sqrt{a}$. This monotonicity cannot hold forever on a machine with finite precision arithmetic. So when it is lost we have reached machine precision.

To use this criterion, we must ensure that $\sqrt{a} < x_0$. This is easily achieved, because one can see geometrically that after the first iteration starting with any positive number, the next iterate is always larger than $\sqrt{a}$. If we start for example with $x_0 = 1$, the next iterate is $(1 + a)/2 \geq \sqrt{a}$. Thus we obtain Algorithm 8.

Algorithm 8 Computing $\sqrt{x}$ machine-independently

```
function y=Sqrt(a);
% SQRT computes the square-root of a positive number
% y=Sqrt(a); computes the square-root of the positive real
% number a using Newton's method, up to machine precision.

xo=(1+a)/2; xn=(xo+a/xo)/2;
while xn<xo
  xo=xn; xn=(xo+a/xo)/2;
end
y=(xo+xn)/2;
```

Notice the elegance of Algorithm 8: there is no tolerance needed for the termination criterion. The algorithm computes the square root on any computer without knowing the machine precision by simply using the fact that there is always only a *finite set of machine numbers*. This algorithm would not work on a machine with exact arithmetic – it relies on finite precision arithmetic. Often these are the best algorithms one can design.

Another example of a *fool-proof* and *machine-independent algorithm* is given in Chapter 4. The bisection algorithm for finding a simple root makes use of the fact that there is only a finite set of machine numbers. Bisection is continued as long as there is a machine number in the interval (a, b). When the interval consists only of the endpoints then the iteration is terminated in a *machine-independent way*. See Algorithm 24 for details.

Machine-independent algorithms are not easy to find. We show in the next subsections two generic stopping criteria that are often used in practice when no machine-independent criterion is available.

1.8.2 Test Successive Approximations

If we are interested in the limit s of a convergent sequence x_k, a commonly used stopping criterion is to check the absolute or relative difference of two *successive approximations*

$$|x_{k+1} - x_k| < \texttt{tol} \quad \text{absolute or} \quad |x_{k+1} - x_k| < \texttt{tol}|x_{k+1}| \quad \text{relative "error"}.$$

The test involves the absolute (or relative) difference of two successive iterates, to which one often refers somewhat sloppily as absolute or relative error. It is of course questionable whether the corresponding errors $|x_{k+1} - s|$ and $|x_{k+1} - s|/|s|$ are indeed small. This is certainly not the case if convergence is very slow (see Chapter 4 , Equation (4.30)), since we can be far away from the solution s and making very small steps toward it. In that case, the above stopping criterion will terminate the iteration prematurely.

Consider as an example the equation $xe^{10x} = 0.001$. A fixed point iteration is obtained by adding x an both sides and dividing by $1 + e^{10x}$,

$$x_{k+1} = \frac{0.001 + x_k}{1 + e^{10x_k}}. \tag{1.19}$$

If we start the iteration with $x_0 = -10$, we obtain the iterates

$$x_1 = -9.9990, \quad x_2 = -9.9980, \quad x_3 = -9.9970.$$

It would be incorrect to conclude that we are close to the solution $s \approx -9.99$, since the only solution of this equation is $s = 0.0009901473844$.

We will see in Chapter 4 that for fixed point iterations the Banach Fixed Point Theorem often allows us to derive a stopping criterion based on the difference of consecutive iterates, which guarantees asymptotically that the current approximation is within a given tolerance of the solution, see Equation (4.31).

1.8.3 Check the Residual

Another possibility to check whether an approximate solution is good enough is to insert this approximation into the equation to be solved, so that one can measure the amount by which the approximation fails to satisfy the equation. This discrepancy is called the *residual* r. For example, in case of the square root above, one might want to check if $r = x_k^2 - a$ is small in absolute value. In the case of a system of linear equations, $A\boldsymbol{x} = \mathbf{b}$, one checks how small the residual

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_k$$

becomes in some norm for an approximate solution $\boldsymbol{x}_k$.

Unfortunately, a small residual does not guarantee that we are close to a solution either ! Take as an example the linear system

$$A\boldsymbol{x} = \boldsymbol{b}, \quad A = \begin{pmatrix} 0.4343 & 0.4340 \\ 0.4340 & 0.4347 \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The exact solution is

$$\boldsymbol{x} = \frac{1}{9}\begin{pmatrix} -43370000 \\ 43400000 \end{pmatrix} = \begin{pmatrix} -4.81888\ldots \\ 4.82222\ldots \end{pmatrix} 10^6.$$

The entries of the matrix A are decimal numbers with 4 digits. The best $4-$digit decimal approximation to the exact solution is

$$\boldsymbol{x}_4 = \begin{pmatrix} -4819000 \\ 4822000 \end{pmatrix}.$$

Now if we compute the residual of that approximation we obtain the rather large residual

$$\boldsymbol{r}_4 = \boldsymbol{b} - A\boldsymbol{x}_4 = \begin{pmatrix} 144.7 \\ 144.6 \end{pmatrix};$$

We can easily guess solutions with smaller residuals, but which clearly are not better solutions. For example, we could have proposed as solution

$$\boldsymbol{x}_1 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \Rightarrow \boldsymbol{r}_1 = \boldsymbol{b} - A\boldsymbol{x}_1 = \begin{pmatrix} 1.0003 \\ 0.0003 \end{pmatrix},$$

which clearly has a smaller residual. In fact, the residual of $\boldsymbol{x} = (0,0)^\top$ is $\boldsymbol{r} = \boldsymbol{b} = (1,0)^\top$, which is even smaller! *Thus, we cannot trust small residuals to always imply that we are close to a solution.*

Chapter 2

Linear Systems of Equations

Prerequisites: Sections 1.2 (finite-precision arithmetic), 1.5 (conditioning) and 1.6 (stability) are required for this chapter.

Solving a system of linear equations is one of the most frequent tasks in numerical computing. The reason is twofold: historically, many phenomena in physics and engineering have been modeled by linear differential equations, since they are much easier to analyze than nonlinear ones. In addition, even when the model is nonlinear, the problem is often solved iteratively as a sequence of linear problems, e.g., by Newton's method (Chapter 4). Thus, it is important to be able to solve linear equations efficiently and robustly, and to understand how numerical artifacts affect the quality of the solution. We start with an introductory example, where we also mention Cramer's rule, a formula used by generations of mathematicians to write down explicit solutions of linear systems, but which is not at all suitable for computations. We then show in Section 2.2 the fundamental technique of Gaussian elimination with pivoting, which is the basis of LU decomposition, the workhorse present in all modern dense linear solvers. This decomposition approach to matrix computations, pioneered by Householder, represents a real paradigm shift in the solution of linear systems and is listed as one of the top ten algorithms of the last century think factorization, not solution. In Section 2.3, we introduce the important concept of the condition number of a matrix, which is the essential quantity for understanding the condition of solving a linear system. We then use Wilkinson's Principle to show how the condition number influences the expected accuracy of the solution of the associated linear system. The special case of symmetric positive definite systems is discussed in Section 2.4, where the LU factorization can be expressed in the very special form $L = U^\top$ due to symmetry, leading to the so-called Cholesky factorization. An alternative for computing the solution of linear systems that does not require pivoting is shown in Section 2.5, where Givens rotations are introduced. We conclude this chapter with special factorization techniques for banded matrices in Section 2.6. The focus of the chapter is on direct methods. Moreover, we consider in this chapter only square linear systems (systems that have as many equations as unknowns) whose matrices are nonsingular, i.e., systems that have a unique solution.

2.1 Introductory Example

As a simple example, we consider the geometric problem of finding the intersection point of three planes α, β and γ given in normal form:

$$\begin{aligned} \alpha: \quad & 4x_1 + x_2 + x_3 = 2, \\ \beta: \quad & x_2 + 2x_3 = 3, \\ \gamma: \quad & -5x_1 + 2x_3 = 5. \end{aligned} \tag{2.1}$$

If the normal vectors of the three planes are not coplanar (i.e., do not lie on the same plane themselves), then there is exactly one intersection point $\boldsymbol{x} = (x_1, x_2, x_3)^\top$ satisfying the three equations simultaneously. Equations (2.1) form a *system of linear equations*, with three equations in three unknowns. More generally, a system of n equations in n unknowns written componentwise is

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + \cdots + a_{2n}x_n &= b_2, \\ \vdots \quad & \quad \vdots \quad \vdots \quad \vdots \\ a_{n1}x_1 + \cdots + a_{nn}x_n &= b_n. \end{aligned} \tag{2.2}$$

The constants a_{ij} are called the coefficients and the b_i form the right-hand side. The coefficients are collected in the matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}. \tag{2.3}$$

The right-hand side is the vector

$$\boldsymbol{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, \tag{2.4}$$

and we collect the x_i in the vector of unknowns

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}. \tag{2.5}$$

In matrix-vector notation, the linear system (2.2) is therefore

$$A\boldsymbol{x} = \boldsymbol{b}. \tag{2.6}$$

For Example (2.1.1) we get

$$A = \begin{pmatrix} 4 & 1 & 1 \\ 0 & 1 & 2 \\ -5 & 0 & 2 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{2.7}$$

Often it is useful to consider the *columns* of a matrix A,

$$A = (\boldsymbol{a}_{:,1}, \boldsymbol{a}_{:,2}, \ldots, \boldsymbol{a}_{:,n}), \tag{2.8}$$

$$\boldsymbol{a}_{:,i} = \begin{pmatrix} a_{1i} \\ \vdots \\ a_{ni} \end{pmatrix} \tag{2.9}$$

denotes the ith column vector. In Matlab we can address $\boldsymbol{a}_{:,i}$ by the expression `A(:,i)`. Similarly we denote by

$$a_{k,:} = (a_{k1}, \ldots, a_{kn})$$

the kth row vector of A and thus

$$A = \begin{pmatrix} \boldsymbol{a}_{1:} \\ \vdots \\ \boldsymbol{a}_{n:} \end{pmatrix}. \tag{2.10}$$

The expression in Matlab for $\boldsymbol{a}_{k:}$ is `A(k,:)`. Notice that in (2.1), the normal vectors to the planes are precisely the rows of A.

An common way to test whether the three normal vectors are coplanar uses *determinants*, which calculate the (signed) volume of the parallelepiped with edges given by three vectors. The determinant is, in fact, defined for a general $n \times n$ matrix A by the real number

$$\det(A) := \sum_{\boldsymbol{k}} (-1)^{\delta(\boldsymbol{k})} a_{1k_1} a_{2k_2} a_{3k_3} \ldots a_{nk_n}, \tag{2.11}$$

where the vector of indices $\boldsymbol{k} = \{k_1, \ldots, k_n\}$ takes all values of the permutations of the indices $\{1, 2, \ldots, n\}$. The sign is defined by $\delta(\boldsymbol{k})$, which equals 0 or 1 depending on the permutation being even or odd. This formula is called the *Leibniz formula for determinants*.

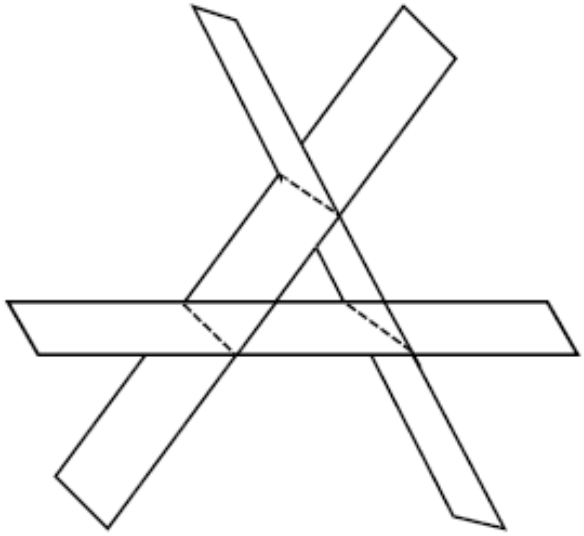
Example 2.1.1. *For $n = 2$ we have*

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}a_{21}, \tag{2.12}$$

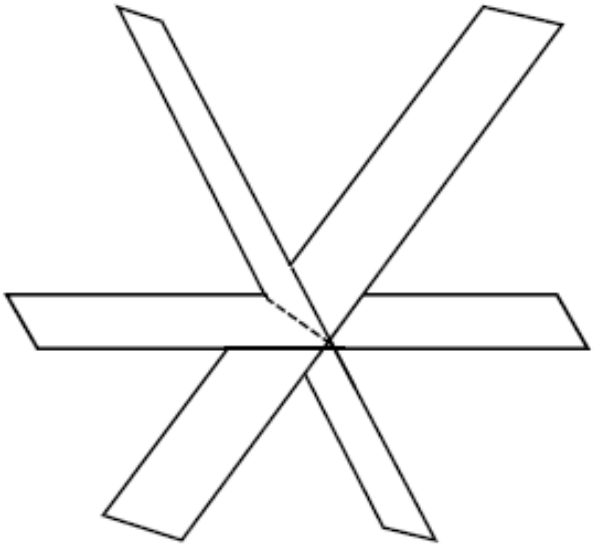
and for n=3 we obtain

$$\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} + a_{12}a_{23}a_{31} - a_{12}a_{21}a_{33} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}. \tag{2.13}$$

In our geometric example, the rows of the coefficient matrix A are precisely the normal vectors to the planes, so $|\det(A)|$ is the volume of the parallelepiped generated by the three vectors. If this volume is zero, then the parallelepiped "collapses" onto the same plane, which implies our system either has no solution (no common intersection point, see Figure 2.1(a)), or has infinitely many solutions (intersection along a whole line



(a)



(b)

Figure 2.1: Three planes with (a) no common intersection, (b) infinitely many intersections along a straight line

or plane, see Figure 2.1(b)). However, if $\det(A) \neq 0$, then there is a unique intersection point, so the solution to (2.1) is unique.

Instead of using Definition (2.11), we can also compute the determinant using the Laplace Expansion. For each row i we have

$$\det(A) = \sum_{j=1}^{n} a_{ij}(-1)^{i+j} \det(M_{ij}), \tag{2.14}$$

where M_{ij} denotes the $(n-1)\times(n-1)$ submatrix obtained by deleting row i and column j of the matrix A. Instead of expanding the determinant as in (2.14) along a row, we can also use an expansion along a column.
The recursive Matlab program(Algorithm 9) computes a determinant using the Laplace Expansion for the first row.

Algorithm 9 Determinant by Laplace Expansion

```
function d=DetLaplace(A);
% DETLAPLACE determinant using Laplace expansion
% d=DetLaplace(A); computes the determinant d of the matrix A
% using the Laplace expansion for the first row.
n=length(A);
if n==1;
  d=A(1,1);
else
  d=0; v=1;
  for j=1:n
      M1j=[A(2:n,1:j-1) A(2:n,j+1:n)];
    d=d+v*A(1,j)*DetLaplace(M1j);
    v=-v;
  end
end
```

The following equation holds for determinants,

$$\det(AB) = \det(A)\det(B). \tag{2.15}$$

This equation allows us to give an explicit expression for the solution of the linear system (2.6). Consider replacing in the identity matrix

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}.$$

the $i-$th column vector $\boldsymbol{e}_i$ by $\boldsymbol{x}$ to obtain the matrix

$$E = (\boldsymbol{e}_1, \dots, \boldsymbol{e}_{i-1}, \boldsymbol{x}, \boldsymbol{e}_{i+1}, \dots, \boldsymbol{e}_n).$$

The determinant is simply

$$\det(E) = x_i, \tag{2.16}$$

as one can see immediately by expanding along the $i-$th row. Furthermore,

$$AE = (A\boldsymbol{e}_1, \ldots, A\boldsymbol{e}_{i-1}, A\boldsymbol{x}, A\boldsymbol{e}_{i+1}, \ldots, A\boldsymbol{e}_n),$$

and because $A\boldsymbol{x} = \boldsymbol{b}$ and $A\boldsymbol{e}_k = \boldsymbol{a}_{:k}$, we obtain

$$AE = (\boldsymbol{a}_{:1}, \ldots, \boldsymbol{a}_{:i-1}, \boldsymbol{b}, \boldsymbol{a}_{:i+1}, \ldots, \boldsymbol{a}_{:n}). \tag{2.17}$$

If we denote the matrix on the right hand side of (2.17) by A_i and if we compute the determinant on both sides we get

$$\det(A)\det(E) = \det(A_i).$$

Using Equation (2.16) we get

Theorem 2.1.1. *(Cramer's Rule) For* $\det(A) \neq 0$*, the linear system* $Ax = b$ *has the unique solution*

$$x_i = \frac{\det(A_i)}{\det(A)}, \quad i = 1, 2, \ldots, n, \tag{2.18}$$

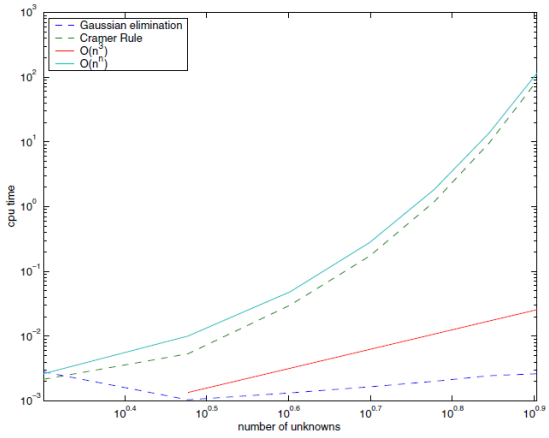
where A_i *is the matrix obtained from* A *by replacing column* $\boldsymbol{a}_{:i}$ *by* $\boldsymbol{b}$*.*

The following Matlab program computes the solution of a linear system with Cramer's rule:

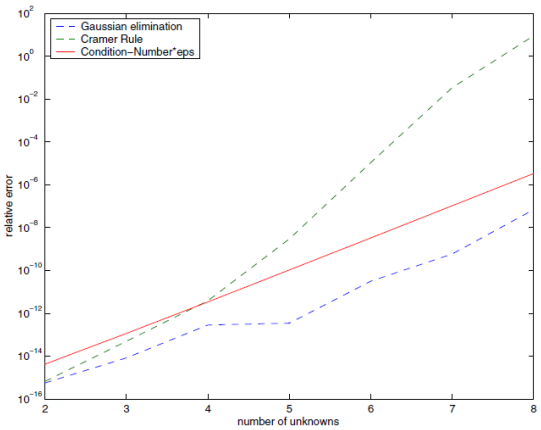
Algorithm 10 Cramer's Rule

```
function x=Cramer(A,b);
% CRAMER solves a linear Sytem with Cramer's rule
% x=Cramer(A,b); Solves the linear system Ax=b using Cramer's
% rule. The determinants are computed using the function
DetLaplace.
n=length(b);
detA=DetLaplace(A);
for i=1:n
  AI=[A(:,1:i-1), b, A(:,i+1:n)];
  x(i)=DetLaplace(AI)/detA;
end
x = x(:);
```

Cramer's rule looks simple and even elegant, but for computational purposes it is a disaster, as we show in Figure 2.2 for the Hilbert matrices. The computational effort with Laplace expansion is $O(n!)$, while with Gaussian elimination (introduced in the next section), it is $O(n^3)$. Furthermore, the numerical accuracy due to finite precision arithmetic is very poor: it is very likely that cancellation occurs within the Laplace expansion, as one can see in Figure 2.2.



(a)



(b)

Figure 2.2: Comparison of speed (above) and accuracy (below) of Cramer's rule with Gaussian Elimination

2.2 Gaussian Elimination

With *Gaussian Elimination*, one tries to reduce a given linear system to an equivalent system with a triangular matrix. As we will see, such systems are very easy to solve.

Example 2.2.1.

$$\begin{aligned} 3x_1 + 5x_2 - x_3 &= 2, \\ 2x_2 - 7x_3 &= -16, \\ -4x_3 &= -8. \end{aligned} \tag{2.19}$$

The matrix U of Equation (2.19) is called an *upper triangular matrix,*

$$A = \begin{pmatrix} 3 & 5 & -1 \\ 0 & 2 & -7 \\ 0 & 0 & -4 \end{pmatrix},$$

since all elements below the main diagonal are zero. The solution of Equation (2.19) is easily computed by back substitution: we compute x_3 from the last equation, obtaining $x_3 = 2$. Then we insert this value into the second last equation and we can solve for $x_2 = -1$. Finally we insert the values for x_2 and x_3 into the first equation and obtain $x_1 = 3$.

If $U \in \mathbb{R}^{n\times n}$ and we solve the $i-$th equation in $U\boldsymbol{x} = \boldsymbol{b}$ for x_i then

$$x_i = (b_i - \sum_{j=i+1}^{n} u_{ij}x_j)/u_{ii}.$$

Therefore we get the following first version for *back substitution*

Algorithm 11 Back substitution

```
function x=BackSubstitution(U,b)
% BACKSUBSTITUTION solves a linear system by backsubstitution
% x=BackSubstitution(U,b) solves Ux=b, U upper triangular by
% backsubstitution

n=length(b);
for k=n:-1:1
  s=b(k);
  for j=k+1:n
    s=s-U(k,j)*x(j);
  end
  x(k)=s/U(k,k);
end
x=x(:);
```

With vector operations, a second variant of back substitution can be formulated using the scalar product. For a third variant, also using vector operations, we can subtract immediately after computing x_i the $i-$th column of U multiplied by x_i from the right-hand side. This simplifies the process to the $SAXPY$ variant[1] of back substitution: This algorithm costs n divisions and $(n-1)+(n-2)+\ldots+1 = \frac{1}{2}n^2 - \frac{1}{2}n$ additions

Algorithm 12 Back substitution, SAXPY-Variant

```
function x=BackSubstitutionSAXPY(U,b)
% BACKSUBSTITUTIONSAXPY solves linear system by backsubstitution
% x=BackSubstitutionSAXPY(U,b) solves Ux=b by backsubstitution by
% modifying the right hand side (SAXPY variant)

n=length(b);
for i=n:-1:1
  x(i)=b(i)/U(i,i);
  b(1:i-1)=b(1:i-1)-x(i)*U(1:i-1,i);
end
x=x(:);
```

and multiplications, and its complexity is thus $O(n^2)$.

We will now reduce in $n-1$ elimination steps the given linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad &\vdots \\ a_{k1}x_1 + a_{k2}x_2 + \ldots + a_{kn}x_n &= b_k \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n \end{aligned} \tag{2.20}$$

to an equivalent system with an upper triangular matrix. A linear system is transformed into an equivalent one by adding to one equation a multiple of another equation. An elimination step consists of adding a suitable multiple in such a way that one unknown is eliminated in the remaining equations.

To eliminate the unknown x_1 in equations #2 to #n, we perform the operations

```
for k=2:n
{new Eq. # k} = {Eq. # k} - ak1/a11{Eq.# 1}
end
```

We obtain a reduced system with an $(n-1)\times(n-1)$ matrix which contains only the unknowns $x_2, \ldots, x_n$. This remaining system is reduced again by one unknown by

[1] SAXPY, which stands for "scalar $a \cdot x$ plus y", is a basic linear algebra operation that overwrites a vector y with the result of $ax+y$, where a is a scalar. This operation is implemented efficiently in several libraries that can be tuned to the machine on which the code is executed.

freezing the second equation and eliminating x_2 in equations #3 to #n. We continue this way until only one equation with one unknown remains. This way we have reduced the original system to a system with an upper triangular matrix. The whole process is described by two nested loops:

```
for i=1:n-1
for k=i+1:n
{new Eq. # k} = {Eq. # k} - aki/aii{Eq. # i}
end
end
```

The coefficients of the $k-$th new equation are computed as

$$a_{kj} := a_{kj} - \frac{a_{ki}}{a_{ii}} a_{ij} \qquad \text{for} \quad j = i+1, \ldots, n. \tag{2.21}$$

and the right-hand side also changes,

$$b_k := b_k - \frac{a_{ki}}{a_{ii}} b_i.$$

Note that the $k-$th elimination step (2.21) is a rank-one change of the remaining matrix. Thus, if we append the right hand side to the matrix A by `A=[A, b]`, then the elimination becomes

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

where the inner loop over k has been subsumed by Matlab's vector notation. Note that we did not compute the zeros in `A(i+1:n,i)`. Rather we used these matrix elements to store the factors necessary to eliminate the unknown x_i.

Example 2.2.2. *We consider the linear system $A\boldsymbol{x} = \boldsymbol{b}$ with `A=invhilb(4)`,*

$$A = \begin{pmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{pmatrix}, \quad b = \begin{pmatrix} -4 \\ 6 \\ -180 \\ 140 \end{pmatrix}$$

The right-hand side $\boldsymbol{b}$ was chosen in such a way that the solution is $\boldsymbol{x} = (1,1,1,1)^\top$. If we apply the above elimination procedure to the augmented matrix $[A, \boldsymbol{b}]$, then we obtain

```
A =
16.0000 -120.0000  240.0000 -140.0000 -4.0000
-7.5000   300.0000 -900.0000  630.0000  30.0000
15.0000 -3.0000    180.0000 -210.0000 -30.0000
-8.7500   2.1000   -1.1667    7.0000    7.0000
>> U=triu(A)
U =
```

```
16.0000 -120.0000  240.0000 -140.0000 -4.0000
0        300.0000 -900.0000  630.0000  30.0000
0        0         180.0000 -210.0000 -30.0000
0        0         0         7.0000    7.0000
```

Thus the equivalent reduced system is

$$\begin{pmatrix} 16 & -120 & 240 & -140 \\ & 300 & -900 & 630 \\ & & 180 & -210 \\ & & & 7 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} -4 \\ 30 \\ -30 \\ 7 \end{pmatrix}$$

and has the same solution $\boldsymbol{x} = (1, 1, 1, 1)^\top$.

There is unfortunately a glitch. Our elimination process may fail if, in step i, the $i-$th equation does not contain the unknown x_i, i.e., if the $(i, i)-$th coefficient is zero. Then we cannot use this equation to eliminate x_i in the remaining equations.

Example 2.2.3.

$$\begin{aligned} x_2 + 3x_3 &= -6 \\ 2x_1 - x_2 + x_3 &= 10 \\ -3x_1 + 5x_2 - 7x_3 &= 10 \end{aligned}$$

Since the first equation does not contain x_1*, we cannot use it to eliminate* x_1 *in the second and third equation.*

Obviously, the solution of a linear system does not depend on the ordering of the equations. It is therefore very natural to reorder the equations in such a way that we obtain a pivot-element $a_{ii} \neq 0$. From the point of view of numerical stability, $a_{ii} \neq 0$ is not sufficient; we also need to reorder equations if the unknown x_i is only "weakly" contained. The following example illustrates this.

Example 2.2.4. *Consider for* ε *small the linear system*

$$\begin{aligned} \varepsilon x_1 + x_2 &= 1, \\ x_1 + x_2 &= 2. \end{aligned} \tag{2.22}$$

If we interpret the equations as lines in the plane, then their graphs show a clear intersection point near $x_1 = x_2 = 1$*, as one can see for* $\varepsilon = 10 - 7$ *in Figure 2.3. The angle between the two lines is about* 45^o.

If we want to solve this system algebraically, then we might eliminate the first unknown in the second equation by replacing the second equation with the linear combination

$$\{Eq.\#2\} - \frac{1}{\varepsilon}\{Eq.\#1\}.$$

This leads to the new, mathematically equivalent system of equations

$$\begin{aligned} \varepsilon x_1 + x_2 &= 1, \\ (1 - 1/\varepsilon)x_2 &= 2 - 1/\varepsilon. \end{aligned}$$

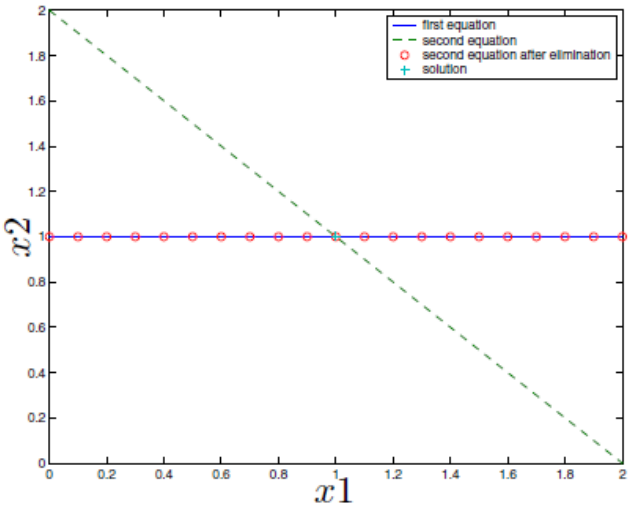


Figure 2.3: Example on how a well-conditioned problem can be transformed into an ill-conditioned one when a small pivot is used in Gaussian elimination

If we again interpret the two equations as lines in the plane, then we see that this time, the two lines are almost coinciding, as shown in Figure 2.3, where we used the circle symbol for the new second line in order to make it visible on top of the first one. The intersection point is now very difficult to find – the problem has become ill conditioned.

What went wrong? We eliminated the unknown using a very small pivot on the diagonal. Doing this, we transformed a well-conditioned problem into an ill-conditioned one. Choosing small pivots makes Gaussian elimination unstable. If, however, we interchange the equations of (2.22) and eliminate afterward, we get

$$\begin{aligned} x_1 + x_2 &= 2 \\ (1-\varepsilon)x_2 &= 1 - 2\varepsilon, \end{aligned} \tag{2.23}$$

a system for which the row vectors of the matrix are nicely linearly independent, and so the intersection point is well defined and can be computed stably.

This observation is the motivation for introducing a pivoting strategy. We consider *partial pivoting*, which means that before each elimination step, we look in the current column for the element with largest absolute value. This element will then be chosen as the pivot. If we cannot find a nonzero pivot element, this means that the corresponding unknown is absent from the remaining equations, i.e., the linear system is singular. In finite precision arithmetic, we cannot expect in the singular case that all possible pivot elements will be exactly zero, since rounding errors will produce rather small (but nonzero) elements; these will have to be compared with the other matrix elements in order to decide if they should be treated as zeros. Therefore, we will consider in the following program a pivot element to be zero if it is smaller than $10^{-14}\|A\|_1$, a reasonable size in practice.

Note that although we have to perform only $n-1$ elimination steps, the for-loop in `Elimination` goes up to n. This is necessary for testing whether a_{nn} becomes zero,

Algorithm 13 Gaussian Elimination with Partial Pivoting function

```
function x=Elimination(A,b)
% ELIMINATION solves a linear system by Gaussian elimination
% x=Elimination(A,b) solves the linear system Ax=b using Gaussian
% Elimination with partial pivoting. Uses the function
% BackSubstitution

n=length(b);
norma=norm(A,1);
A=[A,b]; % augmented matrix
for i=1:n
  [maximum,kmax]=max(abs(A(i:n,i)));      % look for Pivot A(kmax,i)
  kmax=kmax+i-1;
  if maximum < 1e-14*norma;               % only small pivots
    error('matrix is singular')
  end
  if i ~= kmax % interchange rows
    h=A(kmax,:); A(kmax,:)=A(i,:); A(i,:)=h;
  end
A(i+1:n,i)=A(i+1:n,i)/A(i,i);             % elimination step
A(i+1:n,i+1:n+1)=A(i+1:n,i+1:n+1)-A(i+1:n,i)*A(i,i+1:n+1);
end
x=BackSubstitution(A,A(:,n+1));
```

which would indicate that the matrix is singular. The statements corresponding to the actual elimination have an empty index set, and thus have no effect for $i = n$.

Example 2.2.5. *If we consider* `A=magic(4)` *and* $\boldsymbol{b} = (1, 0, 0, 0)^\top$*, then the call* `x=Elimination(A,b)` *will return the error message* `matrix is singular`*. This is correct, since the rank of A is 3.*

In Matlab, a linear system $A\boldsymbol{x} = \boldsymbol{b}$ is solved with the statement `x=A\b`. If A has full rank, the operator \ solves the system using *partial pivoting.*

2.2.1 LU Factorization

Gaussian elimination becomes more transparent if we formulate it using matrix operations. Consider the matrix

$$L_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -l_{21} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -l_{n1} & 0 & 0 & \dots & 1 \end{pmatrix}, \quad l_{j1} := \frac{a_{j1}}{a_{11}}, j = 2, \dots, n. \tag{2.24}$$

Multiplying the linear system $A\boldsymbol{x} = \boldsymbol{b}$ with L_1 from the left, we obtain

$$L_1 A\boldsymbol{x} = L_1\boldsymbol{b}, \tag{2.25}$$

and it is easy to see that this is the system that we get after the first elimination step: the first equation is unchanged and the remaining equations do not contain x_1 any more. Denoting the elements of the matrix $A^{(1)} := L_1 A$ by $a_{ik}^{(1)}$, we can eliminate in the same way the unknown x_2 with the matrix

$$L_2 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & -l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & -l_{n2} & 0 & \dots & 1 \end{pmatrix}, \quad l_{j2} := \frac{a_{j2}^{(1)}}{a_{22}^{(1)}}, j = 3, \dots, n, \tag{2.26}$$

from equations #3 to #n by multiplying the system (2.25) from the left by L_2,

$$L_2 L_1 A\boldsymbol{x} = L_2 L_1\boldsymbol{b}, \tag{2.27}$$

and we obtain the new matrix $A^{(2)} := L_2 A^{(1)} = L_2 L_1 A$. Continuing this way, we obtain the matrices L_k and $A^{(k)}$ for $k = 1, \dots, n-1$, and finally the system

$$A^{(n-1)}\boldsymbol{x} = L_{n-1} \dots L_1 A\boldsymbol{x} = L_{n-1} \dots L_1\boldsymbol{b}, \tag{2.28}$$

where we have now obtained an upper triangular matrix U,

$$A^{(n-1)} = L_{n-1} \dots L_1 A = U \tag{2.29}$$

The matrices L_j are all lower triangular matrices. They are easy to invert, for instance

$$L_1^{-1} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ +l_{21} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ +l_{n1} & 0 & 0 & \dots & 1 \end{pmatrix}. \tag{2.30}$$

Thus we only have to invert the signs. Moving this way the L_i to the right hand side, we obtain

$$A = L_1^{-1} \dots L_{n-1}^{-1} U \tag{2.31}$$

The product of lower triangular matrices is again lower triangular, and therefore

$$L := L_1^{-1} \dots L_{n-1}^{-1} \tag{2.32}$$

is lower triangular. More specifically, we have

$$L = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{pmatrix}, \tag{2.33}$$

where l_{ij} are the multiplication factors which are used for the elimination. Equation (2.31) then reads

$$A = L \cdot U, \tag{2.34}$$

and we have obtained a decomposition of the matrix A into a product of two triangular matrices − the so-called *LU decomposition*. With partial pivoting, the equations (and hence the rows of A) are permuted, so we obtain not a *triangular decomposition* of the matrix A, but that of $\tilde{A} = PA$, where P is a *permutation matrix*, so that $\tilde{A}$ has the same row vectors as A, but in a different order.

Theorem 2.2.1. *(LU Decomposition) Let $A \in \mathbb{R}^{n\times n}$ be a non-singular matrix. Then there exists a permutation matrix P such that*

$$PA = LU, \tag{2.35}$$

where L and U are the lower and upper triangular matrices obtained from Gaussian elimination.

Proof: At the first step of the elimination, we look for the row containing the largest pivot, swap it with the first row, and then perform the elimination. Thus, we have

$$A^{(1)} = L_1 P_1 A,$$

where P_1 interchanges rows #1 and #k_1 with $k_1 > 1$. In the second step, we again swap rows #2 and #k_2 with $k_2 > 2$ before eliminating, i.e.,

$$A^{(2)} = L_2 P_2 A^{(1)} = L_2 P_2 L_1 P_1 A.$$

Continuing this way, we obtain

$$U = A^{(n-1)} = L_{n-1}P_{n-1}\dots L_1P_1A, \tag{2.36}$$

where P_i interchanges rows $\#i$ and $\#k_i$ with $k_i > i$. Our goal is to make all the permutations appear together, instead of being scattered across the factorization. To do so, note that each L_i contains a single column of non-zero entries apart from the diagonal. Thus, it can be written as

$$L_i = I - \boldsymbol{v}_i\boldsymbol{e}_i^\top,$$

where $\boldsymbol{e}_i$ contains 1 at the $i-$th position and zeros everywhere else, and the first i entries of $\boldsymbol{v}_i$ are zero. By direct calculation, we see that

$$\begin{aligned} P_{n-1}\dots P_{i+1}L_i &= P_{n-1}\dots P_{i+1}(I - \boldsymbol{v}_i\boldsymbol{e}_i^\top) \\ &= P_{n-1}\dots P_{i+1} - \tilde{\boldsymbol{v}}_i\boldsymbol{e}_i^\top \\ &= [I - \tilde{\boldsymbol{v}}_i(P_{n-1}\dots P_{i+1}\boldsymbol{e}_i)^\top]P_{n-1}\dots P_{i+1} \end{aligned}$$

where $\tilde{\boldsymbol{v}}_i = P_{n-1}\dots P_{i+1}\boldsymbol{v}_i$ is a permuted version of $\boldsymbol{v}_i$. But the permutation $P_{n-1}\dots P_{i+1}$ only permutes entries $i+1$ to n in a vector; entries 1 to i remain untouched. This means the first i entries of $\tilde{\boldsymbol{v}}_i$ are still zero, and $\boldsymbol{e}_i$ is unchanged by the permutation, i.e., $P_{n-1}\dots P_{i+1}\boldsymbol{e}_i = \boldsymbol{e}_i$. So we in fact have

$$P_{n-1}\dots P_{i+1}L_i = \tilde{L}_iP_{n-1}\dots P_{i+1}$$

with $\tilde{L}_i = I - \tilde{\boldsymbol{v}}_i\boldsymbol{e}_i^\top$ still lower triangular. Now (2.36) implies

$$\begin{aligned} U &= L_{-1}P_{n-1}L_{n-2}P_{n-2}\dots L_2P_2L_1P_1A \\ &= L_{n-1}\tilde{L}_{n-2}P_{n-1}P_{n-2}\dots L_2P_2L_1P_1A \\ &= \dots = (L_{n-1}\tilde{L}_{n-2}\dots\tilde{L}_2\tilde{L}_1)(P_{n-1}\dots P_1)A. \end{aligned}$$

Letting $P = P_{n-1}\dots P_1$ and $L = \tilde{L}_1^{-1}\dots\tilde{L}_{n-2}^{-1}\tilde{L}_{n-1}^{-1}$ completes the proof.

Note that the above proof shows that *we must swap entries in both L and U* when two rows are interchanged. It also means that there *exists* a row permutation such that all the pivots that appear during the elimination are the largest in their respective columns, so for analysis purposes we can assume that A has been "pre-permuted" this way. In practice, of course, the permutation is discovered during the elimination and not known in advance.

A linear system can be solved by the following steps:

1. *Triangular decomposition* of the coefficients matrix $PA = LU$.

2. Apply row changes to the right hand side, $\tilde{\boldsymbol{b}} = P\boldsymbol{b}$, and solve $L\boldsymbol{y} = \tilde{\boldsymbol{b}}$ by *forward substitution.*

3. Solve $U\boldsymbol{x} = \boldsymbol{y}$ by *back substitution.*

The advantage of this arrangement is that for new right hand sides $\boldsymbol{b}$ we do not need to recompute the decomposition. It is sufficient to repeat steps 2 and 3. This leads to substantial computational savings, since the major cost lies in the factorization: to eliminate the first column, one needs $n-1$ divisions and $(n-1)^2$ multiplications and additions. For the second column one needs $n-2$ divisions and $(n-2)^2$ multiplications and additions until the last elimination, where one division and one addition and multiplication are needed. The total number of operations is therefore

$$\sum_{i=1}^{n-1} i + i^2 = \frac{n^3}{3} - \frac{n}{3},$$

which one can obtain from Maple using `sum(i+i^2,i=0..n-1)`. Hence the computation of the LU decomposition costs $O(n^3)$ operations, and is much more expensive than the forward and back substitution, for which the cost is $O(n^2)$.

The LU decomposition can also be used to compute determinants in a numerically sound way, since $\det(A) = \det(L)\det(U) = \det(U)$, and is thus a decomposition useful in its own right. The implementation of the LU decomposition is left to the reader as an exercise see Problem 3.10.

2.2.2 Backward Stability

In order for the above algorithm to give meaningful results, we need to ensure that each of the three steps (LU factorization, forward and back substitution) is backward stable. For the factorization phase, we have seen in Section 2.2 that pivoting is essential for the numerical stability. One might wonder if partial pivoting is enough to guarantee that the algorithm is stable.

Theorem 2.2.2. *(Wilkinson) Let A be an invertible matrix, and $\hat{L}$ and $\hat{U}$ be the numerically computed LU-factors using Gaussian elimination with partial pivoting, $|l_{ij}| \leq 1$ for all i, j. Then for the elements of $\hat{A} := \hat{L}\hat{U}$, we have*

$$|\hat{a}_{ij} - a_{ij}| \leq 2\alpha \min(i-1, j) eps + O(eps^2), \tag{2.37}$$

where $\alpha := \max_{ijk} |\hat{a}_{ij}^{(k)}|$.

Proof: At step k of Gaussian elimination, we compute $\hat{a}_{ij}^{(k)}$ for $i > k, j > k$ using the update formula

$$\begin{aligned} \hat{a}_{ij}^{(k)} &= (\hat{a}_{ij}^{(k-1)} - \hat{l}_{ik}\hat{a}_{kj}^{(k-1)}(1+\varepsilon_{ijk}))(1+\eta_{ijk}) \\ &= \hat{a}_{ij}^{(k-1)} - \hat{l}_{ik}\hat{a}_{kj}^{(k-1)} + \mu_{ijk}, \end{aligned} \tag{2.38}$$

where $|\mu_{ijk}|$ can be bounded using $|\varepsilon_{ijk}| \leq eps$ and $|\eta_{ijk}| \leq eps$:

$$\begin{aligned} |\mu_{ijk}| &\leq \overbrace{|\hat{a}_{ij}^{(k-1)} - \hat{l}_{ik}\hat{a}_{kj}^{(k-1)}|}^{=\hat{a}_{ij}^{(k)} + O(eps)} |\eta_{ijk}| + |\hat{l}_{ik}||\hat{a}_{kj}^{(k-1)}||\varepsilon_{ijk}| + O(eps^2) \\ &\leq 2\alpha\ eps + O(eps^2) \end{aligned}$$

In addition, for $i > j$, we have $\hat{a}_{ij}^{(j)} = 0$ and $\hat{l}_{ij} = \frac{\hat{a}_{ij}^{(j-1)}}{\hat{a}_{jj}^{(j-1)}}(1 + \varepsilon_{ijj})$ which

$$0 = \hat{a}_{ij}^{(j)} = \hat{a}_{ij}^{(j-1)} - \hat{l}_{ij}\hat{a}_{jj}^{(j-1)}\mu_{ijj}, \quad \text{with} \quad |\mu_{ijj}| = |\hat{a}_{ij}^{(j-1)}\varepsilon_{ijj}| \leq \alpha eps.$$

Thus, (2.38) in fact holds whenever $i > j \geq k$ or $j \geq i > k$, with

$$|\mu_{ijk}| \leq 2\alpha\ eps + O(eps^2) \tag{2.39}$$

By the definition of $\hat{A} = \hat{L}\hat{U}$, we have

$$\hat{a}_{ij} = \sum_{k=1}^{\min(i,j)} \hat{l}_{ik}\hat{u}_{kj} = \sum_{k=1}^{\min(i,j)} \hat{l}_{ik}\hat{a}_{kj}^{(k-1)}. \tag{2.40}$$

For the case $i > j$, we obtain, using (2.38) with $i > j \geq k$, a telescopic sum for $\hat{a}_{ij}$:

$$\hat{a}_{ij} = \sum_{k=1}^{j}(\hat{a}_{ij}^{(k-1)} - \hat{a}_{ij}^{(k)} + \mu_{ijk}) = a_{ij} + \sum_{k=1}^{j}\mu_{ijk}, \tag{2.41}$$

since $a_{ij}^{(0)} = a_{ij}$ and $a_{ij}^{(j)} = 0$ for $i > j$. On the other hand, for $i \leq j$, we use (2.40) and (2.38) with $j \geq i > k$ to obtain

$$\hat{a}_{ij} = \sum_{k=1}^{i-1}(\hat{a}_{ij}^{(k-1)} - \hat{a}_{ij}^{(k)} + \mu_{ijk}) + \hat{l}_{ii}\hat{u}_{ij} = a_{ij} + \sum_{k=1}^{i-1}\mu_{ijk}, \tag{2.42}$$

where we used $\hat{l}_{ii} = 1$ and $\hat{u}_{ij} = \hat{a}_{ij}^{(i-1)}$. Combining (2.39), (2.41) and (2.42) yields the desired result.

This theorem shows that Gaussian elimination with partial pivoting is backward stable, if the *growth factor*

$$\rho := \frac{\alpha}{\max|a_{ij}|} \tag{2.43}$$

is not too large, which means that the elements $a_{ij}^{(k)}$ encountered during the elimination process are not growing too much.

Next, we show that the SAXPY variant of back substitution (Algorithm 12) is also backward stable. We show below the floating-point version of back substitution for solving $U\boldsymbol{x} = \boldsymbol{b}$, where the quantities ε_{jk} and η_{jk} all have moduli less than *eps*.

$$\begin{array}{lll}
\hat{b}_k^{(n)} & : & = b_k \\
\text{for} \quad k & & = n, n-1, \ldots, 1 \text{ do} \\
\quad \hat{x}_k & & = \dfrac{\hat{b}_k^{(k)}}{u_{kk}}(1 + \varepsilon_{kk}) \\
\quad \text{for} & & j = 1, \ldots, k-1 \quad \text{do} \\
\quad\quad & & \hat{b}_j^{(k-1)} = (\hat{b}_j^{(k)} - u_{jk}\hat{x}_k(1 + \varepsilon_{jk}))(1 + \eta_{jk}) \\
\quad \text{end} & & \text{do} \\
\text{end} & & \text{do}
\end{array}$$

Theorem 2.2.3. *(Stability of Back Substitution) Let $\hat{\boldsymbol{x}}$ be the numerical solution obtained when solving $U\boldsymbol{x} = \boldsymbol{b}$ using the SAXPY variant of back substitution. Then $\hat{\boldsymbol{x}}$ satisfies $\hat{U}\hat{\boldsymbol{x}} = \boldsymbol{b}$, where*

$$|\hat{u}_{jk} - u_{jk}| \leq (n-k+1)|u_{jk}|eps + O(eps^2).$$

Proof: Define

$$\tilde{b}_j^{(k-1)} = \frac{\hat{b}_j^{(k-1)}}{(1+\eta_{jk})(1+\eta_{j,k+1})\dots(1+\eta_{jn})}$$

for $k > j$. Then we can divide the update formula in the inner loop by $(1+\eta_{jk})(1+\eta_{j,k+1})\dots(1+\eta_{jn})$ to get

$$\tilde{b}_j^{(k-1)} = \tilde{b}_j^{(k)} - \hat{x}_k \cdot \frac{u_{jk}(1+\varepsilon_{jk})}{(1+\eta_{j,k+1})\dots(1+\eta_{jn})}$$

Moreover, the formula for calculating $\hat{x}_k$ in the outer loop implies

$$\tilde{b}_j^{(j)} = \frac{\hat{b}_j^{(j)}}{(1+\eta_{j,j+1})\cdots(1+\eta_{jn})} = \frac{u_{jj}\hat{x}_j}{(1+\varepsilon_{jj})(1+\eta_{j,j+1})\cdots(1+\eta_{jn})}.$$

Thus, using a telescoping sum, we get

$$\begin{aligned} b_j &= \sum_{k=j+1}^{n} (\tilde{b}_j^{(k)} - \tilde{b}_j^{(k-1)}) + \tilde{b}_j^{(j)} \\ &= \frac{u_{jj}\hat{x}_j}{(1+\varepsilon_{jj})(1+\eta_{j,j+1})\cdots(1+\eta_{jn})} + \sum_{k=j+1}^{n} \frac{u_{jk}\hat{x}_k(1+\varepsilon_{jk})}{(1+\eta_{j,k+1})\cdots(1+\eta_{jn})} \end{aligned}$$

which shows that $\hat{U}\hat{x} = b$ with $|\hat{u}_{jk} - u_{jk}| \leq (n-k+1)|u_{jk}|eps + O(eps^2)$, as required.

Since the $|u_{jk}|$ is bounded by $\alpha = \rho \cdot \max(a_{jk})$, we see that back substitution is also backward stable as long as the growth factor ρ is not too large. A similar argument shows that forward substitution is also backward stable.

2.2.3 Pivoting and Scaling

To achieve backward stability, one must choose a pivoting strategy that ensures that the growth factor ρ remains small. Unfortunately, there are matrices for which elements grow exponentially with partial pivoting during the elimination process, for instance the matrix

$$A = \begin{pmatrix} 1 & 0 & \dots & 0 & 1 \\ -1 & 1 & \ddots & \vdots & \vdots \\ -1 & -1 & \ddots & 0 & 1 \\ \vdots & \vdots & \ddots & 1 & 1 \\ -1 & -1 & \dots & -1 & 1 \end{pmatrix}$$

To start the elimination process in this matrix with partial pivoting, we have to add the first row to all the other rows. This leads to the value 2 in the last column of the matrix $A^{(1)}$, but none of the middle columns have changed. So now adding the second row of $A^{(1)}$ to all the following ones leads to the value 4 in the last column, and continuing like this, the last entry of $A^{(n-1)}$ will equal 2^{n-1}.

Partial pivoting is however used today almost universally when solving linear equations, despite the existence of matrices that grow exponentially during the elimination process. In fact, such matrices are rare[2]: a simple Matlab experiment with random matrices shows that Gaussian elimination with partial pivoting is a very stable process. We make use of the function LU in Problem 3.11.

```
N=500;
n=[10 20 30 40 50 60 70 80 90 100];
for j=1:length(n)
m=0;
for i=1:N
A=rand(n(j));
[L,U,P,rho]=LU(A);
m=m+rho;
end;
g(j)=m/N
end;
plot(n,g,'--',n,0.25*n.^(0.71),'-');
legend('average growth factor','0.25*n^{0.71}','Location','NorthWest')
xlabel('matrix size'),ylabel('growth factor \rho')
```

In Figure 2.4, we show the results obtained for a typical run of this algorithm. In fact the elements grow sublinearly for random matrices, and thus, in this case, Theorem 2.2.2 shows that Gaussian elimination with partial pivoting is backward stable.

Partial pivoting may fail to choose the right pivot if the matrix is badly scaled. Consider the linear system $A\boldsymbol{x} = \boldsymbol{b}$

$$\begin{pmatrix} 10^{-12} & 1 & -1 \\ 3 & -4 & 5 \\ 40 & -60 & 0 \end{pmatrix} \boldsymbol{x} = \begin{pmatrix} 17 + 10^{-12} \\ -62 \\ -1160 \end{pmatrix}. \tag{2.44}$$

The exact solution is $\boldsymbol{x} = (1, 20, 3)^\top$. If we multiply the first equation with 10^{14} we get the system $B\boldsymbol{y} = \boldsymbol{c}$,

$$\begin{pmatrix} 100 & 10^{14} & -10^{14} \\ 3 & -4 & 5 \\ 40 & -60 & 0 \end{pmatrix} \boldsymbol{y} = \begin{pmatrix} 17 + 10^{14} + 100 \\ -62 \\ -1160 \end{pmatrix} \tag{2.45}$$

with of course the same solution.

[2]". . . intolerable pivot-growth is a phenomenon that happens only to numerical analysts who are looking for that phenomenon", W. Kahan. Numerical linear algebra. Canad. Math. Bull., 9:757-801, 1966.

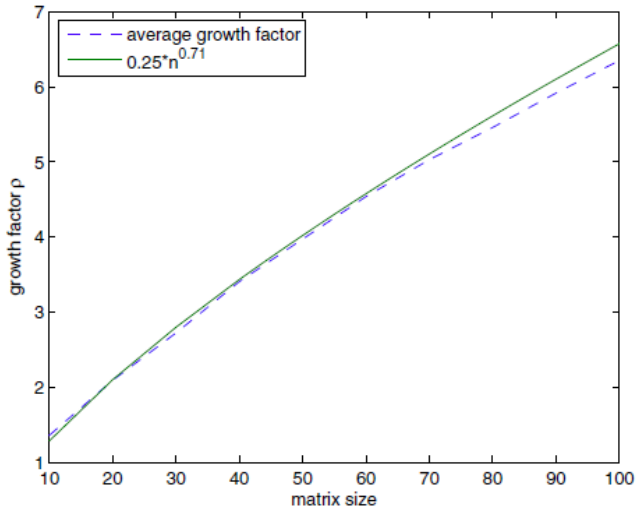


Figure 2.4: Very slow growth of entries encountered during Gaussian elimination with partial pivoting applied to random matrices

With partial pivoting, for the system (2.44) the first pivot in the first elimination will be the element $a_{31} = 40$. For the second system (2.45), however, because of bad scaling, the element $a_{11} = 100$ will be chosen. This has the same bad effect as if we had chosen $a_{11} = 10^{-12}$ as pivot in system (2.44). Indeed, we obtain with Matlab

$$x = (1, 20, 3)^\top, \qquad y = (1.0025, 20.003, 3.0031)^\top,$$

and the second solution is not very accurate, as expected.

Solving the system (2.45) by QR decomposition (Section 2.5) does not help either, as we can see below. But with *complete pivoting*, i.e., if we look for the largest element in modulus in the remaining matrix and interchange both rows (equations) and columns (reordering the unknowns), we get the correct solution (see Problem 3.13 for the function `EliminationCompletePivoting`). With the following Matlab statements,

```
fak=1e14;
A=[ 100/fak 1 -1
3 -4 5
40 -60 0];
xe=[1 20 3 ]'; b=A*xe; x1=A\b;
B=[fak*A(1,:); A(2:3,:)]; c=b; c(1)=c(1)*fak; x2=B\c;
[Q,R]=qr(B); d=Q'*c; x3=R\d;
[x4,Xh,r,U,L,B,P,Q]=EliminationCompletePivoting(B,c,1e-15);
[xe x1 x2 x3 x4]
[norm(xe-x1) norm(xe-x2) norm(xe-x3) norm(xe-x4)]
```

we obtain the results

```
exact          A\b           B\c           QR           compl.piv.
```

```
xe          x1          x2          x3          x4
1.0000      1.0000      1.0025      1.0720      1.0000
20.0000     20.0000     20.0031     20.0456     20.0000
3.0000      3.0000      3.0031      3.0456      3.0000
            ||xe-x1||   ||xe-x2||   ||xe-x3||   ||xe-x4||
            1.3323e-15  5.0560e-03  9.6651e-02  2.9790e-15
```

which show that only Gaussian elimination with complete pivoting leads to a satisfactory solution in this case.

It has been proved that the growth factors ρ_n^c for complete pivoting are bound by

$$\rho_n^c \leq n^{1/2}(2 \cdot 3^{1/2} \dots n^{1-1/n}) \sim n^{1/2} n^{1/4 \log n}$$

It was later conjectured that

$$g(n) := \sup_{A \in \mathbb{R}^{n \times n}} \rho_n^c(A) \leq n.$$

However, this was proven false. The limit $\lim_{n \to \infty} g(n)/n$ is an open problem. Though in practical problems the growth factors for complete pivoting turn out to be smaller than for partial pivoting, the latter is usually preferred because it is less expensive.

2.2.4 Sum of Rank-One Matrices

Gaussian elimination without pivoting, or the computation of the LU decomposition, may be interpreted as a sequence of *rank-one changes*. We consider for that purpose the matrix product as a sum of matrices of rank one:

$$A = LU = \sum_{k=1}^{n} \boldsymbol{l}_{:k} \boldsymbol{u}_{k:},$$

with columns of L and rows of U

$$L = [\boldsymbol{l}_{:1}, \boldsymbol{l}_{:2}, \dots \boldsymbol{l}_{:3}], \quad U = \begin{pmatrix} \boldsymbol{u}_{1:} \\ \boldsymbol{u}_{2:} \\ \vdots \\ \boldsymbol{u}_{n:} \end{pmatrix}.$$

We define the matrices

$$A_j = \sum_{k=j}^{n} \boldsymbol{l}_{:k} \boldsymbol{u}_{k:}, \qquad A_1 = A.$$

Because L and U are triangular, the first $j-1$ rows and columns of A_j are zero. Clearly

$$A_{j+1} = A_j - \boldsymbol{l}_{:j} \boldsymbol{u}_{j:}$$

holds. In order to eliminate row j and column j of A_j we can choose

$$\boldsymbol{u}_{j:} = (\overbrace{0, \dots, 0}^{j-1 \text{ zeros}}, a_{j,j}^{(j)}, a_{j,j+1}^{(j)}, \dots, a_{j,n}^{(j)})$$

and

$$\boldsymbol{l}_{:j} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ a^{(j)}_{j+1,j}/a^{(j)}_{j,j} \\ \vdots \\ a^{(j)}_{n,j}/a^{(j)}_{j,j} \end{pmatrix}.$$

The choice for $\boldsymbol{u}_{j:}$ and $\boldsymbol{l}_{:j}$ is not unique − we could for example divide by the pivot the elements of $\boldsymbol{u}_{j:}$ instead of $\boldsymbol{l}_{:j}$. The result of these considerations is the Matlab function

Algorithm 14 Gaussian Elimination by Rank−one Modifications

```
function [L,U]=LUbyRank1(A);
% LUBYRANK1 computes the LU factorization
% [L,U]=LUbyRank1(A); computes the LU-factorization of A with
% diagonal pivoting as n rank-one modifications. The implementation
% here is for didactic purposes only.

n=max(size(A));
L=[]; U=[];
for j=1:n
  u=[zeros(j-1,1);1; A(j+1:n,j)/A(j,j)];
  v=[zeros(j-1,1); A(j,j:n)'];
  A=A-u*v';
  L=[L u]; U=[U;v'];
end
```

Example 2.2.6.

```
>> A=[ 17 24  1  8 15
       23  5  7 14 16
        4  6 13 20 22
       10 12 19 21  3
       11 18 25  2  9]
>> [L,U]=LUbyRank1(A)
L =
1.0000       0      0      0      0
1.3529  1.0000      0      0      0
0.2353 -0.0128 1.0000      0      0
0.5882  0.0771 1.4003 1.0000      0
0.6471 -0.0899 1.9366 4.0578 1.0000
```

```
U =
17.0000  24.0000  1.0000  8.0000  15.0000
      0 -27.4706  5.6471  3.1765  -4.2941
      0        0 12.8373 18.1585  18.4154
      0        0       0 -9.3786 -31.2802
      0        0       0       0  90.1734
>> norm(L*U-A)
ans =
3.5527e-15
```

We saved the vectors in separate matrices L and U. This would of course not be necessary, since one could overwrite A with the decomposition. Furthermore, we did not introduce pivoting; this function is meant for didactic purposes and should not be used in real computations in the present form.

2.3 Condition of a System of Linear Equations

What can we say about the accuracy of the solution when we solve a system of linear equations numerically? A famous early investigation of this question was done by John von Neumann and H. H. Goldstine in 1947. They conclude their error analysis (over 70 pages long!) with quite pessimistic remarks on the size of linear systems that can be solved on a computer in finite precision arithmetic, see Table 2.1. The number of necessary multiplications they indicate are roughly n^3, which are the operations needed for inversion.

machine precision	10^{-8}	10^{-10}	10^{-12}
n <	15	50	150
# multiplications	3'500	120'000	3'500'000

Table 2.1: Pessimistic Error analysis by John von Neumann and H. H. Goldstine

It is the merit of Jim Wilkinson who discovered first by experiment that the bounds were too pessimistic and who developed the *backward error analysis* which explains much better what happens with calculations on the computer.

Consider the system of linear equations

$$A\boldsymbol{x} = \boldsymbol{b}, \quad \text{with } A \in \mathbb{R}^{n\times n} \text{ nonsingular.}$$

A perturbed system is $\hat{A}\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}}$, and we assume that $\hat{a}_{ij} = a_{ij}(1+\varepsilon_{ij})$ and $\hat{b}_i = b_i(1+\varepsilon_i)$, with $|\varepsilon_{ij}| \le \tilde{\varepsilon}_A$ and $|\varepsilon_i| \le \tilde{\varepsilon}_b$. This perturbation could come from roundoff errors in finite precision arithmetic, or from measurements, or any other source; we are only interested in how much the solution $\hat{\boldsymbol{x}}$ of the perturbed system differs from the solution $\boldsymbol{x}$ of the original system. The element-wise perturbations imply

$$\|\hat{A} - A\| \le \varepsilon_A \|A\|, \quad \|\hat{\boldsymbol{b}} - \boldsymbol{b}\| \le \varepsilon_{\boldsymbol{b}} \|\boldsymbol{b}\| \tag{2.46}$$

and if the norm is the 1−norm or infinity norm, we have $\varepsilon_A = \tilde{\varepsilon}_A$ and $\varepsilon_{\boldsymbol{b}} = \tilde{\varepsilon}_{\boldsymbol{b}}$, otherwise they differ just by a constant.

Theorem 2.3.1. *(Conditioning of the Solution of Linear Systems) Consider two linear systems of equations $A\boldsymbol{x} = b$ and $\hat{A}\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}}$ satisfying (2.46), and assume that A is invertible. If $\varepsilon_A \cdot \kappa(A) < 1$, then we have*

$$\frac{\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|}{\|\boldsymbol{x}\|} \leq \frac{\kappa(A)}{1 - \varepsilon_A \kappa(A)}(\varepsilon_A + \varepsilon_b), \tag{2.47}$$

where $\kappa(A) := \|A\|\|A^{-1}\|$ is the condition number of the matrix A.

Proof: From

$$\hat{\boldsymbol{b}} - \boldsymbol{b} = \hat{A}\hat{\boldsymbol{x}} - A\boldsymbol{x} = (\hat{A} - A)\hat{\boldsymbol{x}} + A(\hat{\boldsymbol{x}} - \boldsymbol{x}),$$

we obtain

$$\hat{\boldsymbol{x}} - \boldsymbol{x} = A^{-1}(-(\hat{A} - A)\hat{\boldsymbol{x}} + \hat{\boldsymbol{b}} - \boldsymbol{b})$$

and therefore, using Assumption (2.46),

$$\|\hat{\boldsymbol{x}} - \boldsymbol{x}\| \leq \|A^{-1}\|(\varepsilon_A \|A\|\|\hat{\boldsymbol{x}}\| + \varepsilon_b \|\boldsymbol{b}\|)$$

Now we can estimate $\|\hat{\boldsymbol{x}}\| = \|\hat{\boldsymbol{x}} - \boldsymbol{x} + \boldsymbol{x}\| \leq \|\hat{\boldsymbol{x}} - \boldsymbol{x}\| + \|\boldsymbol{x}\|$ and $\|\boldsymbol{b}\| = \|A\boldsymbol{x}\| \leq \|A\|\|\boldsymbol{x}\|$, which leads to

$$\|\hat{\boldsymbol{x}} - \boldsymbol{x}\| \leq \|A^{-1}\|\|A\|(\varepsilon_A(\|\hat{\boldsymbol{x}} - \boldsymbol{x}\| + \|\boldsymbol{x}\|) + \varepsilon_b \|\boldsymbol{x}\|)$$

and thus

$$\|\hat{\boldsymbol{x}} - \boldsymbol{x}\|(1 - \varepsilon_A \kappa(A)) \leq \kappa(A)\|\boldsymbol{x}\|(\varepsilon_A + \varepsilon_b)$$

which concludes the proof.

This theorem shows that the condition of the problem of solving a linear system of equations is tightly connected to the condition number of the matrix A, regardless of whether the perturbation appears in the matrix or in the right hand side of the linear system.

If $\kappa(A)$ is large then the solutions $\hat{\boldsymbol{x}}$ and $\boldsymbol{x}$ will differ significantly. According to Wilkinson's Principle (see Section 1.7), the result of a numerical computation is the exact result with slightly perturbed initial data. Thus $\hat{\boldsymbol{x}}$ will be the result of a linear system with perturbed data of order *eps*. As a rule of thumb, we have to expect a relative error of $eps \cdot \kappa(A)$ in the solution.

Computing the condition number is in general more expensive than solving the linear system. For example, if we use the spectral norm, the condition number can be computed by

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} = \texttt{cond(A)} \text{ in Mathlab}$$

Example 2.3.1. *We consider the matrix*

```
>> A=[21.6257 51.2930  1.5724 93.4650
       5.2284 83.4314 37.6507 84.7163
      68.3400  3.6422  6.4801 52.5777
      67.7589  4.5447 42.3687  9.2995];
```

choose the exact solution and compute the right hand side

```
>> x=[1:4]'; b=A*x;
```

Now we solve the system

```
>> xa=A\b;
```

and compare the numerical solution to the exact one.

```
>> format long e
>> [xa x]
>> cond(A)
>> eps*cond(A)
ans =
0.999999999974085 1.000000000000000
1.999999999951056 2.000000000000000
3.000000000039628 3.000000000000000
4.000000000032189 4.000000000000000
ans =
6.014285987206616e+05
ans =
1.335439755935445e-10
```

We see that the numerically computed solution has about 5 incorrect decimal digits, which corresponds very well to the predicted error due to the large condition number of 6.014×10^5 *of the matrix A. The factor eps* $\kappa(A) = 1.33 \times 10^{-10}$ *indicates well the error after the 10th decimal digit.*

Examples of matrices with a very large condition number are the Hilbert matrix H with $h_{ij} = 1/(i+j-1), i, j = 1, \ldots, n$ (`H = hilb(n)` in Matlab) and Vandermonde matrices, which appear for example in interpolation and whose columns are powers of a vector $\boldsymbol{v}$, that is $a_{ij} = v_i^{n-j}$ (`A=vander(v)` in Matlab). Table 2.2 shows the condition numbers for $n = 3, \ldots, 8$, where we have used `v = [1:n]` for the Vandermonde matrices.

n	cond(hilb(n))	cond(vander([1:n]))
3	5.2406e+02	7.0923e+01
4	1.5514e+04	1.1710e+03
5	4.7661e+05	2.6170e+04
6	1.4951e+07	7.3120e+05
7	4.7537e+08	2.4459e+07
8	1.5258e+10	9.5211e+08

Table 2.2: Matrices with large condition numbers

Matrices with a small condition number are for example orthogonal matrices, U such that $UU^\top = I$, which gives for the spectral condition number $\kappa_2(U) = \|U\|_2\|U^{-1}\|_2 = \|U\|_2\|U^\top\|_2 = 1$, since orthogonal transformations preserve Euclidean length.

The condition number $\kappa(A)$ satisfies several properties:

1. $\kappa(A) \geq 1$ provided the matrix norm satisfies the submultiplicative property, since $1 = \|I\| = \|A^{-1}A\| \leq \|A\|\|A^{-1}\|$.
2. $\kappa(\alpha A) = \alpha\kappa(A)$.
3. $\kappa(A) = \frac{\max_{\|\boldsymbol{y}\|=1}\|A\boldsymbol{y}\|}{\min_{\|\boldsymbol{z}\|=1}\|A\boldsymbol{z}\|}$ since

$$\|A^{-1}\| = \max_{\boldsymbol{x}\neq 0}\frac{\|A^{-1}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} = \max_{\boldsymbol{z}\neq 0}\frac{\|\boldsymbol{z}\|}{\|A\boldsymbol{z}\|} = \min_{\boldsymbol{z}\neq 0}\Big(\frac{\|A\boldsymbol{z}\|}{\|\boldsymbol{z}\|}\Big)^{-1}$$

Another useful property of the condition number is that it measures how far A is from a singular matrix, in a relative sense.

Theorem 2.3.2. *Let $E \in \mathbb{R}^{n\times n}$. If $\|E\| \leq 1$, then $I + E$ is non-singular, and we have*

$$\frac{1}{1+\|E\|} \leq \|(I+E)^{-1}\| \leq \frac{1}{1-\|E\|}. \tag{2.48}$$

Proof: To show that $I+E$ is non-singular, let us argue by contradiction by assuming that $I + E$ is singular. Then there exists a non-zero vector $\boldsymbol{x}$ such that $(I + E)\boldsymbol{x} = 0$. Then we have

$$\boldsymbol{x} = -E\boldsymbol{x} \Longrightarrow \|\boldsymbol{x}\| = \|E\boldsymbol{x}\| \leq \|\|E\|\boldsymbol{x}\|.$$

Since $\boldsymbol{x} \neq 0$, we can divide both sides by $\|\boldsymbol{x}\|$ and conclude that $\|E\| \geq 1$, which contradicts the hypothesis that $\|E\| < 1$. Hence $I + E$ is non-singular. To show the first inequality in (2.48), we let $\boldsymbol{y} = (I + E)^{-1}\boldsymbol{x}$. Then

$$\boldsymbol{x} = (I+E)\boldsymbol{y} \Longrightarrow \|\boldsymbol{x}\| \leq (1+\|E\|)\|\boldsymbol{y}\|,$$

which implies

$$\frac{\|(I+E)^{-1}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} = \frac{\|\boldsymbol{y}\|}{\|\boldsymbol{x}\|} \geq \frac{1}{1+\|E\|}.$$

Since this is true for all $\boldsymbol{x} \neq 0$, we can take the maximum of the left hand side over all $\boldsymbol{x} \neq 0$ and obtain

$$\|(I+E)^{-1}\| \geq \frac{1}{1+\|E\|},$$

as required. For the other inequality, note that

$$\|y\| \leq \|x\| + \|y - x\| \leq \|x\| + \| - Ey\| \leq \|x\| + \|E\|\|y\|,$$

which implies

$$(1 - \|E\|)\|y\| \leq \|x\|.$$

Since $1 - \|E\| > 0$, we can divide both sides by $(1 - \|E\|)\|x\|$ without changing the inequality sign. This yields

$$\frac{\|(I+E)^{-1}\boldsymbol{x}\|}{\|\boldsymbol{x}\|} = \frac{\|\boldsymbol{y}\|}{\|\boldsymbol{x}\|} \leq \frac{1}{1-\|E\|}.$$

Maximizing the left hand side over all $\boldsymbol{x} \neq 0$ yields $\|(I + E)^{-1}\| \leq \frac{1}{1-\|E\|}$ as required.

Corollary 2.3.1. *Let $A \in \mathbb{R}^{n\times n}$ be a non-singular matrix, $E \in \mathbb{R}^{n\times n}$ and $r = \|E\|/\|A\|$. If $r\kappa(A) < 1$, then $A + E$ is non-singular and*

$$\frac{\|(A+E)^{-1}\|}{\|A^{-1}\|} \leq \frac{1}{1 - r\kappa(A)}.$$

Proof: By (2.48), we have

$$\begin{aligned} \|(A+E)^{-1}\| &= \|(I + A^{-1}E)^{-1}A^{-1}\| \\ &\leq \|(I + A^{-1}E)^{-1}\|\|A^{-1}\| \leq \frac{\|A^{-1}\|}{1 - \|A^{-1}E\|} \leq \frac{\|A^{-1}\|}{1 - \|A^{-1}\|\|E\|}. \end{aligned}$$

Substituting $\|E\| = r\|A\|$ and dividing by $\|A^{-1}\|$ gives the required result.

Note that the hypothesis implies that $A + E$ will be nonsingular provided that the perturbation E is small enough, in the relative sense, with respect to the condition number $\kappa(A)$. In other words, if $\kappa(A)$ is small, i.e., if A is well conditioned, then a fairly large perturbation is needed to reach singularity. On the other hand, there always exists a perturbation E with $\|E\|_2 = \|A\|_2/\kappa_2(A)$ such that $A + E$ is singular.

2.4 Cholesky Decomposition

2.4.1 Symmetric Positive Definite Matrices

Definition 2.4.1. *(Symmetric Positive Definite Matrices) A matrix is symmetric, $A = A^\top$ and positive definite if*

$$\boldsymbol{x}^\top A\boldsymbol{x} > 0 \quad \textit{for all } \boldsymbol{x} \neq \boldsymbol{0}. \tag{2.49}$$

Positive definite matrices occur frequently in applications, for example for discretizations of coercive partial differential equations, or in optimization, where the Hessian must be positive definite at a strict minimum of a function of several variables. Positive definite matrices have rather special properties, as shown in the following lemma.

Lemma 2.4.2. *Let $A = A^\top \in \mathbb{R}^{n\times n}$ be positive definite. Then*

(a) If $L \in \mathbb{R}^{n\times n}$ with $m \leq n$ has rank m, then $LAL^\top$ is positive definite.

(b) Every principal submatrix of A is positive definite, i.e., for every nonempty subset $J \subset \{1, 2, \ldots, n\}$ with $|J| = m$, the $m \times m$ matrix $A(J, J)$ is positive definite.

(c) The diagonal entries of A are positive, i.e., $a_{ii} > 0$ for all i

(d) The largest entry of A in absolute value must be on the diagonal, and hence positive.

Proof:

(a) Let $\boldsymbol{z} \in \mathbb{R}^m$ be an arbitrary non-zero vector. Then the fact that L has rank m implies $\boldsymbol{y} := L\boldsymbol{z} \neq 0$, since the rows of L would otherwise be linearly dependent, forcing L to have rank less than m. The positive definiteness of A, in turn, gives

$$\boldsymbol{z}^\top LAL^\top \boldsymbol{z} = \boldsymbol{y}^\top A\boldsymbol{y} > 0$$

(b) Let $J = \{j_1, \ldots, j_m\}$ be an arbitrary subset of $\{1, \ldots, n\}$ and $B = A(J, J)$. Define the $n \times m$ matrix R with entries

$$r_{jk} = \begin{cases} 1, & j = j_k \\ 0, & \text{otherwise.} \end{cases}$$

It is then easy to see that $B = R^\top AR$. It follows by letting $L = R^\top$ in (a) that B is positive definite.

(c) This follows from (b) by choosing J to be the one-element set $\{i\}$.

(d) We argue by contradiction. Suppose, on the contrary, that the largest entry in absolute value does not occur on the diagonal. Then there must be a pair (i, j) with $i = j$ such that $|a_{ij}|$ is larger than any entry on the diagonal; in particular, we have $|a_{ij}| > a_{ii}$ and $|a_{ij}| > a_{jj}$. From (b), we know that the principal submatrix

$$B = \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix}$$

must be positive definite. But taking $x^\top = (1, -sign(a_i))^\top$ gives

$$x^\top Bx = a_{ii} + a_{jj} - 2|a_{ij}| < 0,$$

which contradicts the positive definiteness of B. Hence the largest entry in absolute value must occur on the diagonal (and hence be positive).

This gives another characterization of symmetric positive definite matrices.

Lemma 2.4.3. *Let $A = A^\top \in \mathbb{R}^{n\times n}$. Then A is positive definite if and only if all its eigenvalues are positive.*

Proof: Suppose A is symmetric positive definite and λ an eigenvalue of A with eigenvector $\boldsymbol{v} \neq 0$. Then

$$0 < \boldsymbol{v}^\top A\boldsymbol{v} = \boldsymbol{v}^\top(\lambda\boldsymbol{v}) = \lambda\|\boldsymbol{v}\|_2^2$$

We can now divide by $\|v\|_2^2 > 0$ to deduce that $\lambda > 0$.

Now suppose all eigenvalues of A are positive. Then by the spectral theorem, we can write $A = Q\Lambda Q^\top$, where Λ is a diagonal matrix containing the eigenvalues of A and $Q^\top Q = I$. But a diagonal matrix with positive diagonal entries is clearly positive definite, and Q is non-singular (i.e., it has full rank n). Hence, by Lemma 2.4.2(a), A is also positive definite.

Theorem 2.4.4. *Let $A = A^\top \in \mathbb{R}^{n\times n}$ be symmetric positive definite. Then in exact arithmetic, Gaussian elimination with no pivoting does not break down when applied to the system $A\boldsymbol{x} = \boldsymbol{b}$, and only positive pivots will be encountered. Furthermore, we have $U = DL^\top$ for a positive diagonal matrix D, i.e., we have the factorization $A = LDL^\top$.*

Proof: The proof is by induction. As shown before, the diagonal element $a_{11} > 0$ and therefore can be used as pivot. We now show that after the first elimination step, the remaining matrix is again positive definite. For the first step we use the matrix

$$L_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ -\frac{a_{21}}{a_{11}} & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\frac{a_{n1}}{a_{11}} & 0 & 0 & \dots & 1 \end{pmatrix}$$

and multiply from the left to obtain

$$L_1 A = \begin{pmatrix} a_{11} & \boldsymbol{b}^\top \\ \mathbf{0} & A_1 \end{pmatrix}, \quad \boldsymbol{b}^\top = (a_{12}, \dots, a_{1n})$$

Because of the symmetry of A, multiplying the last equation from the right with $L_1^\top$ we obtain

$$L_1 A L_1^\top = \begin{pmatrix} a_{11} & \mathbf{0}^\top \\ \mathbf{0} & A_1 \end{pmatrix} \tag{2.50}$$

and the submatrix A_1 is not changed since the first column of $L_1 A$ is zero below the diagonal. This also shows that A_1 is symmetric. Furthermore, by Lemma 2.4.2(a), we know that $L_1 A L_1^\top$ is in fact positive definite, since L_1 has rank n. Since A_1 is a principal submatrix of $L_1 A L_1^\top$, it is also positive definite. Thus, the second elimination step can be performed again, and the process can be continued until we obtain the decomposition

$$L_{n-1} \cdots L_1 A L_1^\top \cdots L_{n-1}^\top = \begin{pmatrix} a_{11}^{(0)} & & 0 \\ & \ddots & \\ 0 & & a_{nn}^{n-1} \end{pmatrix} =: D,$$

where the diagonal entries $a_{kk}^{(k-1)}$ are all positive. By letting $L = L_1^{-1} \cdots L_{n-1}^{-1}$ we obtain the decomposition $A = LDL^\top$, as required.

Since the diagonal entries of D are positive, it is possible to define its square root

$$D^{1/2} = \operatorname{diag}(\sqrt{a_{11}^{(0)}}, \dots, \sqrt{a_{nn}^{(n-1)}}).$$

Then by letting $R = D^{1/2} L^\top$, we obtain the so-called *Cholesky Decomposition*

$$A = R^\top R.$$

We can compute R directly without passing through the LU decomposition as follows. Multiplying from the left with the unit vector $\boldsymbol{e}_j^\top$, we obtain

$$\boldsymbol{a}_{j:} = \boldsymbol{e}_j^\top R^\top R = \boldsymbol{r}_{:j}^\top R = \sum_{k=1}^{j} r_{kj} \boldsymbol{r}_{k:}. \tag{2.51}$$

If we assume that we already know the rows $\boldsymbol{r}_{1:}, \dots, \boldsymbol{r}_{j-1:}$, we can solve Equation (2.51) for $\boldsymbol{r}_{j:}$ and obtain

$$r_{jj} \boldsymbol{r}_{j:} = a_{j:} - \sum_{k=1}^{j-1} r_{kj} \boldsymbol{r}_{k:} =: \boldsymbol{v}^\top.$$

The right hand side (the vector $\boldsymbol{v}$) is known. Thus multiplying from the right with $\boldsymbol{e}_j$ we get

$$r_{jj}\boldsymbol{r}_{j:}\boldsymbol{e}_j = r_{jj}^2 = v_j \Longrightarrow r_{jj} = \sqrt{v_j} \Longrightarrow \boldsymbol{r}_{j:} = \frac{\boldsymbol{v}^\top}{\sqrt{v_j}}.$$

Thus we have computed the next row of R. We only need a loop over all rows and obtain the function `Cholesky`:(see Algorithm 15)

Algorithm 15 Cholesky Decomposition

```
function R=Cholesky(A)
% CHOLESKY computes the Cholesky decomposition of a matrix
% R=Cholesky(A) computes the Cholesky decomposition A=R'R

n=length(A);
for j=1:n,
  v=A(j,j:n);
  if j>1,
    v=A(j,j:n)-R(1:j-1,j)'*R(1:j-1,j:n);
  end;
  if v(1)<=0
    error('Matrix is not positive definite')
  else
    h=1/sqrt(v(1));
  end
  R(j,j:n)=v*h;
end
```

To compute the Cholesky decomposition there is in Matlab the built-in function `chol`. Using the Cholesky decomposition, we can write the quadratic form defined by the matrix A as a sum of squares:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2 x_i x_j = \boldsymbol{x}^\top A\boldsymbol{x} = \boldsymbol{x}^\top R^\top R\boldsymbol{x} = \|R\boldsymbol{x}\|^2 = \boldsymbol{y}^\top\boldsymbol{y} = \sum_{i=1}^{n} y_i^2, \quad \text{with } \boldsymbol{y} = R\boldsymbol{x}.$$

2.4.2 Stability and Pivoting

We have seen in Section 2.2.2 that for general non-symmetric matrices, pivoting is essential not only to ensure that the elimination process does not break down, but also to ensure stability. However, for symmetric positive definite matrices, we have shown that break down cannot occur in exact arithmetic, even when *no pivoting* is used. Thus, one might suspect that pivoting is also not needed for stability. This is indeed true, provided that the matrix is not too ill conditioned.

Theorem 2.4.5. *Let $A = A^\top \in \mathbb{R}^{n\times n}$ be positive definite. If $c_n\kappa_2(A)eps < 1$ with $c_n = 3n^2 + O(eps)$, then Gaussian elimination (or Cholesky decomposition) does not break down. Moreover, if $\hat{L}$ and $\hat{D}$ are the numerically computed factors in the Cholesky*

decomposition and $\hat{A} = \hat{L}\hat{D}\hat{L}^\top$ *is the numerically computed decomposition, then we have the estimate*

$$|\hat{a}_{ij} - a_{ij}| \leq 3\alpha \min(i-1, j) eps + O(eps^2), \tag{2.52}$$

where $\alpha = \max_{i,j} |a_{i,j}|$

Remark. The above result implies that if Cholesky breaks down, it is either because the matrix A is not positive definite, or that it is very ill conditioned. In practical computations, one rarely checks the condition $c_n \kappa_2(A) eps < 1$ before the actual factorization, because the condition number $\kappa_2(A)$ is more expensive to calculate than the Cholesky factors itself.

If one suspects that A is so ill conditioned that the hypothesis may be violated, pivoting may help reduce the factorization errors due to rounding. For positive definite matrices, complete pivoting reduces to diagonal pivoting, since the largest entry always appears on the diagonal. We can then obtain the estimates in Theorem 2.2.2, provided each submatrix that appears after an elimination step remains positive definite.

Proof: Let$\hat{A}^{(k)}$ be the $(n-k) \times (n-k)$ submatrix $\hat{A}^{(k)} = [\hat{a}_{ij}^{(k)}]_{k+1 \leq i,j \leq n}$ for $0 \leq k \leq n$. Just as in the proof of Theorem 2.2.2, we have the relation

$$\hat{a}_{ij}^{(k)} = (\hat{a}_{ij}^{k-1} - \hat{l}_{ik}\hat{a}_{kj}^{(k-1)}(1+\varepsilon_{ijk}))(1+\eta_{ijk}) = \overbrace{\hat{a}_{ij}^{(k-1)} - \frac{\hat{a}_{ik}^{(k-1)}\hat{a}_{kj}}{\hat{a}_{kk}^{(k-1)}}}^{b_{ij}^{(k)}} + \mu_{ijk},$$

where $\hat{l}_{ik} = \frac{\hat{a}_{ik}^{(k-1)}}{\hat{a}_{kk}^{(k-1)}}(1+\delta_{ik})$ with $|\varepsilon_{ijk}|, |\eta_{ijk}|, |\delta_{ik}| < eps$. Here $B^{(k)} = [b_{ij}^{(k)}]_{k+1 \leq i,j \leq n}$ is the submatrix we would have obtained if we had performed one step of Gaussian elimination on $\hat{A}^{(k-1)}$ in exact arithmetic, so that μ_{ijk} contains all the round-off errors associated with this step. Using the same manipulation as in the proof of Theorem 2.2.2, we can write

$$|\mu_{ijk}| \leq |b_{ij}^{(k)}| eps + \left| \frac{\hat{a}_{ik}^{(k-1)}\hat{a}_{kj}^{(k-1)}}{\hat{a}_{kk}^{(k-1)}} \right| \cdot 2eps + O(eps^2). \tag{2.53}$$

We now use the fact that $B^{(k)}$ is positive definite to bound the first two terms above. By Lemma 2.4.2(d), we have

$$\max_{ij} |b_{ij}^{(k)}| = \max_i b_{ii}^{(k)} = \max_i \left[\hat{a}_{ii}^{(k-1)} - \sum_{j=1}^{k} (\hat{a}_{ij}^{(k-1)})^2 \right] \leq \max_i \hat{a}_{ii}^{(k-1)}.$$

Moreover, we have

$$\max_{ij} \left| \frac{\hat{a}_{ik}^{(k-1)}\hat{a}_{kj}^{(k-1)}}{\hat{a}_{kk}^{(k-1)}} \right| \overset{(*)}{=} \max_i \frac{(\hat{a}_{ik}^{(k-1)})^2}{\hat{a}_{kk}^{(k-1)}} = \max_i (\hat{a}_{ii}^{(k-1)} - b_{ii}^{(k)}) \overset{\dagger}{\leq} \max_i \hat{a}_{ii}^{(k-1)}.$$

where the equality $(*)$ is due to the symmetry of $\hat{a}_{ij}^{(k-1)}$ and the inequality $(\dagger)$ is true because $b_{ii}^{(k)} > 0$. Thus, by letting $\alpha_k = \max_{i,j} |\hat{a}_{ij}^{(k)}| = \max_i \hat{a}_{ii}^{(k)}$, (2.53) becomes

$$|\mu_{ijk}| \leq 3\alpha_{k-1} eps + O(eps^2). \tag{2.54}$$

This implies $\hat{A}^{(k)} = B^{(k)} + E^{(k)}$ with

$$\|E^{(k)}\|_2 \leq 3(n-k)eps\|\hat{A}^{(k-1)}\|_2 + O(eps^2).$$

Thus, if we can show that $\hat{A}^{(k)}$ is positive definite with $c_k\kappa_2(\hat{A}^{(k)})eps < 1$, $c_k = 3(n-k)^2 + O(eps)$, then the induction is complete and we can conclude that the Cholesky factorization does not break down. To do so, we consider the matrix

$$\tilde{A}^{(k-1)} = \hat{A}^{(k-1)} + \begin{pmatrix} 0 & 0 \\ 0 & E^{(k)} \end{pmatrix} = \hat{A}^{(k-1)} + \tilde{E}^{(k)}.$$

In other words, if we perform one step of Gaussian elimination in exact arithmetic on $\tilde{A}^{(k-1)}$, the result would be $\hat{A}^{(k)}$. We argue that all the eigenvalues of $\tilde{A}^{(k-1)}$ are positive, which implies $\tilde{A}^{(k-1)}$ is positive definite. Assume the contrary, i.e., that the smallest eigenvalue of $\tilde{A}^{(k-1)}$ is zero or negative. Then since eigenvalues are a continuous function of the perturbation, there exists $0 < t \leq 1$ such that $\hat{A}^{(k-1)} + t\tilde{E}^{(k)}$ has a zero eigenvalue, i.e., it is singular. But we have

$$\frac{\|t\tilde{E}^{(k)}\|_2\kappa_2(\hat{A}^{(k-1)})}{\|\hat{A}^{(k-1)}\|_2} \leq \frac{3(n-k)t\ eps + O(eps^2)}{3(n-k+1)^2eps + O(eps^2)} < 1,$$

so by Corollary 2.3.1, $\hat{A}^{(k-1)} + t\tilde{E}^{(k)}$ must be non-singular, a contradiction. Hence $\tilde{A}^{(k-1)}$ must be positive definite.

We now want to show that $\kappa_2(\hat{A}^{(k)}) \leq \frac{1}{3(n-k)^2eps+O(eps^2)}$. We have

$$\begin{aligned}
\|\hat{A}^{(k)}\|_2 &= \max_{\|\boldsymbol{z}\|_2=1} \boldsymbol{z}^\top \hat{A}^{(k)}\boldsymbol{z} \\
&= \boldsymbol{z}^\top(\tilde{A}^{(k-1)}(k:n,k:n) - (\hat{a}_{kk}^{(k-1)})^{-1}\boldsymbol{a}_{:k}\boldsymbol{a}_{:k}^\top)\boldsymbol{z} \\
&\leq \boldsymbol{z}^\top[\tilde{A}^{(k-1)}(k:n,k:n)]\boldsymbol{z} \\
&= (0,\boldsymbol{z}^\top)\tilde{A}^{(k-1)}\begin{pmatrix} 0 \\ \boldsymbol{z} \end{pmatrix} \leq \|\tilde{A}^{(k-1)}\|_2.
\end{aligned}$$

On the other hand, since $(\hat{A}^{(k)})^{-1}$ is a principal submatrix of $(\tilde{A}^{(k-1)})^{-1}$, we automatically have $\|(\hat{A}^{(k)})^{-1}\|_2 \leq \|(\tilde{A}^{(k-1)})^{-1}\|_2$. Thus, we have $\kappa_2(\hat{A}^{(k)}) \leq \kappa_2(\tilde{A}^{(k-1)})$. It now suffices to bound $\|\tilde{A}^{(k-1)}\|_2$ and $\|(\tilde{A}^{(k-1)})^{-1}\|_2$ individually:

$$\begin{aligned}
\|\tilde{A}^{(k-1)}\|_2 &\leq \|\hat{A}^{(k-1)}\|_2 + \|E^{(k)}\|_2 \leq \|\hat{A}^{(k-1)}\|_2(1+3(n-k)eps), \\
\|(\tilde{A}^{(k-1)})^{-1}\|_2 &\leq \frac{\|\hat{A}^{(k-1)}\|_2}{1-3(n-k)\kappa_2(\hat{A}^{(k-1)})eps + O(eps^2)} \\
&\leq \frac{\|(\hat{A}^{(k-1)})^{-1}\|_2}{1-\frac{3(n-k)}{c_k-1}} = \frac{c_{k-1}\|(\hat{A}^{(k-1)})^{-1}\|_2}{c_{k-1}-3(n-k)}.
\end{aligned}$$

Multiplying the two quantities above gives

$$\begin{aligned}\kappa_2(\hat{A}^{(k)}) &\leq \frac{\kappa_2(\hat{A}^{(k-1)})c_{k-1}(1+3(n-k)eps)}{c_{k-1}-3(n-k)} \\ &\leq \frac{1}{eps}\frac{1+3(n-k)eps}{c_{k-1}-3(n-k)} \\ &\leq \frac{1}{eps}\frac{1+3(n-k)eps}{3((n-k)^2+(n-k)+1)} \leq \frac{1}{3(n-k)^2 eps},\end{aligned}$$

so the induction step is complete. Finally, (2.54) implies that

$$\alpha_k \leq \alpha_{k-1}(1+3eps).$$

Thus, using the same analysis as in Theorem 2.2.2, we get the estimate (2.52), where

$$\alpha = \max_{0\leq k\leq n-1} \alpha_k \leq \alpha_0(1+3eps)^n = \max_{i,j}|\alpha_{ij}| + O(eps).$$

Thus we complete the proof. □

Note that the above theorem shows that the growth factor for the elimination process is essentially $\rho = 1$, i.e., no entries that appear during the elimination process can be larger than the largest entry in the original matrix A. Thus, Cholesky factorization is backward stable, even without pivoting.

2.5 Elimination with Givens Rotations

In this section, we discuss another method for elimination that requires more operations, but does not require pivoting and can also be used for solving least squares problems.

In the $i-$th step, we eliminate x_i in equations $i+1$ to n as follows: let

$$\begin{aligned}(i) &: a_{ii}x_i + \cdots + a_{in}x_n = b_i \\ &\vdots \qquad\qquad \vdots \\ (k) &: a_{ki}x_i + \cdots + a_{kn}x_n = b_k \\ &\vdots \qquad\qquad \vdots \\ (n) &: a_{ni}x_i + \cdots + a_{nn}x_n = b_n\end{aligned} \tag{2.55}$$

be the reduced system. We wish to eliminate x_i from equation (k). In the case where $a_{ki} = 0$, nothing needs to be done because the unknown x_i is already eliminated. Otherwise, we multiply equation (i) with $-\sin\alpha$ and equation (k) with $\cos\alpha$ and replace equation (k) by the linear combination

$$(k)_{new} := -\sin\alpha \cdot (i) + \cos\alpha \cdot (k). \tag{2.56}$$

Therefore, we choose α so that

$$a_{ki}^{new} := -\sin\alpha \cdot a_{ii} + \cos\alpha \cdot a_{ki} = 0. \tag{2.57}$$

Since $a_{ki} \neq 0$, we compute from (2.57)

$$\cot \alpha = \frac{a_{ii}}{a_{ki}}, \tag{2.58}$$

and obtain, using well known trigonometric identities, the quantities $co = \cos\alpha$ and $si = \sin\alpha$ by

$$\cot = a_{ii}/a_{ik}; \quad si = 1/\sqrt{1+\cot^2}; \quad co = si \times \cot; \tag{2.59}$$

In this elimination step, in addition to replacing equation (k), we also modify equation (i) with

$$(i)_{new} := \cos\alpha \cdot (i) + \sin\alpha \cdot (k). \tag{2.60}$$

This is done for stability purposes. Observe that pivoting now becomes unnecessary: for the situation when $a_{ii} = 0$ and $a_{ki} \neq 0$, we get $\cot\alpha = 0$, and therefore $\sin\alpha = 1$ and $\cos\alpha = 0$. The two assignments (2.56) and (2.60) simply exchange equations (k) and (i), as we would do with pivoting! Admittedly, however, the computational effort is doubled. This leads to the following program:

Algorithm 16 Solving Linear Systems with Givens Rotations

```
function x=EliminationGivens(A,b);
% ELIMINATIONGIVENS solves a linear system using Givens-rotations
% x=EliminationGivens(A,b) solves Ax=b using Givens-rotations. Uses
% the function BackSubstitutionSAXPY.

n=length(A);
for i= 1:n
  for k=i+1:n
    if A(k,i)~=0
    cot=A(i,i)/A(k,i); % rotation angle
    si=1/sqrt(1+cot^2); co=si*cot;
    A(i,i)=A(i,i)*co+A(k,i)*si; % rotate rows
    h=A(i,i+1:n)*co+A(k,i+1:n)*si;
    A(k,i+1:n)=-A(i,i+1:n)*si+A(k,i+1:n)*co;
    A(i,i+1:n)=h;
    h=b(i)*co+b(k)*si; % rotate right hand side
    b(k)=-b(i)*si+b(k)*co; b(i)=h;
    end
  end;
  if A(i,i)==0
    error('Matrix is singular');
  end;
end
x=BackSubstitutionSAXPY(A,b);
```

2.6 Banded matrices

A matrix is called *banded* if it contains nonzero elements only in a few diagonals next to the main diagonal. For example, consider the matrix

$$A = \begin{pmatrix} 2 & 1 & -1 & 0 & 0 & 0 & 0 \\ -4 & 2 & 3 & 0 & 0 & 0 & 0 \\ 0 & -12 & 3 & 1 & 2 & 0 & 0 \\ 0 & 0 & -24 & 4 & -7 & 0 & 0 \\ 0 & 0 & 0 & -40 & 5 & 1 & 4 \\ 0 & 0 & 0 & 0 & -60 & 6 & -23 \\ 0 & 0 & 0 & 0 & 0 & -84 & 7 \end{pmatrix} \tag{2.61}$$

A is a banded matrix with upper bandwidth $p = 2$ and lower bandwidth $q = 1$, thus in total there are $p + q + 1$ nonzero diagonals. It is easy to see that, when computing the LU decomposition without pivoting, the factors L and U occupy the same band, i.e., L is a lower banded matrix with q diagonals and U an upper banded matrix with p nonzero diagonals. For the matrix (2.61), the first Gaussian elimination step in which we eliminate x_1 will only modify the elements a_{21}, a_{22} and a_{23}. Performing the complete LU decomposition without pivoting we obtain the factors $A = LU$ with

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -4 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -6 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 & 1 \end{pmatrix} \quad U = \begin{pmatrix} 2 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 4 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 8 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 1 & 4 \\ 0 & 0 & 0 & 0 & 0 & 12 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 14 \end{pmatrix}.$$

Thus the band of matrix A might be overwritten by the LU decomposition if we, as usual, do not store the diagonal elements of L.

2.6.1 Storing Banded Matrices

Of course, we should not store a banded matrix as a full $n \times n$ matrix, but rather avoid storing zero elements. With lower bandwidth q and upper bandwidth p, a matrix A could be stored as a dense matrix B of size $n \times (p+q+1)$. The nonzero diagonals of A become the columns of B (cf. Figure 2.5). The price we pay for saving memory in this fashion is that accessing the elements becomes more complicated, since the indexing requires more integer operations. Thus, there is always a trade off between saving memory versus saving operations. In addition, the speed of modern microprocessors is affected by complicated memory accesses, because of cache effects.

The mapping of the banded matrix A to the matrix B is computed by the following program:

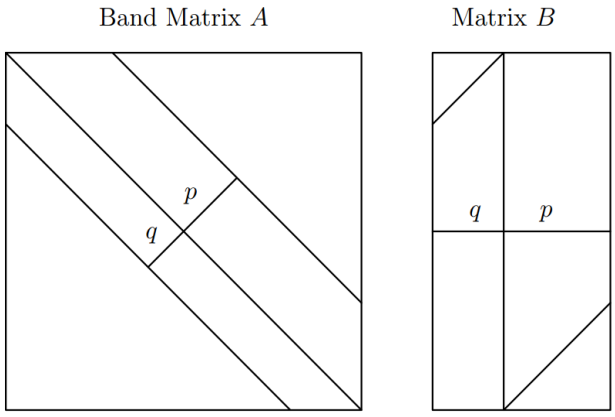


Figure 2.5: Storing of banded matrices

Algorithm 17 Transformation for Banded Matrices

```
function B=StoreBandMatrix(A,q,p)
% STOREBANDMATRIX stores the band of a matrix in a rectangular matrix
% B=StoreBandMatrix(A) stores the band of A (with lower bandwidth p
% and upper bandwidth q) in the rectangular matrix B of dimensions
% n*p+q+1.

n=length(A);
B=zeros(n,p+q+1); % reserve space
for i=1:n
  for j=max(1,i-q):min(n,i+p)
    B(i,j-i+q+1)=A(i,j);
  end
end
```

Our example matrix (2.61) is transformed with `B=StoreBandMatrix(A,1,2)` to

```
B =
     0    2     1    -1
    -4    2     3     0
   -12    3     1     2
   -24    4    -7     0
   -40    5     1     4
   -60    6   -23     0
   -84    7     0     0
```

2.6.2 Tridiagonal Systems

For *tridiagonal systems*, we will not use the above transformation, since it is easier to denote the three diagonals with the vectors $\boldsymbol{c}, \boldsymbol{d}$ and $\boldsymbol{e}$.

$$A = \begin{pmatrix} d_1 & e_1 & & & & \\ c_1 & d_2 & e_2 & & & \\ & c_2 & d_3 & e_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & c_{n-2} & d_{n-1} & e_{n-1} \\ & & & & c_{n-1} & d_n \end{pmatrix} \tag{2.62}$$

Linear systems with a tridiagonal matrix can be solved in $O(n)$ operations. The LU decomposition with *no pivoting* generates two bidiagonal matrices

$$L = \begin{pmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & l_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & l_{n-1} & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_1 & e_1 & & & \\ & u_2 & e_2 & & \\ & & u_3 & \ddots & \\ & & & \ddots & e_{n-1} \\ & & & & u_n \end{pmatrix}.$$

In order to compute L and U, we consider the elements c_k and d_{k+1} of the matrix A. Multiplying $L \cdot U$ and comparing elements we obtain the relations

$$\begin{aligned} l_k u_k = c_k \quad &\text{therefore} \quad l_k = c_k/u_k, \\ l_k e_k + u_{k+1} = d_{k+1} \quad &\text{therefore} \quad u_{k+1} = d_{k+1} - l_k e_k. \end{aligned}$$

The LU decomposition is thus computed by

```
u(1)=d(1);
for k=1:n-1
  l(k)=c(k)/u(k);
  u(k+1)=d(k+1)-l(k)*e(k);
end
```

Forward and back substitutions with L and U are straightforward. Note that we can overwrite the vectors $\boldsymbol{c}$ and $\boldsymbol{d}$ by $\boldsymbol{l}$ and $\boldsymbol{u}$. Furthermore, the right hand side may also be overwritten with the solution. In the French literature, this algorithm is known as Thomas' Algorithm. We obtain the function

Algorithm 18 Gaussian Elimination for Tridiagonal Systems: Thomas Algorithm

```
function [x,a,c]=Thomas(c,a,b,x);
% THOMAS Solves a tridiagonal linear system
% [x,a,c]=Thomas(c,a,b,x) solves the linear system with a
% tridiagonal matrix A=diag(c,-1)+diag(a)+diag(b,1). The right hand
% side x is overwritten with the solution. The LU-decomposition is
% computed with no pivoting resulting in L=eye+diag(c,-1),
% U=diag(a)+diag(b,1).

n=length(a);
for k=1:n-1                              % LU-decomposition with no pivoting
  c(k)=c(k)/a(k);
  a(k+1)=a(k+1)-c(k)*b(k);
end
for k=2:n                                % forward substitution
  x(k)=x(k)-c(k-1)*x(k-1);
end
x(n)=x(n)/a(n);                          % backward substitution
for k=n-1:-1:1
  x(k)=(x(k)-b(k)*x(k+1))/a(k);
end
```

2.6.3 Solving Banded Systems with Pivoting

With partial pivoting, rows are interchanged, destroying the band structure and introducing fill-in. If q is the lower bandwidth and p the upper bandwidth, then after interchanging rows, U will become an upper banded matrix with bandwidth $p+q$. In this section, we will develop a solver for banded systems which uses the matrix B where the nonzero diagonals of A are stored as columns.

We modify the core of the function `Elimination` to take advantage of the band structure and avoid operations with zero elements. It will not be possible to treat the right hand side as $(n+1)$st column of A. Furthermore, we will "de-vectorize" – we cannot always use vector operations after the transformation $a_{i,j} = b_{i,j-i+q+1}$, and therefore rewrite the statements using loops.

The search for a pivot is limited to the lower bandwidth

```
maximum=0;
for k=i:min(i+q,n)
  if abs(B(k,i-k+q+1))>maximum,
    kmax=k; maximum=abs(B(k,i-k+q+1));
  end
end
```

The interchange of rows for pivoting was

```
h=A(kmax,:); A(kmax,:)=A(i,:); A(i,:)=h;
```

and now it becomes

```
for k=i:min(n,i+q+p)
  h=B(kmax,k-kmax+q+1);
  B(kmax,k-kmax+q+1)=B(i,k-i+q+1);
  B(i,k-i+q+1)=h;
end
h=b(kmax); b(kmax)=b(i); b(i)=h;
```

This could still be vectorized to

```
h=B(kmax,i-kmax+q+1:min(n,i+2*q+p-kmax+1));
B(kmax,i-kmax+q+1:min(n,i+2*q+p-kmax+1))=B(i,q+1:min(n,2*q+p+1));
B(i,q+1:min(n,2*q+p+1))=h;
h=b(kmax); b(kmax)=b(i); b(i)=h;
```

The elimination step

```
A(i+1:n,i)=A(i+1:n,i)/A(i,i);
A(i+1:n,i+1:n+1)=A(i+1:n,i+1:n+1)-A(i+1:n,i)*A(i,i+1:n+1);
```

is rewritten with for-loops

```
for k=i+1:n
  A(k,i)=A(k,i)/A(i,i);
```

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

Now observing the upper and lower bandwidth and processing the right hand side separately, we get

```
for k=i+1:min(n,i+q)
  B(k,i-k+q+1)=B(k,i-k+q+1)/B(i,q+1);
end
for k=i+1:min(n,i+q)
  b(k)=b(k)-B(k,i-k+q+1)*b(i);
  for j=i+1:min(n,i+p+q)
    B(k,j-k+q+1)=B(k,j-k+q+1)-B(k,i-k+q+1)*B(i,j-i+q+1);
  end
end
```

Putting all together, we obtain the function `EliminationBandMatrix`:

Algorithm 19 Gaussian Elimination with Partial Pivoting for Banded Matrices

```
function x=EliminationBandMatrix(p,q,B,b);
% ELIMINATIONBANDMATRIX solves a linear system with a banded matrix
% x=EliminationBandMatrix(p,q,B,b); solves the banded linear system
% Ax=b with partial pivoting. The columns of B contain the nonzero
% diagonals of the matrix A. The first q columns of B contain the
% lower diagonals of A (augmented by leading zeros) the remaining
% columns of B contain the diagonal of A and the p upper diagonals,
% augmented by trailing zeros. The vector b contains the right-hand
% side.

n=length(B);
B=[B,zeros(n,q)];                    % augment B with q columns
normb=norm(B,1);
for i=1:n
  maximum=0; % search pivot
  for k=i:min(i+q,n)
    if abs(B(k,i-k+q+1))>maximum,
      kmax=k; maximum=abs(B(k,i-k+q+1));
    end
  end
  if maximum<1e-14*normb;            % only small pivots
    error('matrix is singular')
  end
  if i~=kmax % interchange rows
    h=B(kmax,i-kmax+q+1:min(n,i+2*q+p-kmax+1));
    B(kmax,i-kmax+q+1:min(n,i+2*q+p-kmax+1))=B(i,q+1:min(n,2*q+p+1));
    B(i,q+1:min(n,2*q+p+1))=h;
    h=b(kmax); b(kmax)=b(i); b(i)=h;
  end
  for k=i+1:min(n,i+q)               % elimination step
    B(k,i-k+q+1)=B(k,i-k+q+1)/B(i,q+1);
  end
  for k=i+1:min(n,i+q)
    b(k)=b(k)-B(k,i-k+q+1)*b(i);
    for j=i+1:min(n,i+p+q)
      B(k,j-k+q+1)=B(k,j-k+q+1)-B(k,i-k+q+1)*B(i,j-i+q+1);
    end
  end
end
for i=n:-1:1                         % back substitution
  s=b(i);
  for j=i+1:min(n,i+q+p)
    s=s-B(i,j-i+q+1)*x(j);
  end
  x(i)=s/B(i,q+1);
end
x=x(:);
```

2.6.4 Using Givens Rotations

We have seen in Section 2.5 that Givens rotations can be used as an alternative to LU decomposition to solve dense linear systems. This alternative is also available for banded systems: we show here how to proceed for tridiagonal systems with coefficient matrix A as shown in Equation (2.62). We use Givens rotation matrices $G^{(ik)}$, which differ in only four elements from the identity,

$$\begin{aligned} g_{ii} &= g_{kk} = c = \cos\alpha, \\ g_{ik} &= -g_{ki} = s = \sin\alpha. \end{aligned}$$

Multiplying the linear system from the left by $G^{(ik)}$ changes only two rows, $\boldsymbol{a}_{i:}$ and $\boldsymbol{a}_{k:}$:

$$\begin{aligned} \boldsymbol{\alpha}_{i:}^{new} &:= \quad \cos\alpha \cdot \boldsymbol{\alpha}_{i:}^{old} + \sin\alpha \cdot \boldsymbol{\alpha}_{k:}^{old}, \\ \boldsymbol{\alpha}_{k:}^{new} &:= -\sin\alpha \cdot \boldsymbol{\alpha}_{i:}^{old} + \cos\alpha \cdot \boldsymbol{\alpha}_{k:}^{old}. \end{aligned} \tag{2.63}$$

We can choose the angle α to zero elements in the matrix (see Section 2.5). We illustrate this for $n = 5$. In the first step we choose $G^{(12)}$ which combines the two first rows, and choose α such that $a_{21}^{new} = 0$:

$$G^{(12)} \begin{bmatrix} x & x & & & \\ x & x & x & & \\ & x & x & x & \\ & & x & x & x \\ & & & x & x \end{bmatrix} = \begin{bmatrix} x & x & X & & \\ 0 & x & x & & \\ & x & x & x & \\ & & x & x & x \\ & & & x & x \end{bmatrix}$$

A fill-in element $a_{13} = X$ is generated. In the second step we take $G^{(23)}$ which changes the second and third row such that

$$G^{(23)} \begin{bmatrix} x & x & X & & \\ 0 & x & x & & \\ & x & x & x & \\ & & x & x & x \\ & & & x & x \end{bmatrix} = \begin{bmatrix} x & x & X & & \\ 0 & x & x & X & \\ & 0 & x & x & \\ & & x & x & x \\ & & & x & x \end{bmatrix}$$

zeroing $a_{32} = 0$, and generating the fill-in $a_{24} = X$. The next rotation with

$$G^{(34)} \begin{array}{|ccccc|} \hline x & x & X & & \\ 0 & x & x & X & \\ \hline & 0 & x & x & \\ \hline & & x & x & x \\ \hline & & & x & x \\ \hline \end{array} = \begin{array}{|ccccc|} \hline x & x & X & & \\ 0 & x & x & X & \\ \hline & 0 & x & x & X \\ \hline & & 0 & x & x \\ \hline & & & x & x \\ \hline \end{array}$$

Finally, we obtain A transformed to an upper triangular matrix R with $G^{(45)}$: The

$$G^{(45)} \begin{array}{|ccccc|} \hline x & x & X & & \\ 0 & x & x & X & \\ & 0 & x & x & X \\ \hline & & 0 & x & x \\ \hline & & & x & x \\ \hline \end{array} = \begin{array}{|ccccc|} \hline x & x & X & & \\ 0 & x & x & X & \\ & 0 & x & x & X \\ \hline & & 0 & x & x \\ \hline & & & 0 & x \\ \hline \end{array}$$

solution is then obtained by back-substitution. For the transformation we need only the three diagonals of the matrix A. They will be overwritten with the elements of the upper banded matrix R.

$$A = \begin{array}{|ccccc|} \hline d_1 & e_1 & & & \\ c_1 & d_2 & e_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & e_{n-1} \\ & & & c_{n-1} & d_n \\ \hline \end{array} \longmapsto R = \begin{array}{|ccccc|} \hline d_1 & e_1 & c_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & c_{n-2} \\ & & & \ddots & e_{n-1} \\ & & & & d_n \\ \hline \end{array}$$

The following function(see Algorithm 20) `ThomasGivens` solves a tridiagonal system using Givens rotations. The right hand side is stored in $\boldsymbol{b}$ and overwritten with the solution.

Algorithm 20 Solving Tridiagonal Systems with Givens rotations

```
function [b,d,e,c]=ThomasGivens(c,d,e,b);
% THOMASGIVENS solves a tridiagonal system of linear equations
% [b,d,e,c]=ThomasGivens(c,d,e,b) solves a tridiagonal linear system
% using Givens rotations. The coefficient matrix is
% A=diag(c,-1)+diag(d)+diag(e,1), and the right hand side b is
% overwritten with the solution. The R factor is also returned,
% R=diag(d)+diag(e,1)+diag(c,2).

n=length(d);
e(n)=0;
for i=1: n-1                                    % elimination
  if c(i)~=0
    t=d(i)/c(i); si=1/sqrt(1+t*t); co=t*si;
    d(i)=d(i)*co+c(i)*si; h=e(i);
    e(i)=h*co+d(i+1)*si; d(i+1)=-h*si+d(i+1)*co;
    c(i)=e(i+1)*si; e(i+1)=e(i+1)*co;
    h=b(i); b(i)=h*co+b(i+1)*si;
    b(i+1)=-h*si+b(i+1)*co;
  end;
end;
b(n)=b(n)/d(n);                                 % backsubstitution
b(n-1)=(b(n-1)-e(n-1)*b(n))/d(n-1);
for i=n-2:-1:1,
  b(i)=(b(i)-e(i)*b(i+1)-c(i)*b(i+2))/d(i);
end;
```

Chapter 3

Interpolation

Prerequisites: Chapters 1 and 2 are required.

Interpolation means inserting or blending in a missing value. It is the art of reading between the entries of a tabulated function. We start this chapter with several introductory examples in Section 3.1, through which we explain the interpolation principle. The most common interpolation technique is to use polynomials, and we show in Section 3.2 a classical techniques: Lagrange polynomials. Section 3.3 is devoted to piecewise interpolation, which leads to the classical cubic splines. This section also contains the well-known Morrison-Woodbury formula.

3.1 Introductory Examples

Assume we know only some values $f(x_i)$ for $i = 0, \cdots, n$ of a function f,

x	$x_0,$	$x_1,$	$\ldots,$	$x_i,$	$z,$	$x_{i+1},$	$\ldots,$	x_n
$y = f(x)$	$y_0,$	$y_1,$	$\ldots,$	$y_i,$	?	$y_{i+1},$	$\ldots,$	y_n

(3.1)

Is there a way to compute or approximate the function value $f(z)$ for some given z without evaluating f? Why should we be interested in that problem?

1. An explicit formula for the function f *may not be known to us*, and the tabulated values (3.1) could be the only information we have. The values could have been obtained by some physical measurement, e.g., when we represent the temperature of the air outside a house during a day:

t	8 am	9 am	11 am	1 pm	5 pm
T in °C	12.1	13.6	15.9	18.5	16.1

   We might be interested in finding out what the temperature was at 10 am.

2. An important application of interpolation occurs in the processing of digital images: when a digital picture is enlarged, we have to *increase the number of pixels*. This has to be done by interpolating values for the additional pixels.

3. Another use of interpolation (maybe not so important anymore) is to compute intermediate values of a complicated tabulated function. Before the age of computers, tables of functions were very popular. Astronomers used for their computations a book with tables of the logarithms and trigonometric functions. Many of those tables have been replaced today by programs that compute the values of functions when needed.

   Consider for example the function

$$f(x) = \int_0^x e^{\sin t} dt.$$

   If we know the tabulated values

x	0.4	0.5	0.6	0.7	0.8	0.9
y	0.4904	0.6449	0.8136	0.9967	1.1944	1.4063

(3.2)

   what is the value $f(0.66)$?

4. Finally, interpolation may also be used for data compression: in a large dense table, one can store only every tenth value and interpolate to obtain the deleted values when needed.

If the new value z is within the range of the *interpolation* points x_i then we speak of interpolation. If the desired value z is outside the range, we call the process *extrapolation*. Predictions are always extrapolations; we might, for instance, be interested in predicting what the temperature will be at 7 pm in the example above. Another nice example for extrapolation is the census demonstration in Matlab. The task consists of estimating the population of the USA in the year 2010 based on census data for the years 1900, 1910, . . . , 2000. This example shows very well how sensitive the problem is. Depending on which model is used, one can obtain very different answers.

To interpolate the data given in (3.1) for some desired value z, we choose a model function $g(x)$. Typically this interpolating function should be easy to evaluate and have the following property:

$$g(x_k) = f(x_k) \text{ for some } x_k \text{ in the neighborhood of } \quad z.$$

If g is known then $g(z)$ is taken as an approximation for $f(z)$. The hope and the aim is that the *interpolation error* $|g(z) - f(z)|$ will be small.

Note that if only the values of (3.1) are given and that we do not know anything more about the function f, then the problem of interpolation is ill posed. Any value can be chosen as "approximation" for $f(z)$. Consider the function

$$f(x) = \sin x + 10^{-3} \ln\left((x - 0.35415)^2\right).$$

If it is tabulated for $x = 0.33, 0.34, \cdots, 0.38$, one would not expect the singularity at $z = 0.35415$:

x	0.3300	0.3400	0.3500	0.3600	0.3700	0.3800
$f(x)$	0.3166	0.3250	0.3319	0.3420	0.3533	0.3636

Thus, if an interpolation procedure yields some value of $f(0.35415) \approx 0.3356$, one would probably be ready to accept this.

Often we assume that f is smooth. In that case more can be said about the interpolation error (see Section 3.2.2).

Example 3.1.1. *We would like to interpolate $f(0.66)$ in (3.2) and choose the function*

$$g(x) = \frac{a}{x-b}.$$

The two parameters a and b are determined by requiring that g interpolates both neighbor points of z, i.e.

$$g(0.6) = \frac{a}{0.6-b} = 0.8136,$$
$$g(0.7) = \frac{a}{0.7-b} = 0.9967.$$

With the Maple-statement

```
> solve({a/(0.6-b)=0.8136, a/(0.7-b)=0.9967},{a,b});
```

we obtain $a = -0.4429$ and $b = 1.1443$ thus

$$g(0.66) = \frac{-0.4429}{0.66 - 1.1443} = 0.9145 \approx f(0.66) = 0.9216.$$

By integrating $e^{\sin(t)}$ numerically one can check that the interpolation error is $|g(0.66) - f(0.66)| = 0.0071$, which is rather large. Indeed, the choice of the model function was not very clever; a better result could have been obtained with the linear function $g(x) = ax+b$, in which case we would get $g(0.66) = 0.9235$ and an error of 0.0019.

3.2 Polynomial Interpolation

A common choice of *model functions for interpolation are polynomials*, which are easy to evaluate and smooth, i.e., infinitely differentiable. Given the $n+1$ points in (3.1), we are looking for a polynomial $P(x)$ such that

$$P(x_i) = f(x_i), i = 0, \cdots, n. \tag{3.3}$$

Since by (3.3) we have $n+1$ constraints to satisfy, we need $n+1$ degrees of freedom. Consider the n-th degree polynomial

$$P_n(x) = a_0 + a_1 x + \ldots + a_{n-1}x_{n-1} + a_n x^n. \tag{3.4}$$

The $n+1$ coefficients $\{a_i\}$have to be determined in such a way that (3.3) is satisfied. This leads to the *linear system* of equations

$$\begin{array}{ccccccccccc}
a_0 & + & a_1 x_0 & + & \ldots & + & a_{n-1}x_0^{n-1} & + & a_n x_0^n & = & f(x_0), \\
a_0 & + & a_1 x_1 & + & \ldots & + & a_{n-1}x_1^{n-1} & + & a_n x_1^n & = & f(x_1), \\
\vdots & & \vdots & & & & \vdots & & \vdots & = & \vdots \\
a_0 & + & a_1 x_n & + & \ldots & + & a_{n-1}x_n^{n-1} & + & a_n x_n^n & = & f(x_n).
\end{array}$$

Written in matrix form, the system reads

$$\underbrace{\begin{bmatrix} 1 & x_0 & \dots & x_0^{n-1} & x_0^n \\ 1 & x_1 & \dots & x_1^{n-1} & x_1^n \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & x_n & \dots & x_n^{n-1} & x_n^n \end{bmatrix}}_{V} \underbrace{\begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix}}_{\boldsymbol{a}} = \underbrace{\begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}}_{\boldsymbol{f}} \tag{3.5}$$

The matrix V with the special structure containing the powers of the nodes is called a Vandermonde Matrix. We shall see in the next section that if all the nodes x_i are different, then the matrix V is non-singular and therefore there exists a unique solution to the linear system of equations.

On the other hand, Vandermonde matrices tend to be ill conditioned (see Table 2.2 in Chapter 2) and, as we will see, there are other ways to compute the interpolating polynomial by representing it in another basis of polynomials rather than with monomials.

3.2.1 Lagrange Polynomials

Instead of trying to directly interpolate the function values $f(x_i)$ at the nodes x_i with a polynomial of degree at most n, we can look for a representation of the interpolating polynomial $P_n(x)$ of the form

$$P_n(x) = \sum_{j=0}^{n} f(x_j) l_j(x), \tag{3.6}$$

where the polynomials l_j should be of degree n as well and have to be determined so that $P_n(x)$ interpolates the function f at the nodes x_j. This is clearly the case if $l_i(x_i) = 1$ and $l_i(x_j) = 0$ for $i \neq j$, because then there is only one contribution to the sum from the i-th term if l_i is evaluated at x_i, namely

$$P_n(x_i) = \sum_{j=0}^{n} f(x_j) l_j(x_i) = f(x_i) l_i(x_i) = f(x_i).$$

Now to determine the $l_i(x)$ we have to find the polynomial which equals 1 at x_i and has n zeros at all the other nodes x_j , $i \neq j$. Such a polynomial can be written in factored form directly as

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \qquad i = 0, 1, \cdots, n. \tag{3.7}$$

This polynomial $l_i(x)$ is clearly zero if evaluated at x_j for $j \neq i$, because one of the factors in the numerator vanishes there. On the other hand, when evaluated at x_i, the numerator and denominator become identical, so the polynomial equals one there, as required. These polynomials are called *Lagrange polynomials* and interpolation becomes very simple once the Lagrange polynomials are available, one simply forms the sum given in Equation (3.6).

Obviously the $n+1$ Lagrange polynomials can only exist if the denominators in Equation (3.7) are nonzero. This implies that the nodes must be distinct, i.e., we must have $x_i \neq x_j$ for all $i \neq j$.

How do we know that the interpolation polynomial expanded in powers of x as in (3.4) and the polynomial constructed with the Lagrange basis functions (3.6) represent the same polynomial? One possibility is to expand (3.6) and reorder the terms and check that the expressions are indeed equal. There is, however, a simpler argument that shows that the polynomials are the same. Assume we have computed two interpolating polynomials $Q(x)$ and $P(x)$ each of degree n such that

$$Q(x_j) = f(x_j) = P(x_j), \qquad j = 0, \cdots, n$$

holds. Then we can form the difference

$$d(x) = Q(x) - P(x).$$

d is certainly a polynomial of degree less or equal n. But because of the interpolation property of P and Q, we have $d(x_j) = Q(x_j) - P(x_j) = 0, \quad j = 0, \ldots, n$. A non-zero polynomial of degree less than or equal to n cannot have more than n zeros. But d has $n+1$ distinct zeros; hence, it must be identically zero, meaning that $Q(x) \equiv P(x)$. Thus we have proved

Theorem 3.2.1. ***(Existence and Uniqueness of the InterpolationPolynomial)*** *Given $n+1$ distinct nodes x_j the Vandermonde matrix in (3.5) is non-singular and there exists a unique interpolating polynomial P_n of degree less or equal n with $P(x_j) = f(x_j)$ for $j = 0, \ldots, n$.*

Example 3.2.1. *We interpolate again the value $f(0.66)$ for the function given in (3.2). We chose $n = 2$ and use the three points*

x	*0.6*	*0.7*	*0.8*
y	*0.8136*	*0.9967*	*1.1944*

to determine the interpolation polynomial (a quadratic function). We obtain

$$\begin{aligned} l_0(x) &= \frac{x-0.7}{0.6-0.7}\frac{x-0.8}{0.6-0.8} \quad \Rightarrow \quad l_0(0.66) = 0.28 \\ l_1(x) &= \frac{x-0.6}{0.7-0.6}\frac{x-0.8}{0.7-0.8} \quad \Rightarrow \quad l_1(0.66) = 0.84 \\ l_2(x) &= \frac{x-0.6}{0.8-0.6}\frac{x-0.7}{0.8-0.7} \quad \Rightarrow \quad l_2(0.66) = -0.12 \end{aligned}$$

Thus we get

$$P_2(0.66) = 0.28 \cdot 0.8136 + 0.84 \cdot 0.9967 - 0.12 \cdot 1.1944 = 0.921708.$$

Since the exact value is $f(0.66) = 0.9216978$ the interpolation error is now $-1.01 \cdot 10^{-5}$ and the interpolated value has the same accuracy as the tabulated values.

The following Matlab function(see Algorithm 21) interpolates with the Lagrange interpolation polynomial.

Algorithm 21 Lagrange interpolation

```
function yy=LagrangeInterpolation(x,y,xx)
% LAGRANGEINTERPOLATION interpolation using Lagrange polynomials
%   yy=LagrangeInterpolation(x,y,xx); uses the points (x,y) for the
%   Lagrange Form of the interpolating polynomial P and iterpolates
%   the values yy=P(xx)


n=length(x); nn=length(xx);
for i=1:nn,
  yy(i)=0;
  for k=1:n
    yy(i)=yy(i)+y(k)*prod((xx(i)-x([1:k-1,k+1:n])))...
                    /prod((x(k)-x([1:k-1,k+1:n])));
  end;
end;
```

3.2.2 Interpolation Error

If the function f has continuous derivatives in the range of interpolation, then an error term can be derived for the interpolation formula.

Theorem 3.2.2. *(Interpolation Error) Let $f, f', ..., f^{(n+1)}$ be continuous in the interval $[x_0, x_n]$, where $x_0 < x_1 < \cdots < x_n$. If the polynomial P_n interpolates f in the nodes x_j then*

$$R_n(x) := f(x) - P_n(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_n)}{(n+1)!} f^{(n+1)}(\xi), \tag{3.8}$$

where ξ is some value between the nodes $x_0, x_1, ..., x_n$ and x.

Proof. For $x = x_k$ the theorem trivially holds, since both sides vanish, so let us consider a fixed x such that $x \neq x_k$, $k = 0, \cdots, n$. Define

$$L(t) = \prod_{i=0}^{n} (t - x_i)$$

and consider the function

$$F(t) = f(t) - P_n(t) - cL(t), \quad \text{with} \quad c = \frac{f(x) - P_n(x)}{L(x)}.$$

Clearly

$$F(x_k) = 0, \quad k = 0, 1, \cdots, n,$$

but also $F(x) = 0$ because of the special choice of the constant c. Thus F has at least $n + 2$ distinct zeros. By Rolle's theorem, the continuity of F' implies that there is a

zero of F' between any two zeros of F. Therefore F' has at least $n+1$ distinct zeros. If we continue to take derivatives and count the zeros we finally find that

$$F^{(n+1)}(t) = f^{(n+1)}(t) - P_n^{(n+1)}(t) - cL^{(n+1)}(t) \tag{3.9}$$

has at least one zero ξ. Since P_n is of degree n we have

$$P_n^{(n+1)}(t) \equiv 0,$$

and, because L is a polynomial of degree $n+1$ with leading coefficient 1, we obtain

$$L^{(n+1)}(t) = (n+1)!.$$

If we insert this for $t = \xi$ in Equation (3.9) and if we solve the equation

$$F^{(n+1)}(\xi) = 0$$

for c then we get using the definition of c

$$c = \frac{f(x) - P_n(x)}{L(x)} = \frac{f^{(n+1)}(\xi)}{(n+1)!},$$

which is what we wanted to prove. □

To estimate the interpolation error in Example 3.2.1 with the expression in (3.8), we need to compute the maximum of $|f'''|$ in the interval $[0.6, 0.8]$. Since

$$f'''(x) = (-\sin x + \cos^2 x)e^{\sin x},$$

and since f''' is monotonically decreasing in this interval with $f'''(0.6) = 0.2050$ and $f'''(0.8) = -0.4753$, we conclude

$$\max_{0.6 \le x \le 0.8} |f'''(x)| = |f'''(0.8)| = 0.4753.$$

Furthermore

$$\max_{0.6 \le x \le 0.8} |L(x)| = 3.849 \cdot 10^{-4},$$

and thus for all $x \in (0.6, 0.8)$ the error can be bounded by

$$|R_n(x)| \le 3.049 \cdot 10^{-5}.$$

For $x = 0.66$ we have$|L(0.66)| = 3.36 \cdot 10^{-4}$ and thus we get the sharper bound

$$|R_n(0.66)| \le 2.6617 \cdot 10^{-5}.$$

The estimate is about twice as large as the exact interpolation error.

Remarks:

- Normally the derivative $f^{(n+1)}$ is not available and so it is difficult to use the error term given above unless a bound on this derivative is known.

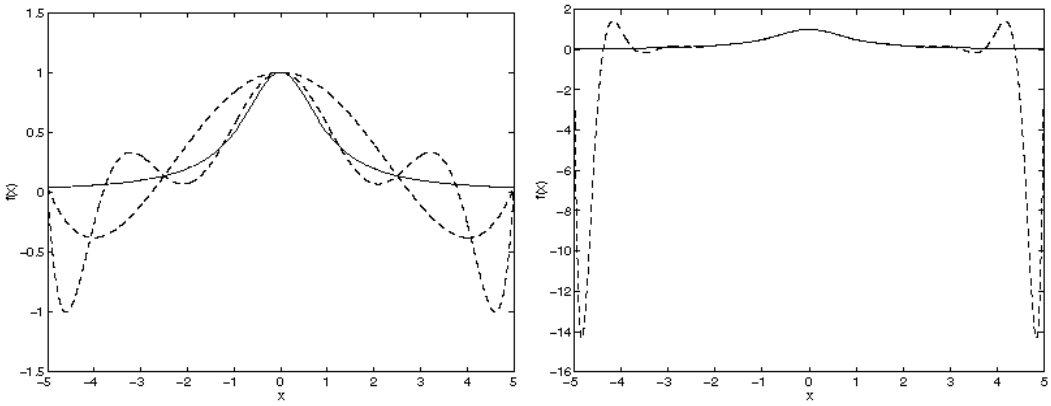


Figure 3.1: Runge's famous example, interpolated with equidistant nodes, on the left with a polynomial of degree four and eight, on the right with a polynomial of degree sixteen.

- From the product $L(x)$ in the error term, one can expect large interpolation errors towards the ends of the interval in which the interpolation is performed, since many of the terms in the product will be large. This is especially the case if many nodes are used. An impressive example was given by Runge and is reproduced in Figure 3.1. The function which is interpolated is

$$f(x) = \frac{1}{1+x^2},$$

and the nodes are chosen to be equidistant on the interval $x \in [-5, 5]$. The polynomials indeed still interpolate the function values, but between the nodes the interpolation error is unacceptably large for higher degree polynomials, especially near the boundary. A remedy is to use non-equidistant nodes which are more closely spaced at the ends of the interval, for example Chebyshev nodes, which are derived from the roots of the *Chebyshev polynomials*; Another possibility is to use piecewise polynomials, as we will see in the Section 3.3, which leads to spline interpolation in Section 3.3.1.

3.3 Piecewise Interpolation with Polynomials

As already shown with Runge's example (see Figure 3.1), the interpolation polynomial may not always produce the result that one would like. The following example demonstrates this clearly.

Example 3.3.1. *The interpolating polynomial through the points*

x	1	2.5	3	5	13	18	20
y	2	3	4	5	7	6	3

(3.10)

has the graph shown in Figure 3.2.

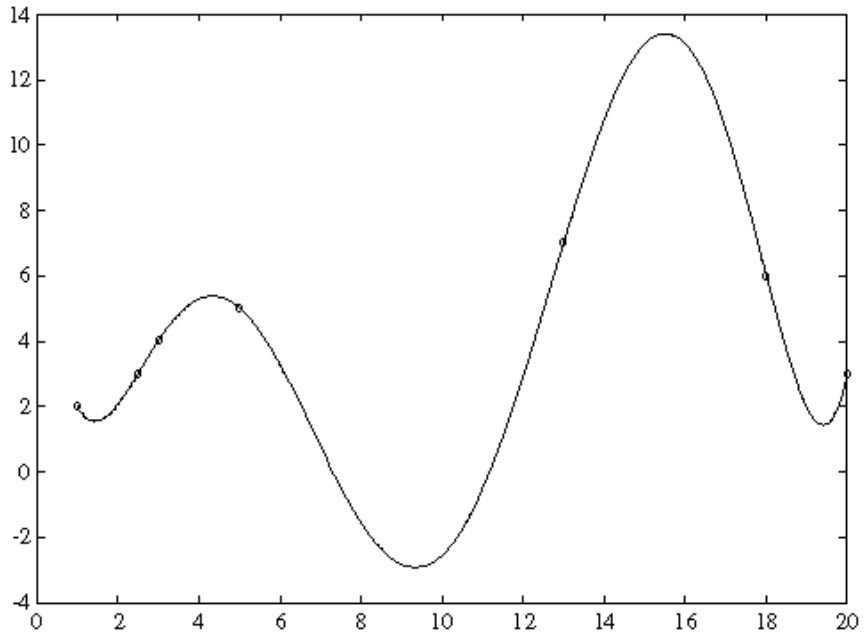


Figure 3.2: Undesirable Interpolation Result Leading to Negative Values

If the values in Table 3.10 were points of a function that must be positive by some physical argument, then it would not be desirable to approximate the function by its interpolation polynomial, since the latter is negative in the interval $(7, 11)$*. If we choose to interpolate piecewise with polynomials of lower degrees, e.g. by second degree polynomials for three consecutive points each, then we obtain Figure 3.3. This time the graph of the interpolation function is all positive; however, the derivative is no longer continuous at the points where the polynomial pieces meet.*

The idea of spline interpolation is to approximate piecewise with polynomials of lower degree in order to obtain a non-oscillating approximation that is also smooth (i.e., as many times differentiable as possible) at the knots where the pieces meet.

3.3.1 Classical Cubic Splines

To avoid discontinuous derivatives, we need to prescribe not only a common function value but also the same value for the derivative at the node where two piecewise polynomials meet. Assume the points

x	x_1	x_2	$\cdots$	x_n
y	y_1	y_2	$\cdots$	y_n

are given. The simplest possibility is to choose each x_i as node, and to interpolate in each interval

$$[x_i, x_{i+1}], i = 1, 2, ..., n-1$$

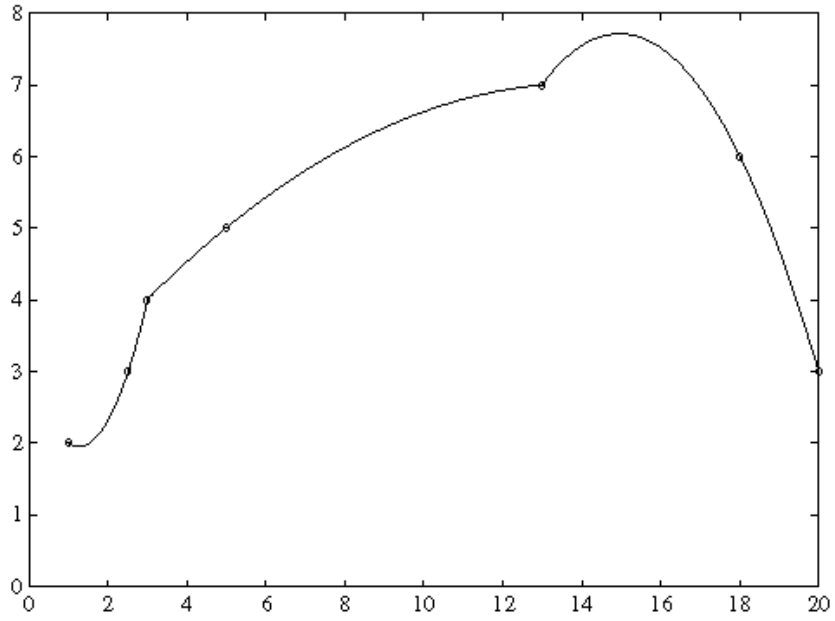


Figure 3.3: Piecewise Interpolation with Parabolas

by one polynomial P_i. The index i does not refer here to the degree of the polynomial but to the number of the interval. The polynomial P_i should satisfy the following conditions:

$$\begin{array}{rclcrcll} P_i(x_i) & = & y_i, & & P_i(x_{i+1}) & = & y_{i+1} & \text{interpolate} \\ P_i'(x_i) & = & y_i', & & P_i'(x_{i+1}) & = & y_{i+1}' & \text{continuous derivative} \end{array} \tag{3.11}$$

Of course we still have to determine what our derivatives y_i' should be, since in general they are not given. The polynomial P_i has to satisfy four conditions – thus it is uniquely determined if we choose a degree of three. To simplify computations we make a change of variables

$$t = \frac{x - x_i}{h_i} \qquad \text{with} \qquad h_i = x_{i+1} - x_i, \tag{3.12}$$

where t is a local variable in the i-th interval. Now

$$Q_i(t) = P_i(x_i + th_i) \tag{3.13}$$

is a polynomial of degree three in t. The derivative is

$$Q_i'(t) = h_i P_i'(x_i + th_i) \tag{3.14}$$

and thus the conditions (3.11) become

$$\begin{array}{rclcrcl} Q_i(0) & = & y_i, & & Q_i(1) & = & y_{i+1}, \\ Q_i'(0) & = & h_i y_i', & & Q_i'(1) & = & h_i y_{i+1}'. \end{array} \tag{3.15}$$

A short computation yields

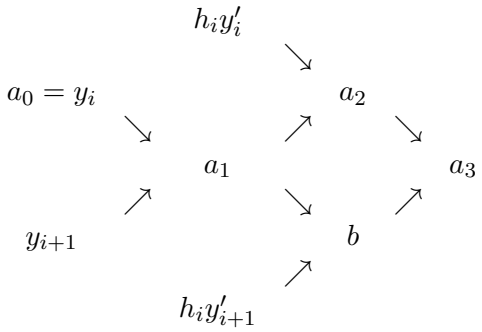
$$\begin{array}{rcl} Q_i(t) & = & y_i(1 - 3t^2 + 2t^3) + y_{i+1}(3t^2 - 2t^3) \\ & & + h_i y_i'(t - 2t^2 + t^3) + h_i y_{i+1}'(-t^2 + t^3). \end{array} \tag{3.16}$$

Definition 3.3.1. ***(Cubic Hermite Polynomials, Cardinal Form)*** *The four polynomials*

$$\begin{aligned} H_0^3(t) &= 1 - 3t^2 + 2t^3 & H_1^3(t) &= t - 2t^2 + t^3 \\ H_3^3(t) &= 3t^2 - 2t^3 & H_2^3(t) &= -t^2 + t^3 \end{aligned}$$

are called cubic Hermite polynomials, and (3.16) is the cardinal form of the interpolating polynomial.

Q_i could be evaluated by the expression (3.16); however, a more efficient scheme is based on Hermite interpolation. For this, one forms differences in the following scheme by subtracting the value above from the one below:



(3.17)

This way we obtain the coefficients a_0, a_1, a_2 and a_3 and we can compute

$$Q_i(t) = a_0 + (a_1 + (a_2 + a_3 t)(t-1))t \tag{3.18}$$

using only 3 multiplications and 8 additions/subtractions. The verification that (3.18) yields the same polynomial as (3.16) is left as an exercise.

In the next section, we will investigate several possibilities for choosing the derivatives y_i'. Assuming that these are known, and therefore also the polynomials Q_i for $i = 1, \cdots, n-1$, the composed global function g is called a cubic spline function. To interpolate with g for a value $x = z$, we proceed in three steps:

1. Determine the interval which contains z, i.e. compute the index i for which $x_i \le z < x_{i+1}$.

2. Compute the local variable $t = (z - x_i)/(x_{i+1} - x_i)$.

3. Evaluate $g(z) = Q_i(t)$ with (3.17) and (3.18).

To find the interval which contains z, we can use a binary search. This is done in the following Matlab function `SplineInterpolation`.

3.3.2 Derivatives for the Spline Function

As we saw in the previous section, we need derivatives at the nodes in order to construct a spline function. In principle, we could prescribe any value for them; however, it makes sense to estimate the derivatives from the given function values. A simple estimate for

Algorithm 22 Generic Cubic Spline

```
function g=SplineInterpolation(x,y,ys,z);
% SPLINEINTERPOLATION generic interpolation with cubic splines
%   g=SplineInterpolation(x,y,ys,z); interpolates at the scalar
%   location z the data (x,y,ys) with a cubic spline function.
%   Here ys is the desired derivative at x.

n=length(x);
a=1; b=n; i=a;
while a+1~=b,
  i=floor((a+b)/2);
  if x(i)<z, a=i; else b=i; end
end
i=a; h=(x(i+1)-x(i));
t=(z-x(i))/h;
a0=y(i); a1=y(i+1)-a0; a2=a1-h*ys(i);
a3= h*ys(i+1)-a1; a3=a3-a2;
g=a0+(a1+(a2+a3*t)*(t-1))*t;
```

the derivative at point (x_i, y_i) is given by the slope of the straight line through the neighboring points (see Figure 3.4),

$$y'_i = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}} = \frac{y_{i+1} - y_{i-1}}{h_i + h_{i+1}}, i = 2, 3, \cdots, n-1. \tag{3.19}$$

Using (3.19), we can compute derivatives for all inner nodes. For the two boundary points, we have several possibilities:

1. Use the slope of the line through the first two (respectively the last two) points

   $$y'_1 = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{respectively} \quad y'_n = \frac{y_n - y_{n-1}}{x_n - x_{n-1}}.$$

   With Matlab the derivatives `ys` for this case can be computed by

   ```
   n=length(x);
   ys=(y(3:n)-y(1:n-2))./(x(3:n)-x(1:n-2));
   ys=[(y(2)-y(1))/(x(2)-x(1)); ys; (y(n)-y(n-1))/(x(n)-x(n-1))];
   ```

2. *Natural boundary conditions*: they are defined such that the second derivative vanishes. From the equations

   $$Q''_1(0) = 0 \quad \text{and} \quad Q''_{n-1}(1) = 0$$

   we obtain

   $$y'_1 = \frac{3}{2}\frac{y_2 - y_1}{h_1} - \frac{1}{2}y'_2 \quad \text{respectitively} \quad y'_n = \frac{3}{2}\frac{y_n - y_{n-1}}{h_{n-1}} - \frac{1}{2}y'_{n-1}.$$

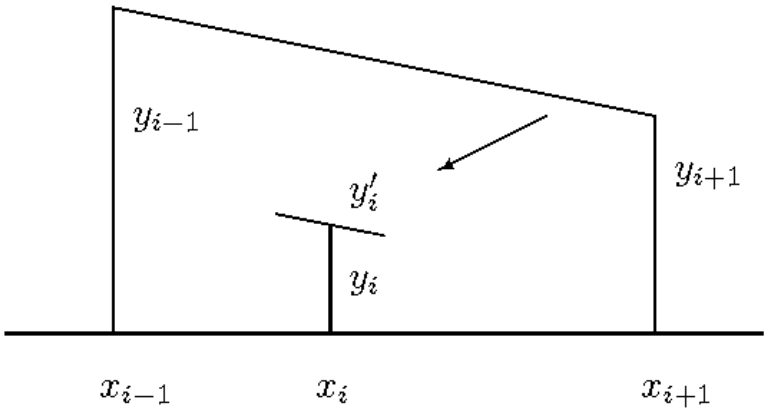


Figure 3.4: Slope of the line through neighboring points

3. An important special case are periodic boundary conditions. If the function values belong to a periodic function then $y_1 = y_n$ and we can choose

$$y'_1 = y'_n = \frac{y_2 - y_{n-1}}{h_1 + h_{n-1}}$$

This corresponds again to the slope of the line through neighboring points.

Spline functions which are computed with these constructed derivatives are called defective spline functions.

Let us now interpolate again the points given in Table 3.10. Using the slopes of the neighboring points and the natural boundary conditions we obtain the defective spline of Figure 3.5. We have also plotted the derivatives g' and g''. We see that g' (dotted line) is continuous while g'' (dashed line) is discontinuous.

The question is whether it is possible to choose the derivatives y'_i in such a way that the second derivative g'' will also be continuous. We would like to have

$$P''_i(x_{i+1}) = P''_{i+1}(x_{i+1}) \quad \text{for} \quad i = 1, 2, \cdots, n-2. \tag{3.20}$$

Written in the local variable t with $Q_i(t) = P_i(x_i + h_i t)$, the conditions (3.20) become

$$\frac{Q''_i(1)}{h_i^2} = \frac{Q''_{i+1}(0)}{h_{i+1}^2} \quad \text{for} \quad i = 1, 2, \cdots, n-2. \tag{3.21}$$

If we differentiate Q_i in (3.16) we get

$$Q''_i(t) = y_i(-6 + 12t) + y_{i+1}(6 - 12t) + h_i y'_i(-4 + 6t) + h_i y'_{i+1}(-2 + 6t), \tag{3.22}$$

and if we insert this into (3.21), we obtain

$$\frac{6}{h_i^2}(y_i - y_{i+1}) + \frac{2}{h_i}y'_i + \frac{4}{h_i}y'_{i+1} = \frac{6}{h_{i+1}^2}(y_{i+2} - y_{i+1}) - \frac{4}{h_{i+1}}y'_{i+1} - \frac{2}{h_{i+1}}y'_{i+2} \tag{3.23}$$

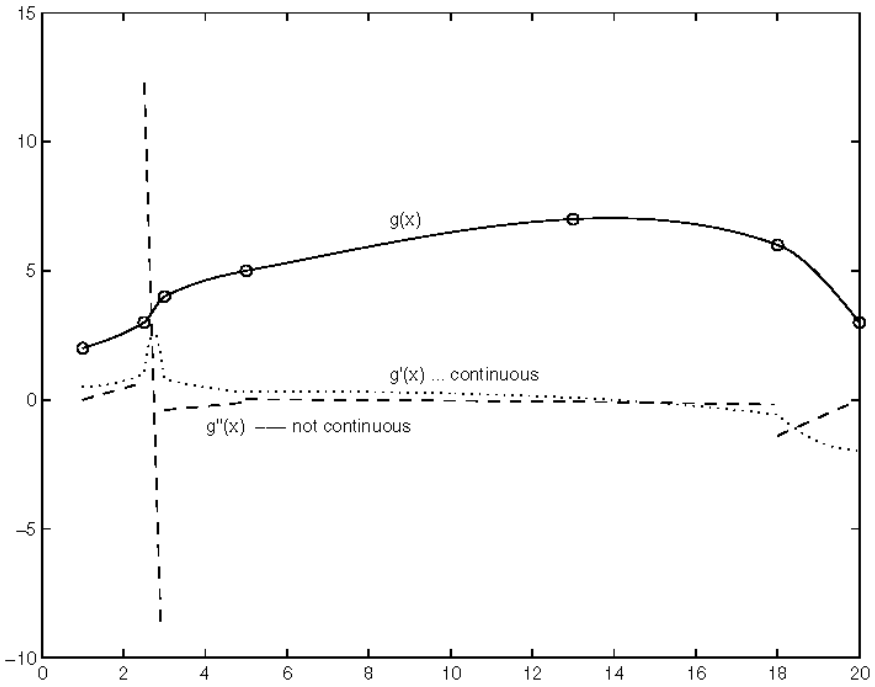


Figure 3.5: Defective Spline

Equation (3.23) is a linear equation for the unknown derivatives y'_i, y'_{i+1} and y'_{i+2}. We can write such an equation for $i = 1, 2, \cdots, n-2$. In matrix notation, we get a linear system of $n-2$ equations with n unknowns,

$$A\boldsymbol{y'} = c \tag{3.24}$$

with $\boldsymbol{y'} = (y'_1, y'_2, \cdots, y'_n)^\top$ being the vector of unknowns and

$$A = \begin{pmatrix} b_1 & a_1 & b_2 & & & \\ & b_2 & a_2 & b_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & b_{n-2} & a_{n-2} & b_{n-1} \end{pmatrix}$$

a *tridiagonal matrix* with the elements

$$b_i = \frac{1}{h_i} \quad \text{and} \quad a_i = \frac{2}{h_i} + \frac{2}{h_{i+1}}, \quad i = 1, 2, \cdots, n-2.$$

The right hand side of Equation (3.24) is

$$\boldsymbol{c} = \begin{pmatrix} 3(d_2 + d_1) \\ 3(d_3 + d_2) \\ \vdots \\ 3(d_{n-1} + d_{n-2}) \end{pmatrix}$$

where we have used the abbreviation

$$d_i = \frac{y_{i+1} - y_i}{h_i^2}, \quad i = 1, 2, \cdots, n-1. \tag{3.25}$$

Thus we have obtained $n-2$ linear equations for the unknown derivatives. We need two more equations to determine them uniquely. Just as for the defective splines, we can ask for further boundary conditions. We consider three possibilities:

1. *Natural boundary conditions*: $P_1''(x_1) = P_{n-1}''(x_n) = 0$. These conditions give two more equations:

$$\begin{aligned} \frac{2}{h_1}y_1' + \frac{1}{h_1}y_2' &= \quad 3d_1, \\ \frac{1}{h_{n-1}}y_{n-1}' + \frac{2}{h_{n-1}}y_n' &= \quad 3d_{n-1}, \end{aligned} \tag{3.26}$$

which are obtained from $Q_1''(0) - Q_{n-1}''(1) = 0$ using (3.22). If we add them to the system of equations (3.24), then we can compute the derivatives y_i' by solving a linear system of equations with a *tridiagonal matrix*. Using the tridiagonal solver `Thomas` (see Algorithm 18 in Chapter 2), we obtain the derivatives in this case with the statements

```
h=x(2:n)-x(1:n-1);
a=2./h(1:n-2)+2./h(2:n-1);
b=1./h(1:n-1);
aa=[2/h(1); a; 2/h(n-1)]
bb=3*[d(1); d(2:n-1)+d(1:n-2); d(n-1)]
ys=Thomas(b,aa,b,bb)
```

The spline function $s : [x_1, x_n] \longrightarrow \mathbb{R}$ that is determined piecewise by the $P_i(x), i = 1, 2, \cdots, n-1$ this way is a simplified model of the shape of a thin wooden spline that passes through the given points, as the following theorem shows.

Theorem 3.3.2. *For a given set of points $(x_i, y_i), i = 1, 2, ..., n$, let $s : [x_1, x_n] \longrightarrow \mathbb{R}$ be the classical cubic spline function satisfying $s(x_i) = y_i, i = 1, 2, ..., n$. Then for any twice continuously differentiable function $f : [x_1, x_n] \longrightarrow \mathbb{R}$ satisfying $f(x_i) = y_i, i = 1, 2, ..., n$ and*

$$s''(x_n)(f'(x_n) - s'(x_n)) = s''(x_1)(f'(x_1) - s'(x_1)), \tag{3.27}$$

we have that

$$\int_{x_1}^{x_n} (s''(x))^2 dx \leq \int_{x_1}^{x_n} (f''(x))^2 dx, \tag{3.28}$$

i.e. the spline function $s(x)$ minimizes the energy defined by the integral in (3.28)

Proof. Let $s(x)$ be the minimizer of $\int_{x_1}^{x_n}(s''(x))^2 dx$. Every twice continuously differentiable function passing through the points $(x_i, y_i), i = 1, 2, \cdots, n$can be written in the form

$$f(x) := s(x) + \varepsilon h(x),$$

where $\varepsilon \in \mathbb{R}$ and $h(x)$ is a twice continuously differentiable function with zeros at $x_i, h(x_i) = 0, i = 1, 2, \cdots, n$. The minimality condition (3.28) becomes

$$\begin{aligned}\int_{x_1}^{x_n}(s''(x))^2dx &\leq \int_{x_1}^{x_n}(s''(x)+\varepsilon h''(x))^2dx \\ &= \int_{x_1}^{x_n}(s''(x))^2dx + 2\varepsilon\int_{x_1}^{x_n} s''(x)h''(x)dx + \varepsilon^2\int_{x_1}^{x_n}(h''(x))^2dx.\end{aligned}$$

For a fixed function $h(x)$, this condition is satisfied for all ε if and only if

$$\int_{x_1}^{x_n} s''(x)h''(x)dx = 0 \tag{3.29}$$

as one can see by differentiation. Integration by parts leads to

$$s''(x)h'(x)|_{x_0}^{x_n} - \int_{x_1}^{x_n} s'''(x)h'(x)dx = 0. \tag{3.30}$$

Now condition (3.27) implies that the boundary term in (3.30) vanishes. Since $s'''(x)$ is constant in each interval $(x_i - 1, x_i), i = 2, 3, \cdots, n$, say equal to the constant C_i, the second term in (3.30) becomes

$$\int_{x_1}^{x_n} s'''(x)h'(x)dx = \sum_{i=2}^{n} C_i \int_{x_{i-1}}^{x_i} h'(x)dx = \sum_{i=2}^{n} C_i(h(x_i) - h(x_{i-1})) = 0,$$

since h vanishes at the nodes, and therefore (3.29) holds, which implies (3.28) and concludes the proof. □

We see that in addition to the natural boundary conditions $s''(x_1) = s''(x_n) = 0$ from (3.27), the so called *clamped boundary conditions* $s'(x_1) = f'(x_1)$ and $s'(x_n) = f'(x_n)$ also lead to an energy minimizing spline among all interpolating functions f with these slopes at the endpoints. When interpolating a function f with a spline, it is better to use clamped boundary conditions, since with free boundary conditions, the approximation order is polluted from the boundary and drops from $O(h^4)$ to $O(h^2)$. As an alternative, one can use the approach described next, which does not require the knowledge of derivatives of f, but then looses the energy minimizing property toward the boundaries.

2. *Not-a-knot condition* of de Boor: Here we want the two polynomials in the first two and in the last two intervals to be the same:

$$P_1(x) \equiv P_2(x) \quad \text{and} \quad P_{n-2}(x) \equiv P_{n-1}(x) \tag{3.31}$$

We get here the two equations:

$$\begin{aligned}\tfrac{1}{h_1}y_1' + \left(\tfrac{1}{h_1} + \tfrac{1}{h_2}\right)y_2' &= 2d_1 + \tfrac{h_1}{h_1+h_2}(d_1+d_2), \\ \left(\tfrac{1}{h_{n-2}} + \tfrac{1}{h_{n-1}}\right)y_{n-1}' + \tfrac{1}{h_{n-1}}y_n' &= 2d_{n-1} + \tfrac{h_{n-1}}{h_{n-1}+h_{n-2}}(d_{n-1}+d_{n-2}).\end{aligned} \tag{3.32}$$

At first sight condition (3.31) might look strange. The motivation is as follows. If the function values y_i are equidistant with step-size h and belong to a smooth differentiable function f, then according to the error estimate (3.8) we would expect

an interpolation error $\sim h^4$. It can be shown in this case for a spline function with natural boundary conditions that the interpolation error is $\sim h^2$. The natural boundary condition is not at all *natural* from the viewpoint of approximating a function. In fact it is not clear why the function $f$that we wish to approximate should always have a vanishing second derivative at the endpoints. De Boor's Condition (3.31) yields an approximation $\sim h^4$.

To obtain (3.32), it is sufficient to demand that the first two (respectively the last two) polynomials have the same third derivative. Since the third derivative of a cubic polynomial is constant, the two polynomials must be the same. For the first two polynomials, the equation

$$P_1'''(x_2) = P_2'''(x_2)$$

is equivalent to

$$\frac{Q_1'''(1)}{h_1^3} = \frac{Q_2'''(0)}{h_2^3},$$

which is

$$\frac{1}{h_1^2}y_1' + (\frac{1}{h_1^2} - \frac{1}{h_2^2})y_2' - \frac{1}{h_2^2}y_3' = 2\left(\frac{d_1}{h_1} - \frac{d_2}{h_2}\right). \tag{3.33}$$

Equation (3.33) can be added to the system (3.24) as its first equation. Similarly, the second equation of (3.31) produces an equation that is added as the last equation to the system (3.24). Thus, we obtain once again n equations with n unknowns. Unfortunately, the matrix is no longer tridiagonal. If we wish to solve the system with a tridiagonal solver, we have to replace (3.33) by an equivalent one which contains only the unknowns y_1' and y_2' Denote by (I) Equation (3.33) and by (II) the first equation of system (3.24). Both equations contain the unknowns y_1', y_2' and y_3'. With the linear combination

$$(I) + \frac{1}{h_2}(II)$$

we eliminate the unknown y_3' and obtain after the division with $(\frac{1}{h_1} + \frac{1}{h_2})$ the equation

$$\frac{1}{h_1}y_1' + \left(\frac{1}{h_1} + \frac{1}{h_2}\right)y_2' = 2d_1 + \frac{h_1}{h_1 + h_2}(d_1 + d_2), \tag{3.34}$$

which we use now instead of (3.33) to preserve the tridiagonal structure. Similarly we obtain the second equation of (3.32).

To solve a linear system with a tridiagonal matrix we make again use of the function `Thomas` (see Algorithm 18). Note that the Matlab-function `YY=spline(X,Y,XX)` provides in `YY`, the values of the interpolating function at `XX`. The spline is ordinarily constructed using the not-a-knot end conditions.

3. *Periodic boundary conditions*: if y_i are function values of a periodic function, then $y_1 = y_n$. In addition, we require that the first and second derivatives be the same:

$$P_1'(x_1) = P_{n-1}'(x_n), P_1''(x_1) = P_{n-1}''(x_n). \tag{3.35}$$

The first condition in (3.35) is equivalent to

$$y_1' = y_n', \tag{3.36}$$

and the second, when expressed in Q_i, becomes

$$\frac{Q_1''(0)}{h_1^2} = \frac{Q_{n-1}''(1)}{h_{n-1}^2},$$

which yields the equation

$$2\left(\frac{1}{h_1} + \frac{1}{h_{n-1}}\right) y_1' + \frac{1}{h_1} y_2' + \frac{1}{h_{n-1}} y_{n-1}' = 3(d_1 + d_{n-1}), \tag{3.37}$$

If we use (3.36) to eliminate the unknown y_n', we obtain a system of linear equations $B\boldsymbol{y'} = \boldsymbol{c}$ with $(n-1)$ unknowns and $(n-1)$ equations. The matrix has the form

$$B = \begin{pmatrix} a_0 & b_1 & 0 & \cdots & 0 & b_{n-1} \\ b_1 & a_1 & b_2 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & & & \ddots & \ddots & b_{n-2} \\ b_{n-1} & 0 & \cdots & 0 & b_{n-2} & a_{n-2} \end{pmatrix} \tag{3.38}$$

where we have again used the abbreviations

$$b_i = \frac{1}{h_i}, \qquad i = 1, 2, \cdots, n-1,$$

and

$$\begin{aligned} a_0 &= \tfrac{2}{h_{n-1}} + \tfrac{2}{h_1}, \\ a_i &= \tfrac{2}{h_i} + \tfrac{2}{h_{i+1}}, \quad i = 1, 2, \cdots, n-2. \end{aligned}$$

The right hand side $\boldsymbol{c}$ of the system is computed by the coefficients d_i defined in Equation (3.25),

$$\boldsymbol{c} = \begin{pmatrix} 3(d_1 + d_{n-1}) \\ 3(d_2 + d_1) \\ \vdots \\ 3(d_{n-1} + d_{n-2}) \end{pmatrix} \tag{3.39}$$

Except for the two matrix elements at the bottom left and top right the matrix would be tridiagonal.

3.3.3 Sherman-Morrison-Woodbury Formula

Let A be an $n \times n$ matrix and U, V be $n \times p$ with $p \leq n$ (often $p \ll n$). Then every solution $\boldsymbol{x}$ of the linear system

$$(A + UV^\top)\boldsymbol{x} = \boldsymbol{b} \tag{3.40}$$

is also a solution of the *augmented system*

$$\begin{aligned} A\boldsymbol{x} + U\boldsymbol{y} &= \boldsymbol{b}, \\ V^\top \boldsymbol{x} - \boldsymbol{y} &= 0. \end{aligned} \tag{3.41}$$

Now if we assume that the matrices in the following algebraic manipulations are invertible, then by solving the first equation for $\boldsymbol{x}$, we obtain

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} - A^{-1}U\boldsymbol{y}. \tag{3.42}$$

Introducing this in the second equation of (3.41), we get

$$\boldsymbol{y} = (I + V^\top A^{-1}U)^{-1}V^\top A^{-1}\boldsymbol{b}, \tag{3.43}$$

therefore

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} - A^{-1}U(I + V^\top A^{-1}U)^{-1}V^\top A^{-1}\boldsymbol{b}.$$

But from (3.40) and (3.41), we also have

$$\boldsymbol{x} = (A + UV^\top)^{-1}\boldsymbol{b}$$

and

$$\boldsymbol{y} = V^\top (A + UV^\top)^{-1}\boldsymbol{b}.$$

Equating both expressions for $\boldsymbol{x}$ and $\boldsymbol{y}$ with the expressions (3.42) and (3.43) we get the matrix equations:

$$\begin{aligned} V^\top (A + UVT)^{-1} &= (I + V^\top A^{-1}U)^{-1}V^\top A^{-1}, \\ (A + UV^\top)^{-1} &= A^{-1} - A^{-1}U(I + V^\top A^{-1}U)^{-1}V^\top A^{-1}. \end{aligned} \tag{3.44}$$

Equation (3.44) is called the *Shermann-Morrison-Woodbury Formula*. The (small) $p \times p$ matrix $I + V^\top A^{-1}U$ is called *capacitance matrix*. The Sherman- Morrison–oodbury formula is useful for computing the inverse of a rank-p change of the matrix A. Let us consider a few applications:

1. If A is sparse and/or $A\boldsymbol{v} = \boldsymbol{b}$ is easy to solve, then it pays to use the Sherman-Morrison-Woodbury formula to compute the solution of $(A + UV^\top)\boldsymbol{x} = \boldsymbol{b}$. The algorithm is

   (a) Solve $A\boldsymbol{y} = \boldsymbol{b}$.

   (b) Compute the $n \times p$ matrix W by solving $AW = U$. This can be combined with the first step by simultaneously solving linear systems with the same coefficient matrix A with $p + 1$ right hand sides.

   (c) Form the capacitance matrix $C = I + V^\top W \in \mathbb{R}^{p \times p}$ and solve the linear system $C\boldsymbol{z} = V^\top \boldsymbol{y}$.

   (d) the solution is $\boldsymbol{x} = \boldsymbol{y} - W\boldsymbol{z}$.

2. Rank-1 change of the identity matrix. The Sherman-Morrison- Woodbury formula becomes
$$(I + \boldsymbol{u}\boldsymbol{v}^\top)^{-1} = I - \frac{1}{1 - \boldsymbol{v}^\top \boldsymbol{u}} \boldsymbol{u}\boldsymbol{v}^\top.$$
If we have to solve a linear system $B\boldsymbol{x} = \boldsymbol{b}$ with $B = I + \boldsymbol{u}\boldsymbol{v}^\top$, then the solution can be computed in $O(n)$ operations as a linear combination of the vectors $\boldsymbol{b}$ and $\boldsymbol{u}$:
$$\boldsymbol{x} = \boldsymbol{b} - \frac{\boldsymbol{v}^\top \boldsymbol{b}}{1 - \boldsymbol{v}^\top \boldsymbol{u}} \boldsymbol{u}.$$

3. Splines with periodic boundary conditions. The matrix B defined in Equation (3.38) is a rank-1 change of a tridiagonal matrix. With $\boldsymbol{e} = \boldsymbol{e}_1 + \boldsymbol{e}_{n-1} = (1, 0, \cdots, 0, 1)^\top$ we have
$$B = A + \frac{1}{h_{n-1}} \boldsymbol{e}\boldsymbol{e}^\top$$
with the tridiagonal matrix
$$A = \begin{pmatrix} \tilde{a}_0 & b_1 & & & \\ b_1 & a_1 & b_2 & & \\ & b_2 & \ddots & \ddots & \\ & & \ddots & a_{n-3} & b_{n-2} \\ & & & b_{n-2} & \tilde{a}_{n-2} \end{pmatrix}$$
The coefficients a_i and b_i are the same as given in Equation (3.38), but
$$\tilde{a}_0 = \frac{1}{h_{n-1}} + \frac{2}{h_1}, \tilde{a}_{n-2} = \frac{2}{h_{n-2}} + \frac{1}{h_{n-1}}.$$
The solution of the linear system $B\boldsymbol{y'} = \boldsymbol{c}$ for the derivatives requires three steps. We make use of the function `Thomas` (Algorithm 18) to solve the linear systems with tridiagonal matrices:

(a) Solve $A\boldsymbol{u} = \boldsymbol{e}$ with `Thomas`.

(b) Solve $A\boldsymbol{v} = \boldsymbol{c}$ with `Thomas`.

(c) $\boldsymbol{y'} = \boldsymbol{v} - \frac{v_1 + v_{n-1}}{u_1 + u_{n-1} + h_{n-1}} \boldsymbol{u}$

3.3.4 Spline Curves

Given n points in the plane (x_i, y_i) for $i = 1, 2, \cdots, n$, we would like to connect them by a curve. The numbering of the points is crucial; reordering them will give us another curve. Plane curves are represented by parametric functions
$$(x(s), y(s)) \quad \text{with} \quad s_1 \leq s \leq s_n.$$
We can interpret the given points as function values for some (yet to be determined) parametrization
$$x(s_i) = x_i, \quad y(s_i) = y_i, \quad i = 1, 2, \cdots, n.$$

The sequence $\{s_i\}$ of the parameter values can be chosen arbitrarily, we only have to pay attention to monotonicity that means the sequence must be strictly increasing $s_i < s_i + 1$. Often the parameter is chosen to be the arc length, therefore it seems reasonable to parametrize using the distance between successive points

$$s_1 = 0, \quad s_{i+1} = s_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}, \quad i = 1, 2, \cdots, n-1. \tag{3.45}$$

After computing the sequence s_i according to (3.45) we have the data

s	s_1	s_2	$\cdots$	s_n
x	x_1	x_2	$\cdots$	x_n

for $x(s)$

and

s	s_1	s_2	$\cdots$	s_n
y	y_1	y_2	$\cdots$	y_n

for $y(s)$.

Both functions $x(s)$ and $y(s)$ can now be interpolated with any of the variants for spline interpolation and the curve can then be plotted. For closed curves, it is important to use the periodicity condition to avoid a cusp at the endpoints.

Example 3.3.2. *For the points*

x	*1.31*	*2.89*	*5.05*	*6.67*	*3.12*	*2.05*	*0.23*	*3.04*	*1.31*
y	*7.94*	*5.50*	*3.47*	*6.40*	*3.77*	*1.07*	*3.77*	*7.41*	*7.94*

we obtain the curve of Figure 3.6 by using defective splines with periodic boundary conditions.

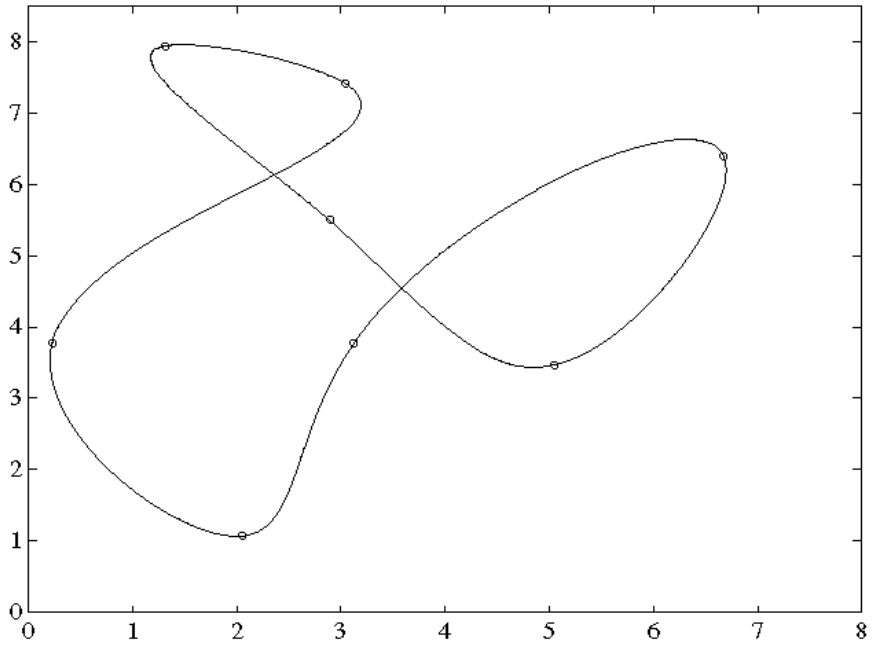


Figure 3.6: Spline Curve

Chapter 4

Nonlinear Equations

Prerequisites: This chapter requires Sections 1.5 (conditioning), 1.8 (stopping criteria), Chapter 2 (linear equations), as well as polynomial interpolation (3.2).

Solving a nonlinear equation in one variable means: given a continuous function f on the interval $[a, b]$, we wish to find a value $s \in [a, b]$ such that $f(s) = 0$. Such a value s is called a *zero or root of the function f or a solution of the equation $f(x) = 0$*. For a multivariate function $f : R^n \longrightarrow R^n$, solving the associated system of equations means finding a vector $s \in R^n$ such that $\boldsymbol{f}(s) = 0$. After an introductory example, we show in Section 4.2 the many techniques for finding a root of a scalar function: the fundamental bisection algorithm, fixed point iteration including convergence rates. Section 4.3 is devoted to the special case of finding zeros of polynomials. In Section 4.4, we leave the scalar case and consider non-linear systems of equations, where fixed point iterations are the only realistic methods for finding a solution. The main workhorse for solving non-linear systems is then Newton's method, and variants thereof.

4.1 Introductory Example

We use *Kepler's Equation* as our motivating example: consider a *two-body problem* like a satellite orbiting the earth or a planet revolving around the sun. Kepler discovered that the orbit is an ellipse and the central body F (earth, sun) is in a focus of the

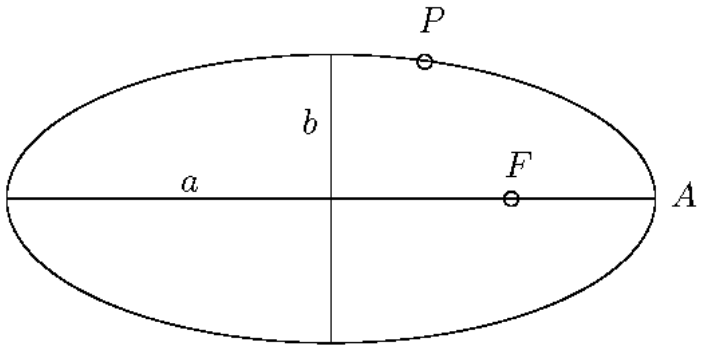


Figure 4.1: Satellite P orbiting the earth F

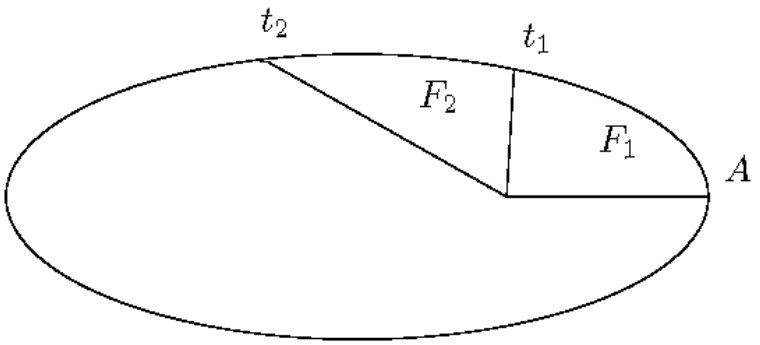


Figure 4.2: Kepler's second law: if $F_1 = F_2$ then $t_2 = 2t_1$

ellipse. If the ellipse is eccentric (i.e., not a circle), then the speed of the satellite P is not uniform: near the earth it moves faster than far away. Figure 4.1 depicts the situation. Kepler also discovered the law of this motion by carefully studying data from the observations by Tycho Brahe. It is called *Kepler's second law* and says that the travel time is proportional to the area swept by the radius vector measured from the focus where the central body is located, see Figure 4.2. We would like to use this law to predict where the satellite will be at a given time.

Assume that at $t = 0$ the satellite is at point A, the perihelion of the ellipse, nearest to the earth. Assume further that the time for completing a full orbit is T. The question is: where is the satellite at time t (for $t < T$)?

We need to compute the area ΔFAP that is swept by the radius vector as a function of the angle E (see Figure 4.3). E is called the eccentric anomaly. The equation of the ellipse with semi-axis a and b is

$$x(E) = a\cos E, \quad y(E) = b\sin E.$$

To compute the infinitesimal area dI between two nearby radius vectors, we will use the cross product, see Figure 4.4. The infinitesimal vector of motion $(x'(E)dE, y'(E)dE)$ can be obtained by Taylor expansion from the difference of the two radius vectors. Taking the cross product, we get

$$dI = \frac{1}{2}\left|\begin{pmatrix} x(E) \\ y(E) \\ 0 \end{pmatrix} \times \begin{pmatrix} x'(E) \\ y'(E) \\ 0 \end{pmatrix}\right| dE = \frac{1}{2}(x(E)y'(E) - x'(E)y(E))dE$$

Inserting the derivatives

$$x'(E) = -a\sin E, \quad y'(E) = b\cos E$$

and integrating $I = \int_0^E dI$ we obtain the simple expression for the area

$$I = \Delta 0AP = \frac{1}{2}abE.$$

To obtain the area ΔFAP we now have to subtract from I the area of the triangle $\Delta 0FP$. This area is given by

$$\frac{aeb\sin E}{2}$$

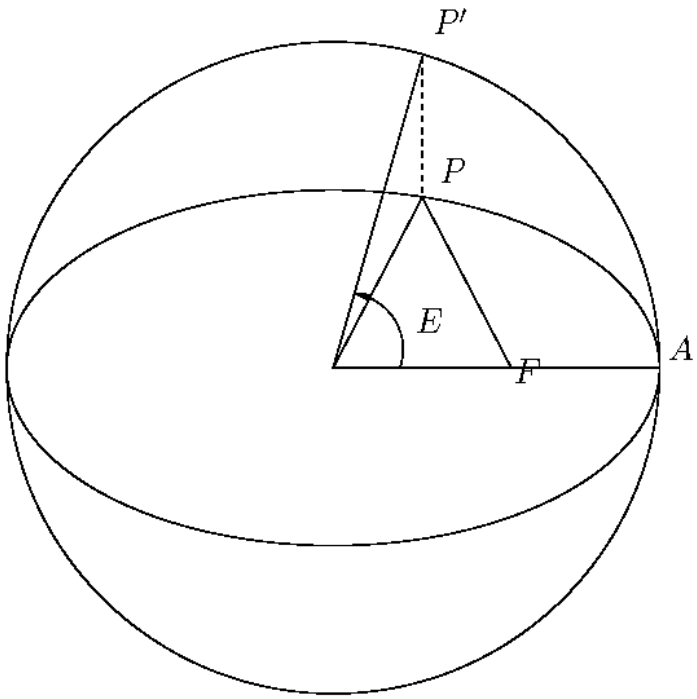


Figure 4.3: Definition of E

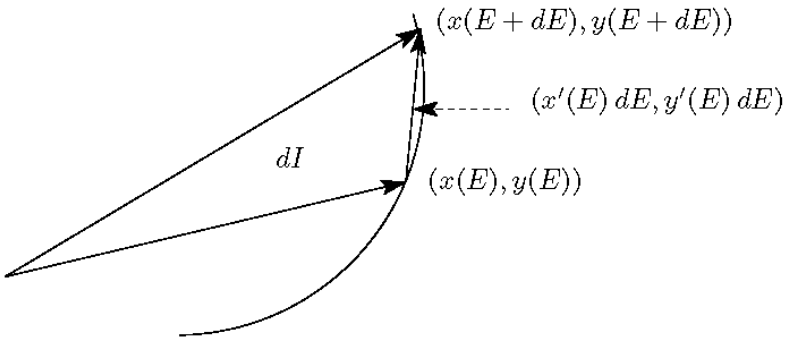


Figure 4.4: Computing an infinitesimal area

where $e = \frac{\sqrt{a^2-b^2}}{a}$ is called the *eccentricity* of the ellipse. Thus we obtain the following function of $S(E)$ for the area swept by the radius:

$$S(E) = \frac{1}{2}ab(E - e\sin E).$$

Now according to Kepler's second law, $S(E)$ is proportional to the time t. Thus $E - e\sin E \sim t$. The proportionality factor must be $2\pi/T$, and therefore

$$E - e\sin E = \frac{2\pi}{T}t, \qquad \textit{Kepler's Equation}. \tag{4.1}$$

Kepler's Equation (4.1) defines the implicit function $E(t)$, i.e. gives the relation between the location of the satellite (angle E) and the time t. If we want to know where the satellite is for a given time t, then we have to solve the nonlinear Equation (4.1) for E.

Some typical values for a satellite are $T = 90$ minutes, and $e = 0.8$. If we want to know the position of the satellite for $t = 9$ minutes, then we have to solve

$$f(E) = E - 0.8\sin E - \frac{2\pi}{10} = 0. \tag{4.2}$$

4.2 Scalar Nonlinear Equations

Finding roots of scalar nonlinear equations is already a difficult task, and there are many numerical methods devoted to it. We show in this section some of the most popular ones, and not all of these methods can be generalized to higher dimensional problems. We start with the simplest but also most robust method called bisection.

4.2.1 Bisection

The first method for solving an equation $f(x) = 0$ which we will discuss in this chapter is called bisection. We assume that we know an interval $[a, b]$ for which $f(a) < 0$ and $f(b) > 0$. If f is continuous in $[a, b]$ then there must exist a zero $s \in [a, b]$. To find it, we compute the midpoint x of the interval and check the value of the function $f(x)$. Depending on the sign of $f(x)$, we can decide in which subinterval $[a, x]$ or $[x, b]$ the zero must lie. Then we continue this process of bisection in the corresponding subinterval until the size of the interval containing s becomes smaller than some given tolerance:

Algorithm 23 Bisection - First Version

```
while b-a>tol
  x=(a+b)/2
      if f(x)<0, a=x; else b=x; end
end
```

At each step of Algorithm 23, we compute a new interval (a_k, b_k) containing s. We have

$$b_k - a_k = \frac{1}{2}(b_{k-1} - a_{k-1}) = \frac{1}{2^k}(b - a).$$

Since the kth approximation of s is $x_k = (a_k + b_k)/2$, we obtain for the error

$$|x_k - s| \leq b_k - a_k = \frac{1}{2^k}(b - a) \to 0, \quad k \to \infty. \tag{4.3}$$

If we want the error to satisfy $|x_k - s| \leq tol$, then it suffices to have $(b-a)/2^k \leq tol$, so that

$$k > \ln\left(\frac{b-a}{tol}\right) / \ln 2. \tag{4.4}$$

Algorithm 23 can be improved. First, it does not work if $f(a) > 0$ and $f(b) < 0$. This can easily be fixed by multiplying the function f by -1. Second, the algorithm will also fail if the tolerance tol is too small: assume for example that the computer works with a mantissa of 12 decimal digits and that

$$a_k = 5.34229982195, \quad b_k = 5.34229982200$$

Then $a_k + b_k = 10.68459964395$ is the exact value and the rounded value to 12 digits is 10.6845996440. Now $x_k = (a_k + b_k)/2 = 5.34229982200 = b_k$. *Thus there is no machine number with 12 decimal digits between a_k and b_k and the midpoint is rounded to b_k.* Since $b_k - a_k = 5e-11$, a required tolerance of say $tol = 1e-15$ would be unreasonable and would produce an infinite loop with Algorithm 23.

However, it is easy to test if there is still at least one machine number in the interval. If for $x = (a + b)/2$ the condition `(a<x) & (x<b)` holds, then there exists such a number, otherwise we must terminate the iteration. Thus, we obtain the following Matlab function (Algorithm 24):

Algorithm 24 Bisection

```
function [x,y]=Bisection(f,a,b,tol)
% BISECTION computes a root of a scalar equation
%   [x,y]=Bisection(f,a,b,tol) finds a root x of the scalar function
%   f in the interval [a,b] up to a tolerance tol. y is the
%   function value at the solution

fa=f(a); v=1; if fa>0, v=-1; end;
if fa*f(b)>0
  error('f(a) and f(b) have the same sign')
end
if (nargin<4), tol=0; end;
x=(a+b)/2;
while (b-a>tol) & ((a < x) & (x<b))
  if v*f(x)>0, b=x; else a=x; end;
x=(a+b)/2;
end
if nargout==2, y=f(x); end;
```

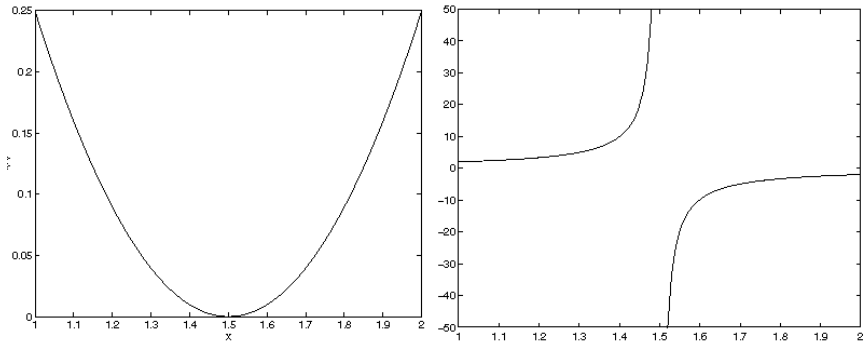


Figure 4.5: Cases where the Bisection Algorithm 24 fails

Algorithm 24 is an example of a "fool-proof" and machine-independent algorithm (see Section 1.8.1): for any continuous function whose values have opposite signs at the two end points, the algorithm will find a zero of this function. With $tol = 0$, it will compute a zero of f to machine precision. The algorithm makes use of finite precision arithmetic and would not work on a computer with exact arithmetic.

Example 4.2.1. *As a first example, we consider the function* $f(x) = x + e^x$,

```
>> [x,y]=Bisection(@(x) x+exp(x),-1,0)
x =
-0.567143290409784
y =
-1.110223024625157e-16
```

and we obtain the zero to machine precision. As a second example, we consider Kepler's Equation (4.2),

```
>> [E,f]=Bisection(@(E) E-0.8*sin(E)-2*pi/10,0,pi,1e-6)
E =
1.419135586110581
f =
-1.738227842773554e-07
```

where we asked for a precision of $1e-6$.

If the assumptions for bisection are not met, i.e., f has values with the same sign for a and b (see left figure in Figure 4.5) or f is not continuous in (a, b) (see right figure in Figure 4.5), then Algorithm 24 will fail.

4.2.2 Fixed Point Iteration

Consider the equation

$$f(x) = 0, \tag{4.5}$$

where $f(x)$ is a real function defined and continuous in the interval $[a, b]$. Assume that $s \in [a, b]$ is a zero of $f(x)$. In order to compute s, we transform (4.5) algebraically into

k	x_k	k	x_k	k	x_k
0	0.5000000000	10	0.5669072129	20	0.5671424776
1	0.6065306597	11	0.5672771960	21	0.5671437514
2	0.5452392119	12	0.5670673519	22	0.5671430290
3	0.5797030949	13	0.5671863601	23	0.5671434387
4	0.5600646279	14	0.5671188643	24	0.5671432063
5	0.5711721490	15	0.5671571437	25	0.5671433381
6	0.5648629470	16	0.5671354337	26	0.5671432634
7	0.5684380476	17	0.5671477463	27	0.5671433058
8	0.5664094527	18	0.5671407633	28	0.5671432817
9	0.5675596343	19	0.5671447237	29	0.5671432953

Table 4.1: Iteration $x_{k+1} = exp(-x_k)$

fixed point form,

$$x = F(x), \tag{4.6}$$

where F is chosen so that $F(x) = x \iff f(x) = 0$. A simple way to do this is, for example, $x = x + f(x) =: F(x)$, but other choices are possible. Finding a zero of $f(x)$ in $[a, b]$ is then equivalent to finding a fixed point $x = F(x)$ in $[a, b]$. The fixed point form suggests the *fixed point iteration*

$$x_0 \text{ initial guess}, \quad x_{k+1} = F(x_k),\ k = 0, 1, 2, \cdots . \tag{4.7}$$

The hope is that iteration (4.7) will produce a convergent sequence $x_k \to s$.

For example, consider

$$f(x) = xe^x - 1 = 0. \tag{4.8}$$

When trying to solve this equation in Maple, we find

```
solve(x*exp(x)-1,x);
```

$$\text{LambertW}(1)$$

the well-known *LambertW function*, which we will encounter again later in this chapter.

A first fixed point iteration for computing LambertW(1) is obtained by rearranging and dividing (4.8) by e^x,

$$x_{k+1} = e^{-x_k}. \tag{4.9}$$

With the initial guess $x_0 = 0.5$ we obtain the iterates shown in Table 4.1. Indeed x_k seems to converge to $s = 0.5671432...$

A second fixed point form is obtained from $xe^x = 1$ by adding x on both sides to get $x + xe^x = 1 + x$, factoring the left-hand side to get $x(1 + e^x) = 1 + x$, and dividing by $1 + e^x$, we obtain

$$x = F(x) = \frac{1 + x}{1 + e^x}. \tag{4.10}$$

k	x_k	k	x_k	k	x_k
0	0.5000000000	6	−0.6197642518	12	−0.1847958494
1	0.6756393646	7	0.7137130874	13	0.9688201302
2	0.3478126785	8	0.2566266491	14	−0.5584223793
3	0.8553214091	9	0.9249206769	15	0.7610571653
4	−0.1565059553	10	−0.4074224055	16	0.1319854380
5	0.9773264227	11	0.8636614202	17	0.9813779498

Table 4.2: Chaotic iteration with $x_{k+1} = x_k + 1 - x_k e^{x_k}$

This time the convergence is much faster — we need only three iterations to obtain a 10-digit approximation of s,

$$\begin{aligned} x_0 &= 0.5000000000, \\ x_1 &= 0.5663110032, \\ x_2 &= 0.5671431650, \\ x_3 &= 0.5671432904. \end{aligned}$$

Another possibility for a fixed point iteration is

$$x = x + 1 - xe^x. \tag{4.11}$$

This iteration function does not generate a convergent sequence. We observe here from Table 4.2 a chaotic behavior: no convergence but also no divergence to infinity.

Finally we could also consider the fixed point form

$$x = x + xe^x - 1. \tag{4.12}$$

With this iteration function the iteration diverges to minus infinity:

$$\begin{aligned} x_0 &= 0.5000000000 \\ x_1 &= 0.3243606353 \\ x_2 &= -0.2270012400 \\ x_3 &= -1.4079030215 \\ x_4 &= -2.7523543838 \\ x_5 &= -3.9278929674 \\ x_6 &= -5.0052139570 \end{aligned}$$

From these examples, we can see that there is an infinite number of possibilities in choosing an iteration function $F(x)$. Hence the question is, when does the iteration converge? The fixed point iteration has a very nice geometric interpretation: we plot $y = F(x)$ and $y = x$ in the same coordinate system (see Figure 4.6). The intersection points of the two functions are the solutions of $x = F(x)$. The computation of the sequence x_k with

$$\begin{aligned} x_0 &\quad \text{choose initial value} \\ x_{k+1} &= F(x_k), k = 0, 1, 2, \cdots \end{aligned}$$

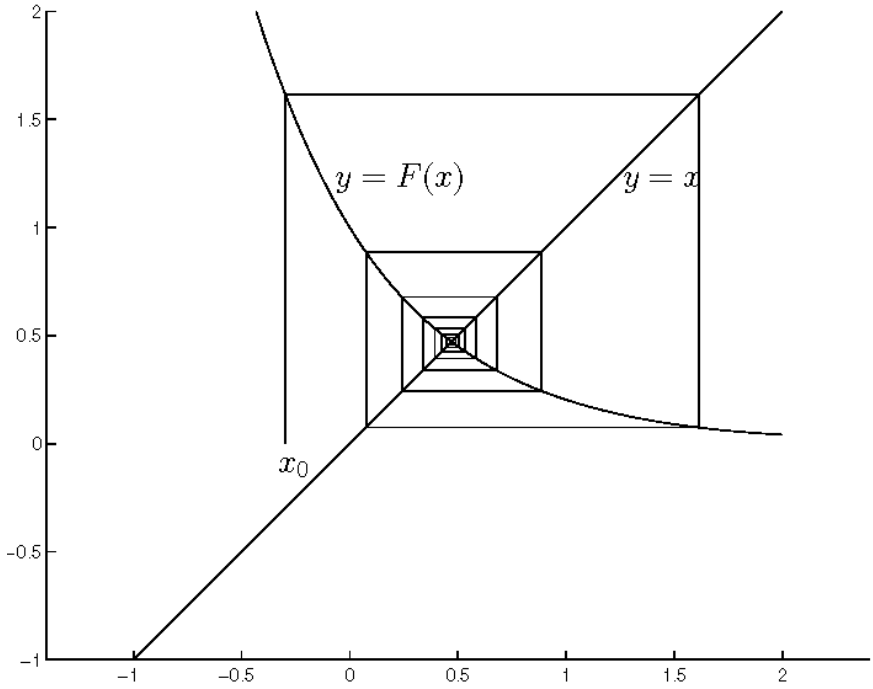


Figure 4.6: $x = F(x)$

can be interpreted geometrically via sequences of lines parallel to the coordinate axes:

x_0	start with x_0 on the $x-$axis
$F(x_0)$	go parallel to the $y-$axis to the graph of F
$x_1 = F(x_0)$	move parallel to the $x-$axis to the graph $y = x$
$F(x_1)$	go parallel to the $y-$axis to the graph of F
etc.	

One can distinguish four cases, two where $|F'(s)| < 1$ and the algorithm converges and two where $|F'(s)| > 1$ and the algorithm diverges. An example for each case is given in Figure 4.7. A more general statement for convergence is the theorem of Banach, Theorem 4.4.1, which is explained in Section 4.4.

4.2.3 Convergence Rates

In the previous section, we have seen geometrically that a fixed point iteration converges if $|F'(s)| < 1$. We have also observed that in case of convergence, some iterations (like Iteration (4.10)) converge much faster than others (like Iteration(4.9)). In this section, we would like to analyze the convergence speed. We observe already from Figure 4.7 that the smaller $|F'(s)|$, the faster the convergence.

Definition 4.2.1. ***(Iteration Error)*** *The error at iteration step k is defined by $e_k = x_k - s$.*

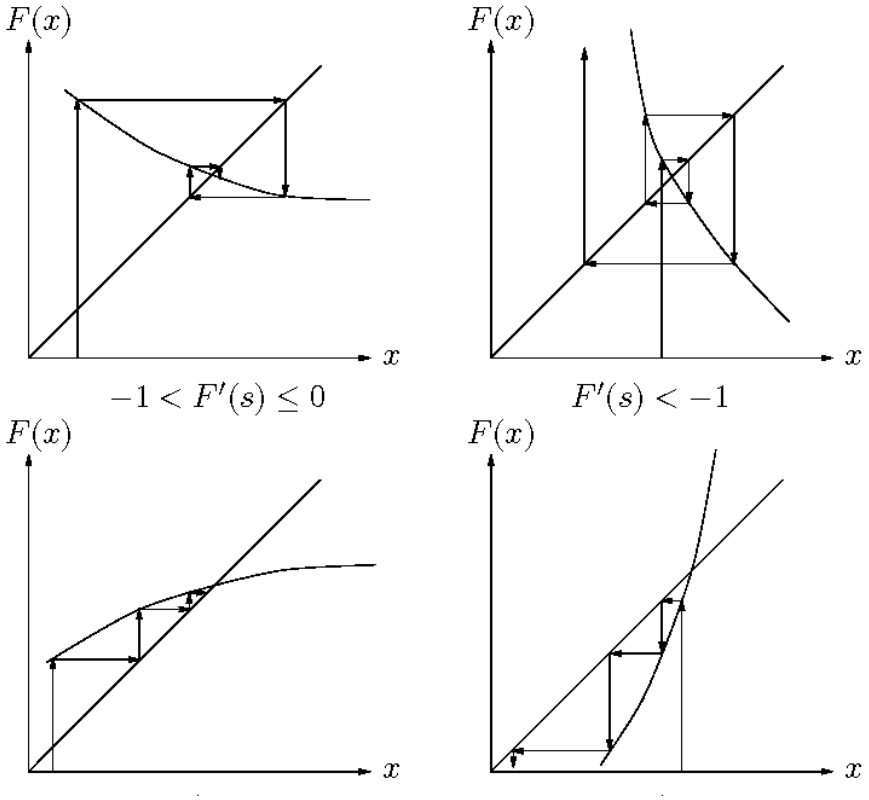


Figure 4.7: Four different scenarios for the fixed point iteration

Subtracting the equation $s = F(s)$ from $x_{k+1} = F(x_k)$ and expanding in a Taylor series, we get

$$x_{k+1} - s = F(x_k) - F(s) = F'(s)(x_k - s) + \frac{F''(s)}{2!}(x_k - s)^2 + \cdots ,$$

or expressed in terms of the error,

$$e_{k+1} = F'(s)e_k + \frac{F''(s)}{2!}e_k^2 + \frac{F'''(s)}{3!}e_k^3 + \cdots . \tag{4.13}$$

If $F'(s) \neq 0$, we conclude from (4.13), assuming that the error e_k goes to zero, that

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = F'(s).$$

This means that asymptotically $e_{k+1} \sim F'(s)e_k$. *Thus for large k the error is reduced in each iteration step by the factor* $|F'(s)|$. This is called *linear convergence* because the new error is a linear function of the previous one. Similarly we speak of *quadratic convergence* if $F'(s) = 0$ but $F''(s) \neq 0$, because $e_{k+1} \sim (F''(s)/2)e_k^2$. More generally we define:

Definition 4.2.2. ***(Convergence Rate)*** *The rate of convergence of* $x_{k+1} = F(x_k)$ *is*
linear*: if* $F'(s) \neq 0$ *and* $|F'(s)| < 1$,
quadratic*: if* $F'(s) = 0$ *and* $F''(s) \neq 0$,
cubic*: if* $F'(s) = 0$ *and* $F''(s) = 0$, *but* $F'''(s) \neq 0$,
of order *m: if* $F'(s) = F''(s) = \cdots = F^{(m-1)}(s) = 0$, *but* $F^{(m)}(s) \neq 0$.

Example 4.2.2.

1. *Consider Iteration (4.9):* $x_{k+1} = F(x_k) = e^{-x_k}$. *The fixed point is* $s = 0.5671432904$. $F'(s) = -F(s) = -s = -0.5671432904$. *Because* $0 < |F'(s)| < 1$ *we have linear convergence. With linear convergence the number of correct digits grows linearly. For* $|F'(s)| = 0.5671432904$ *the error is roughly halved in each step. So in order to obtain a new decimal digit one has to perform p iterations, where* $(0.5671432904)^p = 0.1$. *This gives us* $p = 4.01$. *Thus after about 4 iterations we obtain another decimal digit, as one can see by looking at Table 4.1.*

2. *If we want to solve Kepler's equation,* $E - e\sin E = \frac{2\pi}{T}t$ *for E, an obvious iteration function is*

$$E = F(E) = \frac{2\pi}{T}t + e\sin E.$$

Because $|F'(E)| = e|\cos E| < 1$, *the fixed point iteration always generates a linearly convergent sequence.*

3. *Iteration (4.10)* $x_{k+1} = F(x_k)$ *with* $F(x) = \frac{1+x}{1+e^x}$ *converges quadratically, because*

$$F'(x) = \frac{1 - xe^x}{(1+e^x)^2} = -\frac{f(x)}{(1+e^x)^2},$$

and since $f(s) = 0$ we have $F'(s) = 0$, and one can check that $F''(s) \neq 0$. With quadratic convergence, the number of correct digits doubles at each step asymptotically. If we have 3 correct digits at step k, $e_k = 10^{-3}$, then $e_{k+1} \approx e_k^2 = 10^{-6}$, and thus we have 6 correct digits in the next step $k + 1$.

The doubling of digits can be seen well when computing with Maple with extended precision (we separated the correct digits with a $*$):

```
> Digits:=59;
> x:=0.5;
> for i from 1 by 1 to 5 do x:=(1+x)/(1+exp(x)); od;
x:= .5
x:= .56*631100319721815304164915138173728187008095203663475541 08
x:= .567143*165034862212786512096659696366513431350818708556747 7
x:= .56714329040978*102869957664941534720617055786604397310562 79
x:= .567143290409783872999968662 2*088916713037266116513649733766
x:= .567143290409783872999968662210355549753815787186512508135 13*
```

4.3 Zeros of Polynomials

Zeros of polynomials used to be important topic when calculations had to be performed by hand. Whenever possible, one tried to transform a given equation to a polynomial equation. For example if we wish to compute a solution of the goniomatric equation

$$\tan\alpha = \sin\alpha - \cos\alpha,$$

then using the change of variables

$$t = \tan\frac{\alpha}{2}, \quad \sin\alpha = \frac{2t}{1+t^2}, \quad \cos\alpha = \frac{1-t^2}{1+t^2}, \quad \tan\alpha = \frac{2t}{1-t^2},$$

we obtain

$$\frac{2t}{1-t^2} = \frac{2t}{1+t^2} - \frac{1-t^2}{1+t^2},$$

or equivalently, the following algebraic equation of degree 4.

$$t^4 + 4t^3 - 2t^2 + 1 = 0,$$

Surely this equation was easier to solve than the equivalent one involving trigonometric functions in the old days before computers were available.

The algebraic eigenvalue problem $A\boldsymbol{x} = \lambda\boldsymbol{x}$ can be reduced to computing the zeros of a polynomial, since the eigenvalues are the roots of the the characteristic polynomial:

$$P_n(\lambda) = \det(A - \lambda I).$$

This is however not an advisable numerical strategy for computing eigenvalues, as we will see soon.

The Fundamental Theorem of Algebra states that every polynomial of degree $n \geq 1$,

$$P_n(x) = a_0 + a_1 x + \cdots + a_n x^n, \tag{4.14}$$

has at least one zero in C. If s_1 is such a zero then we can factor P_n using Horner's scheme discussed in the next section to obtain

$$P_n(x) = P_{n-1}(x)(x - s_1). \tag{4.15}$$

The remaining zeros of P_n are also zeros of P_{n-1}. By continuing with the polynomial P_{n-1} we have deflated the zero s_1 from P_n. If s_2 is another zero of P_{n-1} then again we can deflate it and we obtain:

$$P_{n-1}(x) = P_{n-2}(x)(x - s_2) \Rightarrow P_n(x) = P_{n-2}(x)(x - s_1)(x - s_2).$$

Continuing in this way, we reach finally

$$P_n(x) = P_0(x)(x - s_1)\cdots(x - s_n).$$

Since $P_0(x) = \text{const}$, by comparing with (4.14) wemust have that $P_0(x) = a_n$, and we have shown that P_n can be factored into n linear factors, and that a polynomial of degree n can have at most n zeros.

The following theorem estimates the region in the complex plane where the zeros of a polynomial can be.

Theorem 4.3.1. *All the zeros of the polynomial $P_n(x) = a_0 + a_1x + \cdots + a_nx^n$ with $a_n \neq 0$ lie in the disk around the origin in the complex plane with radius*

$$r = 2 \max_{1\leq k\leq n} \sqrt[k]{\left|\frac{a_{n-k}}{a_n}\right|}$$

Proof. We want to show that if $|z| > r$, then $|P_n(z)| > 0$. Let $|z| > 2\sqrt[k]{\left|\frac{a_{n-k}}{a_n}\right|}$ for all k. Then

$$2^{-k}|z|^k > \sqrt[k]{\left|\frac{a_{n-k}}{a_n}\right|} \iff 2^{-k}|a_n||z|^n > |a_{n-k}|z^{n-k}|. \tag{4.16}$$

Because of the triangle inequality $||x| - |y|| \leq |x + y| \leq |x| + |y|$, we conclude that

$$|P_n(z)| \geq |a_nz^n| - \sum_{k=1}^{n} |a_{n-k}||z^{n-k}|,$$

and because of Equation (4.16) the right-hand side becomes

$$\geq |a_n||z^n| - \sum_{k=1}^{n} 2^{-k}|a_n||z^n| = |a_n||z^n|(1 - \sum_{k=1}^{n} 2^{-k}) = |a_n||z^n|(\frac{1}{2})^n > 0. \qquad \square$$

In the work of Ruffini, Galois and Abel it was proved that, in general, explicit formulas for the zeros only exist for polynomials of degree $n \leq 4$. Maple knows these explicit formulas: for $n = 3$, they are known as the Cardano formulas, and the formulas for $n = 4$ were discovered by Ferrari. To solve a general polynomial equation of degree 4 algebraically, $x^4 + bx^3 + cx^2 + dx + e = 0$, the following commands are needed:

```
> solve(x^4+b*x^3+c*x^2+d*x+e=0,x);
> allvalues(%);
```

The resulting expressions are several pages long and maybe not very useful.

4.3.1 Condition of the Zeros

Zeros of polynomials became less popular when Jim Wilkinson discovered that they are often very ill conditioned. As part of a test suite for a new computer, Wilkinson constructed from the following polynomial of degree 20 (Wilkinson's polynomial) with roots $x_i = 1, 2, \cdots, 20$ by expanding the product:

$$P_{20}(x) = \prod_{i=1}^{20}(x - i) = x^{20} - 210x^{19} + 20615x^{18} - \cdots + 20!.. \tag{4.17}$$

Then he used a numerical method to compute the zeros and he was astonished to observe that his program found some complex zeros, rather than reproducing the zeros $x_i = 1, 2, \cdots, 20$. After having checked very carefully that there was no programming error, and that the hardware was also working correctly, he then tried to understand the results by a backward error analysis. Let z be a simple zero of

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0.$$

Let ε be a small parameter and let g be another polynomial of degree n. Consider the zero $z(\varepsilon)$ (corresponding to z, i.e.,$z(0) = z$) of the perturbed polynomial h,

$$h(x) := P_n(x) + \varepsilon g(x).$$

We expand $z(\varepsilon)$ in a series

$$z(\varepsilon) = z + \sum_{k=1}^{\infty} p_k \varepsilon^k.$$

The coefficient $p_1 = z'(0)$ can be computed by differentiating

$$P_n(z(\varepsilon)) + \varepsilon g(z(\varepsilon)) = 0$$

with respect to ε. We obtain

$$P_n'(z(\varepsilon))z'(\varepsilon) + g(z(\varepsilon)) + \varepsilon g'(z(\varepsilon))z'(\varepsilon) = 0,$$

and therefore

$$z'(\varepsilon) = -\frac{g(z(\varepsilon))}{P_n'(z(\varepsilon)) + \varepsilon g'(z(\varepsilon))}.$$

Thus, for $\varepsilon = 0$,

$$z'(0) = -\frac{g(z)}{P_n'(z)}. \tag{4.18}$$

We now apply (4.18) to Wilkinson's polynomial (4.17). Wilkinson perturbed only the coefficient $a_{19} = -210$ by 2^{-23} (which was the machine precision for single precision on some early computers). This modification corresponds to the choice of $g(x) = x^{19}$ and $\varepsilon = -2^{-23}$, and we obtain for the zero $z_r = r$ the perturbation

$$\delta z_r \approx 2^{-23}\frac{r^{19}}{|P_{20}'(r)|} = 2^{-23}\frac{r^{19}}{(r-1)!(20-r)!}.$$

For $r = 16$, the perturbation is maximal and becomes

$$\delta z_{16} \approx 2^{-23} \frac{16^{19}}{15!4!} \approx 287.$$

Thus, the zeros are ill conditioned. Wilkinson computed the exact zeros using multiple precision and found e.g. the zeros $16.730 \pm 2.812\ i$. We can easily reconfirm this calculation using Maple:

```
> Digits:=50;
> p:=1:
> for i from 1 by 1 to 20 do
> p:=p*(x-i)
> od:
> Z:=fsolve(p-2^(-23)*x^19,x,complex,maxsols=20);
> plot(map(z->[Re(z),Im(z)],[Z]),x=0..22,style=point,symbol=circle);
```

We do not actually need to work with 50 decimal digits; we obtain the same results even in Matlab using standard IEEE arithmetic. We can also use Maple to simulate the experiment of Wilkinson, using 7 decimal-digit arithmetic. For this simulation, it is important to represent the coefficients of the expanded polynomial as 7-digit numbers:

```
> Digits:=7;
> p:=1:
> for i from 1 by 1 to 20 do
> p:=p*(x-i)
> od:
> PP:=expand(p);
> PPP:=evalf(PP);
> Z:=fsolve(PPP,x,complex,maxsols=20);
> plot(map(z->[Re(z),Im(z)],[Z]),x=0..28,style=point,symbol=circle);
```

Figure 4.8 shows how most of the zeros become complex numbers.

In Matlab, polynomials are represented by

$$P_n(x) = A(1)x^n + A(2)x^{n-1} + \cdots + A(n)x + A(n+1),$$

while in usual mathematical notation one prefers

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

In order to switch between the two representations, we must keep in mind the transformation

$$A(i) = a_{n+1-i}, i = 1, \cdots, n+1 \iff a_j = A(n+1-j), j = 0, \cdots, n.$$

In the following Matlab script, we change the second coefficient of Wilkinson's polynomial by subtracting a small perturbation: $p_2 := p_2 - \lambda p_2$ where $\lambda = 0 : 1e-11 : 1e-8$. Note that the perturbations are cumulative; at the last iteration, the total perturbation gives $\tilde{p}_2 = p_2(1-\mu)$, with $\mu \approx 5.0 \times 10^{-6}$.

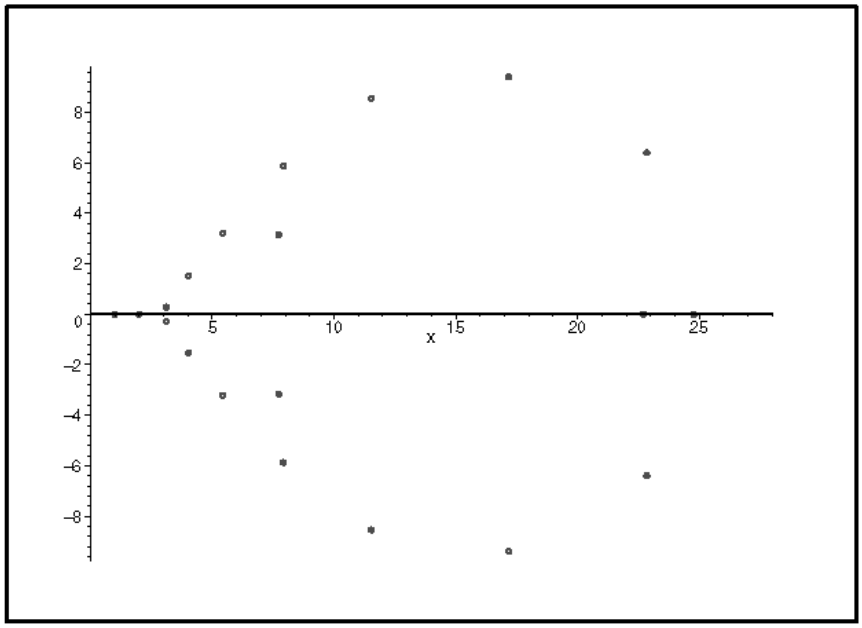


Figure 4.8: Zeros of Wilkinson's Polynomial computed with 7 digits.

Algorithm 25 Experiment with Wilkinson's Polynomial

```
axis([-5 25 -10 10])
hold
P=poly(1:20)
for lamb=0:1e-11:1e-8
  P(2)=P(2)*(1-lamb);
  Z=roots(P);
  plot(real(Z),imag(Z),'o')
end
```

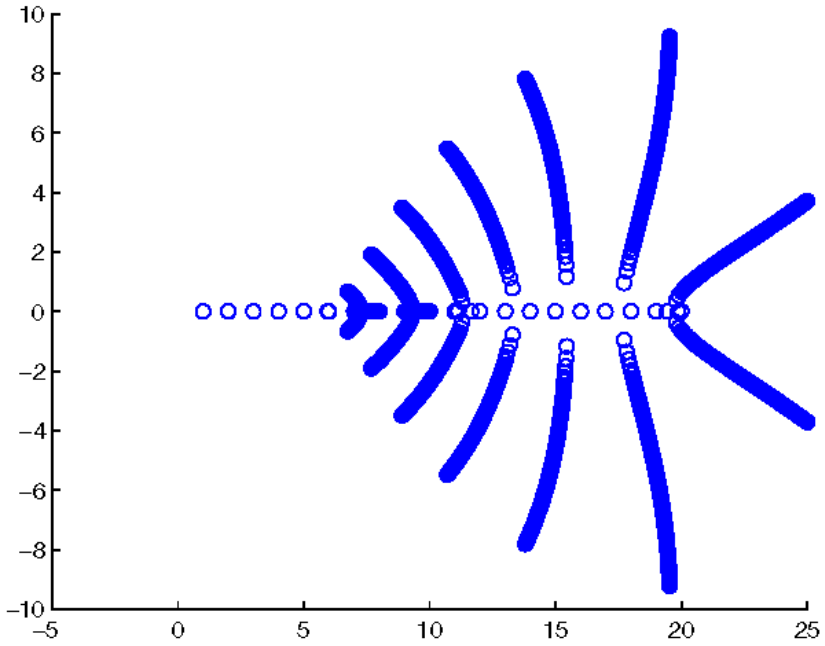


Figure 4.9: Roots of the perturbed Wilkinson polynomial

By computing the roots and plotting them in the complex plane, we can observe in Figure 4.9 how the larger roots are rapidly moving away from the real line.

We finally remark that multiple roots are always ill conditioned. To illustrate that, assume that we change the constant coefficient of $P_n(x) = (x-1)^n$ by the machine precision ε. The perturbed polynomial becomes $(x-1)^n - \varepsilon$ and its roots are solutions of

$$(x-1)^n = \varepsilon \Rightarrow x(\varepsilon) = 1 + \sqrt[n]{\varepsilon}.$$

The new roots are all simple and lie on the circle of radius $\sqrt[n]{\varepsilon}$ with center 1. For $\varepsilon = 2.2204e-16$ and $n = 10$ we get $\sqrt[10]{\varepsilon} = 0.0272$ which shows that the multiple roots "explode" quite dramatically into n simple ones.

4.3.2 Companion Matrix

As we have seen in the previous section, zeros of polynomials may be ill conditioned, so one had to find new methods for computing eigenvalues. New algorithms that work directly on the matrix, instead of forming the characteristic polynomial, have been successfully developed.

Definition 4.3.2. ***(Companion Matrix)*** *The monic polynomial* $P_n(x) = x_n + a_{n-1}x_{n-1} +$

$\cdots + a_0$ *has the companion matrix*

$$A = \begin{bmatrix} -a_{n-1} & -a_{n-2} & \dots & -a_1 & -a_0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \ddots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 & 0 \end{bmatrix}$$

By expanding the determinant $\det(\lambda I - A)$, we see that $P_n(\lambda)$ is the characteristic polynomial of A.

Example 4.3.1. *Consider for $n = 4$ the matrix $C = \lambda I - A$ and compute its determinant by the expanding the first column. We get*

$$\begin{aligned} \det \begin{pmatrix} \lambda + a_3 & a_2 & a_1 & a_0 \\ -1 & \lambda & 0 & 0 \\ 0 & -1 & \lambda & 0 \\ 0 & 0 & -1 & \lambda \end{pmatrix} &= (\lambda + a_3) \det \begin{pmatrix} \lambda & 0 & 0 \\ 1 & \lambda & 0 \\ 0 & -1 & \lambda \end{pmatrix} + 1 \cdot \det \begin{pmatrix} a_2 & a_1 & a_0 \\ -1 & \lambda & 0 \\ 0 & -1 & \lambda \end{pmatrix} \\ &= \lambda^4 + a_3\lambda^3 + a_2\lambda^2 + 1 \cdot \det \begin{pmatrix} a_1 & a_0 \\ -1 & \lambda \end{pmatrix} \\ &= \lambda^4 + a_3\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 \end{aligned}$$

The Matlab function `compan(a)` computes this companion matrix. In Maple we get with

```
> p:=z^5+2*z^4+3*z^3+4*z^2+5*z+6;
> with(LinearAlgebra);
> A:=CompanionMatrix(p);
```

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & -6 \\ 1 & 0 & 0 & 0 & -5 \\ 0 & 1 & 0 & 0 & -4 \\ 0 & 0 & 1 & 0 & -3 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix}$$

a different definition of the companion matrix, which is often used in textbooks. Both definitions are equivalent, since by computing $\det(\lambda I - A)$ we obtain the characteristic polynomial of A.

4.3.3 Newton Method Using Taylor Expansions

One of implementation of Newton's method is based on expanding the polynomial at new points in the complex plane. Let x be an approximation of a zero, then

$$P_n(x + h) = a_0 + a_1 h + \cdots + a_n h^n.$$

Because $a_0 = P_n(x)$ and $a_1 = P_n'(x)$, the Newton step simply becomes

$$x := x - a_0/a_1.$$

We compute a new Taylor expansion of the polynomial at the new approximation and the effect is that the new coefficient a_0 decreases with each iteration. If finally x is a root, then $a_0 = 0$, and

$$P_n(x+h) = h(a_1 + a_2 h + \cdots + a_n h^{n-1}),$$

and thus the remaining roots are zeros of

$$P_{n-1}(x+h) = a_1 + a_2 h + \cdots + a_n h^{n-1}.$$

Therefore, deflation becomes here very simple.

Algorithm 26 Newton for Zeros of Polynomials

```
function z=NewtonTaylorRoots(a)
% NEWTONTAYLORROOTS computes the zeros of a polynomial
% z=NewtonTaylorRoots(a) computes the zeros of the polynomial given
% in the vector a using Newton's method by re-expanding the
% polynomial

n=length(a); degree=n-1;
z=[]; x=0; h=1+sqrt(-1);
for m=degree:-1:1
  while abs(a(m+1))>norm(a)*eps                    % Newton iteration
    x=x+h; a=Taylor(a,h); h=-a(m+1)/a(m);
  end
a=a(1:m);                                         % deflation
z=[z; x];
end
```

4.4 Nonlinear Systems of Equations

Solving n nonlinear equations in n variables means that, for a given continuous function $\boldsymbol{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, we want to find a value $\boldsymbol{x} \in \Omega \in \mathbb{R}^n$ such that $\boldsymbol{f}(\boldsymbol{x}) = 0$. We use the notation $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^\top$ and $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x}))^\top$.

In one dimension, the measure of distance between two numbers is the absolute value of the difference between those numbers. In several dimensions we have to use norms as measures of distance (see Chapter 1, Section 1.5.1). We also need the generalization of Taylor expansions from the one-dimensional case to multivariate, *vector-valued functions* $\boldsymbol{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. Recall that for the function $g : \mathbb{R} \longrightarrow \mathbb{R}$, a second-order Taylor expansion (with a third-order remainder term) is given by

$$g(t) = g(0) + g'(0)t + \frac{1}{2!}g''(0)t^2 + \frac{1}{3!}g'''(\tau)t^3, \tag{4.19}$$

where $0 \leq \tau \leq t$. In order to obtain the Taylor expansion for $\boldsymbol{f}$, we consider a perturbation in the direction $\boldsymbol{h} = (h_1, ..., h_n)^\top$ and consider each component f_i separately, i.e., we consider the scalar function

$$g(t) := f_i(\boldsymbol{x} + t\boldsymbol{h}).$$

In order to expand this scalar function using (4.19), we need the derivatives of g,

$$\begin{aligned} g'(0) &= \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}(\boldsymbol{x}) h_j \\ g''(0) &= \sum_{j=1}^{n}\sum_{k=1}^{n} \frac{\partial^2 f_i}{\partial x_j \partial x_k}(\boldsymbol{x}) h_j h_k \\ g'''(\tau) &= \sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} \frac{\partial^3 f_i}{\partial x_j \partial x_k \partial x_l}(\boldsymbol{x} + \tau\boldsymbol{h}) h_j h_k h_l. \end{aligned}$$

Introducing these derivatives in the Taylor expansion (4.19) of g and evaluating at $t = 1$, we naturally arrive at the Taylor expansion of each component f_i,

$$\begin{aligned} f_i(\boldsymbol{x} + \boldsymbol{h}) &= f_i(\boldsymbol{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}(\boldsymbol{x}) h_j + \frac{1}{2!}\sum_{j=1}^{n}\sum_{k=1}^{n} \frac{\partial^2 f_i}{\partial x_j \partial x_k}(\boldsymbol{x}) h_j h_k \\ &\quad + \frac{1}{3!}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} \frac{\partial^3 f_i}{\partial x_j \partial x_k \partial x_l}(\boldsymbol{x} + \tau_i\boldsymbol{h}) h_j h_k h_l. \end{aligned} \tag{4.20}$$

Since this explicit notation with sums is quite cumbersome, one often uses the notation of multilinear forms, and writes simultaneously for all components of $\boldsymbol{f}$

$$\boldsymbol{f(x + h) = f(x) + f'(x)(h) + \frac{1}{2!} f''(x)(h, h) + R(h)}, \tag{4.21}$$

where the remainder term can be estimated by $\boldsymbol{R(h)} = O(\|\boldsymbol{h}\|^3)$. Note that the first-order term can be written as a matrix-vector multiplication

$$\boldsymbol{f'}(\boldsymbol{x})(\boldsymbol{h}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \dots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \cdot \boldsymbol{h}.$$

The $m \times n$ matrix is often called the Jacobian matrix of $\boldsymbol{f}$, which we will denote by $J(\boldsymbol{x})$. In the special case of a scalar function with many arguments, $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, which occurs often in optimization, we obtain

$$f(\boldsymbol{x + h}) = f(\boldsymbol{x}) + f'(\boldsymbol{x})(\boldsymbol{h}) + \frac{1}{2!} f''(\boldsymbol{x})(\boldsymbol{h}, \boldsymbol{h}) + O(\|\boldsymbol{h}\|^3), \tag{4.22}$$

and now in the linear term, we see the transpose of the gradient appearing, $f'(\boldsymbol{x}) = \nabla f(\boldsymbol{x})^\top$. In the quadratic term, the bilinear form $f''(\boldsymbol{x})(\boldsymbol{h},\boldsymbol{h})$ can also be written in matrix notation as $f''(\boldsymbol{x})(\boldsymbol{h},\boldsymbol{h}) = \boldsymbol{h}^\top H(\boldsymbol{x})\boldsymbol{h}$, where $H(\boldsymbol{x})$ is a symmetric matrix with entries

$$(H(\boldsymbol{x}))_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}(\boldsymbol{x}).$$

This matrix is known as the *Hessian matrix* of f. Thus, the Taylor expansion for the function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ can be written in matrix form as

$$f(\boldsymbol{x}+\boldsymbol{h}) = f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^\top \boldsymbol{h} + \frac{1}{2!}\boldsymbol{h}^\top H(\boldsymbol{x})\boldsymbol{h} + O(\|\boldsymbol{h}\|^3). \tag{4.23}$$

4.4.1 Fixed Point Iteration

A fixed point iteration in several dimensions is very similar to one dimension. To find a zero of the function $\boldsymbol{f(x)}$ for $\boldsymbol{x} \in \Omega$, we need to define a continuous function $\boldsymbol{F(x)}$ on Ω such that

$$\boldsymbol{f(x)} = 0 \quad \Longleftrightarrow \quad \boldsymbol{x = F(x)}. \tag{4.24}$$

For example, one could define

$$\boldsymbol{F(x)} := \boldsymbol{x - f(x)},$$

but again there is an infinite number of possibilities for choosing a function $\boldsymbol{F(x)}$ which satisfies (4.24). To find a zero of $\boldsymbol{f(x)}$ in Ω is then equivalent to finding a fixed point of $\boldsymbol{F(x)}$ in Ω. For that purpose, the following fixed point iteration is used:

```
x1=initial guess;
x2=F(x1);
while norm(x2-x1)>tol*norm(x2)
  x1=x2;
  x2=F(x1);
end
```

Note that one can use any of the norms introduced in Section 1.5.1. The question of when the above iteration converges can be answered by Theorem 4.4.1 in the next section: $\boldsymbol{F(x)}$ has to be a contraction.

4.4.2 Theorem of Banach

We need some definitions to state the theorem. A *Banach space* $\mathcal{B}$ is a *complete normed vector space* over some number field $\mathcal{K}$ such as $\mathbb{R}$ or $\mathbb{C}$. "Normed" means that there exists a norm $\|\ \|$ with the following properties:

1. $\|\boldsymbol{x}\| \geq 0$, $\forall \boldsymbol{x} \in \mathcal{B}$, and $\|\boldsymbol{x}\| = 0 \ \Leftrightarrow \ \boldsymbol{x} = 0$
2. $\|\gamma\boldsymbol{x}\| = |\gamma|\|\boldsymbol{x}\|, \forall \gamma \in \mathcal{K}$ and $\boldsymbol{x} \in \mathcal{B}$
3. $\|\boldsymbol{x}+\boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|, \forall \boldsymbol{x},\boldsymbol{y} \in \mathcal{B}$ (*triangular inequality*)

"Complete" means that every Cauchy sequence converges in $\mathcal{B}$.

Let $A \subset \mathcal{B}$ be a closed subset and F a mapping $F : A \to A$. F is called Lipschitz continuous on A if there exists a constant $L < \infty$ such that $\|F(\boldsymbol{x}) - F(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|$ $\forall \boldsymbol{x}, \boldsymbol{y} \in A$. Furthermore, F is called a contraction if L can be chosen such that $L < 1$.

Theorem 4.4.1 (Banach Fixed Point Theorem)**.** *Let A be a closed subset of a Banach space $\mathcal{B}$, and let F be a contraction $F : A \to A$. Then:*

a) The contraction F has a unique fixed point $\boldsymbol{s}$, which is the unique solution of the equation $\boldsymbol{x} = F(\boldsymbol{x})$.

b) The sequence $\boldsymbol{x}_{k+1} = F(\boldsymbol{x}_k)$ converges to $\boldsymbol{s}$ for every initial guess $\boldsymbol{x}_0 \in A$.

c) We have the a posteriori estimate :

$$\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq \frac{L^{k-l}}{1-L}\|\boldsymbol{x}_{l+1} - \boldsymbol{x}_l\|, \qquad for\ 0 \leq l < k. \tag{4.25}$$

Proof. Because F is Lipschitz continuous, we have $\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\| = \|F(\boldsymbol{x}_k) - F(\boldsymbol{x}_{k-1})\| \leq L\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\|$ and therefore

$$\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\| \leq L^{k-l}\|\boldsymbol{x}_{l+1} - \boldsymbol{x}_l\|, \qquad for\ 0 \leq l \leq k. \tag{4.26}$$

We claim that $\{\boldsymbol{x}_k\}$ is a Cauchy sequence. We need to show that for a given $\varepsilon > 0$, there exists a number K such that $\forall m > 1$ and $\forall k \geq K$ the difference $\|\boldsymbol{x}_{k+m} - \boldsymbol{x}_k\| < \varepsilon$. This is the case since, using the triangle inequality, we obtain

$$\begin{aligned} \|\boldsymbol{x}_{k+m} - \boldsymbol{x}_k\| &= \|\boldsymbol{x}_{k+m} - \boldsymbol{x}_{k+m-1} + \boldsymbol{x}_{k+m-1} - \boldsymbol{x}_{k+m-2} \pm \cdots - \boldsymbol{x}_k\| \\ &\leq \sum_{i=k}^{k+m-1} \|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i\| \\ &\leq (L^{m-1} + L^{m-2} + \cdots + L + 1)\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\|. \end{aligned}$$

Thus we have

$$\|\boldsymbol{x}_{k+m} - \boldsymbol{x}_k\| \leq \frac{1-L^m}{1-L}\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\| \tag{4.27}$$

Using (4.26), we get

$$\|\boldsymbol{x}_{k+m} - \boldsymbol{x}_k\| \leq \frac{1-L^m}{1-L}L^k\|\boldsymbol{x}_1 - \boldsymbol{x}_0\| \tag{4.28}$$

and since $L < 1$, we can choose k so that the right hand side of (4.28) is smaller than ε. We have proved that $\{\boldsymbol{x}_k\}$ is a Cauchy sequence and hence converges to some $\boldsymbol{s} \in \mathcal{B}$; since A is a closed subset of $\mathcal{B}$, we deduce that $\boldsymbol{s} \in A$.

We now show by contradiction that $\boldsymbol{s}$ is unique. If we had two fixed points $\boldsymbol{s}_1 = F(\boldsymbol{s}_1)$ and $\boldsymbol{s}_2 = F(\boldsymbol{s}_2)$ then

$$\|\boldsymbol{s}_1 - \boldsymbol{s}_2\| = \|F(\boldsymbol{s}_1) - F(\boldsymbol{s}_2)\| \leq L\|\boldsymbol{s}_1 - \boldsymbol{s}_2\|,$$

thus $1 \leq L$. But since we assumed $L < 1$ this is a contradiction and there cannot be two fixed points. Thus we have proved a) and b).

In order to prove c), we use (4.27) and then (4.26) to obtain

$$\|\boldsymbol{x}_{k+m} - \boldsymbol{x}_k\| \leq \frac{1-L^m}{1-L} L^{k-l} \|\boldsymbol{x}_{l+1} - \boldsymbol{x}_l\|,$$

which holds for $0 \leq l < k$ and for all m. If we let $m \leftarrow \infty$ we finally obtain

$$\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq \frac{1}{1-L} L^{k-l} \|\boldsymbol{x}_{l+1} - \boldsymbol{x}_l\|.$$

□

Consequences and remarks:

1. If we set $l = 0$ in Equation (4.25) we obtain an *a priori error estimate*:

$$\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq \frac{L^k}{1-L} \|\boldsymbol{x}_1 - \boldsymbol{x}_0\|.$$

   Using this error estimate and knowing L, we can predict how many iteration steps are necessary to obtain an error $\|\boldsymbol{s} - \boldsymbol{x}_k\| < \varepsilon$, namely

$$k > \frac{\ln(\frac{\varepsilon(1-L)}{\|\boldsymbol{x}_1 - \boldsymbol{x}_0\|})}{\ln L}. \tag{4.29}$$

2. We can also obtain the convenient *a posteriori error estimate* by substituting $l = k - 1$ in Equation (4.25),

$$\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq \frac{L}{1-L} \|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\|.$$

   If $\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\| < \varepsilon$, then

$$\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq \frac{L}{1-L} \varepsilon. \tag{4.30}$$

   Note that the bound on the right hand side of Equation (4.30) can be much larger than ε. In fact if e.g. $L = 0.9999$, then $\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq 0.9999\varepsilon$. Thus even if successive iterates agree to 6 decimal digits, the approximation may contain only two correct decimal digits.

   Only when $L \leq 0.5$ can we conclude from $\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\| < \varepsilon$ that $\|\boldsymbol{s} - \boldsymbol{x}_k\| \leq \varepsilon$. In this case we may then terminate the iteration by checking successive iterates and conclude that the error is also smaller than ε.

   A way to estimate L from the iteration at step $k+1$ is to use (4.26) for $k = l+1$, which gives

$$L \approx \frac{\|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\|}{\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\|}.$$

   This, together with the a posteriori estimate, leads to the stopping criterion

$$\frac{\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\| \|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\|}{\|\boldsymbol{x}_k - \boldsymbol{x}_{k-1}\| - \|\boldsymbol{x}_{k+1} - \boldsymbol{x}_k\|} < \texttt{tol} \tag{4.31}$$

   which guarantees asymptotically that the error is less than `tol`.

4.4.3 Newton's Method

We want to find $\boldsymbol{x}$ such that $\boldsymbol{f}(\boldsymbol{x}) = 0$. Expanding $\boldsymbol{f}$ at some approximation $\boldsymbol{x}_k$, we obtain

$$\boldsymbol{f}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x}_k) + J(\boldsymbol{x}_k)\boldsymbol{h}, \qquad \text{with} \quad \boldsymbol{h} = \boldsymbol{x} - \boldsymbol{x}_k,$$

where $J(\boldsymbol{x}_k)$ denotes the Jacobian evaluated at $\boldsymbol{x}_k$,

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \dots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}.$$

Instead of solving $\boldsymbol{f}(\boldsymbol{x}) = 0$, we solve the linearized system

$$\boldsymbol{f}(\boldsymbol{x}_k) + J(\boldsymbol{x}_k)\boldsymbol{h} = 0$$

for the *Newton correction* $\boldsymbol{h}$, and obtain a new, hopefully better approximation

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{h} = \boldsymbol{x}_k - J(\boldsymbol{x}_k)^{-1}\boldsymbol{f}(\boldsymbol{x}_k). \tag{4.32}$$

Thus, Newton's method is the fixed point iteration

$$\boldsymbol{x}_{k+1} = \boldsymbol{F}(\boldsymbol{x}_k) = \boldsymbol{x}_k - J(\boldsymbol{x}_k)^{-1}\boldsymbol{f}(\boldsymbol{x}_k).$$

It is important to note that the occurrence of the inverse of the Jacobian in the formula implies that one has to solve a linear system at each step of the Newton iteration.

Theorem 4.4.2. *[Quadratic Convergence of Newton's Method] Suppose that $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ is three times continuously differentiable, and that Jacobian $f'(x)$ is invertible in a neighborhood of $\boldsymbol{s}$, where $f(\boldsymbol{s}) = 0$. Then, for $\boldsymbol{x}_k$ sufficiently close to $\boldsymbol{s}$, the error $\boldsymbol{e}_k := \boldsymbol{x}_k - \boldsymbol{s}$ in Newton's method satisfies*

$$\boldsymbol{e}_{k+1} = \frac{1}{2}(f'(\boldsymbol{x}_k))^{-1} f''(\boldsymbol{x}_k)(\boldsymbol{e}_k, \boldsymbol{e}_k) + O(\|\boldsymbol{e}_k\|^3). \tag{4.33}$$

Hence Newton's method converges locally quadratically.

Proof. We expand $f(\boldsymbol{x})$ in a Taylor series, see (4.21), and obtain

$$f(\boldsymbol{x}) = f(\boldsymbol{x}_k) + f'(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2} f''(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k, \boldsymbol{x} - \boldsymbol{x}_k) + O(\|\boldsymbol{x} - \boldsymbol{x}_k\|^3).$$

Setting $\boldsymbol{x} := \boldsymbol{s}$, where $f(\boldsymbol{s}) = 0$, and subtracting the Newton iteration formula

$$0 = f(\boldsymbol{x}_k) + f'(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k),$$

we obtain for the error $\boldsymbol{e}_k := \boldsymbol{x}_k \boldsymbol{s}$ the relation

$$0 = f'(\boldsymbol{x}_k)(-\boldsymbol{e}_{k+1}) + \frac{1}{2} f''(\boldsymbol{x}_k)(\boldsymbol{e}_k, \boldsymbol{e}_k) + O(\|\boldsymbol{e}_k\|^3),$$

which concludes the proof. □

A short Matlab implementation of Newton's method for an arbitrary system of nonlinear equations is given in Algorithm 27.

Algorithm 27 Newton's method for a system of non-linear equations

```
function x=Newton(f,x0,tol,maxiter,fp);
% NEWTON solves a nonlinear system of equations
% x=Newton(f,x0,tol,maxiter,fp); solves the nonlinear system of
% equations f(x)=0 using Newtons methods, starting with the initial
% guess x0 up to a tolerance tol, doing at most maxit iterations. An
% analytical Jacobian can be given in the parameter fp

numJ=nargin<5;
if nargin<4, maxit=100; end;
if nargin<3, tol=1e6; end;

x=x0; i=0; dx=ones(size(x0));
while norm(f(x))>tol & norm(dx)>tol & i<maxiter
  if numJ, J=NumericalJacobian(f,x); else J=fp(x); end;
    dx=-J\f(x); x=x+dx; i=i+1;
  end
  if i>=maxiter
     error('Newton did not converge: maximum number of iterations exceeded');
end;
```

Note that in this implementation, one has the choice of giving the Jacobian matrix in analytic form, or letting the procedure compute an approximate *Jacobian matrix numerically*. For a numerical approximation, one usually uses a finite difference, i.e.

$$\frac{\partial f_i}{\partial x_j}(x_1,\ldots,x_n) \approx \frac{f_i(x_1,\ldots,x_j+h,\ldots,x_n)-f_i(x_1,\ldots,x_n)}{h}, \tag{4.34}$$

for some discretization parameter h. A Matlab implementation of this approach is given in Algorithm 28.

Algorithm 28 Finite difference approximation for the Jacobian

```
function J=NumericalJacobian(f,x);
% NUMERICALJACOBIAN computes a Jacobian numerically
%   J=NumericalJacobian(f,x); computes numerically a Jacobian matrix
%   for the function f at x

for i=1:length(x)
  xd=x; h=sqrt(eps*(1+abs(xd(i)))); xd(i)=xd(i)+h;
  J(:,i)=(f(xd)-f(x))/h;
end;
```

In order to determine a numerically sensible choice for the discretization parameter h, one has to consider two approximation errors in the expression: first, the smaller one chooses h, the more accurate the finite difference approximation will be, since in the limit, it converges mathematically to the derivative. Numerically, however, there is a difference that needs to be computed in the numerator, and we have seen that differences of quantities that are very close in size suffer from cancellation, so h should not be too small. In order to get more insight, assume that f is a scalar function. Expanding using a Taylor series gives

$$\frac{f(x+h)-f(x)}{h} = f'(x) + \frac{h}{2}f''(x) + O(h^2), \tag{4.35}$$

which shows clearly that h needs to be small in order to get a good approximation of the derivative $f'(x)$. On the other hand, let us study the roundoff error when computing the approximation,

$$\begin{aligned}&\frac{f((x+h)(1+\varepsilon_1))(1+\varepsilon_2) - f(x)(1+\varepsilon_3)}{h}\\ &\approx \frac{f(x+h)-f(x)}{h} + \frac{1}{h}(f'(x+h)(x+h)\varepsilon_1 + f(x+h)\varepsilon_2 - f(x)\varepsilon_3)\end{aligned}$$

where $|\varepsilon_i| \leq eps$, the machine precision. A good choice is to balance the two sources of error, i.e.

$$\frac{h}{2}(\ldots) \approx \frac{1}{h}(\ldots)eps,$$

which indicates that a good choice for h is

$$h = \sqrt{eps}, \qquad \text{or} \qquad h = \sqrt{eps(1+|x|)}.$$

Example 4.4.1. *We would like to compute the intersection points of the circle of radius $r = 2$ centered at the origin with the ellipse with center $M = (3, 1)$ and semi-axes (parallel to the coordinate axes) a and $b = 2$.*

We will solve this problem using the parametric representation of the circle and the ellipse. The circle is given by $(2\cos t, 2\sin t)$, $0 \leq t < 2\pi$. A point on the ellipse has the coordinates $(3 + a\cos s, 1 + 2\sin s)$, $0 \leq s < 2\pi$.

To find an intersection point we must solve the following nonlinear system of equations for s and t:

$$\begin{aligned}2\cos t &= 3 + a\cos s,\\ 2\sin t &= 1 + 2\sin s,\end{aligned}$$

or

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{pmatrix}2\cos x_1 - 3 - a\cos x_2\\ 2\sin x_1 - 1 - 2\sin x_2\end{pmatrix} = \begin{pmatrix}0\\0\end{pmatrix} \quad \textit{with} \quad \boldsymbol{x} = \begin{pmatrix}t\\s\end{pmatrix}.$$

The Jacobian is

$$J = \begin{pmatrix}-2\sin x_1 & a\sin x_2\\ 2\cos x_1 & -2\cos x_2\end{pmatrix}.$$

We would like to compute the intersection points for $a = 1.3 : 0.5 : 7$. From the geometry it is clear that there are two points for a specific value of a in that range.

In the following Matlab script, we compute the solutions using Newton's method, with two different initial guesses for the unknown parameters $\boldsymbol{x} = (t, s)$. We also plot the conputed intersection points, the circle and the different ellipses (see Figure 4.10).

Algorithm 29 Intersection of Circle and Ellipse

```
PlotEllipse([0 0],2,2);
axis([-4 4, -3 5]); axis square; hold;
X1=[]; X2=[];
for a= 1.3:0.5:7
  f=@(x) [2*cos(x(1))-3-a*cos(x(2));2*sin(x(1))-1-2*sin(x(2))];
  fp=@(x) [-2*sin(x(1)) a*sin(x(2)); 2*cos(x(1)) -2*cos(x(2))];
  x=Newton(f,[0;4],1e-10,20);                    % first intersection
  X1=[X1; 2*cos(x(1)) 2*sin(x(1))];
  plot(2*cos(x(1)),2*sin(x(1)),'o');
  x=Newton(f,[1;3],1e-10,20);                    % second intersection
  X2=[X2; 2*cos(x(1)) 2*sin(x(1))];
  plot(2*cos(x(1)),2*sin(x(1)),'o');
  PlotEllipse([3 1],a,2);
  pause
end;
hold off
```

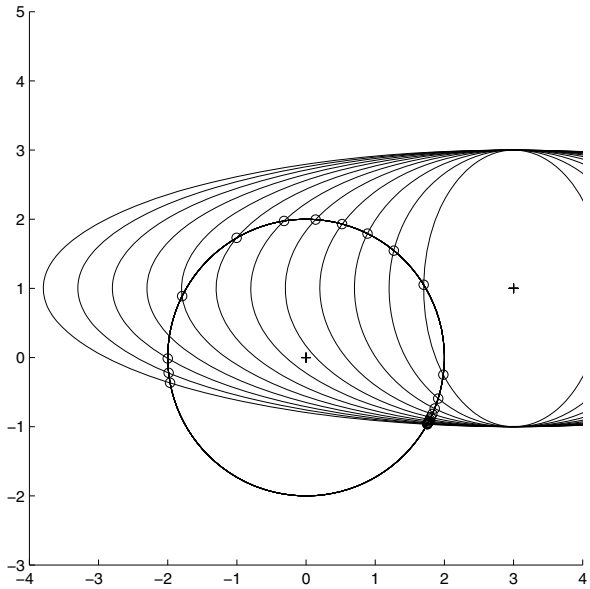


Figure 4.10: Computing Intersection Points with Newton's Method

Table 4.3: Coordinates of Intersection Points

First points			Second points	
1.9845	−0.2486		1.7005	1.0528
1.9104	−0.5919		1.2685	1.5463
1.8594	−0.7367		0.8880	1.7920
1.8261	−0.8157		0.5217	1.9308
1.8038	−0.8640		0.1376	1.9953
1.7883	−0.8956		−0.3184	1.9745
1.7772	−0.9174		−1.0031	1.7302
1.7690	−0.9331		−1.7924	0.8873
1.7628	−0.9447	same!	1.7628	−0.9447
1.7580	−0.9536		−2.0000	−0.0136
1.7543	−0.9605		−1.9877	−0.2218
1.7512	−0.9660		−1.9662	−0.3662

Looking at Figure 4.10 and at the computed coordinates of the intersection points (see Table 4.3) we see that we missed one point! The convergence result in Theorem 4.4.2 is a local result, meaning Newton's method needs good starting values to converge. In this case, our initial guesses are not good enough to ensure that we do not converge to some other point we are not interested in.

Concerning the global convergence behavior of Newton's method, only very little is known. One can only completely analyze the behavior of Newton's method for certain simple examples; one of the best known is the following.

Example 4.4.2. *We consider $f(z) = z^3 - 1$, with $z \in \mathbb{C}$. The equation $f(z) = 0$ has three complex roots on the unit circle. We could write this function in real variables: if we set $z := x + iy$, we obtain*

$$f(x, y) = \begin{pmatrix} x^3 - 3xy^2 - 1 \\ 3x^2y - y^3 \end{pmatrix}.$$

It is however easier to use Newton's method directly in the complex formulation, which leads to the iteration

$$z_{k+1} = z_k - \frac{z_k^3 - 1}{3z_k^2}.$$

It is interesting to find out which of the complex roots will be found by Newton's method as we vary the starting values z_0; the set of initial values that lead to convergence to the same root is called the basin of attraction of that root. The short Matlab Algorithm 30 computes the basin of attraction for each root.

Algorithm 30 Newton's method applied to the complex equation $z^3 - 1 = 0$

```
n=1000; m=30;
x=-1:2/n:1;
[X,Y]=meshgrid(x,x);
Z=X+1i*Y;
for i=1:m
Z=Z-(Z.^3-1)./(3*Z.^2);
end;
image((round(imag(Z))+2)*10);          % transform roots to 10,20,30
```

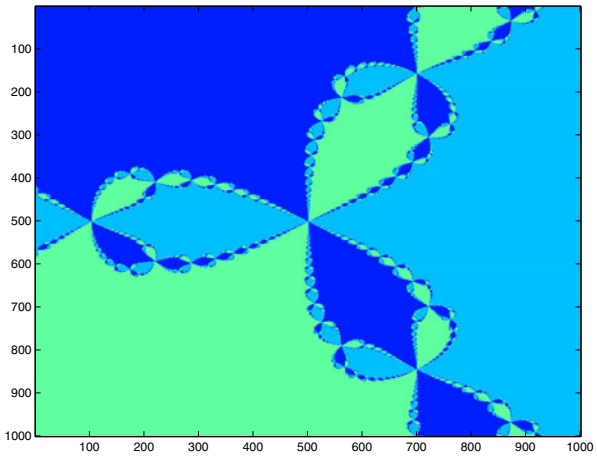


Figure 4.11: Solving the complex equation $z^3 - 1 = 0$ for many starting values: a fractal appears

We show the result in Figure 4.11. We observe that Newton's method does not always converge to the root of the equation that is closest to the initial guess. On the contrary, the basins of attraction of the roots have a very complicated structure, and similarly for their boundaries: they are fractal. It is worthwhile playing with the parameters in the example Algorithm 30 to zoom into specific regions of the Figure, or even to reduce the number of iterations.

There are many variants of Newton's method to address convergence problems or irregularities of the method, mostly developed in the context of optimization.

Chapter 5

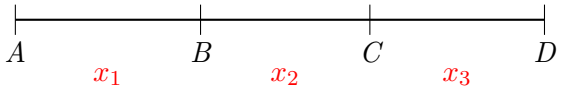
Least Squares Problems

Least squares problems appear very naturally when one would like to estimate values of parameters of a mathematical model from measured data, which are subject to errors. They appear however also in other contexts, and form an important subclass of more general optimization problems. After several typical examples of least squares problems, we start in Section 5.2 with the linear least squares problem and the natural solution given by the normal equations. There were two fundamental contributions to the numerical solution of linear least squares problems in the last century: the first one was the development of the QR factorization by Golub in 1965, and the second one was the implicit QR algorithm for computing the singular value decomposition (SVD) by Golub and Reinsch (1970). We introduce the SVD, which is fundamental for the understanding of linear least squares problems, in Section 5.3.

5.1 Introductory Examples

We start this chapter with several typical examples leading to least squares problems.

Example 5.1.1. *Measuring a road segment*



Assume that we have performed 5 measurements

$$AD = 89m, AC = 67m, BD = 53m, AB = 35m \text{ and } CD = 20m,$$

and we want to determine the length of the segments $x_1 = AB$, $x_2 = BC$ and $x_3 = CD$.

According to the observations we get a linear system with more equations than unknowns:

$$\begin{aligned} x_1 + x_2 + x_3 &= 89 \\ x_1 + x_2 &= 67 \\ x_2 + x_3 &= 53 \\ x_1 &= 35 \\ x_3 &= 20 \end{aligned} \quad \Longleftrightarrow \quad A\boldsymbol{x} = \boldsymbol{b},\ A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},\ \boldsymbol{b} = \begin{pmatrix} 89 \\ 67 \\ 53 \\ 35 \\ 20 \end{pmatrix}$$

Notice that if we use the last three equations then we get the solution $x_1 = 35$, $x_2 = 33$ and $x_3 = 20$. However, if we check the first two equations by inserting this solution we get

$$x_1 + x_2 + x_3 - 89 = -1,$$
$$x_1 + x_2 - 67 = 1.$$

So the equations contradict each other because of the measurement errors, and the over-determined system has no solution.

A remedy is to find an approximate solution that satisfies the equations as well as possible. For that purpose one introduces the residual vector

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$$

One then looks for a vector $\boldsymbol{x}$ that minimizes in some sense the residual vector.

Example 5.1.2. *The amount f of a component in a chemical reaction decreases with time t exponentially according to:*

$$f(t) = a_0 + a_1 e^{-bt}$$

If the material is weighed at different times, we obtain a table of measured values: The

t	t_1	...	t_m
y	y_1	...	y_m

problem now is to estimate the model parameters a_0, a_1 and b from these observations. Each measurement point (t_i, y_i) yields an equation:

$$f(t_i) = a_0 + a_1 e^{-bt_i} \approx y_i, i = 1, \ldots m. \tag{5.1}$$

If there were no measurement errors, then we could replace the approximate symbol in (5.1) by an equality and use three equations from the set to determine the parameters. However, in practice, measurement errors are inevitable. Furthermore, the model equations are often not quite correct and only model the physical behavior approximately. The equations will therefore in general contradict each other and we need some mechanism to balance the measurement errors, e.g. by requiring that (5.1) be satisfied as well as possible.

Example 5.1.3. *The next example comes from coordinate metrology. Here a coordinate measuring machine measures two sets of points on two orthogonal lines (see Figure 5.1). If we represent the line g_1 by the equations*

$$g_1 : \quad c_1 + n_1 x + n_2 y = 0, \quad n_1^2 + n_2^2 = 1 \tag{5.2}$$

then $\boldsymbol{n} = (n_1, n_2)^\top$ is the normal vector on g_1. The normalizing equation $n_1^2 + n_2^2 = 1$ ensures the uniqueness of the parameters c, n_1 and n_2.

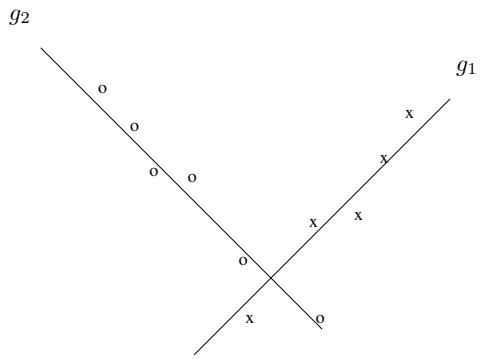


Figure 5.1: Measured points on two orthogonal lines.

If we insert the coordinates of a measured point $P_i = (x_i, y_i)$ into Equation (5.2), we obtain the residual $r_i = c_1 + n_1 x_i + n_2 y_i$ and $d_i = |r_i|$ is the distance of P_i from g_1. The equation of a line g_2 orthogonal to g_1 is

$$g_2 : \quad c_2 - n_2 x + n_1 y = 0, \quad n_1^2 + n_2^2 = 1 \tag{5.3}$$

If we now insert the coordinates of q measured points Q_i into (5.3) and of p points P_i into Equation (5.2), we we obtain the following system of equations for determining the parameters c_1, c_2, n_1 and n_2:

$$\begin{pmatrix} 1 & 0 & x_{P_1} & y_{P_1} \\ 1 & 0 & x_{P_2} & y_{P_2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & x_{P_p} & y_{P_p} \\ 0 & 1 & y_{Q_1} & -x_{Q_1} \\ 0 & 1 & y_{Q_2} & -x_{Q_2} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & y_{Q_q} & -x_{Q_q} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ n_1 \\ n_2 \end{pmatrix} \approx \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \textit{subject to } n_1^2 + n_2^2 = 1 \tag{5.4}$$

Note the difference between the least squares equations and the constraint: whereas equations of the type g_1 or g_2 only need to be satisfied approximately, the constraint $n_1^2 + n_2^2 = 1$ must be satisfied exactly by the solution.

Example 5.1.4. *In control theory, one often considers a system like the one in Figure 5.2. The vectors $\boldsymbol{u}$ and $\boldsymbol{y}$ are the measured input and output signals at various points in time. Let $y_{t+i} = y(t + i\triangle t)$. A simple model assumes a linear relationship between the output and the input signal of the form.*

$$y_{t+n} + a_{n-1} y_{t+n-1} + \cdots + a_0 y_t \approx b_{n-1} u_{t+n-1} + b_{n-2} u_{t+n-2} + \cdots + b_0 u_t \tag{5.5}$$

The problem is to determine the parameters ai and bi from measurements of u and y. For each time step we obtain a new equation of the form (5.5). If we write them all

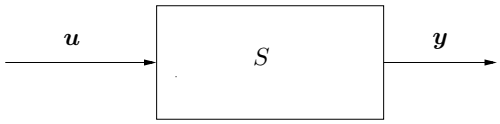


Figure 5.2: System with input $\boldsymbol{u}$ and output $\boldsymbol{y}$

together, we get a system of linear equations:

$$\begin{pmatrix} y_{n-1} & y_{n-2} & \dots & y_0 & -u_{n-1} & -u_{n-2} & \dots & -u_0 \\ y_n & y_{n-1} & \dots & y_1 & -u_n & -u_{n-1} & \dots & -u_1 \\ y_{n+1} & y_n & \dots & y_2 & -u_{n+1} & -u_n & \dots & -u_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \begin{pmatrix} a_{n-1} \\ a_{n-2} \\ \vdots \\ a_0 \\ b_{n-1} \\ n_{n-2} \\ \vdots \\ b_n \end{pmatrix} \approx \begin{pmatrix} -y_n \\ -y_{n+1} \\ -y_{n+2} \\ \vdots \end{pmatrix} \tag{5.6}$$

A matrix is said to be Toeplitz if it has constant elements on the diagonals. Here in (5.6), the matrix is composed of two Toeplitz matrices. The number of equations is not fixed, since one can generate new equations simply by adding a new measurement.

Example 5.1.5. *In robotics and many other applications, one often encounters the Procrustes problem or one of its variants. Consider a given body (e.g., a pyramid like in Figure 5.3) and a copy of the same body. Assume that we know the coordinates of m points $\boldsymbol{x_i}$ on the first body, and that the corresponding points $\boldsymbol{\xi_i}$ have been measured*

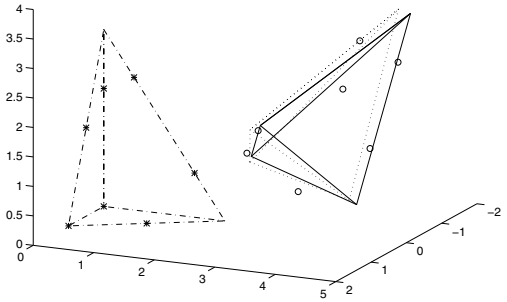


Figure 5.3: Procrustes or registration problem

on the other body in another position in space. We would like to rotate and translate the second body so that it can be superimposed onto the first one as well as possible. In other words, we seek an orthogonal matrix Q (the product of three rotations) and a translation vector $\boldsymbol{t}$ such that $\boldsymbol{\xi_i} \approx Q\boldsymbol{x_i} + \boldsymbol{t}$ for $i = 1, \dots, m$.

The above examples are illustrations of different classes of approximation problems. For instance, in Examples 5.1.1 and 5.1.4, the equations are linear. However, in Example

5.1.2 (chemical reactions), the system of equations (5.1) is nonlinear. In the metrology example 5.1.3, the equations are linear, but they are subject to the nonlinear constraint $n_1^2 + n_2^2 = 1$. Finally, we have also an nonlinear problem in Example 5.1.5, and it is not clear how to parametrize the unknown matrix Q. Nonetheless, in all the above examples, we would like to satisfy some equations as well as possible; this is indicated by the approximation symbol "$\approx$" and we have to define what we mean by that.

There are also least squares problems that are not connected with measurements like in the following example:

Example 5.1.6. *We consider two straight lines g and h in space. Assume they are given by a point and a direction vector:*

$$g : \boldsymbol{X} = \boldsymbol{P} + \lambda \boldsymbol{t}$$
$$h : \boldsymbol{Y} = \boldsymbol{Q} + \mu \boldsymbol{s}$$

If they intersect each other, then there must exist a λ and a μ such that

$$\boldsymbol{P} + \lambda \boldsymbol{t} = \boldsymbol{Q} + \mu \boldsymbol{s} \tag{5.7}$$

Rearranging (5.7) yields

$$\begin{pmatrix} t_1 & -s_1 \\ t_2 & -s_2 \\ t_3 & -s_3 \end{pmatrix} \begin{pmatrix} \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} Q_1 - P_1 \\ Q_2 - P_2 \\ Q_3 - P_3 \end{pmatrix} \tag{5.8}$$

a system of three linear equations with two unknowns. If the equations are consistent, then we can use two of them to determine the intersection point. If, however, we have a pair of skew lines (i.e., if (5.8) has no solution) then we may be interested in finding the the point $\boldsymbol{X}$ on g and $\boldsymbol{Y}$ on h which are closest, i.e. for which the distance vector $\boldsymbol{r} = \boldsymbol{X} - \boldsymbol{Y}$ has minimal length $\|\boldsymbol{r}\|_2^2 \to \min$. Thus, we are interested in solving (5.8) as a least squares problem.

5.2 Linear Least Squares Problem and the Normal Equations

Linear least squares problems occur when solving overdetermined linear systems, i.e. we are given more equations than unknowns. In general, such an overdetermined system has no solution, but we may find a meaningful approximate solution by minimizing some norm of the residual vector.

Given a matrix $A \in \mathbb{R}^{m \times n}$ with $m > n$ and a vector $\boldsymbol{b} \in \mathbb{R}^m$ we are looking for a vector $\boldsymbol{x} \in \mathbb{R}^n$ for which the norm of the residual $\boldsymbol{r}$ is minimized, i.e.

$$\|\boldsymbol{r}\| = \|\boldsymbol{b} - A\boldsymbol{x}\| \to \min \tag{5.9}$$

The calculations are simplest when we choose the 2-norm. Thus we will minimize the square of the length of the residual vector

$$\|\boldsymbol{r}\|_2^2 = r_1^2 + r_2^2 + \cdots + r_m^2 \to \min \tag{5.10}$$

To see that this minimum exists and is attained by some $\boldsymbol{x} \in \mathbb{R}^n$, note that $E = \{\boldsymbol{b} - A\boldsymbol{x} | \boldsymbol{x} \in \mathbb{R}^n\}$ is non-empty, closed and convex subset of $\mathbb{R}^m$. Since $\mathbb{R}^m$ equipped with the Euclidean inner product is a Hilbert space, E contains a unique element of smallest norm, so there exists an $\boldsymbol{x} \in \mathbb{R}^n$(not necessarily unique) such that $\|\boldsymbol{b} - A\boldsymbol{x}\|_2$ is minimized.

The minimization problem (5.10) gave rise to the name Least Squares Method. The theory was developed independently by Carl Friedrich Gauss in 1795 and Adrien-Marie Legendre who published it first in 1805. On January 1, 1801, using the least squares method, Gauss made the best prediction of the orbital positions of the planetoid Ceres based on measurements of G. Piazzi, and the method became famous because of this.

We characterize the least squares solution by the following theorem.

Theorem 5.2.1. *(Least Squares Solution) Let*

$$S = \{\boldsymbol{x} \in \mathbb{R}^n \text{ with } \|\boldsymbol{b} - A\boldsymbol{x}\|_2 \to \min\}$$

be the set of solutions and let $\boldsymbol{r_x} = \boldsymbol{b} - A\boldsymbol{x}$ denote the residual for a specific $\boldsymbol{x}$. Then

$$\boldsymbol{x} \in S \iff A^\top \boldsymbol{r_x} = 0 \iff \boldsymbol{r_x} \perp \mathcal{R}(A) \tag{5.11}$$

where $\mathcal{R}(A)$ denotes the subspace spanned by the columns of A.

Proof. We prove the first equivalence, from which the second one follows easily.

"$\Leftarrow$": Let $A^\top \boldsymbol{r_x} = 0$ and $\boldsymbol{z} \in \mathbb{R}^n$ be an arbitrary vector. It follows that $\boldsymbol{r_z} = \boldsymbol{b} - A\boldsymbol{z} = \boldsymbol{b} - A\boldsymbol{x} + A(\boldsymbol{x} - \boldsymbol{z})$, thus $\boldsymbol{r_z} = \boldsymbol{r_x} + A(\boldsymbol{x} - \boldsymbol{z})$. Now

$$\|\boldsymbol{r_z}\|_2^2 = \|\boldsymbol{r_x}\|_2^2 + 2(\boldsymbol{x} - \boldsymbol{z})^\top A^\top \boldsymbol{r_x} + \|A(\boldsymbol{x} - \boldsymbol{z})\|_2^2$$

But $A^\top \boldsymbol{r_x} = 0$ and therefore $\|\boldsymbol{r_z}\|_2 > \|\boldsymbol{r_x}\|_2$. Since this holds for every $\boldsymbol{z}$ then $\boldsymbol{x} \in S$.

"$\Rightarrow$": We show this by contradiction: assume $A^\top \boldsymbol{r_x} = \boldsymbol{z} \neq 0$. We consider $\boldsymbol{u} = \boldsymbol{x} + \varepsilon \boldsymbol{z}$ with $\varepsilon > 0$:

$$\boldsymbol{r_u} = \boldsymbol{b} - A\boldsymbol{u} = \boldsymbol{b} - A\boldsymbol{x} - \varepsilon A\boldsymbol{z} = \boldsymbol{r_x} - \varepsilon A\boldsymbol{z}$$

Now $\|\boldsymbol{r_u}\|_2^2 = \|\boldsymbol{r_x}\|_2^2 - 2\varepsilon \boldsymbol{z}^\top A^\top \boldsymbol{r_x} + \varepsilon^2 \|A\boldsymbol{z}\|_2^2$. Because $A^\top \boldsymbol{r_x} = \boldsymbol{z}$ we obtain

$$\|\boldsymbol{r_u}\|_2^2 = \|\boldsymbol{r_x}\|_2^2 - 2\varepsilon \|\boldsymbol{z}\|_2^2 + \varepsilon^2 \|A\boldsymbol{z}\|_2^2$$

We conclude that, for sufficient small ε, we can obtain $\|\boldsymbol{r_u}\|_2^2 < \|\boldsymbol{r_x}\|_2^2$. This is a contradiction, since $\boldsymbol{x}$ cannot be in the set of solutions in this case. Thus the assumption was wrong, i.e., we must have $A^\top \boldsymbol{r_x} = 0$, which proves the first equivalence in (5.11). □

The least squares solution has an important statistical property which is expressed in the following $Gauss-Markoff$ Theorem. Let the vector $\boldsymbol{b}$ of observations be related to an unknown parameter vector $\boldsymbol{x}$ by the linear relation

$$A\boldsymbol{x} = \boldsymbol{b} + \boldsymbol{\varepsilon} \tag{5.12}$$

where $A \in \mathbb{R}^{m \times n}$is a known matrix and $\boldsymbol{\varepsilon}$ is a vector of random errors. In this standard linear model it is assumed that the random variables ε_j are uncorrelated and all have zero mean and the same variance.

Theorem 5.2.2. *(Gauss-Markoff) Consider the standard linear model (5.12). Then the best linear unbiased estimator of any linear function $c^\top x$ is the least square solution of $\|A\boldsymbol{x} - \boldsymbol{b}\|_2^2 \to \min$.*

Proof. Consult a statistics textbook, for example, A. M. Mood and F. A. Graybill. Introduction to the Theory of Statistics. McGraw-Hill Book Company, New York, 2nd edition, 1963. (p. 181) □

Equation (5.11) can be used to determine the least square solution. From $A^\top \boldsymbol{r}_{\boldsymbol{x}} = 0$ it follows that $A^\top(\boldsymbol{b} - A\boldsymbol{x}) = 0$, and we obtain the *Normal Equations of Gauss*:

$$A^\top A\boldsymbol{x} = A^\top \boldsymbol{b} \tag{5.13}$$

Example 5.2.1. *We return to Example 5.1.1 and solve it using the Normal Equations.*

$$A^\top A\boldsymbol{x} = A^\top \boldsymbol{b} \Longleftrightarrow \begin{pmatrix} 3 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 191 \\ 209 \\ 162 \end{pmatrix}$$

The solution of this 3×3 system is

$$\boldsymbol{x} = \begin{pmatrix} 35.125 \\ 32.500 \\ 20.625 \end{pmatrix}$$

The residual for this solution becomes

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x} = \begin{pmatrix} 0.7500 \\ -0.6250 \\ -0.1250 \\ -0.1250 \\ -0.6250 \end{pmatrix} \quad \textit{with } \|\boldsymbol{r}\|_2 = 1.1726$$

Notice that for the solution $\boldsymbol{x} = (35, 33, 20)^\top$ obtained by solving the last three equations we obtain a larger residual $\|\boldsymbol{r}\|_2 = \sqrt{2} = 1.4142$.

There is also a way to understand the normal equations geometrically from (5.11). We want to find a linear combination of columns of the matrix A to approximate the vector $\boldsymbol{b}$. The space spanned by the columns of A is the range of A, $\mathcal{R}(A)$, which is a hyperplane in $\mathbb{R}^m$, and the vector $\boldsymbol{b}$ in general does not lie in this hyperplane, as shown in Figure 5.4. Thus, minimizing $\|\boldsymbol{b} - A\boldsymbol{x}\|_2$ is equivalent to minimizing the length of the residual vector $\boldsymbol{r}$, and thus the residual vector has to be orthogonal to $\mathcal{R}(A)$, as shown in Figure 5.4.

The normal equations (5.13) concentrate data since $B = A^\top A$ is a small $n \times n$ matrix, whereas A is $m \times n$. The matrix B is symmetric, and if rank(A)=n, then it is also positive definite. Thus, the natural way to solve the normal equations is by means of the Cholesky decomposition

1. Form $B = A^\top A$ (we need to compute only the upper triangle since B is symmetric) and compute $\boldsymbol{c} = A^\top \boldsymbol{b}$.

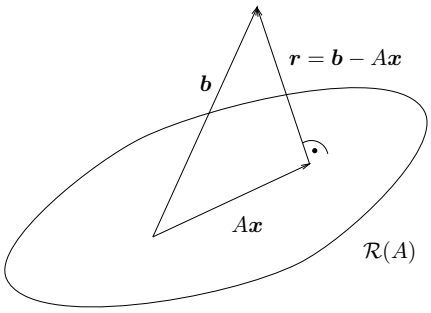


Figure 5.4: r is orthogonal to $R(A)$

2. Decompose $B = R^\top R$ (Cholesky) where R is an upper triangular matrix.
3. Compute the solution by forward-($R^\top \boldsymbol{y} = \boldsymbol{c}$) and back-substitution ($R\boldsymbol{x} = \boldsymbol{y}$).

We will see later on that there are numerically preferable methods for computing the least squares solution. They are all based on the use of orthogonal matrices (i.e. matrices B for which $B^\top B = I$).

Notice that when solving linear systems $A\boldsymbol{x} = \boldsymbol{b}$ with n equations and n unknowns by Gaussian elimination, reducing the system to triangular form, we make use of the fact that equivalent systems have the same solutions:

$$A\boldsymbol{x} = \boldsymbol{b} \Longleftrightarrow BA\boldsymbol{x} = B\boldsymbol{b} \quad if\ B\ is\ nonsingular$$

For a system of equations $A\boldsymbol{x} \approx \boldsymbol{b}$ to be solved in the least squares sense, it no longer holds that multiplying by a nonsingular matrix B leads to an equivalent system. This is because the transformed residual $B\boldsymbol{r}$ may not have the same norm as $\boldsymbol{r}$ itself. However, if we restrict ourselves to the class of *orthogonalmatrices*,

$$A\boldsymbol{x} \approx \boldsymbol{b} \Longleftrightarrow BA\boldsymbol{x} \approx B\boldsymbol{b} \quad if\ B\ is\ orthogonal.$$

then the least squares problems remain equivalent, since $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$ and $B\boldsymbol{r} = B\boldsymbol{b} - BA\boldsymbol{x}$ have the same length,

$$\|B\boldsymbol{r}\|_2^2 = (B\boldsymbol{r})^\top (B\boldsymbol{r}) = \boldsymbol{r}^\top B^\top B\boldsymbol{r} = \boldsymbol{r}^\top \boldsymbol{r} = \|\boldsymbol{r}\|_2^2$$

Orthogonal matrices and the matrix decompositions containing orthogonal factors therefore play an important role in algorithms for the solution of linear least squares problems. Often it is possible to simplify the equations by pre-multiplying the system by a suitable orthogonal matrix.

5.3 Singular Value Decomposition (SVD)

The singular value decomposition (SVD) of a matrix A is a very useful tool in the context of least squares problems. It is also very helpful for analyzing properties of a matrix. With the SVD one x-rays a matrix!

Theorem 5.3.1. *(Singular Value Decomposition, SVD) Let $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. Then there exist orthogonal matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ and a diagonal matrix $\Sigma = diag\{\sigma_1, \dots \sigma_n\} \in \mathbb{R}^{m\times n}$ with $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$, such that*

$$A = U\Sigma V^\top$$

holds. The column vectors of $U = [\boldsymbol{u_1}, \dots, \boldsymbol{u_m}]$ are called the left singular vectors and similarly $V = [\boldsymbol{v_1}, \dots, \boldsymbol{v_n}]$ are the right singular vectors. The values σ_i are called the singular values of A. If $\sigma_r > 0$ is the smallest nonzero singular value, then the matrix A has rank r.

Proof. The 2-norm of A is defined by $\|A\|_2 = \max_{\|\boldsymbol{x}\|_2=1} \|A\boldsymbol{x}\|_2$. Thus there exists a vector $\boldsymbol{x}$ with $\|\boldsymbol{x}\|_2 = 1$ such that

$$\boldsymbol{z} = A\boldsymbol{x}, \quad \|\boldsymbol{z}\|_2 = \|A\|_2 =: \sigma$$

Let $\boldsymbol{y} := \boldsymbol{z}/\|\boldsymbol{z}\|_2$. This yields $A\boldsymbol{x} = \sigma\boldsymbol{y}$ with $\|\boldsymbol{x}\|_2 = \|\boldsymbol{y}\|_2 = 1$.

Next we extend $\boldsymbol{x}$ into an orthonormal basis of $\mathbb{R}^n$. If $V \in \mathbb{R}^{n\times n}$ is the matrix containing the basis vectors as columns, then V is an orthogonal matrix that can be written as $V = [\boldsymbol{x}, V_1]$, where $V_1^\top \boldsymbol{x} = 0$. Similarly, we can construct an orthogonal matrix $U \in \mathbb{R}^{m\times m}$ satisfying $U = [\boldsymbol{y}, U_1], U_1^\top \boldsymbol{y} = 0$. Now

$$A_1 = U^\top AV = \begin{bmatrix} \boldsymbol{y}^\top \\ U_1^\top \end{bmatrix} A \begin{bmatrix} \boldsymbol{x} & V_1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{y}^\top A\boldsymbol{x} & \boldsymbol{y}^\top AV_1 \\ U_1^\top A\boldsymbol{x} & U_1^\top AV_1 \end{bmatrix} = \begin{bmatrix} \sigma & \boldsymbol{w}^\top \\ 0 & B \end{bmatrix}$$

because $\boldsymbol{y}^\top A\boldsymbol{x} = \boldsymbol{y}^\top \sigma\boldsymbol{y} = \sigma\boldsymbol{y}^\top\boldsymbol{y} = \sigma$ and $U_1^\top A\boldsymbol{x} = \sigma U_1^\top \boldsymbol{y} = 0$ since $U_1 \perp \boldsymbol{y}$.

We claim that $\boldsymbol{w}^\top := \boldsymbol{y}^\top AV_1 = 0$. In order to prove this, we compute

$$A_1 \begin{pmatrix} \sigma \\ \boldsymbol{w} \end{pmatrix} = \begin{pmatrix} \sigma^2 + \|\boldsymbol{w}\|_2^2 \\ B\boldsymbol{w} \end{pmatrix}$$

and conclude from that equation that

$$\|A_1 \begin{pmatrix} \sigma \\ \boldsymbol{w} \end{pmatrix}\|_2^2 = (\sigma^2 + \|\boldsymbol{w}\|_2^2)^2 + \|B\boldsymbol{w}\|_2^2 \geq (\sigma^2 + \|\boldsymbol{w}\|_2^2)^2$$

Now since V and U are orthogonal, $\|A_1\|_2 = \|U^\top AV\|_2 = \|A\|_2 = \sigma$ holds and

$$\sigma^2 = \|A_1\|_2^2 = \max_{\|\boldsymbol{x}\|_2 \neq 0} \frac{\|A_1\boldsymbol{x}\|_2^2}{\|\boldsymbol{x}\|_2^2} \geq \frac{\|A_1 \begin{pmatrix} \sigma \\ \boldsymbol{w} \end{pmatrix}\|_2^2}{\|\begin{pmatrix} \sigma \\ \boldsymbol{w} \end{pmatrix}\|_2^2} \geq \frac{(\sigma^2 + \|\boldsymbol{w}\|_2^2)^2}{\sigma^2 + \|\boldsymbol{w}\|_2^2}$$

The last equation reads

$$\sigma^2 \geq \sigma^2 + \|\boldsymbol{w}\|_2^2$$

and we conclude that $\boldsymbol{w} = 0$. Thus we have obtained

$$A_1 = U^\top AV = \begin{bmatrix} \sigma & 0 \\ 0 & B \end{bmatrix}$$

We can now apply the same construction to the sub-matrix B and thus finally end up with a diagonal matrix. □

If we write the equation $A = U\Sigma V^\top$ in partitioned form, in which Σ_r contains only the nonzero singular values, we get

$$A = [U_1, U_2] \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} [V_1, V_2]^\top \tag{5.14}$$

$$= U_1 \Sigma_r V_1^\top \tag{5.15}$$

$$= \sum_{i=1}^{r} \sigma_i \boldsymbol{u_i} \boldsymbol{v_i}^\top \tag{5.16}$$

Equation (5.14) is the full decomposition with square matrices U and V . When making use of the zeros we obtain the "economy" or "reduced" version of the SVD given in (5.15). In Matlab there are two variants to compute the SVD:

```
[U S V ]=svd(A) % gives the full decomposition
[U S V ]=svd(A,0) % gives an m by n matrix U
```

The call `svd(A,0)` computes a version between full and economic with a nonsquare matrix $U \in \mathbb{R}^{m\times n}$ This form is sometimes referred to as the "thin SVD".

Example 5.3.1. *The matrix A has rank one and its economy SVD is given by*

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} (2\sqrt{3}) \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{pmatrix}$$

With Matlab we get the thin version

```
>> [U,S,V]=svd(ones(4,3),0)
U =
-0.5000  0.8660 -0.0000
-0.5000 -0.2887 -0.5774
-0.5000 -0.2887  0.7887
-0.5000 -0.2887 -0.2113
S =
3.4641 0      0
0      0.0000 0
0      0      0
V =
-0.5774  0.8165  0
-0.5774 -0.4082 -0.7071
-0.5774 -0.4082  0.7071
```

Theorem 5.3.2. *If $A = U\Sigma V^\top$, then the column vectors of V are the eigenvectors of the matrix $A^\top A$ associated with the eigenvalues $\sigma_i^2, i = 1, \ldots, n$. The column vectors of U are the eigenvectors of the matrix $AA^\top$.*

Proof.

$$A^\top A = (U\Sigma V^\top)^\top U\Sigma V^\top = VDV^\top, \quad D = \Sigma^\top\Sigma = diag(\sigma_1^2, \ldots, \sigma_n^2) \tag{5.17}$$

Thus $A^\top AV = VD$ and σ_i^2 is an eigenvalue of $A^\top A$. Similarly

$$AA^\top = U\Sigma V^\top (U\Sigma V^\top)^\top = U\Sigma\Sigma^\top U^\top \tag{5.18}$$

where $\Sigma\Sigma^\top = diag(\sigma_1^2, \ldots, \sigma_n^2, 0, \ldots, 0) \in \mathbb{R}^{m\times m}$ □

Theorem 5.3.3. *Let $A = U\Sigma V^\top$. Then*

$$\|A\|_2 = \sigma_1 \quad \|A\|_F = \sqrt{\sum_{i=1}^{n} \sigma_i^2}$$

Proof. Since U and V are orthogonal, we have $\|A\|_2 = \|U\Sigma V^\top\|_2 = \|\Sigma\|_2$. Now

$$\|\Sigma\|_2^2 = \max_{\|\boldsymbol{x}\|_2=1} \|\Sigma\boldsymbol{x}\|_2^2 = \max_{\|\boldsymbol{x}\|_2=1} (\sigma_1^2 x_1^2 + \cdots + \sigma_n^2 x_n^2) \leq \sigma_1^2(x_1^2 + \cdots + x_n^2) = \sigma_1^2$$

and since the maximum is attained for $\boldsymbol{x} = \boldsymbol{e_1}$ it follows that $\|A\|_2 = \sigma_1$. For the *Frobenius norm* we have

$$\|A\|_F = \sqrt{\sum_{i,j} a_{i,j}^2} = \sqrt{tr(A^\top A)} = \sqrt{\sum_{i=1}^{n} \sigma_i^2}$$

since the trace of a matrix equals the sum of its eigenvalues. □

In (5.16), we have decomposed the matrix A as a sum of rank-one matrices of the form $\boldsymbol{u_i}\boldsymbol{v_i}^\top$. Now we have

$$\|\boldsymbol{u_i}\boldsymbol{v_i}^\top\|_2^2 = \max_{\|\boldsymbol{x}\|_2=1} \|\boldsymbol{u_i}\boldsymbol{v_i}^\top\boldsymbol{x}\|_2^2 = \max_{\|\boldsymbol{x}\|_2=1} |\boldsymbol{v_i}^\top\boldsymbol{x}|^2 \|\boldsymbol{u_i}\|_2^2 = \max_{\|\boldsymbol{x}\|_2=1} |\boldsymbol{v_i}^\top\boldsymbol{x}|^2$$

and since

$$\max_{\|\boldsymbol{x}\|_2=1} |\boldsymbol{v_i}^\top\boldsymbol{x}|^2 = \max_{\|\boldsymbol{x}\|_2=1} (\|\boldsymbol{v_i}\|_2\|\boldsymbol{x}\|_2 \cos\alpha)^2 = \cos^2\alpha$$

where α is the angle between the two vectors, we obtain

$$\max_{\|\boldsymbol{x}\|_2=1} \|\boldsymbol{v_i}^\top\boldsymbol{x}\|_2^2 = \|\boldsymbol{v_i}^\top\boldsymbol{v_i}\|_2^2 = 1$$

We see from (5.16) that the matrix A is decomposed into a weighted sum of matrices which have all the same norm, and the singular values are the weights. The main contributions in the sum are the terms with the largest singular values. Therefore we may approximate A by a lower rank matrix by dropping the smallest singular values, i.e., replacing their values by zero. In fact we have the

Theorem 5.3.4. *Let $A \in \mathbb{R}^{m\times n}$ have rank r and let $A = U\Sigma V^\top$. Let $\mathcal{M}$ denote the set of $m \times n$ matrices with rank $p < r$. The solution of*

$$\min_{X\in\mathcal{M}} \|A - X\|_2$$

is given by $A_p = \sum_{i=1}^{p} \sigma_i \boldsymbol{u_i}\boldsymbol{v_i}^\top$ and we have

$$\min_{X\in\mathcal{M}} \|A - X\|_2 = \|A - A_p\|_2 = \sigma_{p+1}.$$

Proof. We have $U^\top A_p V = diag(\sigma_1^2, \ldots, \sigma_p^2, 0, \ldots, 0)$ thus $A_p \in \mathcal{M}$ and

$$\|A - A_p\|_2 = \sigma_{p+1}$$

Let $B \in \mathcal{M}$ Mand let the linear independent vectors $\boldsymbol{x_1}, \ldots, \boldsymbol{x_{n-p}}$ span the null space of B, so that $B\boldsymbol{x_j} = 0$. The two sets of vectors $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_{n-p}}\}$ and $\{\boldsymbol{v_1}, \ldots, \boldsymbol{v_{p+1}}\}$ contain altogether $n + 1$ vectors. Hence, they must be linearly dependent, so we can write

$$\alpha_1\boldsymbol{x_1} + \alpha_2\boldsymbol{x_2} + \cdots + \alpha_{n-p}\boldsymbol{x_{n-p}} + \beta_1\boldsymbol{v_1} + \beta_2\boldsymbol{v_2} + \cdots + \beta_{p+1}\boldsymbol{v_{p+1}} = 0.$$

Moreover, not all $\alpha_i = 0$, otherwise the vectors $\boldsymbol{v_i}$ would be linearly dependent! Denote by

$$\boldsymbol{h} = -\alpha_1\boldsymbol{x_1} - \alpha_2\boldsymbol{x_2} - \cdots - \alpha_{n-p}\boldsymbol{x_{n-p}} = \beta_1\boldsymbol{v_1} + \beta_2\boldsymbol{v_2} + \cdots + \beta_{p+1}\boldsymbol{v_{p+1}} \neq 0$$

and form the unit vector $\boldsymbol{z} = \boldsymbol{h}/\|\boldsymbol{h}\|_2 = \gamma_1\boldsymbol{v_1} + \gamma_2\boldsymbol{v_2} + \cdots + \gamma_{p+1}\boldsymbol{v_{p+1}}$. Then $B\boldsymbol{z} = 0$, $\boldsymbol{z}^\top\boldsymbol{z} = \gamma_1^2 + \cdots + \gamma_{p+1}^2 = 1$ and

$$A\boldsymbol{z} = U\Sigma V^\top\boldsymbol{z} = \sum_{i=1}^{p+1} \sigma_i\gamma_i\boldsymbol{u_i}.$$

It follows that

$$\begin{aligned}\|A - B\|_2^2 \geq \|(A - B)\boldsymbol{z}\|_2^2 = \|A\boldsymbol{z}\|_2^2 &= \sum_{i=1}^{p+1} \sigma_i^2\gamma_i^2 \\ &\geq \sigma_{p+1}^2 \sum_{i=1}^{p+1} \gamma_i^2 = \sigma_{p+1}^2\|z\|_2^2 = \sigma_{p+1}^2\end{aligned}$$

Thus, the distance from A to any other matrix in $\mathcal{M}$ is greater or equal to the distance to A_p. This proves the theorem. □

5.3.1 Pseudoinverse

Definition 5.3.5. *(Pseudoinverse) Let $A = U\Sigma V^\top$ be the singular value decomposition with*

$$\Sigma = \begin{pmatrix}\Sigma_r \\ 0\end{pmatrix} \in \mathbb{R}^{m\times n}, \quad \Sigma_r := diag(\sigma_1, \ldots, \sigma_r, 0, \ldots, 0) \in \mathbb{R}^{n\times n}$$

with $\sigma_1 \geq \cdots \geq \sigma_r > 0$. Then the matrix $A^+ = V\Sigma^+ U^\top$ with

$$\Sigma^+ = (\Sigma_r^+ \; 0) \in \mathbb{R}^{n\times m}, \quad \Sigma_r^+ := diag(\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_r}, 0, \ldots, 0) \in \mathbb{R}^{n\times n} \tag{5.19}$$

is called the pseudoinverse of A.

We have discussed the SVD only for the case in which $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. This was mainly for simplicity, since the SVD exists for any matrix: if $A = U\Sigma V^\top$, then $A^\top = V\Sigma^\top U^\top$ is the singular value decomposition of $A^\top \in \mathbb{R}^{n\times m}$. Usually the SVD is computed such that the singular values are ordered decreasingly. The representation $A^+ = V\Sigma^+ U^\top$ of the pseudoinverse is thus already a SVD, except that the singular values $\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_r}$ are ordered increasingly. By simultaneously permuting rows and columns one can reorder the decomposition and bring it into standard form with decreasing elements in Σ^+.

Theorem 5.3.6. *(Penrose Equations) $Y = A^+$ is the only solution of the matrix equations*

$$\begin{aligned} &(i) AYA = A && (ii) YAY = Y \\ &(iii) (AY)^\top = AY && (iv) (YA)^\top = YA \end{aligned}$$

Proof. It is simple to verify that A^+ is a solution: inserting the SVD e.g. into (i), we get

$$AA^+A = U\Sigma V^\top V\Sigma^+ U^\top U\Sigma V^\top = U\Sigma\Sigma^+\Sigma V^\top = U\Sigma V^\top = A.$$

More challenging is to prove uniqueness. To do this, assume that Y is any solution to (i)-(iv). Then

$$\begin{aligned} Y &= YAY \quad \text{because of (ii)} \\ &= (YA)^\top Y = A^\top Y^\top Y \quad \text{because of (iv)} \\ &= (AA^+A)^\top Y^\top Y = A^\top (A^+)^\top A^\top Y^\top Y \quad \text{because of (i)} \\ &= A^\top (A^+)^\top YAY \quad \text{because of (iv)} \\ &= A^\top (A^+)^\top Y = (A^+A)^\top Y \quad \text{because of (ii)} \\ &= A^+AY \quad \text{because of (iv)} \\ &= A^+AA^+AY \quad \text{because of (ii)} \\ &= A^+(AA^+)^\top (AY)^\top = A^+(A^+)^\top A^\top Y^\top A^\top \quad \text{because of (iii)} \\ &= A^+(A^+)^\top A^\top \quad \text{because of (i)} \\ &= A^+(AA^+)^\top = A^+AA^+ \quad \text{because of (iii)} \\ Y &= A^+ \quad \text{because of (ii)} \end{aligned}$$

□

5.3.2 Fundamental Subspaces

There are four fundamental subspaces associated with a matrix $A \in \mathbb{R}^{m\times n}$:

Definition 5.3.7. *(Fundamental Subspaces of a Matrix)*

1. $\mathcal{R}(A) = \{\boldsymbol{y} | \boldsymbol{y} = A\boldsymbol{x}, \quad \boldsymbol{x} \in \mathbb{R}^n\} \subset \mathbb{R}^m$ *is the range or column space.*
2. $\mathcal{R}(A)^\perp$ *the orthogonal complement of* $\mathcal{R}(A)$.
If $\boldsymbol{z} \in \mathcal{R}(A)^\perp$, *then* $\boldsymbol{z}^\top \boldsymbol{y} = 0$, $\forall \boldsymbol{y} \in \mathcal{R}(A)$.
3. $\mathcal{R}(A^\perp) = \{\boldsymbol{z} | \boldsymbol{z} = A^\top \boldsymbol{y},\ \boldsymbol{y} \in \mathbb{R}^m\} \subset \mathbb{R}^n$ *the row space.*
4. $\mathcal{N}(A) = \{\boldsymbol{x} | A\boldsymbol{x} = 0\}$ *the null space.*

Theorem 5.3.8. *The following relations hold:*

1. $\mathcal{R}(A)^\perp = \mathcal{N}(A^\top)$. *Thus,* $\mathbb{R}^m = \mathcal{R}(A) \oplus \mathcal{N}(A^\top)$.
2. $\mathcal{R}(A^\top)^\perp = \mathcal{N}(A)$. *Thus,* $\mathbb{R}^n = \mathcal{R}(A^\top) \oplus \mathcal{N}(A)$.

In other words, $\mathcal{R}^m$ *can be written as a direct sum of the range of* A *and the null space of* $A^\top$, *and an analogous result holds for* $\mathbb{R}^n$.

Proof. Let $\boldsymbol{z} \in \mathcal{R}(A)^\perp$. Then for any $\boldsymbol{x} \in \mathcal{R}^n$, we have $A\boldsymbol{x} \in \mathcal{R}(A)$, so by definition we have

$$0 = (A\boldsymbol{x})^\top \boldsymbol{z} = \boldsymbol{x}^\top (A^\top \boldsymbol{z}).$$

Since this is true for all $\boldsymbol{x}$, it follows that $A^\top \boldsymbol{z} = 0$, which means $\boldsymbol{z} \in \mathcal{N}(A^\top)$ and therefore $\mathcal{R}(A)^\perp \subset \mathcal{N}(A^\top)$.

On the other hand, let $\boldsymbol{y} \in \mathcal{R}(A)$ and $\boldsymbol{z} \in \mathcal{N}(A^\top)$. Then we have

$$\boldsymbol{y}^\top \boldsymbol{z} = (A\boldsymbol{x})^\top \boldsymbol{z} = \boldsymbol{x}^\top (A^\top \boldsymbol{z}) = \boldsymbol{x}^\top 0 = 0$$

which means that $\boldsymbol{z} \in \mathcal{R}(A)^\perp$. Thus also $\mathcal{N}(A^\top) \subset \mathcal{R}(A)^\perp$.

The second statement is verified in the same way. □

One way to understand the problem of finding least squares approximations is via projections onto the above subspaces. Recall that $P : \mathbb{R}^n \to \mathbb{R}^n$ is a projector onto a subspace $V \subset \mathbb{R}^n$ if $P^2 = P$ and $\mathcal{R}(P) = V$. Additionally, if P is symmetric, then it is an orthogonal projector. The following lemma shows a few properties of orthogonal projectors.

Lemma 5.3.9. *Let* $V \neq 0$ *be a non-trivial subspace of* $\mathbb{R}^n$. *If* P_1 *is an orthogonal projector onto* V, *then* $\|P_1\|_2 = 1$ *and* $P_1 \boldsymbol{v} = \boldsymbol{v}$ *for all* $\boldsymbol{v} \in V$. *Moreover, if* P_2 *is another orthogonal projector onto* V, *then* $P_1 = P_2$.

Proof. We first show that $P_1 \boldsymbol{v} = \boldsymbol{v}$ for all $\boldsymbol{v} \in V$. Let $0 \neq \boldsymbol{v} \in V$. Since $V = \mathcal{R}(P_1)$, there exists $\boldsymbol{x} \in \mathbb{R}^n$ such that $\boldsymbol{v} = P_1 \boldsymbol{x}$. Thus,

$$P_1 \boldsymbol{v} = P_1^2 \boldsymbol{x} = P_1 \boldsymbol{x} = \boldsymbol{v}$$

Taking norms on both sides gives

$$\|P_1 \boldsymbol{v}\|_2 = \|\boldsymbol{v}\|_2$$

which implies $\|P_1\|_2 \geq 1$. To show that $\|P_1\|_2 = 1$, let $\boldsymbol{y} \in \mathbb{R}^n$ be arbitrary. Then

$$\|P_1 \boldsymbol{y}\|_2^2 = \boldsymbol{y}^\top P_1^\top P_1 \boldsymbol{y} = \boldsymbol{y}^\top P_1^2 \boldsymbol{y} = \boldsymbol{y}^\top P_1 \boldsymbol{y}.$$

The Cauchy-Schwarz inequality now gives

$$\|P_1\boldsymbol{y}\|_2^2 \leq \|\boldsymbol{y}\|_2\|P_1\boldsymbol{y}\|_2$$

which shows, upon dividing both sides by $\|P_1\boldsymbol{y}\|_2$, that $\|P_1\boldsymbol{y}\|_2 \leq \|\boldsymbol{y}\|_2$. Hence, we conclude that $\|P_1\|_2 = 1$.

Now let P_2 be another orthogonal projector onto V. To show equality of the two projectors, we show that $(P_1 - P_2)\boldsymbol{y} = 0$ for $\boldsymbol{y} \in \mathbb{R}^n$. Indeed, we have

$$\begin{aligned}\|(P_1 - P_2)\boldsymbol{y}\|_2^2 &= \boldsymbol{y}^\top(P_1 - P_2)^\top(P_1 - P_2)\boldsymbol{y} \\ &= \boldsymbol{y}^\top(P_1 - P_2)^2\boldsymbol{y} \\ &= \boldsymbol{y}^\top(P_1 - P_1P_2 - P_2P_1 + P_2)\boldsymbol{y} \\ &= \boldsymbol{y}^\top(I - P_1)P_2\boldsymbol{y} + \boldsymbol{y}^\top(I - P_2)P_1\boldsymbol{y}\end{aligned} \tag{5.20}$$

But for any $\boldsymbol{v} \in V$, we have

$$(I - P_1)\boldsymbol{v} = \boldsymbol{v} - P_1\boldsymbol{v} = \boldsymbol{v} - \boldsymbol{v} = 0$$

and similarly for $I - P_2$. Since $P_2\boldsymbol{y} \in V$, we have $(I - P_1)P_2\boldsymbol{y} = 0$, so the first term in (5.20) vanishes. Exchanging the roles of P_2 and P_1 shows that the second term in (5.20) also vanishes, so $P_1 = P_2$. □

Thanks to the above lemma, we see that the orthogonal projector onto a given V is in fact unique; we denote this projector by P_V. With the help of the pseudoinverse, we can describe orthogonal projectors onto the fundamental subspaces of A.

Theorem 5.3.10. *(Projectors Onto Fundamental Subspaces)*

$$\begin{aligned}&1.P_{\mathcal{R}(A)} = AA^+ && 2.P_{\mathcal{R}(A^\top)} = A^+A \\ &3.P_{\mathcal{N}(A^\top)} = I - AA^+ && 4.P_{\mathcal{N}(A)} = I - A^+A\end{aligned}$$

Proof. We prove only the first relation; the other proofs are similar. Because of Relation (iii) in Theorem 5.3.6 we have $(AA^+)^\top = AA^+$. Thus $P_{\mathcal{R}(A)}$ is symmetric. Furthermore $(AA^+)(AA^+) = (AA^+A)A^+ = AA^+$ because of (i). Thus $P_{\mathcal{R}(A)}$ is symmetric and idempotent and is therefore an orthogonal projector. Now let $\boldsymbol{y} = A\boldsymbol{x} \in \mathcal{R}(A)$; then $P_{\mathcal{R}(A)}\boldsymbol{y} = AA^+\boldsymbol{y} = AA^+A\boldsymbol{x} = A\boldsymbol{x} = \boldsymbol{y}$. So elements in $\mathcal{R}(A)$ are projected onto themselves. Finally take $\boldsymbol{z} \perp \mathcal{R}(A) \iff A^\top\boldsymbol{z} = 0$ then $P_{\mathcal{R}(A)}\boldsymbol{z} = AA^+\boldsymbol{z} = (AA^+)^\top\boldsymbol{z} = (A^+)^\top A^\top\boldsymbol{z} = 0$. □

Note that the projectors can be computed using the SVD. Let $U_1 \in \mathbb{R}^{m\times r}$, $U_2 \in \mathbb{R}^{m\times(n-r)}$, $V_1 \in \mathbb{R}^{n\times r}$, $V_2 \in \mathbb{R}^{n\times(n-r)}$ and $\Sigma_r \in \mathbb{R}^{r\times r}$ in the following SVD

$$A = \begin{pmatrix}U_1 & U_2\end{pmatrix}\begin{pmatrix}\Sigma_r & 0 \\ 0 & 0\end{pmatrix}\begin{pmatrix}V_1^\top \\ V_2^\top\end{pmatrix}$$

Then inserting this decomposition into the expressions for the projectors of Theorem 5.3.10 we obtain:

$$\begin{aligned}&1.P_{\mathcal{R}(A)} = U_1U_1^\top && 2.P_{\mathcal{R}(A^\top)} = V_1V_1^\top \\ &3.P_{\mathcal{N}(A^\top)} = U_2U_2^\top && 4.P_{\mathcal{N}(A)} = V_2V_2^\top\end{aligned}$$

5.3.3 Solution of the Linear Least Squares Problem

We are now ready to describe the general solution for the linear least squares problem. We are given a system of equations with more equations than unknowns,

$$A\boldsymbol{x} \approx \boldsymbol{b}$$

In general $\boldsymbol{b}$ will not be in $\mathcal{R}(A)$ and therefore the system will not have a solution. A consistent system can be obtained if we project $\boldsymbol{b}$ onto $\mathcal{R}(A)$:

$$A\boldsymbol{x} = AA^+\boldsymbol{b} \Longleftrightarrow A(\boldsymbol{x} - A^+\boldsymbol{b}) = 0$$

We conclude that $\boldsymbol{x} - A^+\boldsymbol{b} \in \mathcal{N}(A)$. That means

$$\boldsymbol{x} - A^+\boldsymbol{b} = (I - A^+A)\boldsymbol{w}$$

where we have generated an element in $\mathcal{N}(A)$ by projecting an arbitrary vector $\boldsymbol{w}$ onto it. Thus we have shown

Theorem 5.3.11. *(General Least Squares Solution) The general solution of the linear least squares problem $A\boldsymbol{x} \approx \boldsymbol{b}$ is*

$$\boldsymbol{x} = A^+\boldsymbol{b} + (I - A^+A)\boldsymbol{w}, \quad \boldsymbol{w} \text{ arbitrary.} \tag{5.21}$$

Using the expressions for projectors from the SVD we obtain for the general solution

$$\boldsymbol{x} = V_1\Sigma_r^{-1}U_1^\top\boldsymbol{b} + V_2\boldsymbol{c} \tag{5.22}$$

where we have introduced the arbitrary vector $\boldsymbol{c} := V_2^\top\boldsymbol{w}$. Notice that if we calculate $\|\boldsymbol{x}\|_2^2$ using e.g. (5.21), we obtain

$$\begin{aligned}\|\boldsymbol{x}\|_2^2 &= \|A^+\boldsymbol{b}\|_2^2 + 2\boldsymbol{w}^\top \underbrace{(I - A^+A)^\top A^+\boldsymbol{b}}_{=0} + \|(I - A^+A)\boldsymbol{w}\|_2^2 \\ &= \|A^+\boldsymbol{b}\|_2^2 + \|(I - A^+A)\boldsymbol{w}\|_2^2 \geq \|A^+\boldsymbol{b}\|_2^2.\end{aligned}$$

This calculation shows that any solution to the least squares problem must have norm greater than or equal to that of $A^+\boldsymbol{b}$; in other words, the pseudoinverse produces the minimum-norm solution to the least squares problem $A\boldsymbol{x} \approx \boldsymbol{b}$. Thus, we have obtained an algorithm for computing both the general and the minimum norm solution of the linear least squares problem with (possibly) rank deficient coefficient matrix:

If A has full rank (rank(A)=n) then the solution of the linear least squares problem is unique:

$$\boldsymbol{x} = A^+\boldsymbol{b} = V\Sigma^+U^\top\boldsymbol{b}.$$

The matrix A^+ is called pseudoinverse because in the full rank case the solution $A\boldsymbol{x} \approx \boldsymbol{b} \Longrightarrow \boldsymbol{x} = A^+\boldsymbol{b}$ is the analogue of the solution $\boldsymbol{x} = A^{-1}\boldsymbol{b}$ of a linear system $A\boldsymbol{x} = \boldsymbol{b}$ with nonsingular matrix $A \in \mathbb{R}^{n\times n}$.

The general least squares solution presented in Theorem 5.3.11 is also valid for a consistent system of equations $A\boldsymbol{x} = \boldsymbol{b}$ where $m \leq n$, i.e. an underdetermined linear system with fewer equations than unknowns. In this case the $\boldsymbol{x} = V_1\Sigma_r^{-1}U_1^\top\boldsymbol{b}$ solves the problem

$$\min \|\boldsymbol{x}\|_2 \quad \text{subject to } A\boldsymbol{x} = \boldsymbol{b}$$

Algorithm 31 General solution of the linear least squares problem $A\boldsymbol{x} \approx \boldsymbol{b}$

1. Compute the `SVD: [U,S,V]=svd(A)`.
2. Make a rank decision, i.e. choose r such that $\sigma_r > 0$ and $\sigma_{r+1} = \cdots = \sigma_n = 0$. This decision is necessary because rounding errors will prevent the zero singular values from being exactly zero.
3. Set `V1=V(:,1:r), V2= V(:,r+1:n), Sr=S(1:r,1:r), U1=U(:,1:r)`.
4. The solution with minimal norm is `xm=V1*(Sr\U1'*b)`.
5. The general solution is `x=xm+V2*c` with an arbitrary $c \in \mathbb{R}^{n-r}$

5.3.4 SVD and Rank

Consider the matrix

$$A = \begin{pmatrix} -1.9781 & 4.4460 & -0.1610 & -3.8246 & 3.8137 \\ 2.7237 & -2.3391 & 2.3753 & -0.0566 & -4.1472 \\ 1.6934 & -0.1413 & -1.5614 & -1.5990 & 1.7343 \\ 3.1700 & -7.1943 & -4.5438 & 6.5838 & -1.1887 \\ 0.3931 & -3.1482 & 3.1500 & 3.6163 & -5.9936 \\ -7.7452 & 2.9673 & -0.1809 & 4.6952 & 1.7175 \\ -1.9305 & 8.9277 & 2.2533 & -10.1744 & 5.2708 \end{pmatrix}$$

The singular values are computed by `svd(A)` as

$$\begin{aligned} \sigma_1 &= 20.672908496836218 \\ \sigma_2 &= 10.575440102610981 \\ \sigma_3 &= 8.373932796689537 \\ \sigma_4 &= 0.000052201761324 \\ \sigma_5 &= 0.000036419750608 \end{aligned}$$

and we can observe a gap between σ_3 and σ_4. The two singular values σ_4 and σ_5 are about 10^5 times smaller than σ_3. Clearly the matrix A has rank 5. However, if the matrix elements comes from measured data and if the measurement uncertainty is $5 \cdot 10^{-5}$, one could reasonably suspect that A is in fact a perturbed representation of a rank-3 matrix, where the perturbation is due to measurement errors of the order of $5 \cdot 10^{-5}$. Indeed, reconstructing the matrix by setting $\sigma_4 = \sigma_5 = 0$ we get

```
[U,S,V]=svd(A);
S(4,4)=0; S(5,5)=0;
B=U*S*V'
```

We see no difference between A and B when using the Matlab-format short because, according to Theorem 5.3.4, we have $\|A - B\|_2 = \sigma_4 = 5.220210^{-5}$. Thus, in this case, one might as well declare that the matrix has numerical rank 3.

In general, if there is a distinct gap between the singular values one can define a threshold and remove small nonzero singular values which only occur because of the rounding effects of finite precision arithmetic or maybe because of measurement errors in the data. The default tolerance in Matlab for the rank command is `tol=max(size(A)) *eps(norm(A))`. Smaller singular values are considered to be zero.

There are full rank matrices whose singular values decrease to zero with no distinct gap. It is well known that the Hilbert matrix is positive definite. The Matlab statement `eig(hilb(14))` gives us as smallest eigenvalue (which is here equal to σ_{14}) the negative value -3.483410^{-17}! With `svd(hilb(14))` we get $\sigma_{14} = 3.900710^{-18}$ which is also not correct. Using the Maple commands

```
with(LinearAlgebra);
Digits:=40;
n:=14;
A:=Matrix(n,n);
for i from 1 to n do
for j from 1 to n do
A[i,j]:=1/(i+j-1);
end do;
end do;
evalm(evalf(Eigenvalues(A)));
```

we obtain for $n = 14$ the singular values

$$
\begin{aligned}
&1.8306\\
&4.1224 \cdot 10^{-01}\\
&5.3186 \cdot 10^{-02}\\
&4.9892 \cdot 10^{-03}\\
&3.6315 \cdot 10^{-04}\\
&2.0938 \cdot 10^{-05}\\
&9.6174 \cdot 10^{-07}\\
&3.5074 \cdot 10^{-08}\\
&1.0041 \cdot 10^{-09}\\
&2.2100 \cdot 10^{-11}\\
&3.6110 \cdot 10^{-13}\\
&4.1269 \cdot 10^{-15}\\
&2.9449 \cdot 10^{-17}\\
&9.8771 \cdot 10^{-20}
\end{aligned}
$$

For `A=hilb(14)` Matlab computes `rank(A)=12` which is mathematically not correct but a reasonable numerical rank for such an ill-conditioned matrix. The SVD is the best tool to assign numerically a rank to a matrix.

5.4 Algorithms Using Orthogonal Matrices

5.4.1 QR Decomposition

Consider a matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ and rank(A)=n. Then there exists the Cholesky decomposition of $A^\top A = R^\top R$ where R is an upper triangular matrix. Since R is non- singular we can write $R^{-\mathrm{T}}A^\top AR^{-1} = I$ or

$$(AR^{-1})^\top (AR^{-1}) = I$$

This means that the matrix $Q_1 := AR^{-1}$ has orthogonal columns. Thus we have found the QR decomposition

$$A = Q_1 R \tag{5.23}$$

Here, $Q_1 \in \mathbb{R}^{m\times n}$ and $R \in \mathbb{R}^{n\times n}$. We can always augment Q_1 to an $m \times m$ orthogonal matrix $Q := [Q_1, Q_2]$ and instead consider the decomposition

$$A = [Q_1, Q_2]\begin{pmatrix} R \\ 0 \end{pmatrix} \tag{5.24}$$

The decomposition (5.24) is what Matlab computes with the command `[Q,R] = qr(A)`. The decomposition (5.24) exists for any matrix A with full column rank.

We have shown in Section 5.2 that for an orthogonal matrix B the problems

$$A\boldsymbol{x} \approx \boldsymbol{b} \quad \text{and} \quad BA\boldsymbol{x} \approx B\boldsymbol{b}$$

are equivalent. Now if $A = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$, then $B = Q^\top$ is orthogonal and $A\boldsymbol{x} \approx \boldsymbol{b}$ and $Q^\top A\boldsymbol{x} \approx Q^\top \boldsymbol{b}$ are equivalent. But

$$Q^\top A = \begin{pmatrix} R \\ 0 \end{pmatrix}$$

and the equivalent system becomes

$$\begin{pmatrix} R \\ 0 \end{pmatrix}\boldsymbol{x} \approx \begin{pmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \end{pmatrix}, \quad \text{with } \begin{pmatrix} \boldsymbol{y_1} \\ \boldsymbol{y_2} \end{pmatrix} = Q^\top \boldsymbol{b}$$

The square of the norm of the residual,

$$\|\boldsymbol{r}\|_2^2 = \|\boldsymbol{y_1} - R\boldsymbol{x}\|_2^2 + \|\boldsymbol{y_2}\|_2^2$$

is obviously minimal for $\hat{\boldsymbol{x}}$ where

$$R\hat{\boldsymbol{x}} = \boldsymbol{y_1}, \quad \hat{\boldsymbol{x}} = R^{-1}\boldsymbol{y_1}, \quad \text{and} \quad \min \|\boldsymbol{r}\|_2 = \|\boldsymbol{y_2}\|_2$$

This approach is numerically preferable to the normal equations, since it does not change the condition number. This can be seen by noting that the singular values are

not affected by orthogonal transformations: If $A = U\Sigma V^\top = Q\begin{pmatrix} R \\ 0 \end{pmatrix}$ then the singular value decomposition of $\begin{pmatrix} R \\ 0 \end{pmatrix}$ is

$$\begin{pmatrix} R \\ 0 \end{pmatrix} = (Q^\top U)\Sigma V^\top$$

and thus R and A have the same singular values, which leads to

$$\kappa(A) = \kappa(R)$$

In the following section we will show how to compute the QR decomposition.

5.4.2 Method of Householder

Definition 5.4.1. *(Elementary Householder Matrix) An elementary Householder matrix is a matrix of the form $P = I - \boldsymbol{u}\boldsymbol{u}^\top$ with $\|\boldsymbol{u}\|_2 = \sqrt{2}$*

Elementary Householder matrices have the following properties:

1. P is symmetric.
2. P is orthogonal, since

$$P^\top P = (I - \boldsymbol{u}\boldsymbol{u}^\top)(I - \boldsymbol{u}\boldsymbol{u}^\top) = I - \boldsymbol{u}\boldsymbol{u}^\top - \boldsymbol{u}\boldsymbol{u}^\top + \boldsymbol{u}\underbrace{\boldsymbol{u}^\top\boldsymbol{u}}_{2}\boldsymbol{u}^\top = I$$

3. $P\boldsymbol{u} = -\boldsymbol{u}$ and if $\boldsymbol{x}\perp\boldsymbol{u}$ then $P\boldsymbol{x} = \boldsymbol{x}$. If $\boldsymbol{y} = \alpha\boldsymbol{x} + \beta\boldsymbol{u}$ then $P\boldsymbol{y} = \alpha\boldsymbol{x} - \beta\boldsymbol{u}$. Thus P is a reflection across the hyperplane $\boldsymbol{u}^\top\boldsymbol{x} = 0$.

P will be used to solve the following basic problem: Given a vector $\boldsymbol{x}$, find an orthogonal matrix P such that

$$P\boldsymbol{x} = \begin{pmatrix} \sigma \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \sigma\boldsymbol{e_1}$$

Since P is orthogonal we have $\|P\boldsymbol{x}\|_2^2 = \|\boldsymbol{x}\|_2^2 = \sigma^2$ thus $\sigma = \pm\|\boldsymbol{x}\|_2$. Furthermore $P\boldsymbol{x} = (I - \boldsymbol{u}\boldsymbol{u}^\top)\boldsymbol{x} = \boldsymbol{x} - \boldsymbol{u}(\boldsymbol{u}^\top\boldsymbol{x}) = \sigma\boldsymbol{e_1}$, thus $\boldsymbol{u}(\boldsymbol{u}^\top\boldsymbol{x}) = \boldsymbol{x} - \sigma\boldsymbol{e_1}$ and we obtain by normalizing

$$\boldsymbol{u} = \frac{\boldsymbol{x} - \sigma\boldsymbol{e_1}}{\|\boldsymbol{x} - \sigma\boldsymbol{e_1}\|_2}\sqrt{2}$$

We can still choose the sign of σ, and we choose it such that no cancellation occurs in computing $\boldsymbol{x} - \sigma\boldsymbol{e_1}$

$$\sigma = \begin{cases} \|\boldsymbol{x}\|_2, & x_1 < 0 \\ -\|\boldsymbol{x}\|_2, & x_1 \geq 0 \end{cases}$$

For the denominator we get$\|\boldsymbol{x} - \sigma\boldsymbol{e_1}\|_2^2 = (\boldsymbol{x} - \sigma\boldsymbol{e_1})^\top(\boldsymbol{x} - \sigma\boldsymbol{e_1}) = \boldsymbol{x}^\top\boldsymbol{x} - 2\sigma\boldsymbol{e_1}^\top\boldsymbol{x} + \sigma^2$. Note that $-2\sigma\boldsymbol{e_1}^\top = 2|x_1|\|\boldsymbol{x}\|_2$, so the calculations simplify and we get

$$\boldsymbol{u} = \frac{\boldsymbol{x} - \sigma\boldsymbol{e_1}}{\sqrt{\|\boldsymbol{x}\|_2(|x_1| + \|\boldsymbol{x}\|_2)}}$$

In order to apply this basic construction for the computation of the QR decomposition, we construct a sequence of n elementary matrices P_i:

$$P_i = \begin{pmatrix} I & 0 \\ 0 & I - \boldsymbol{u_i}\boldsymbol{u_i}^\top \end{pmatrix}$$

We choose $\boldsymbol{u_i} \in \mathbb{R}^{m-i+1}$ such that zeros are introduced in the $i-$th column of A below the diagonal when multiplying P_iA. We obtain after n steps

$$P_nP_{n-1}\cdots P_1A = \begin{pmatrix} R \\ 0 \end{pmatrix}$$

and, because each P_i is symmetric, Q becomes

$$Q = (P_nP_{n-1}\cdots P_1)^\top = P_1P_2\cdots P_n$$

If we store the new diagonal elements (which are the diagonal of R) in a separate vector d we can store the Householder vectors $\boldsymbol{u_i}$ in the same location where we introduce zeros in A. This leads to the following *implicit QR factorization* algorithm:

Algorithm 32 Householder QR Decomposition

```
function [A,d]=HouseholderQR(A);
% HOUSEHOLDERQR computes the QR-decomposition of a matrix
% [A,d]=HouseholderQR(A) computes an implicit QR-decomposition A=QR
% using Householder transformations. The output matrix A contains
% the Householder vectors u and the upper triangle of R. The
% diagonal of R is stored in the vector d.

[m,n]=size(A);
for j=1:n,
s=norm(A(j:m,j));
if s==0, error('rank(A)<n'), end
if A(j,j)>=0, d(j)=-s; else d(j)=s; end
fak=sqrt(s*(s+abs(A(j,j))));
A(j,j)=A(j,j)-d(j);
A(j:m,j)=A(j:m,j)/fak;
if j<n, % transformation of the rest of the matrix G:=G-u*(u'*G)
A(j:m,j+1:n)=A(j:m,j+1:n)-A(j:m,j)*(A(j:m,j)'*A(j:m,j+1:n));
end
end
```

Algorithm `HouseholderQR` computes an upper triangular matrix R_h which is very similar to R_c obtained by the Cholesky decomposition $A^\top A = R_c^\top R_c$. The only difference is that R_h may have negative elements in the diagonal, whereas in R_c the diagonal entries are positive. Let D be a diagonal matrix with

$$d_{ii} = \begin{cases} 1 & r_{ii}^h > 0 \\ -1 & r_{ii}^h < 0 \end{cases}$$

then $R_c = DR_h$. The matrix Q is only implicitly available through the Householder vectors $\boldsymbol{u}_i$. This is not an issue because it is often unnecessary to compute and store the matrix Q explicitly; in many cases, Q is only needed as an operator that acts on vectors by multiplication. Using the implicit representation we can, for instance, compute the transformed right hand side of the least square equations $\boldsymbol{y} = Q^\top \boldsymbol{b}$ by applying the reflections $\boldsymbol{y} = P_n P_{n-1} \cdots P_1 \boldsymbol{b}$. This procedure is numerically preferable to forming the explicit matrix Q and then multiplying with $\boldsymbol{b}$.

Algorithm 33 Transformation $z = Q^\top y$

```
function z=HouseholderQTy(A,y);
% HOUSEHOLDERQTY applies Householder reflections transposed
% z=HouseholderQTy(A,y); computes z=Q'y using the Householder
% reflections Q stored as vectors in the matrix A by
% A=HousholderQR(A)
[m,n]=size(A); z=y;
for j=1:n,
z(j:m)=z(j:m)-A(j:m,j)*(A(j:m,j)'*z(j:m));
end;
```

If we wish to compute $\boldsymbol{z} = Q\boldsymbol{y}$ then because $Q = P_1 P_2 \cdots P_n$ it is sufficient to reverse the order in the for-loop:

Algorithm 34 Transformation $z = Qy$

```
function z=HouseholderQy(A,y);
% HOUSEHOLDERQY applies Householder reflections
% z=HouseholderQy(A,y); computes z=Qy using the Householder
% reflections Q stored as vectors in the matrix A by
% A=HousholderQR(A)
[m,n]=size(A); z=y;
for j=n:-1:1,
z(j:m)=z(j:m)-A(j:m,j)*(A(j:m,j)'*z(j:m));
end;
```

Example 5.4.1. *We compute the QR decomposition of a section of the Hilbert matrix. We also compute the explicit matrix Q by applying the Householder transformations to the column vectors of the identity matrix:*

```
m=8;n=6;
H=hilb(m); A=H(:,1:n);
[AA,d]=HouseholderQR(A)
Q=[];
for i=eye(m), % compute Q explicit
z=HouseholderQy(AA,i); Q=[Q z];
end
R=triu(AA);
for i=1:n, R(i,i)=d(i); end % add diagonal to R
[norm(Q'*Q-eye(m)) norm(A-Q*R)]
[q,r]=qr(A); % compare with Matlab qr
[norm(q'*q-eye(m)) norm(A-q*r)]
```

The resulting matrix after the statement `[AA,d]=HouseholderQR(A)` *contains the Householder vectors and the upper part of R. The diagonal of R is stored in the vector* $\boldsymbol{d}$*:*

```
AA =
1.3450 -0.7192 -0.5214 -0.4130 -0.3435 -0.2947
0.3008  1.1852 -0.1665 -0.1612 -0.1512 -0.1407
0.2005  0.3840 -1.0188  0.0192  0.0233  0.0254
0.1504  0.3584  0.2452 -1.3185  0.0015  0.0023
0.1203  0.3242  0.3945 -0.4332 -1.0779  0.0001
0.1003  0.2925  0.4703 -0.2515  0.0111 -1.3197
0.0859  0.2651  0.5059 -0.0801  0.3819 -0.5078
0.0752  0.2417  0.5190  0.0641  0.8319  0.0235
d =
-1.2359 -0.1499 0.0118 0.0007 0.0000 0.0000
ans =
1.0e-15 *
0.6834 0.9272
ans =
1.0e-15 *
0.5721 0.2461
```

We see that the results (orthogonality of Q and reproduction of A by QR) compare well with the Matlab function `qr`*.*