VSB – Technical University of Ostrava
Faculty of Civil Engineering

Algorithmization of Engineering Computations

Prof. Ing. Martin Krejsa, Ph.D.

Learning materials

Ostrava, 2024

Learning materials of the **Algorithmization of Engineering Computations**

Study program: Civil Engineering

Title of the branch: Building Structures

Key Words:

Algorithm, Matlab, matrix, vector, function, polynomial function, algebraic nonlinear equation, linear equations, numerical integration, differential equation, iteration, interpolation, approximation.

Introduction

The study materials for „Algorithmization of Engineering Computations“ presented in this thesis are designed for Bachelor’s level civil engineering students specializing in the Building Structures. The aim of these materials is to familiarize students with basic programming techniques and the creation of algorithms for solving engineering tasks in the construction industry.

The MATLAB interactive computing system is a logical choice of tool for the purposes of these materials, allowing the application of algorithmic procedures and creation of simple calculation tools without complicated user interface programming. MATLAB is a user-friendly tool for executing mathematical and numerical procedures and includes a library with a wide range of functions in addition to standard programming tools. Results can easily be displayed as graphs.

The choice of application environment for the materials took into account compatibility with other programming and advanced calculation algorithm subjects taught at the Department of Structural Mechanics.

In Ostrava, February 2024 Prof. Ing. Martin Krejsa, Ph.D.

Table of Contents

Chapter 1

Matlab

Objectives

This chapter introduces:

- the **Matlab user interface**,
- definitions for variables and management of these variables in the Matlab environment,
- an example of creating an algorithm using **logical decision making**.

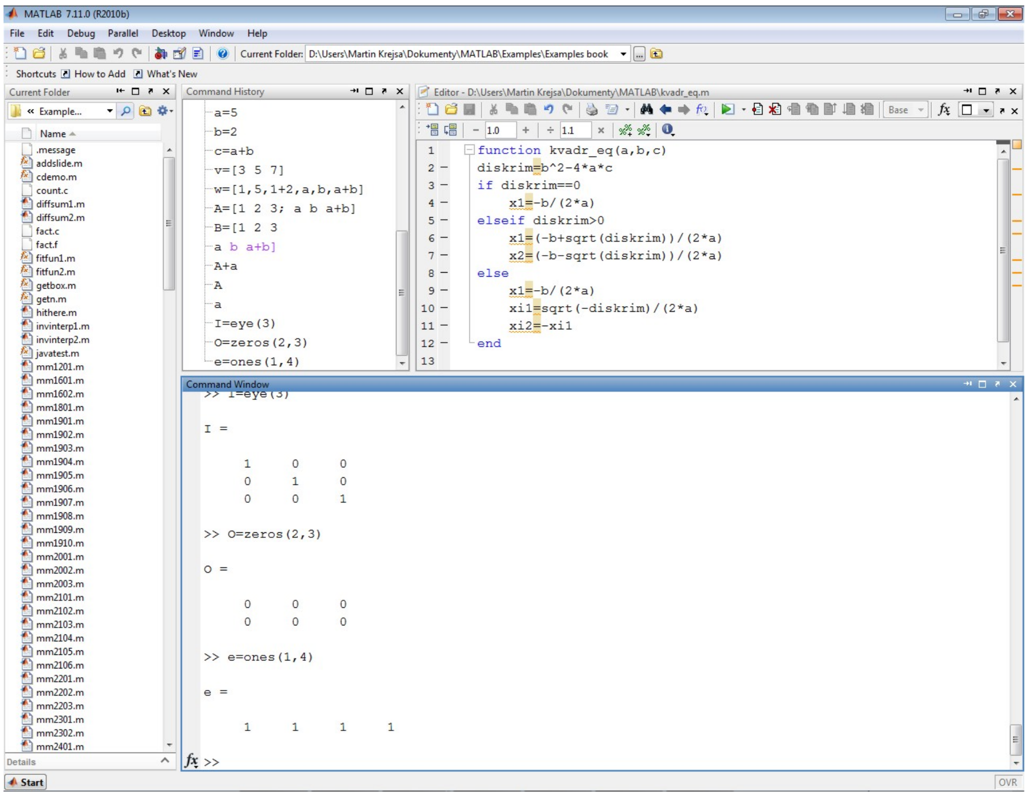
MATLAB [5] (see Fig. 1.1) is a programming environment which uses scripting language for scientific and technical numerical calculations, modeling and algorithm design. Simulink is an extension of MATLAB for simulating and modelling dynamic systems. It uses Matlab algorithms for the numerical solution of mainly nonlinear differential equations.

The name MATLAB is an abbreviation of MATrix LABoratory, arising from matrices being the key data structures used in MATLAB calculations.

For those interested:

MATLAB is commercial software, however. A welcome alternative for creating algorithms using compatible commands and scripts is Octave software. Octave is free to downloaded, for example from [2, 7]. Octave is open-source and is therefore available in a range of versions for Unix (Linux), Mac OS or MS Windows. The program's somewhat bare user interface contains only a command line in text mode for entering a specific commands and listing resulting values.

The MATLAB system itself contains a number of commands for managing variables, basic algebraic operations, calculations with vectors and matrices, and commands for higher mathematical operators. Because it has a large number of options,



```
a=5
b=2
c=a+b
v=[3 5 7]
w=[1,5,1+2,a,b,a+b]
A=[1 2 3; a b a+b]
B=[1 2 3
a b a+b]
A+a
A
a
I=eye(3)
O=zeros(2,3)
e=ones(1,4)
```

```
function kvadr_eq(a,b,c)
diskrim=b^2-4*a*c
if diskrim==0
    x1=-b/(2*a)
elseif diskrim>0
    x1=(-b+sqrt(diskrim))/(2*a)
    x2=(-b-sqrt(diskrim))/(2*a)
else
    x1=-b/(2*a)
    xi1=sqrt(-diskrim)/(2*a)
    xi2=-xi1
end
```

```
>> I=eye(3)

I =

     1     0     0
     0     1     0
     0     0     1

>> O=zeros(2,3)

O =

     0     0     0
     0     0     0

>> e=ones(1,4)

e =

     1     1     1     1

>>
```

Fig. 1.1 The MATLAB workspace.

Variable name	*Description*
`help`	lists the help contents
`help name`	displays the help for the functionality specified by name
`lookfor keyword`	lists all help items for the entered keyword
`info`	information about MathWorks

Tab. 1.1 Overview of help commands.

users are encouraged to make use of the extensive help files accessed via respective commands (Table 1.1).

MATLAB remembers the format of the numeric values it works with, but the user can change the format for displaying values with the `format` command and relevant specifications (Table 1.2).

1.1 Entering Variables

The basic variable type in MATLAB is a matrix. A simple variable containing one value is represented as a $[1, 1]$ matrix. Variables are entered with commands defined

Command	*Command description*	*The number π in the given format*
format short	fixed decimal point, 5 displayed digits	3.1416
format long	fixed decimal point, 15 displayed digits (double variable type), or 7 (single type)	3.141592653589793
format shorte	floating decimal point, 5 displayed digits	3.1416e+000
format longe	floating decimal point, 15 displayed digits (double variable type), or 7 (single type)	3.141592653589793e+000
format shorteng	engineering format, 5 displayed digits and exponent	3.1416e+000
format longeng	engineering format, 16 displayed digits and exponent	3.14159265358979e+000
format rat	results are displayed as a fraction (default setting)	355/113
format hex	results are displayed as a hexadecimal value	400921fb54442d18

Tab. 1.2 Overview of possible format types for displaying numerical values.

in the program's command line. If the command ends with a semicolon, the result of the command is not displayed. If the entry includes a basic arithmetic operation, certain symbols are used (Table 1.3). A full stop (or period) is used as a decimal point.

A variable is defined automatically by assigning its value with the equality sign. For example:

```
a=5
b=2
c=a+b
d='Result'
```

Predefined special variables can be used to assign values to variables (Table 1.4). Some elementary functions have an input parameter (Table 1.5).

1.2 Vectors and Matrices

Square brackets denote a vector. Spaces or commas separate elements in a line. For example:

Operation	*Symbol*	*Example*
Sum	`+`	`4+11`, `a+b`
Subtraction	`-`	`18-5`, `a-b`
Product	`*`	`7.13*5`, `a*b`
Fraction ($\frac{1}{8}$)	`/` or `\`	`1/8` = `8\1`
Power (2^8)	`^`	`2^8`

Tab. 1.3 List of symbols of arithmetic operations.

Variable name	*Description*
`ans`	variable containing the result of an arithmetic operation
`pi`	Ludolphine number π
`inf`	infinity ∞
`nargin`	number of input parameters of the given function
`nargout`	number of output parameters of the given function
`eps`	the smallest usable number in this format
`realmin`	the absolute smallest usable positive real number
`realmax`	the absolute largest usable positive real number

Tab. 1.4 List of predefined variables.

Name of the function	*Description*
`abs(x)`	absolute value of x
`cos(x)`, `sin(x)`, `tan(x)`	trigonometric function (x parameter in radians)
`acos(x)`, `asin(x)`, `atan(x)`	inverse trigonometric functions
`exp(x)`	exponential function (e^x)
`log(x)`	natural logarithm x
`log10(x)`	decimal logarithm x
`sqrt(x)`	square root x

Tab. 1.5 List of elementary functions.

```
u=[3 5 7]
v=[1,5,1+2,a,b,a+b]
w=[0,pi,2*pi]
```

To create vectors, more advanced techniques are applied (Table 1.6).

Matrices are denoted in a similar way to vectors, except that rows are separated with a semicolon or the `<Enter>` key. For example:

```
A=[1 2 3; a b a+b]
B=[1 2 3
a b a+b]
```

Command	*Description*	*Result*
`x=[1:5]`	creates row vector **x**, starting with 1, ending with 5, adding the value of 1	`x=[1,2,3,4,5]`
`x=[2:3:11]`	creates row vector **x**, starting with 2, ending with 11, adding the value of 3	`x=[2,5,8,11]`
`x=linspace(1,25,5)`	creates row vector **x**, starting with 1, ending with 25, the vector contains 5 elements	`x=[1,7,13,19,25]`

Tab. 1.6 Commands for constructing vectors.

Comment 1.1. The names of variables in Matlab are case-sensitive. The command

```
A+a
```

generates the following output for the matrices **A** and variables **a**:

```
ans =
     6     7     8
    10     7    12
```

i.e., to each element of the matrix **A** the content of variable **a**, i.e., 5, is added.

MATLAB standard functions can be used to directly generate matrices or vectors of specific dimensions, for example:

```
I=eye(3)
O=zeros(2,3)
e=ones(1,4)
```

In the first case, a square matrix with values 1 on the diagonal and 0 outside the diagonal is generated. The latter two generate matrices or vectors containing the values 0 or 1, respectively, in all of their elements.

1.2.1 Access to Matrices and Vectors

Matrices let us reference individual elements. Table 1.7 lists the possible sample reference variations for the above matrix `A=[1 2 3; a b a+b]` (i.e., containing `[1 2 3; 5 2 7]`).

Command	*Description*	*Result*
`A(2,1)`	element on the 2nd row and 1st column of matrix **A**	5
`A(2,[1 3])`	1st and 3rd elements from the 2nd row of matrix **A**	$5, 7$
`A(1,1:3)`	1st, 2nd and 3rd elements on the 1st row of the matrix **A**	$1, 2, 3$
`A(1,1:end)`	as above	$1, 2, 3$
`A(1,:)`	as above	$1, 2, 3$
`A(2,1:2:3)`	1st and 3rd elements from the 2nd row of matrix **A**, `a:b:c` defines the arithmetic sequence where the 1st element equals `a`, the last element equals `c`, and `b` is the difference between adjacent elements (does not need to be stated if it equals 1)	$5, 7$
`A(1:end,2:end)`	2nd and 3rd elements from both rows (submatrix)	$2, 3; 2, 7$

Tab. 1.7 Commands for access to matrix and vector elements.

1.2.2 Matrix Operations

Matrix operations can be performed on both vectors and matrices. Matrices are transposed with the ' (apostrophe) operator. For example, the command `A'` transposes the original matrix **A**:

```
ans =
   1    5
   2    2
   3    7
```

To perform operations between individual elements of a matrix or vector, the `.` (full stop) sign is placed in front of the operator. For example, for the vector `u=[3 5 7]`, the result of the `u*u'` command is:

```
ans =  83
```

whereas entering the `u.*u` command displays the following vector:

```
ans =
     9    25    49
```

Similarly, other element-by-element operations can be performed with vectors and matrices (Table 1.8) for generally defined vectors $a = a_1, a_2, \ldots, a_n$; $b = b_1, b_2, \ldots, b_n$ and scalar c.

Several other matrix functions can also be performed; Table 1.9 lists the basic functions.

Operation	*Command*	*Result*
scalar sum	`a+c`	$a_1 + c, a_2 + c, \ldots, a_n + c$
scalar product	`a*c=a.*c`	$a_1 \cdot c, a_2 \cdot c, \ldots, a_n \cdot c$
vector sum	`a+b`	$a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n$
vector product	`a.*b`	$a_1 \cdot b_1, a_2 \cdot b_2, \ldots, a_n \cdot b_n$
vector division from the left	`a./b`	$a_1/b_1, a_2/b_2, \ldots, a_n/b_n$
vector division from the right	`a.\b`	$a_1\backslash b_1, a_2\backslash b_2, \ldots, a_n\backslash b_n$
scalar exponentiation	`a.^c`	$a_1^c, a_2^c, \ldots, a_n^c$
scalar exponentiation	`c.^a`	$c^{a_1}, c^{a_2}, \ldots, c^{a_n}$
vector exponentiation	`a.^b`	$a_1^{b_1}, a_2^{b_2}, \ldots, a_n^{b_n}$

Tab. 1.8 Element-by-element operations with vectors and matrices.

Function name	*Description*
`det(A)`	determinant of matrix **A**
`inv(A)`	calculation of inverse matrix for **A**
`diag(A)`	list of elements on the diagonal of matrix **A**

Tab. 1.9 Basic matrix functions.

1.3 Managing Variables in MATLAB

In MATLAB, variables, vectors and matrices are managed with several commands:

- the `who` or `whos` commands list all the currently defined variables (the latter includes the size of the variable),
- the `size(variable)` command returns the dimensions of the given variable,
- the `clear(variable)` command deletes the given variable from memory,
- the `clear` command without a parameter deletes all the entered variables from memory.

1.4 Using the Graphics Output

In MATLAB's graphics mode, the basic tool is the `plot` command. Its syntax is `plot(x,y,options)`, where `x` and `y` are the coordinates of the points to be drawn. The `options` parameter defines how the graphics output is displayed. It contains characters for defining the color and drawing style of the points and their connecting lines:

- color of the points and their connecting lines: **b** (blue), **g** (green), **r** (red), **c** (blue-green), **m** (purple), **y** (yellow), **k** (black), **w** (white),

- style of the connecting line: `-` (points are connected by a solid line), `:` (points are connected by a dotted line), `-.` (points are connected by a dash-dotted line), `--` (points are connected by a broken line),
- style of point drawing: `*` (points are drawn as stars), `.` (dots), `x` (crosses), `+` (plus signs), `o` (circles), `s` (squares), `d` (diamonds).

Example 1.2. To draw connecting lines for five points with coordinates,

x	10	11	12	13	14	15
y	1	5	7	4	6	10

can be used the command:

```
plot([10:15],[1,5,7,4,6,10],'r-*')
```

Fig. 1.2 displays the final graphics output.

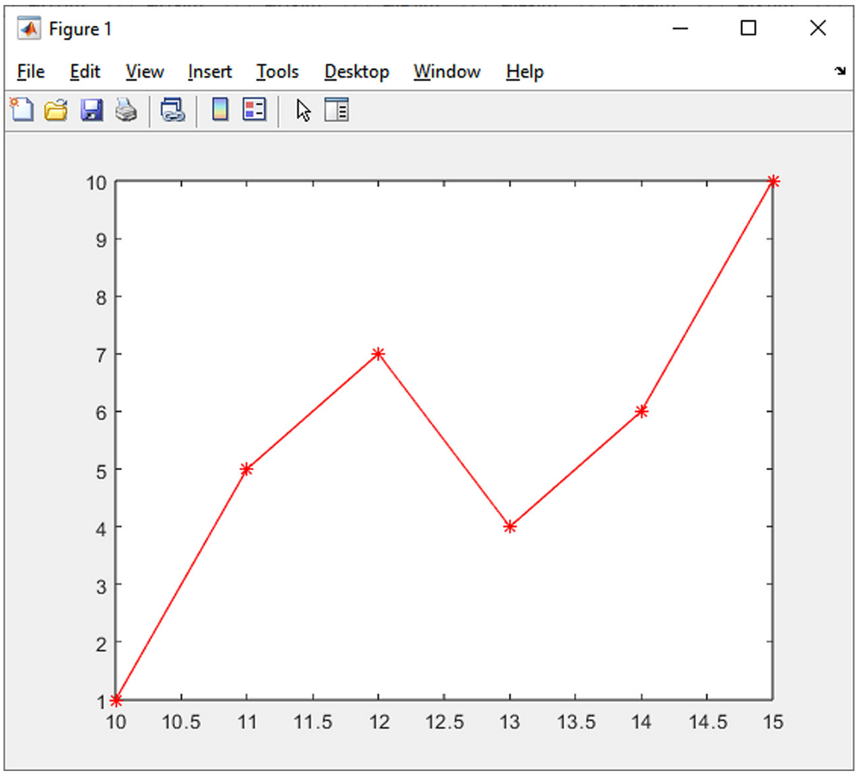


Fig. 1.2 Example of a MATLAB graphics output.

Another useful command in the graphical environment is `hold on`. It displays the graphic outputs of multiple commands in a single window (the `hold off` command is used to return to the original state).

1.4.1 Creating Graphs for Functions

When producing a graph for a function, it is first necessary to discretize x. The corresponding function values $f(x)$ are then determined at the obtained points. The resulting graph of the discretized function is displayed with the `plot` command.

Example 1.3. A graph of the sine function is plotted with the following sequence and the `plot` command:

```
x=linspace(0,2*pi,30);
y=sin(x);
plot(x,y,'b-')
```

A title, perhaps descriptions of the axes, can be added to the graph:

```
title('Graph of the function y=sin(x)');
xlabel('x');
ylabel('sin(x)')
```

Fig. 1.3 depicts the final graph of the sine function.

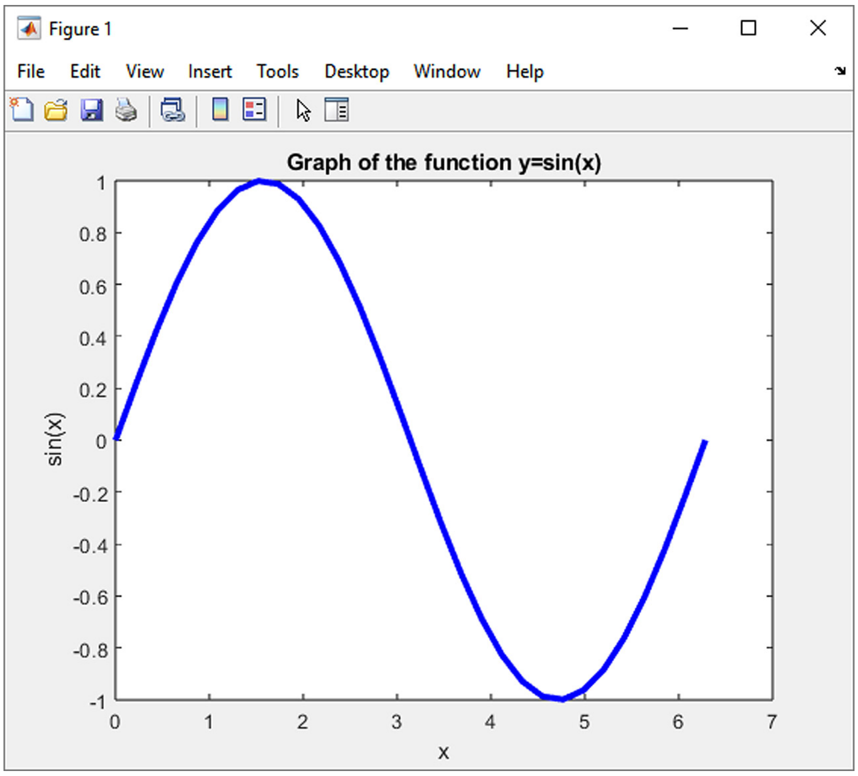


Figure 1
File Edit View Insert Tools Desktop Window Help
Graph of the function y=sin(x)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
sin(x)
0 1 2 3 4 5 6 7
x

Fig. 1.3 Graph of the sine function.

1.5 Creating Scripts

In Matlab, a sequence of commands can be entered in the command line and the system will consecutively process these commands. If the calculation should be performed iteratively, all the commands must be entered again, which is time--consuming and undesirable.

The solution is to save a sequence of Matlab commands in a text file as a script; these files use the extension `*.m` (referred to as an “m-file”). Using the commands stored in the m-file, the calculation is started by specifying the file name (without the extension) while searching the current directory and the directories listed in the `path` system variable.

In this way, a special calculation function (referred to as an “m-function”) is created in a separate file. The function can be called from the command line by entering the name of the function or a list of input parameters in parentheses, separated by commas. **The file name should match the function name in its header.**

The m-file can also contain control commands typical for more advanced programming platforms and include commands for loops and logical conditions.

1.5.1 Loop Commands

Matlab allows the use of two types of programming loop:

- **for loop**, with the syntax

```
for i=initial value:step:end value
    command sequence
end
```

- **while loop** with the syntax:

```
while logical condition
    command sequence
end
```

The creation of an algorithms which uses both types of loop is described in greater detail in the following chapters.

1.5.2 Logical Conditions

The syntax for a sequence of commands divided into three blocks according to the results of a logical condition is

```
if logical condition 1
    command sequence 1
elseif logical condition 2
    command sequence 2
else
    command sequence 3
end
```

To define a condition whose result is either "true" or "false", we use the following logical operators: & (`and`), | (`or`), ˜ (`not`). The supported relational operators are: <, <=, >=, > and == (is equal to).

Example 1.4. Create an m-function and use logical operators to find the solution of a quadratic equation:

```
function quadr_eq(a,b,c)
discrim=b^2-4*a*c
if discrim==0
    x1=-b/(2*a)
elseif discrim>0
    x1=(-b+sqrt(discrim))/(2*a)
    x2=(-b-sqrt(discrim))/(2*a)
else
    x1=-b/(2*a)
    xi1=sqrt(-discrim)/(2*a)
    xi2=-xi1
end
```

Entering input parameters `quadr_eq(5,10,1)` and calling the function called from the MATLAB command line returns two real roots of the quadratic equation:

```
discrim =
    80

x1 =
  -0.105572809000084

x2 =
  -1.894427190999916
```

Calling the function with the `quadr_eq(10,5,1)` parameters obtains the result:

```
discrim =
   -15
```

```
x1 =
  -0.250000000000000

xi1 =
   0.193649167310371

xi2 =
  -0.193649167310371
```

The results can be checked with a special MATLAB function named `roots(c)`, where `c` is a vector of polynomial coefficients sorted in descending order by powers of x. For the general expression of the n-th degree polynomial:

$$f(x) = c_n \cdot x^n + c_{n-1} \cdot x^{n-1} + \cdots + c_1 \cdot x + c_0 \ , \tag{1.1}$$

where vector `c` contains elements $c_n, c_{n-1}, \ldots, c_1, c_0$.

For the first calculation in the example in 1.4, the sequence of commands is

```
c=[5,10,1]
roots(c)
```

with the result

```
ans =
  -1.894427190999916
  -0.105572809000084
```

Comment 1.5. A number of publications provide instructions for using MATLAB and Octave. Some of these are freely available in electronic form (e.g., [12]).

Chapter 2

Fundamentals of Algorithmization

Objectives

This chapter describes:

- the concept of an algorithm,
- the foundations for creating simple algorithms.

An algorithm is a precise instruction or procedure for use in solving a given type of task. Algorithms most often appear in programming and refer to the theoretical principle of solving a problem (as opposed to the exact notation in a specific programming language). Algorithms are used in many scientific branches. A kitchen recipe, for example, may also be understood as a certain kind of algorithm. In a narrower sense, the word algorithm refers only to procedures that satisfy the requirements indicated by the properties of algorithms.

2.1 Properties of Algorithms

The following list of properties of an algorithm is given, for example, in [13]:

- **Finiteness**

  Each algorithm must terminate in a finite number of steps. This number of steps may be arbitrarily large (depending on the range and values of the input data), but it must be finite for each individual input. Procedures which do not satisfy this condition are referred to as computational methods. A special example of an infinite computational method is a reactive process that continuously responds to the surrounding environment. However, some authors think of such procedures also as algorithms.

- **Generality**

  Algorithms do not solve individual, specific problems (e.g., “calculate $3 \cdot 7$”); they solve a general class of similar problems (e.g., “calculate the product of two integers”) which have a wide set of possible inputs.

- **Definiteness**

  Each step of the algorithm must be precisely defined. In each situation, the aim, steps and execution of the algorithm must be clear so that the same results are always obtained for the same input. Since ordinary language generally does not provide absolute precision and unambiguous expression, programming languages have been designed so that the commands executed by algorithms are written with clearly defined meanings. A computation method expressed in a programming language is referred to as a *program*. Some algorithms, however, are not deterministic (e.g., probabilistic algorithms with (pseudo)random selection).

- **Results**

  An algorithm has at least one output representing a quantity expressed in the desired relationship to the specified inputs. It thus provides a solution to the problem described in the algorithm (the algorithm progresses from processing values to producing an output).

- **Simplicity**

  The algorithm consists of a finite number of simple (elementary) steps.

In practice, the algorithms of interest are those which in some way have a certain quality. These algorithms meet various criteria which are measured, for example, by the number of steps required to run the algorithm, simplicity, efficiency or elegance. The effectiveness of algorithms is studied in the branches of computer science called *algorithmic analysis* and *complexity theory*, using methods for selecting the best algorithms from several well-known examples that solve specific problems.

2.2 Elementary Algorithms

The following section examines several elementary algorithms which are the basis of many computing techniques and procedures.

2.2.1 Swapping the Contents of Two Variables

The simplest algorithm may be considered as a procedure which swaps the contents of two variables. A third, auxiliary variable is needed for this operation. For variables a and b and auxiliary variable c, the algorithm takes the form:

```
c=a;
a=b;
b=c;
```

Although this is a truly elementary algorithm, it can be used together with logical operations (Ch. 1.5.2); for example, to sort the vector d in ascending order, using 3 elements (c is again an auxiliary variable):

```
function sort_it(d)
  if length(d)==3
    if d(1)>d(2)
      c=d(1);
      d(1)=d(2);
      d(2)=c;
    end
    if d(2)>d(3)
      c=d(2);
      d(2)=d(3);
      d(3)=c;
    end
    if d(1)>d(2)
      c=d(1);
      d(1)=d(2);
      d(2)=c;
    end
    d
  end
end
```

Example 2.1. Sort the vector `[8 24 2]` in ascending order, using the three-element vector sorting function created above.

Solution. First, it is necessary to assign input values to vector d:

```
d=[8 24 2]
```

We then call the `sort_it` function:

```
sort_it(d)
```

The result is

```
d =
     2     8    24
```

▲

This method of sorting, however, is very inefficient and certainly cannot be considered universal, but it is suitable for explaining the basics of algorithmization. Chapter 5 discusses the quality of algorithms dedicated to sorting vectors.

Chapter 3

Calculation of Function Values

Objectives

This chapter demonstrates:

- the basic approaches to creating algorithms for calculating function values,
- the differences between algorithms in terms of effectiveness,
- the use of **for** loops in algorithms,
- the use of "tabulated functions".

3.1 Calculating the Value of a Polynomial

The following example of one of the most basic algorithms for calculating a polynomial was taken from [11]. It describes the best way to determine the value of the polynomial:

$$f(x) = 2 \cdot x^4 + 3 \cdot x^3 - 3 \cdot x^2 + 5 \cdot x - 1 \tag{3.1}$$

for a specific value of x, e.g., $x = \frac{1}{2}$. An algorithm with the least number of mathematical operations is viewed as the best method of calculation.

Method 1:

The first method directly determines the required value:

$$f(x = \frac{1}{2}) = 2 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} + 3 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} - 3 \cdot \frac{1}{2} \cdot \frac{1}{2} + 5 \cdot \frac{1}{2} - 1 = \frac{5}{4}\,. \tag{3.2}$$

In this calculation, 10 multiplication and 4 sum/difference operations are performed.

Method 2:

A somewhat more advantageous solution is a procedure in which the powers of the input parameter x are determined in a step-by-step manner:

$$\frac{1}{2} \cdot \frac{1}{2} = \left(\frac{1}{2}\right)^2 \tag{3.3}$$

$$\left(\frac{1}{2}\right)^2 \cdot \frac{1}{2} = \left(\frac{1}{2}\right)^3 \tag{3.4}$$

$$\left(\frac{1}{2}\right)^3 \cdot \frac{1}{2} = \left(\frac{1}{2}\right)^4 \tag{3.5}$$

Now the final calculation of the polynomial is performed:

$$f(x = \frac{1}{2}) = 2 \cdot \left(\frac{1}{2}\right)^4 + 3 \cdot \left(\frac{1}{2}\right)^3 - 3 \cdot \left(\frac{1}{2}\right)^2 + 5 \cdot \left(\frac{1}{2}\right) - 1 = \frac{5}{4} \,. \tag{3.6}$$

This algorithm is more efficient than the first, using a total of 7 multiplication operations and the same number of 4 operations for sum/subtraction.

Method 3:

The third method of calculation, known as *Horner's method*, assumes the following modification to the solved polynomial (3.1):

$$\begin{aligned} f(x) &= -1 + x \cdot (5 - 3 \cdot x + 3 \cdot x^2 + 2 \cdot x^3) = \\ &= -1 + x \cdot (5 + x \cdot (-3 + 3 \cdot x + 2 \cdot x^2)) = \\ &= -1 + x \cdot (5 + x \cdot (-3 + x \cdot (3 + 2 \cdot x))) \,. \end{aligned} \tag{3.7}$$

Proceeding from the inside out, the sequence of calculation operations for $x = \frac{1}{2}$ is

$$\frac{1}{2} \cdot 2 + 3 = 4 \tag{3.8}$$

$$\frac{1}{2} \cdot 4 - 3 = -1 \tag{3.9}$$

$$\frac{1}{2} \cdot (-1) + 5 = \frac{9}{2} \tag{3.10}$$

$$\frac{1}{2} \cdot \frac{9}{2} - 1 = \frac{5}{4} \,. \tag{3.11}$$

In total, only four multiplication and sum/subtraction operations are needed to determine the value of the polynomial. Algorithm 1 below is the most efficient algorithm that can be schematically expressed for a generally expressed polynomial (1.1).

```
Input  : $n$, $\mathbf{c} = \{c_1, c_2, c_3, \ldots, c_n, c_{n+1}\}$, $x$
Output: $f(x)$

$y \leftarrow c_{n+1}$
for $i \leftarrow n, n-1, \ldots, 2, 1$ do
  | $y \leftarrow y \cdot x + c_i$
end
$f(x) \leftarrow y$
```

Algorithm 1: Horner's method.

Algorithm 1 is implemented in MATLAB using the following m-file:

```
function y=horner(d,c,x)
y=c(d+1);
for i=d:-1:1
  y=y*x+c(i);
end
```

where parameter d is the degree of the entered polynomial, c is a vector with $d+1$ constant coefficients of the polynomial, and x is the value for which the polynomial is determined. For the polynomial (3.1) and the specified value $x = \frac{1}{2}$, the function is called by the command:

```
horner(4,[-1 5 -3 3 2],1/2)
```

The MATLAB output is:

```
ans =
    1.2500
```

Example 3.1. Apply the previous procedure to determine the deflection in the middle of the span of a statically indeterminate beam. Figure 3.1 illustrates the statics of the beam. To determine the polynomial equations of the deflection curve and slope, we use the method of direct integration of the differential equation of the 4th order $EI_y w_z(x)'''' = q_z(x)$, where E is the modulus of elasticity in tension and compression [MPa], I_y the relevant moment of inertia [m^4], and q_z is the continuous force load acting in the vertical direction [kN/m]. Table 3.1 gives the specific input data.

Solution. The right-hand side of the 4th-order differential equation

$$EI_y w_z(x)'''' = q_z(x) \tag{3.12}$$

has a term corresponding to the continuous force load, i.e., $q(x) = q = \text{const}$.

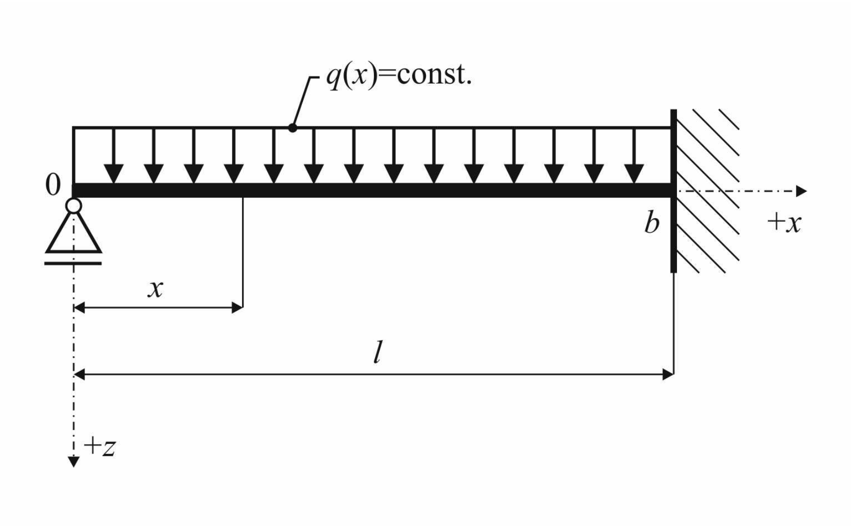


Fig. 3.1 Statics free body diagram of the statically indeterminate beam.

Continuous force load q_z :	4 kN/m
Beam span l :	6 m
Width of the rectangular cross-section b :	0.02 m
Height of the rectangular cross-section h :	0.15 m
Moment of inertia I_y :	$\frac{1}{12} \cdot 0.02 \cdot 0.15^3 = 5.625 \cdot 10^{-6}$ m^4
Modulus of elasticity in tension and compression E:	$2.1 \cdot 10^{11}$ Pa

Tab. 3.1 Input data for exercise 3.1.

The equation (3.12) can then be gradually integrated:

$$EI_y w_z(x)'''' = q_z \ , \tag{3.13}$$

$$EI_y w_z(x)''' = -V_z(x) = q_z \cdot x + C_1 \ , \tag{3.14}$$

$$EI_y w_z(x)'' = -M_y(x) = q_z \cdot \frac{x^2}{2} + C_1 \cdot x + C_2 \ , \tag{3.15}$$

$$EI_y w_z(x)' = q_z \cdot \frac{x^3}{6} + C_1 \cdot \frac{x^2}{2} + C_2 \cdot x + C_3 \ , \tag{3.16}$$

$$EI_y w_z(x) = q_z \cdot \frac{x^4}{24} + C_1 \cdot \frac{x^3}{6} + C_2 \cdot \frac{x^2}{2} + C_3 \cdot x + C_4 \ . \tag{3.17}$$

The integration constants $C_1, \ldots, C_4$ are determined from the boundary conditions:

a) For $x = 0$, is $M_y(x) = 0$, and thus

$$M_y(x = 0) = -q \cdot \frac{0^2}{2} - C_1 \cdot 0 - C_2 = 0 \;, \tag{3.18}$$

from which it follows that $C_2 = 0$,

b) For $x = 0$, is $w_z(x) = 0$ and thus

$$w_z(x = 0) = \frac{1}{EI_y} \cdot \left(q_z \cdot \frac{0^4}{24} + C_1 \cdot \frac{0^3}{6} + 0 \cdot \frac{0^2}{2} + C_3 \cdot 0 + C_4 \right) = 0 \;, \tag{3.19}$$

from which it follows that $C_4 = 0$,

c) For $x = l$, is $w_z^{'}(x) = 0$, and thus

$$w_z^{'}(x = l) = \frac{1}{EI_y} \cdot \left(q_z \cdot \frac{l^3}{6} + C_1 \cdot \frac{l^2}{2} + 0 \cdot l + C_3 \right) = 0 \;, \tag{3.20}$$

d) For $x = l$, is $w_z(x) = 0$, and thus

$$w_z(x = l) = \frac{1}{EI_y} \cdot \left(q_z \cdot \frac{l^4}{24} + C_1 \cdot \frac{l^3}{6} + 0 \cdot \frac{l^2}{2} + C_3 \cdot l + 0 \right) = 0 \;. \tag{3.21}$$

The last two equations represent a system of two linear equations with two unknown integration constants C_1 and C_3. By solving the system, can be obtained:

$$C_1 = -\frac{3}{8}\, q_z l \;, \tag{3.22}$$

and

$$C_3 = \frac{1}{48}\, q_z l^3 \;. \tag{3.23}$$

By substituting the resulting values of the integration constants $C_1, \ldots, C_4$ into the relations (3.14) to (3.17), it is possible to obtain the resulting equations for a pair of static quantities (the shifting force V_z and bending moment M_y), also for both deformation quantities (the deflection w_z and slope $w_z^{'} = \varphi_y$):

$$V_z(x) = -\left(q_z x - \frac{3}{8}\, q_z l \right) = \frac{3}{8}\, q_z l - q_z x \;, \tag{3.24}$$

$$M_y(x) = -\left(q_z \frac{x^2}{2} - \frac{3}{8}\, q_z l x + 0 \right) = \frac{3}{8}\, q_z l x - q_z \frac{x^2}{2} \;, \tag{3.25}$$

$$\begin{aligned} w_z(x)^{'} = \varphi_y(x) &= \frac{1}{EI_y} \cdot \left(q_z \frac{x^3}{6} - \frac{3}{8}\, q_z l\, \frac{x^2}{2} + 0 \cdot x + \frac{1}{48}\, q_z l^3 \right) = \\ &= \frac{1}{EI_y} \cdot \left(q_z \frac{x^3}{6} - \frac{3}{16}\, q_z l x^2 + \frac{1}{48}\, q_z l^3 \right) \;, \end{aligned} \tag{3.26}$$

$$\begin{aligned} w_z(x) &= \frac{1}{EI_y}\cdot\left(q_z\frac{x^4}{24}-\frac{3}{8}q_zl\,\frac{x^3}{6}+0\cdot\frac{x^2}{2}+\frac{1}{48}q_zl^3x+0\right) = \\ &= \frac{1}{2EI_y}\cdot\left(q_z\frac{x^4}{12}-\frac{3}{24}q_zlx^3+\frac{1}{24}q_zl^3x\right)\ . \end{aligned} \tag{3.27}$$

Using MATLAB and the `horner` m-function to determine the required deflection, the sequence of executed commands is

```
qz=4000;
l=6;
b=0.02;
h=0.15;
Iy=1/12*b*h^3;
E=2.1*10^11;
c=[0 1/(48*E*Iy)*qz*l^3 0 -3/(48*E*Iy)*qz*l qz/(24*E*Iy)];
horner(4,c,l/2)*1000
```

The resulting deflection in millimeters in the middle of the span is

```
ans =
  22.857142857142865
```

▲

Example 3.2. Applying the procedure described above, we determine the deflection in the middle of the span of the statically indeterminate beam (Fig. 3.2).

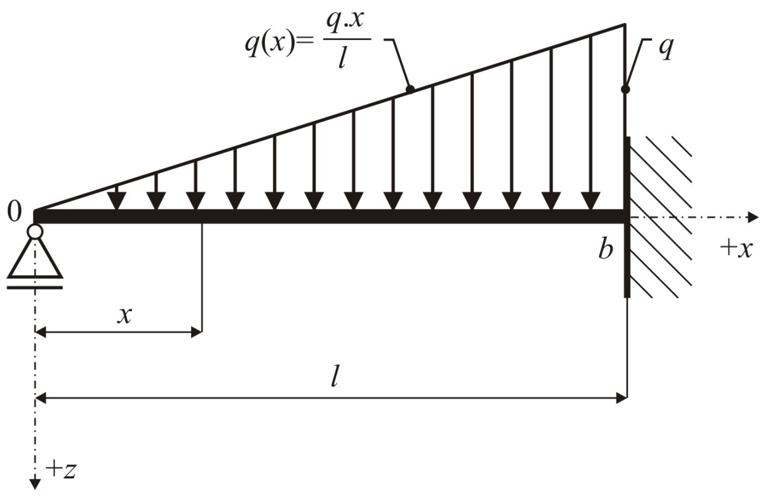


Fig. 3.2 Statics free body diagram of the solved statically indeterminate beam.

Example 3.3. Determine the deflection in the middle of the span of the statically determined beam in Figure 3.3. Use the Clebsch method to determine the equation of the deflection curve across the investigated interval $< 0; \frac{2}{3}\, l >$.

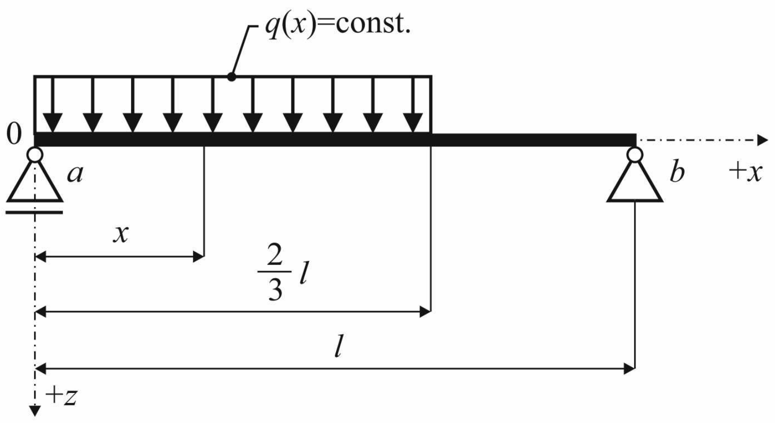


Fig. 3.3 Statics free body diagram of a solved statically determined beam.

3.1.1 Tabulated Functions

Listing the values of a function with the x parameter is best executed with a `for` loop and screen display function in the prescribed formats `disp` and `sprintf`.

Comment 3.4. The `disp(x)` function displays the content of `text`-type variable x.

Comment 3.5. The `sprintf` function converts the data into a text string in the required format using "conversion specifiers", which start with the % sign. The `%f` specifier is for normal conversion and converts a numerical value into a form with a fixed decimal point, the `%e` specifier expresses a numerical value in exponential form, and the `%g` specifier automates selection. Between the `%` character and the `f`, `e` or `g` specifiers, we can additionally insert the number of characters of the required format width, or a period and the number of characters after the decimal point. The `\n` parameter produces a new line.

Example 3.6. Tabulate the deflection curve function of the beam described in Exercise 3.1. Determine the resulting deflections in cross-sections, with a spacing of 10 cm.

Solution. The sequence of commands to list the resulting deflections at the tracked points x might look like this:

```
format long
disp('  x [mm]     w(x) [mm]')
disp('______________________')
for x=0:.1:1
disp(sprintf('%8.1f %12.4f',x*1000,horner(4,c,x)*1000))
end
```

The output will then look like this:

```
  x [mm]     w(x) [mm]

______________________
     0.0        0.0000
   100.0        1.5226
   200.0        3.0377
   300.0        4.5383
   400.0        6.0176
    ...          ...
  5700.0        0.6297
  5800.0        0.2881
  5900.0        0.0741
  6000.0        0.0000
```

▲

Example 3.7. Extend the output from the previous exercise by including the value of the shear force V_z, the bending moment M_y, and the slope φ_y.

3.1.2 Drawing the Graph of a Targeted Function

The graph of the solved function can be plotted using the procedure described in Ch. 1.4.1.

Example 3.8. Draw a graph of the deflection curve of the beam described in Exercise 3.1.

Solution. A possible solution is a sequence of commands that uses a **for** loop, the derived vector c from Exercise 3.1, and the `horner` m-function:

```
x=linspace(0,1,100);
for i=1:100, y(i)=horner(4,c,x(i))*1000; end
plot(x,y,'b-');
title('Deflection curve of the beam w(x)');
xlabel('x');
ylabel('w(x)');
```

Figure 3.4 depicts the resulting graph of the deflection curve.

▲

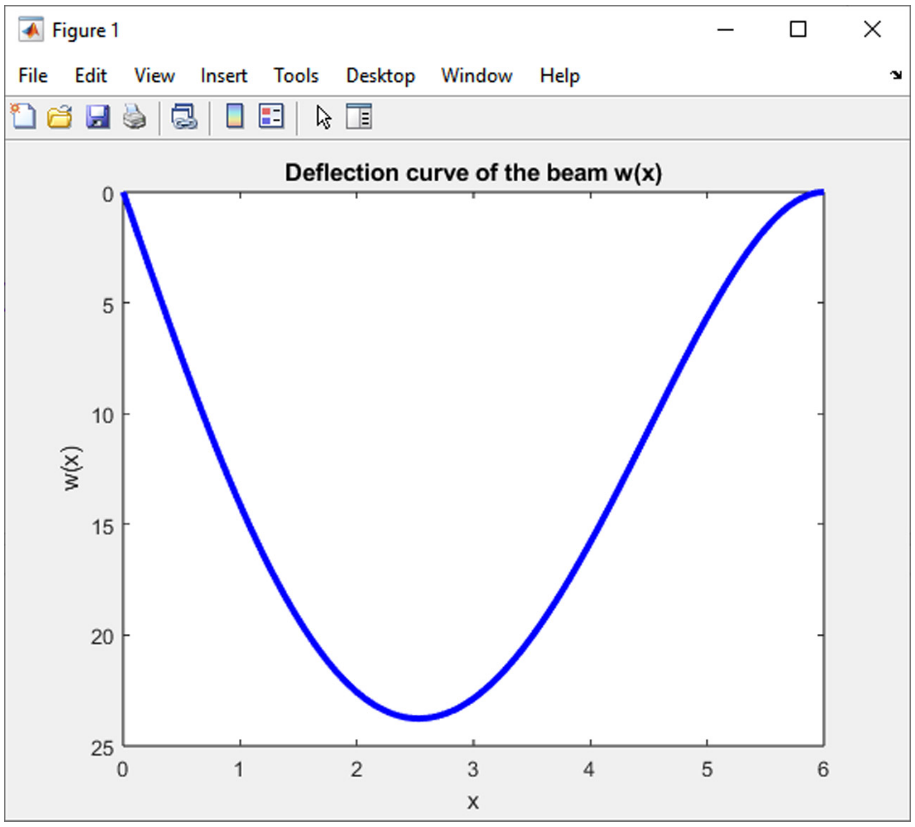


Fig. 3.4 The deflection curve of the statically indeterminate beam given in Exercise 3.1.

Examples to Practice

1. Use an approach similar to the above to create the graph of the deflection curve of the statically indeterminate beam depicted in Fig. 3.2.

2. Draw a graph of the deflection curve of a statically determined beam depicted in Fig. 3.3.

3.1.3 Determining the Maximum of a Discretized Function

We can perform a simplified calculation of the largest value of the function in a predefined interval in three steps: first by discretizing the x axis, then determining the values of the function for all x in the required range, and finally using algorithm 2 to determine the largest number in the vector.

```
Input  : $n$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Output: $maximum$
$maximum \leftarrow b_1$
for $i \leftarrow 2, 3, \ldots, n-1, n$ do
    if $b_i > maximum$ then
        $maximum \leftarrow b_i$
    end
end
```

Algorithm 2: Algorithm for Determining the Largest Number in a Vector.

Implementing the above algorithm in the MATLAB system, we can create the `maximum` m-function with the following sequence of commands:

```
function m=maximum(b)
n=length(b)
m=b(1);
if n>1
  for i=2:n
    if b(i)>m
      m=b(i);
    end
  end
end
```

Comment 3.9. The `length` function returns the dimension of the vector contained in the input parameter.

Example 3.10. Determine the value of the largest deflection of the beam described in Exercise 3.1 using the tabulated values of the deflection curve given in Exercise 3.6.

Solution. First, it is necessary to create a vector $b = b_1, b_2, \ldots, b_n$ with deflection values in the observed cross-sections ($b_i = w_z(x_i)$) of coordinates x_i and a spacing of 10 cm (n will therefore be equal to 61 when the span is $l = 6$ m):

```
i=0;
for x=0:.1:l
  i=i+1;
  b(i)=horner(4,c,x)*1000;
end
```

At this point, we call the function which finds the largest number in vector b. Entering the `maximum(b)` command, the output of the m-function and the result of the largest deflection (in mm) are:

```
n =
    61

ans =
   23.7654
```

▲

Comment 3.11. This method of calculating the largest value of the function is approximate only because it is hampered by the error caused by discretization of the x axis. An approach that leads to an exact solution is shown in Chapter 4.

Examples to Practice

1. Using the `maximum` m-function, determine the largest deflection on the statically indeterminate beam depicted on Fig. 3.2.

2. Determine the largest deflection on the statically determined beam depicted in Figure 3.3.

Chapter 4

Solving Nonlinear Algebraic Equations

Objectives

This chapter provides a detailed introduction to:

- the principle of iterative methods,
- basic algorithms for determining the roots of non-linear algebraic equations,
- using the **while** loop in an algorithm.

4.1 Iteration

The word *iteration* carries the same sense as repetition (*iteretur* in Latin – to repeat). In mathematics, iteration refers to the solution of a problem through successive repetitions that approach and lead to the desired result.

4.1.1 Taylor Series

A simple iterative calculation can be demonstrated with the *Taylor series*, which is a special power series named after the English mathematician Brook Taylor, who published it in 1712 (the method of approximating a function by a power series was discovered as early as 1671 by James Gregory).

Under certain assumptions of the $f(x)$ function around the point a, this function can be expressed (expanded) as a power series. This type of expression of a function using the Taylor series is referred to as a *Taylor expansion*:

$$\begin{aligned} f(x) &= f(a) + \frac{f(a)'}{1!}\cdot(x-a) + \frac{f(a)''}{2!}\cdot(x-a)^2 + \frac{f(a)^{(3)}}{3!}\cdot(x-a)^3 + \cdots = \\ &= \sum_{k=0}^{\infty} \frac{f(a)^{(k)}}{k!}\cdot(x-a)^k\,. \end{aligned} \tag{4.1}$$

For an approximate expression of the function values, it is not necessary to express every term of the Taylor series; terms with higher derivatives can be ignored. In this way, a Taylor polynomial is obtained and can be used to approximate the values of a function that has a derivative at a given point, using a polynomial whose coefficients depend on the derivatives of the function at that point.

Example 4.1. Using the first ten terms of the Taylor expansion, determine the value of the e^x function for $x = 1$.

Solution. The relation for the Taylor expansion for determining the value of the e^x function takes the following form:

$$f(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^{(n)}}{n!}\,. \tag{4.2}$$

The relation (4.2) is valid for $x \in <-\infty, \infty>$. The given calculation can be executed with Algorithm 3.

```
Input  : $n$, $x$
Output: $f(x)$
$y \leftarrow 1$
for $i \leftarrow 1, 2, 3, \ldots, n-1$ do
    $y \leftarrow y + \frac{x^i}{i!}$
end
$f(x) \leftarrow y$
```

Algorithm 3: Determining the value of the e^x function using a Taylor expansion.

The corresponding MATLAB script contains the following instructions:

```
function y=e_on_x(n,x)
if n<2
  error('The number of terms n must be 2 at least!')
end
y=1;
for i=1:n-1
  y=y+(x^i)/factorial(i);
end
```

After calling the m-function

```
e_on_x(10,1)
```

we obtain the result:

```
ans =
   2.718281525573192
```

The progress of the gradually refined result and increasing number of terms in the Taylor expansion is demonstrated in the table below:

```
   i      f(xi)         f(xi)-e^x

------------------------------------
   1   1.00000000   -1.71828183e+000
   2   2.00000000   -7.18281828e-001
   3   2.50000000   -2.18281828e-001
   4   2.66666667   -5.16151618e-002
   5   2.70833333   -9.94849513e-003
   6   2.71666667   -1.61516179e-003
   7   2.71805556   -2.26272903e-004
   8   2.71825397   -2.78602051e-005
   9   2.71827877   -3.05861778e-006
  10   2.71828153   -3.02885853e-007
```

The table's third column indicates the difference between the partial result and the exact value of e^x, which is called with MATLAB's `exp(x)` function.

▲

Comment 4.2. The `e_on_x` m-file also applies the `error` function, which displays an error message and terminates the function.

Comment 4.3. To calculate the value of $n!$, MATLAB uses the `factorial` function.

4.1.2 Loop Terminating Condition

In the exercise in Section 4.1, was entered the exact number of the required terms of the expansion and used the Taylor expansion to calculate the approximate value of the function (i.e., the `for` loop was used). In practice, it is generally applied the "loop terminating condition", which takes the form

$$|x_k - x_{k-1}| < \varepsilon \ , \tag{4.3}$$

where x_{k-1}, x_k are the terms of the Taylor expansion in the $(k-1)$th and kth positions, and ε, referred to as the tolerance in the error of a result, is a sufficiently small number.

4.1.3 Recurring Pattern

Using knowledge of one or more preceding elements, a recurring pattern determines the terms of a sequence. Each recurring pattern must include the entry of the first or several first terms of the sequence. The disadvantage of using a recurring pattern is that any element of the sequence can be determined only if the preceding terms are known. The first element of a given sequence, referred to as the "zero approximation", must always be estimated.

Example 4.4. Using "Heron's formula" determine the value of the $f(x) = \sqrt{x}$ function for $x = 2$, with a tolerance in the error of the result $\varepsilon = 0.001$.

Solution. Heron's formula for determining the value of the $f(x) = \sqrt{x}$ function is recurrent and takes the form

$$f(x) = y_k = \frac{1}{2}\left(y_{k-1} + \frac{x}{y_{k-1}}\right) , \tag{4.4}$$

where y_{k-1}, y_k are again the $(k-1)$th and kth terms of the sequence.

The relation (4.4) is valid for $x > 0$. The calculation procedure for a given task can be expressed with Algorithm 4.

```
Input  : $x$, $\varepsilon$
Output: $f(x)$
$y_1 \leftarrow x$
$y_2 \leftarrow \frac{1}{2} \cdot \left(y_1 + \frac{x}{y_1}\right)$
while $|y_1 - y_2| \geqq \varepsilon$ do
    $y_1 \leftarrow y_2$
    $y_2 \leftarrow \frac{1}{2} \cdot \left(y_1 + \frac{x}{y_1}\right)$
end
$f(x) \leftarrow y_2$
```

Algorithm 4: Determining the value of the $f(x) = \sqrt{x}$ function with Heron's formula.

The MATLAB script contains the following instructions:

```
function y=sq_root(x,tol)
if x<=0
  error('The condition x>0 is not met!')
end
y1=x; y2=1/2*(y1+x/y1);
while abs(y1-y2)>tol
  y1=y2;
  y2=1/2*(y1+x/y1);
end
y=y2;
```

After calling the m-function

```
sq_root(2,0.001)
```

a corresponding result can be obtained:

```
ans =
   1.414213562374690
```

The process of the gradually refined result is shown in the table below:

```
   i     f(xi)        f(xi)-x^0.5

-----------------------------------
   1   2.00000000   5.85786438e-001
   2   1.50000000   8.57864376e-002
   3   1.41666667   2.45310429e-003
   4   1.41421569   2.12390141e-006
   5   1.41421356   1.59472435e-012
```

The table's third column indicates the difference between the partial result and the exact value of $\sqrt{x}$, which is called with MATLAB's `sqrt(x)` function.

We can also display the deviation from the exact solution with the command:

```
sq_root(2,0.001)^2
```

to display:

```
ans =
   2.000000000004511
```

▲

Example 4.5. Verify the validity of the following Taylor expansion for calculating the value of $\sin(x)$:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{(2n+1)}}{(2n+1)!} \,, \tag{4.5}$$

where the desired tolerance of the error in the result $\varepsilon = 0.001$. The (4.5) relation is valid for $x \in < -\infty, \infty >$.

Example 4.6. Verify the validity of the following Taylor expansion for calculating the value of the $\cos(x)$ function:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n}}{(2n)!} \ . \tag{4.6}$$

where the desired tolerance of the error in the result $\varepsilon = 0.001$. The (4.6) relation is valid for $x \in < -\infty, \infty >$.

4.2 Iterative Methods of Solving Non-linear Algebraic Equations

The principles described in Chapter 4.1 can be used to solve non-linear algebraic equations. The individual methods described in the text below differ mainly in speed of convergence and universality in use.

Convergence is a term which denotes convergence or developments that lead to convergence. Properties that converge are said to be convergent. Objects, processes and properties which participate in convergence are referred to as convergent, for example, a *convergent series* in mathematics. In mathematics, convergence is closely related to the concept of limit. The opposite of convergence is **divergence**.

4.2.1 Simple Iteration

The solved equation

$$f(x) = 0 \ , \tag{4.7}$$

must first be modified into the form

$$x = g(x) \ . \tag{4.8}$$

This procedure can be done in several ways. A prerequisite of the calculation is the condition that an interval $< a_0, b_0 >$ falls into the domain of the definition and the domain of continuity of the functions $f(x)$ and $g(x)$, which also contains a common root of both equations. It is also necessary to choose the value of the zero approximation from this interval.

Comment 4.7. The disadvantage of this method is the convergence of the solution to the root that does not lie in the $< a_0, b_0 >$ interval in the case of an inappropriately selected function $g(x)$. The solved root is not common to both equations; it is the root of equation $g(x)$ only.

The calculation for k iteration steps can be executed with Algorithm 5.

```
**Input**  : $x_0$, $k$
**Output**: $f(x)$
**for** $i \leftarrow 1, 2, 3, \ldots, k$ **do**
    $x_i \leftarrow g(x_{i-1})$
**end**
$f(x) \leftarrow x_k$
```

Algorithm 5: Determining the root of the equation $f(x)$ using the simple iteration method.

The notation in the form of an m-function then looks as follows:

```
function xc=simple_it(g,x0,k)
x(1)=x0;
for i=1:k
  x(i+1)=g(x(i));
end
x'
xc=x(k+1);
```

The list of parameters of the `simple_it` m-function includes the solved function g. This can be defined via a variable using the `inline` MATLAB function.

Example 4.8. Use the simple iteration method to approximate the root of the equation

$$f(x) = \left(\frac{x}{2}\right)^2 - \sin(x) = 0\ . \tag{4.9}$$

Use the following input parameters: number of iteration loops $k = 10$, zero approximation $x_0 = 1.5$.

Solution. Equation (4.9) can be simplified into the form

$$x = 2\sqrt{\sin(x)}\ . \tag{4.10}$$

Equation (4.10) can then be input into MATLAB with the command

```
g=inline('2*sqrt(sin(x))')
```

Now you can start the calculation by:

```
xc=simple_it(g,1.5,10)
```

The list of the first ten iterations is shown below (the tenth, final iteration may be considered the solved root of the nonlinear equation):

```
    1.500000000000000
    1.997493415863046
    1.908232350897023
    1.942788324690179
    1.930393907098011
    1.934981663979237
    1.933302091730397
    1.933919512286077
    1.933692884811449
    1.933776115524553
    1.933745554580009
```

Substituting the calculated root 1.933745554580009 into the original equation (4.9) yields the following:

```
ans =
    -1.085057641792009e-005
```

This value sufficiently demonstrates the inaccuracy of the solution. ▲

Example 4.9. Calculate the root of the nonlinear equation given in Exercise 4.8 using the simple iteration method with zero approximation $x_0 = 2.0$ and $k = 20$ iteration loops. Indicate the accuracy achieved by the solution.

Example 4.10. Modify the algorithm for the simple iteration method so that the loop is terminated by the (4.3) condition. Then solve Exercise 4.8 using $\varepsilon = 0.05$, indicating the tolerance of the error of the result.

4.2.2 Bisection Method (Interval Halving)

The root of the real nonlinear equation $f(x) = 0$, which is continuous for each value of $x \in < a_0; b_0 >$, can be solved with this method approximately (with a specified tolerance of ε). It is also assumed following condition: $f(a_0) \cdot f(b_0) < 0$, i.e., $\operatorname{sign} f(a_0) = -\operatorname{sign} f(b_0)$.

The procedure for calculating the root of the nonlinear equation $f(x) = 0$ using the bisection method is executed by Algorithm 6.

After n computational steps, the examined interval with the sought root of the equation $f(x) = 0$ will have a width of

$$b_n - a_n = \frac{1}{2}(b_{n-1} - a_{n-1}) = \frac{1}{2^2}(b_{n-2} - a_{n-2}) = \ \ldots \ = \frac{1}{2^n}(b_0 - a_0) \ . \tag{4.11}$$

```
Input  : $\varepsilon > 0$, $a_0$, $b_0$
Output: $x_c$
$a \leftarrow a_0$
$b \leftarrow b_0$
while $|a - b| \geqq \varepsilon$ do
    $c \leftarrow \frac{a+b}{2}$
    if $f(c) = 0$ then
        $x_c \leftarrow c$                    /* $c$ is the final equation root */
        end of the calculation;
    end
    if $\mathrm{sign}(f_c) \cdot \mathrm{sign}(f_a) < 0$ then
        $b \leftarrow c$         /* the new interval boundaries are $< a, c >$ */
    else
        $a \leftarrow c$         /* the new interval boundaries are $< c, b >$ */
    end
end
$x_c \leftarrow \frac{a+b}{2}$
```

Algorithm 6: Algorithm of the bisection method (interval halving).

To estimate the error (inaccuracy) of the result, it holds that

$$|a_n - \alpha| \leqq \frac{b_0 - a_0}{2^n} , \tag{4.12}$$

or

$$|b_n - \alpha| \leqq \frac{b_0 - a_0}{2^n} , \tag{4.13}$$

where α is the exact value of the root of the equation $f(x) = 0$.

Comment 4.11. The bisection method converges very slowly. The authors of [6] report that accuracy improves by one decimal place after 3.3 computational steps, where $10^{-1} \approx 2^{-3.3}$. However, the function $f(x)$ does not affect the speed of convergence. The advantage is the simplicity of the application and the ability to accurately estimate the number of steps needed to achieve the required accuracy.

Example 4.12. Determine the approximation of the root of the equation using the bisection method:

$$f(x) = x^3 + x - 1 . \tag{4.14}$$

Use the following input parameters: $\varepsilon = 0.001$, $a_0 = 0$ and $b_0 = 1$.

Solution. We can program a function to calculate the root of a nonlinear equation using the bisection method and an m-file as follows:

```
function xc=bisect(f,a,b,tol)
if sign(f(a))*sign(f(b))>=0
  error('The condition f(a)*f(b)<0 is not met!')
end
fa=f(a);
fb=f(b);
while(b-a)>tol
  c=(a+b)/2;
  fc=f(c);
  if fc==0
    xc=c                  %c is the solution
    break
  end
  if sign(fc)*sign(fa)<0  %The new interval boundaries are <a,c>
    b=c;
    fb=fc;
  else                    %The new interval boundaries are <c,b>
    a=c;
    fa=fc;
  end
end
xc=(a+b)/2;
```

The list of parameters also includes the solved function f. This can be defined with a variable which uses the `inline` MATLAB function:

```
f=inline('x^3+x-1')
```

The calculation can now be started by entering the command

```
xc=bisect(f,0,1,0.001)
```

The result is

```
xc =
    0.6821
```

To demonstrate the bisect function, it is possible to see how the contents of the variables a, b and c change during iterations of the algorithm. The columns indicated $f(a)$, $f(b)$ and $f(c)$ contain the value -1 or $+1$ depending on whether the respective value of the function $f(a)$, $f(b)$ and $f(c)$ is negative or positive.

```
  i     a     f(a)    c    f(c)    b    f(b)

-------------------------------------------
  0   0.0000  -1   0.5000  -1   1.0000   1
  1   0.5000  -1   0.7500   1   1.0000   1
  2   0.5000  -1   0.6250  -1   0.7500   1
  3   0.6250  -1   0.6875   1   0.7500   1
  4   0.6250  -1   0.6563  -1   0.6875   1
  5   0.6563  -1   0.6719  -1   0.6875   1
  6   0.6719  -1   0.6797  -1   0.6875   1
  7   0.6797  -1   0.6836   1   0.6875   1
  8   0.6797  -1   0.6816  -1   0.6836   1
  9   0.6816  -1   0.6826   1   0.6836   1
 10   0.6816  -1   0.6821  -1   0.6826   1
```

▲

Example 4.13. Applying the bisection method, determine the approximation of the root of the equation (4.9) given in Exercise 4.8. Use input parameters: $\varepsilon = 0.05$, $a_0 = 1.5$ and $b_0 = 2.0$.

Example 4.14. Using the bisection method, determine the largest deflection on the beam described in Exercise 3.1.

Solution. In the cross-section with the largest deflection, we have $w_z(x)' = \varphi_y(x) = 0$. To determine this, it is necessary to solve the roots of the 3rd-degree polynomial. If the derived equation for slope (3.26) is applied to (1.1) with the general expression of the n-th degree polynomial, the vector c with elements $[c_n, c_{n-1}, \ldots, c_1, c_0]$ takes the form

$$\mathbf{c} = \left[\ \frac{q_z}{6} \ , \ \ -\frac{3}{16}\, q_z l \ , \ \ 0 \ , \ \ \frac{1}{48}\, q_z l^3 \ \right] \ . \tag{4.15}$$

The root of a polynomial using the bisection method can be calculated using a slightly modified version of the m-function from Exercise 4.12:

```
function xc=bisection(a,b,d,tol)
if sign(horner(3,d,a))*sign(horner(3,d,b))>=0
  error('The condition f(a)*f(b)<0 is not met!')
end
fa=horner(3,d,a);
fb=horner(3,d,b);
while b-a>tol
  c=(a+b)/2;
  fc=horner(3,d,c);
  if fc==0
```

```
    xc=c                    %c is the solution
    break
  end
  if sign(fc)*sign(fa)<0  %The new interval boundaries are <a,c>
    b=c;
    fb=fc;
  else                    %The new interval boundaries are <c,b>
    a=c;
    fa=fc;
  end
end
xc=(a+b)/2
```

The largest deflection is obtained by substituting the resulting root of the slope polynomial from the interval $(0; l)$ into the equation of the deflection curve (3.27).

The entire sequence of commands for determining the zero-slope cross-section and the largest deflection on the beam is

```
format long
qz=4000;
l=6;
b=0.02;
h=0.15;
Iy=1/12*b*h^3;
E=2.1*10^11;
c=[0 1/(48*E*Iy)*qz*l^3 0 -3/(48*E*Iy)*qz*l qz/(24*E*Iy)];
fi=[1/(48*E*Iy)*qz*l^3 0 -3/(16*E*Iy)*qz*l qz/(6*E*Iy)];
xmax=bisection(0,l-0.00001,fi,0.00000000000001)
slope=horner(3,fi,xmax)
wmax=horner(4,c,xmax)*1000
```

In the given task, the largest deflection in the cross-section has the coordinate

```
xmax =
   2.529210992451759
```

Slope in this cross-section is

```
slope =
   1.734723475976807e-017
```

The maximum deflection in millimetres is

```
wmax =
   23.769036533008371
```

▲

Comment 4.15. Figure 4.1 illustrates a beam slope graph from the previous example of command sequences and the working principle of the bisection method algorithm:

```
x=linspace(0,l,100);
plot([0,l],[0,0],'r-')
for i=1:100, y(i)=horner(3,fi,x(i)); end
plot(x,y,'b-');
title('Graph of the slope fi(x)');
xlabel('x');
ylabel('fi(x)');
```

The graph shows the path of the iterative calculation from the center of the interval towards the solved root of the slope equation.

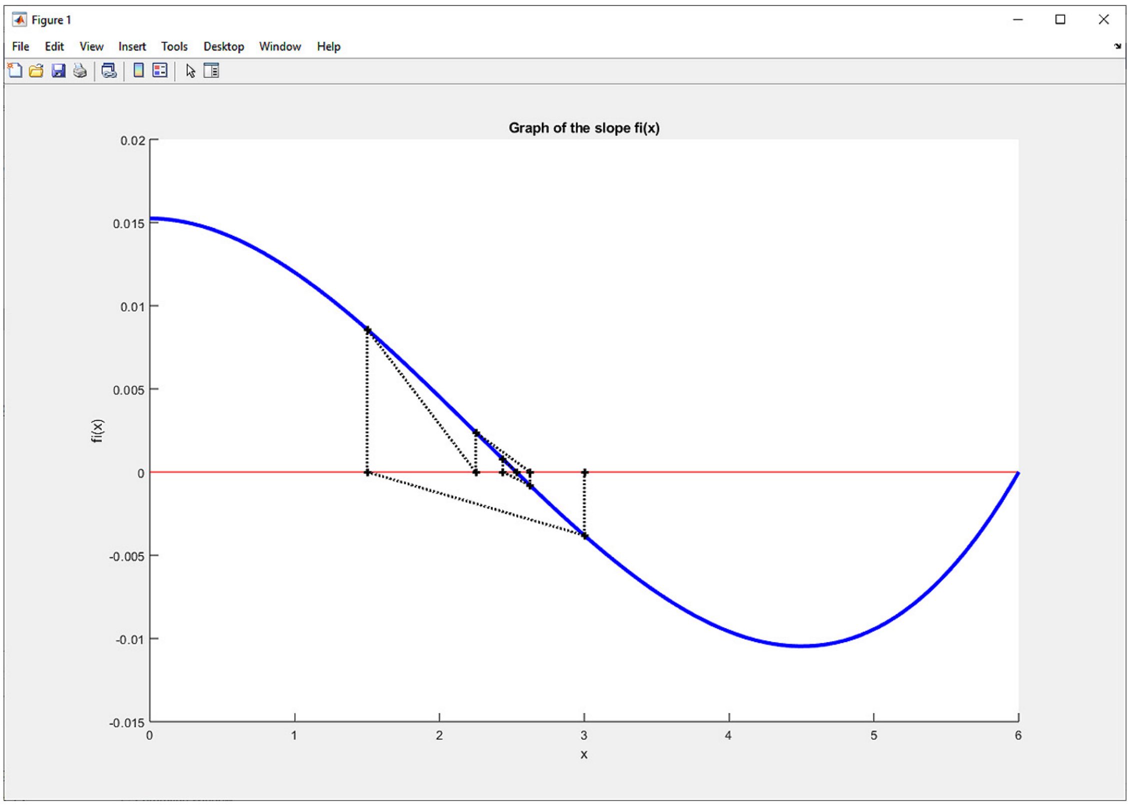


Fig. 4.1 Graph of the slope of the statically indeterminate beam from Example 3.1 and iteration of the cross-section with the largest deflection using the bisection method.

Comment 4.16. The graph in Figure 4.1 shows that the limit value of the slope at point b is zero. If the limits $a_0 = 0$ and $b_0 = l$ were selected when calling the bisection function, the required condition for solution by this method would therefore not be met: $f(a_0) \cdot f(b_0)$ <0. For this reason, we adjust the value of the input parameter $b_0 = l$ to $b_0 = l - 0.00001$:

```
xmax=bisection(0,l-0.00001,fi,0.00000000000001,1000)
```

4.2.3 Regula Falsi Method

Using the regula falsi method, the root of the real nonlinear equation $f(x) = 0$, which is continuous for $x \in < a; b >$, can also be solved approximately (with the specified tolerance ε). Let us assume that $f(a_0) \cdot f(b_0) < 0$, i.e., $\operatorname{sign} f(a_0) = -\operatorname{sign} f(b_0)$ remains valid since it corresponds to the existence of a real root of the given equation in the interval $< a; b >$.

The regula falsi method first determines the intersection of the secant of the curve $f(x)$ constructed at the points $[a, f(a)]$ and $[b, f(b)]$ according to the relation

$$s = a - \frac{f(a)}{f(b) - f(a)} \cdot (b - a)\ . \tag{4.16}$$

If $\operatorname{sign}(f(s)) = \operatorname{sign}(f(a))$, the examined interval changes to $< s, b >$, otherwise to $< a, s >$ and the calculation intersection of the secant of the curve $f(x)$ continues using the recurrent formula (4.16) until the terminating condition is met.

The procedure for calculating the root of the nonlinear equation $f(x) = 0$ by the regula falsi method is executed by Algorithm 7.

```
Input  : $\varepsilon > 0$, $a$, $b$
Output: $x_c$
$s \leftarrow a - \frac{f(a)}{f(b) - f(a)} \cdot (b - a)$
while $|f(s)| \geqq \varepsilon$ do
    if $\operatorname{sign}(f(s)) = \operatorname{sign}(f(a))$ then
        $a \leftarrow s$          /* The new interval boundaries are $< s, b >$ */
    else
        $b \leftarrow s$          /* The new interval boundaries are $< a, s >$ */
    end
    $s \leftarrow a - \frac{f(a)}{f(b) - f(a)} \cdot (b - a)$
end
$x_c \leftarrow s$
```

Algorithm 7: Regula falsi method algorithm.

Example 4.17. Using the regula falsi method, determine the approximation of the root of the equation (4.14) from Example 4.12. Select input parameters similar to $\varepsilon = 0.001$, $a_0 = 0$ and $b_0 = 1$.

Solution. The function for calculating the root of a nonlinear equation using the regula falsi method as a MATLAB script can be written as

```
function xc=regula(f,a,b,tol)
if sign(f(a))*sign(f(b))>=0
  error('The condition f(a)*f(b)<0 is not met!')
end
fa=f(a);
fb=f(b);
s=a-fa/(fb-fa)*(b-a);
fs=f(s);
while abs(fs)>=tol
  if sign(fs)==sign(fa)  %The new interval boundaries are <s,b>
    a=s;
    fa=fs;
  else                   %The new interval boundaries are <a,s>
    b=s;
    fb=fs;
  end
  s=a-fa/(fb-fa)*(b-a);
  fs=f(s);
end
xc=s;
```

The list of parameters includes the solved function f, which can be defined by the already known function `inline`. The calculation is started with the command:

```
xc=regula(f,0,1,0.001)
```

The result is:

```
xc =
   0.682175815962540
```

The operation of the regula function is shown in the table below, indicating the changing content of variables a, b and s during iteration and whether the corresponding value of $f(a)$, $f(s)$ and $f(s)$ is negative or positive.

```
  i     a     f(a)    s     f(s)    b     f(b)

--------------------------------------------
  0   0.0000  -1   0.5000  -1   1.0000   1
  1   0.5000  -1   0.6364  -1   1.0000   1
  2   0.6364  -1   0.6712  -1   1.0000   1
  3   0.6712  -1   0.6797  -1   1.0000   1
  4   0.6797  -1   0.6817  -1   1.0000   1
  5   0.6817  -1   0.6822  -1   1.0000   1
```

▲

Comment 4.18. The regula falsi method always converges to the resulting root if its existence is guaranteed by the input parameters. Convergence using the regula falsi method is quicker than with the bisection method, which corresponds to the course of calculations in Examples 4.12 and 4.17. Although the bisection method achieved a solution after ten steps with the required tolerance of the error in the result $\varepsilon = 0.001$, the regula falsi method found the sought root after only five iterations.

Example 4.19. Using the example of the beam described in Exercise 3.1, determine the largest deflection on the structure in a manner similar to Exercise 4.14, but this time using the regula falsi method.

Solution. Using the regula falsi method with the input data $a = 0$, $b = l - 0.001$ and $\varepsilon = 0.0001$, the largest deflection in the cross-section has the coordinate:

```
xmax =
   2.529471870690230
```

Slope in this cross-section is:

```
slope =
   -2.201632719885452e-006
```

and the maximum deflection itself in millimetres is:

```
wmax =
   23.769036245827937
```

▲

Comment 4.20. To illustrate how the regula falsi method algorithm works, let us examine the beam slope graph created from the previous example (Fig. 4.2). The graph indicates the path of the iterative calculation from the initial approximation towards the solved root of the slope equation.

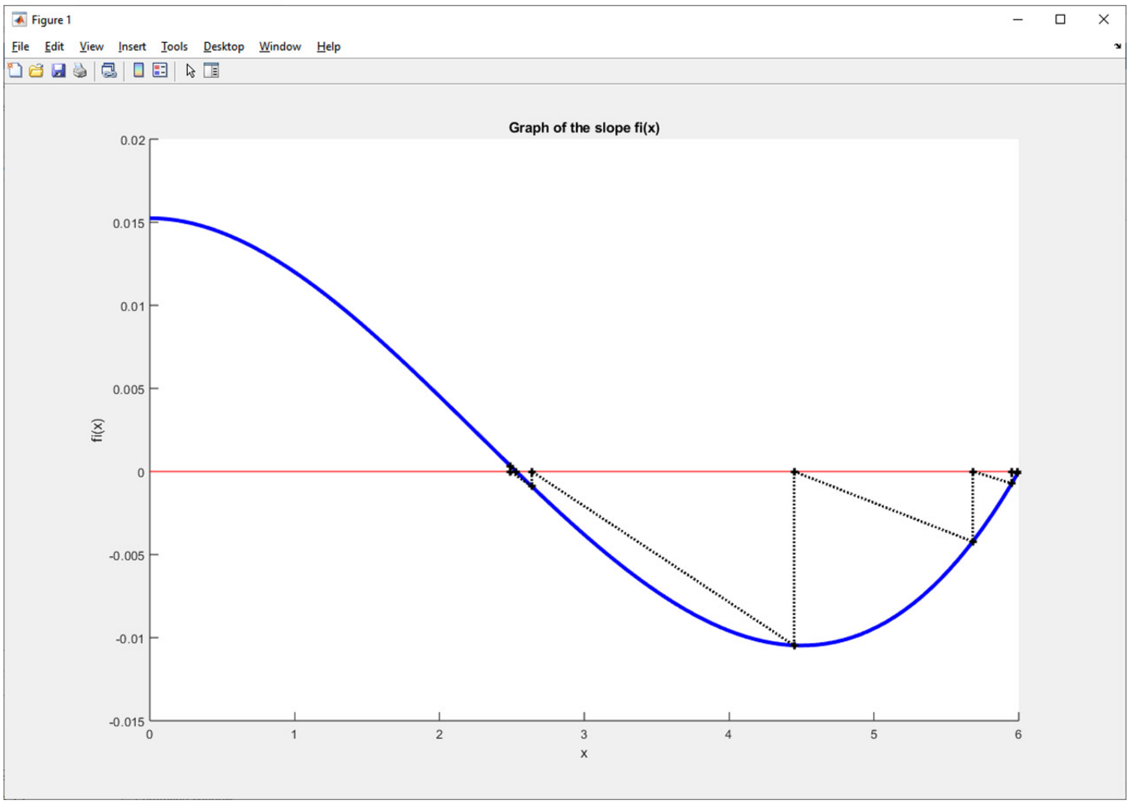


Fig. 4.2 Graph of the slope of the statically indeterminate beam in Example 3.1 and iteration of the cross-section with the largest deflection using the regula falsi method.

4.2.4 Secant Method

By applying the secant calculation method, the solved function $f(x)$ is instead a linear function:

$$g(x) = \frac{f(x_k - f(x_{k-1}))}{x_k - x_{k-1}} \cdot (x - x_k) + f(x_k) \ . \tag{4.17}$$

Determining the root of the equation $g(x)$ in (4.17) then requires application of the recurrence formula:

$$x_{k+1} = x_k - f(x_k) \cdot \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \ . \tag{4.18}$$

The root x_{k+1} may be considered an approximation of the root of the equation $f(x) = 0$. The relation (4.18) represents a two-step iterative formula since two initial approximations x_0 and x_1 must be determined to start the calculation.

Comment 4.21. If the two initial approximations x_0 and x_1 are not appropriately selected, the secant method may not converge. It is therefore necessary to perform a convergence check for the calculation algorithm or determine the initial approximations using another method.

Example 4.22. Solve the root of the equation from Example 4.17 using the secant method. Choose the input parameters $x_0 = 0$, $x_1 = 1$ and $\varepsilon = 0.001$.

Solution. When we define the algorithm and the instructions of the m-file, we can start with the regula falsi calculation method, which must be adjusted according to the secant method's different recurrence formula (4.18).

The m-function, named `m_secant`, can be called with the command:

```
xc=m_secant(f,0,1,0.001)
```

For the given input parameters, the result is:

```
xc =
   0.682020419648186
```

The secant method's calculation procedure is best followed by successively listing the values of x_{k-1}, x_k and x_{k+1} and its functional values:

```
  i   x(k-1)  f(x(k-1))     x(k)    f(x(k))    x(k+1)  f(x(k+1))
----------------------------------------------------------------
  0   0.0000  -1.000000   1.0000   1.000000   0.5000  -0.375000
  1   1.0000   1.000000   0.5000  -0.375000   0.6364  -0.105935
  2   0.5000  -0.375000   0.6364  -0.105935   0.6901   0.018636
  3   0.6364  -0.105935   0.6901   0.018636   0.6820  -0.000737
```

Three iteration loops are sufficient to achieve a result with a tolerance of $\varepsilon =$ $= 0.001$. ▲

Example 4.23. For the beam described in Exercise 3.1, determine the largest deflection on the structure in a manner similar to the procedure in Exercise 4.14 or 4.19, but instead use the bisection method.

Solution. Using the secant method with the input data $a = 0$, $b = l/2$ and $\varepsilon =$ $= 0.0001$, the largest deflection in the cross-section has the coordinate:

```
xmax =
   2.534161490683230
```

The slope in this cross-section is:

```
slope =
   -4.176775334049400e-005
```

The maximum deflection in millimetres is:

```
wmax =
   23.768933137770031
```

▲

Comment 4.24. To illustrate how the secant method algorithm works, let us examine the beam slope graphs created from the previous example (Fig. 4.3 and 4.4). The graphs show the path of the iterative calculation from the initial approximation x_k to the solved root of the slope equation. The graphs differ in the values of the input parameter b, entered as $b = l - 0.85$ and $b = l - 1.0$, respectively. As mentioned above in the note, the method is very sensitive to the values of the first two approximations x_0 and x_1.

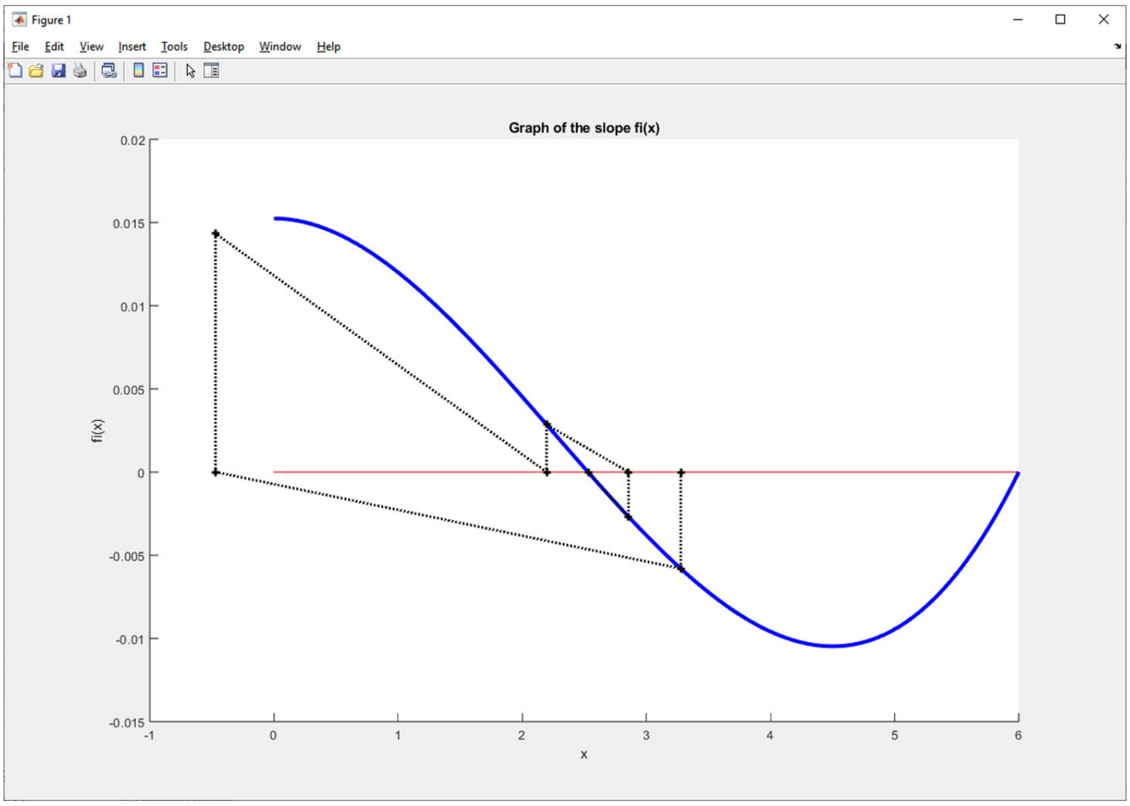


Fig. 4.3 Graph of the slope of the beam from Example 3.1 and iteration of the cross-section with the largest deflection using the secant method with the input parameters $a = 0$, $b = l - 0.85$ and $\varepsilon = 0.0001$.

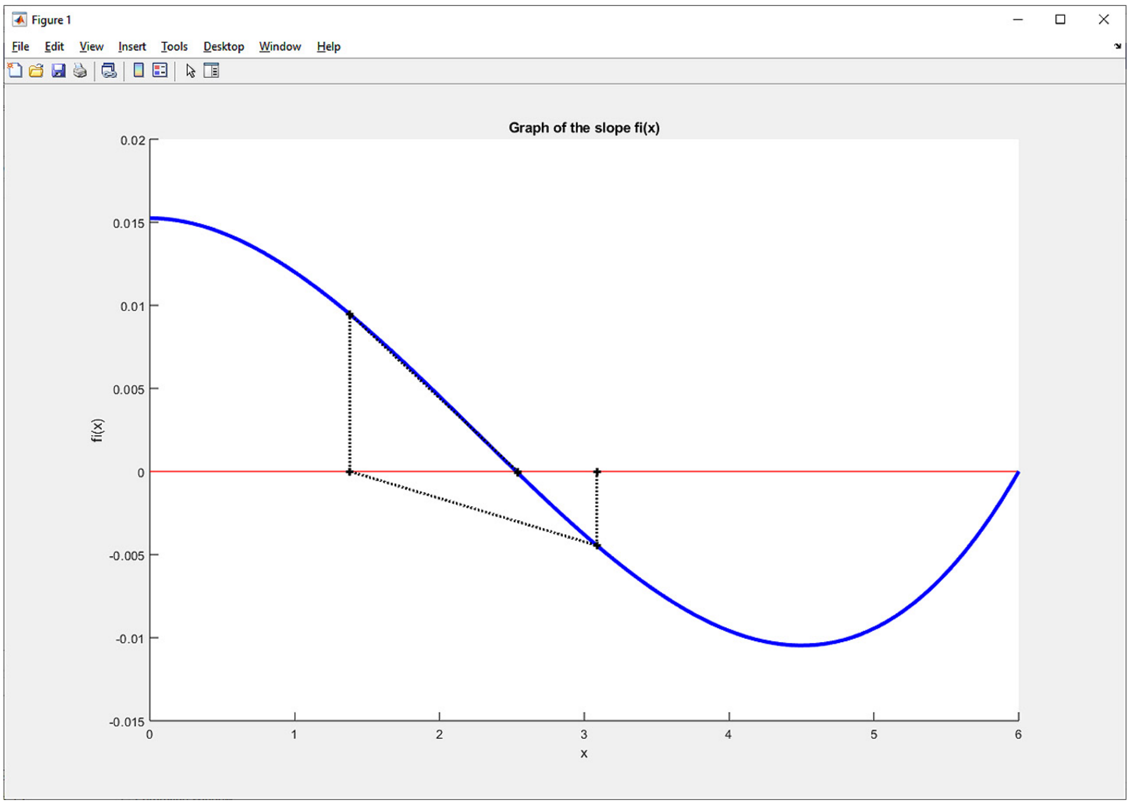


Fig. 4.4 Graph of the slope of the beam from Example 3.1 and iteration of the cross--section with the largest deflection using the secant method with input parameters $a = 0$, $b = l - 1.0$ and $\varepsilon = 0.0001$.

4.2.5 Newton's Method (Tangent Method)

If the simple real root of the equation $f(x) = 0$ lies in the interval $< a, b >$, in which the derivatives $f^{'}(x), f^{''}(x)$ also exist, the function f can be expressed in the form of a Taylor expansion at the point x_0:

$$f(x) = f(x_0) + f^{'}(x_0) \cdot (x - x_0) + \frac{1}{2} \cdot f^{''}(\xi_0) \cdot (x - x_0)^2 \; . \tag{4.19}$$

The equation $f(x) = 0$ can be approximated by a linear equation consisting of the first two terms of this expansion (4.19):

$$f(x_0) + f^{'}(x_0) \cdot (x - x_0) = 0 \; . \tag{4.20}$$

The root of the linear equation (4.20) is determined as follows:

$$x_1 = x_0 - \frac{f(x_0)}{f^{'}(x_0)} \; . \tag{4.21}$$

If Equation (4.20) is expressed generally for x_k:

$$f(x_k) + f^{'}(x_k) \cdot (x - x_k) = 0 \ , \tag{4.22}$$

in finding a solution, we obtain a sequence defined by a recurrence formula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f^{'}(x_k)} \ . \tag{4.23}$$

This describes the idea behind Newton's method.

Comment 4.25. According to [8], the zero approximation x_0 should be selected so that it falls between the extreme points a, b such that $\text{sign}(f(x_0)) = \text{sign}(f^{''}(x_0))$. If an inflection point lies between the real root of the initial approximation, the sequence can converge diverge or oscillate to one of the other roots of the function being solved.

Comment 4.26. The algorithm for Newton's method corresponds to the calculation procedure of the simple iteration method for the function:

$$\varphi(x) = x - \frac{f(x)}{f^{'}(x)} \ . \tag{4.24}$$

Comment 4.27. Compared to the previous procedures, the recurrent sequence formula contains the derivative of the function $f(x)$. Numerical differentiation is explained in Chapter 8. The following example focuses only on the task where the derivative of the solved function $f^{'}(x)$ is known.

Example 4.28. For the beam described in Exercise 3.1, determine the largest deflection on the structure in a manner similar to Exercises 4.14, 4.19 or 4.23. Use Newton's method to determine the cross-section with the greatest deflection.

Solution. As we have already determined in Exercise 3.1, the derivative of the slope according to (3.15) is given by:

$$w_z(x)^{''} = \varphi_y^{'}(x) = -\frac{1}{EI_y} \cdot M_y(x) \ , \tag{4.25}$$

and specifically, according to (3.25):

$$\varphi_y^{'}(x) = \frac{1}{EI_y} \cdot \left(q_z \frac{x^2}{2} - \frac{3}{8} q_z l x \right) \ . \tag{4.26}$$

If the derived equation for the slope (3.26) is related to (1.1) by the general expression of the n-th degree polynomial, the vector `c` with elements $[c_n, c_{n-1}$ až $c_1, c_0]$ take the following form:

$$\mathbf{c} = \left[\ \frac{q_z}{2EI_y}\ ,\quad -\frac{3q_z l}{8EI_y}\ ,\quad 0\ \right]\ . \qquad (4.27)$$

The sequence of commands for a given calculation may look like this:

```
format long
qz=4000;
l=6;
b=0.02;
h=0.15;
Iy=1/12*b*h^3;
E=2.1*10^11;
c=[0 1/(48*E*Iy)*qz*l^3 0 -3/(48*E*Iy)*qz*l qz/(24*E*Iy)];
fi=[1/(48*E*Iy)*qz*l^3 0 -3/(16*E*Iy)*qz*l qz/(6*E*Iy)];
m=[0 -3*qz*l/(8*E*Iy) qz/(2*E*Iy)];
xmax=newton_it(3,fi,m,0.0001)
horner(3,fi,xmax)
horner(4,c,xmax)*1000
```

where `newton_it` is an m-function that is called with the input parameter of the zero approximation ($x_0 = 3$ metres) and the tolerance of the error of the result ($\varepsilon = 0.0001$), containing the commands:

```
function xc=newton_it(a,fi,m,tol)
f1=horner(2,m,a);
f2=horner(3,fi,a);
s=a-f2/f1;
fs=horner(3,fi,s);
while abs(fs)>=tol
  a=s;
  f1=horner(2,m,a);
  f2=horner(3,fi,a);
  s=a-f2/f1;
  fs=horner(3,fi,s);
end
xc=s;
```

The solution for the specified input data:

```
xmax =
   2.529166666666667
```

Slope in this cross-section:

```
slope =
   3.740855052600245e-007
```

The maximum deflection in millimetres was then determined:

```
wmax =
   23.769036524717556
```

If we list the values x_i, $f(x_i)$, $f^{'}(x_i)$, x_{i+1} and $f(x_{i+1})$:

```
  i      x(i)     f(x(i))    df(x(i))    x(i+1)  f(x(i+1))
-------------------------------------------------------
  0    3.0000  -0.007619  -0.003810   2.5000   0.000247
  1    2.5000  -0.008466   0.000247   2.5292   0.000000
```

we discover that this method produces a sufficiently accurate solution after the first iteration step. ▲

Examples to Practice

1. Using the calculation procedures explained above, determine the largest deflection on the statically indeterminate beam depicted in Figure 3.2.

2. Calculate the largest deflection on the statically determined beam depicted in Figure 3.3.

Chapter 5

Methods for Sorting a Set of Elements

Objectives

The section introduces:

- the foundations of a sorting algorithm which has numerous engineering applications,
- double `for` loops,
- procedures for reading data to and from a text file.

5.1 Sorting Algorithms

Sorting algorithms can be applied to a wide range of engineering tasks, for example processing experimentally obtained data or measurements. Numerous examples are summarized in [1].

5.1.1 Bubble Sort

Bubble sort is a simple, easily implemented sorting algorithm named after the procedure it is based on. Imagine that the sorted numbers are bubbles rising to the surface of water at different speeds according to their sizes. The elements with larger “bubble” are sorted towards the end of the ascending list. The algorithm iteratively sorts a sequence of n elements, comparing every two adjacent elements and swapping them if they are not in the correct order. Elements are compared until the whole list is sorted.

The algorithm is universal (it compares pairs of elements), functions locally (auxiliary memory is not required), and is stable (elements with the same key do not change their relative position). It is also a natural sorting algorithm (it processes a partially sorted list more quickly than an unsorted one).

```
Input  : $\mathbf{x} = \{x_1, x_2, x_3, \ldots, x_{n-1}, x_n\}$
Output: $\mathbf{x}$
for $j \leftarrow 1, 2, 3, \ldots, n-1$ do
    for $i \leftarrow n-1, n-2, \ldots, j+1, j$ do
        if $x_i > x_{i+1}$ then
            $c \leftarrow x_i$
            $x_i \leftarrow x_{i+1}$
            $x_{i+1} \leftarrow c$
        end
    end
end
```

Algorithm 8: The *bubble sort* algorithm.

The Matlab code for the calculation performed by Algorithm 8 is given below:

```
function y=bubblesort(x);
n=length(x);
for j=1:n-1
  for i=n-1:-1:j
    if x(i)>x(i+1);
      c=x(i);
      x(i)=x(i+1);
      x(i+1)=c;
    end
  end
end
y=x;
```

This procedure is too inefficient for use in larger applications, however and is mainly useful for teaching purposes or simple applications. This type of sorting algorithm is one of the slowest and requires a large number of memory writes compared to other algorithms of the same complexity.

Comment 5.1. From the point of view of using `for` loops, we note that the algorithm contains a double loop with control variables i and j. Sorting an array of n elements requires $n-1$ executions of the outer loop, while the inner loop sequentially executes $(n-1), (n-2), (n-3), \ldots, 2, 1$ times. The final value of the control variable j is therefore not constant but depends on the value of the control variable i. The total number of operations for this type of sorting is $\frac{n^2 - n}{2}$, which corresponds to the task's complexity with an approximate number of calculation operations n^2.

5.1.2 Select Sort

Select sort (often abbreviated *selectsort*) is a simple sorting algorithm based on the direct selection of a minimum. Because of its simple execution, it is often used to organize small amounts of data. Algorithms with less time complexity are used for larger volumes of data.

The algorithm sequentially processes the sorted vector and searches for the minimum of the remaining elements for a given element of the vector. The algorithm's procedure is described below:

1. the algorithm finds the element with the smallest value in the sequence of n values,
2. this element is swapped with the element in the first position to place the element with the smallest value at this point in the sequence,
3. the rest of the sequence is sorted by repeating these steps for the remaining $n - 1$ elements.

The *selectsort* algorithm is therefore based on the idea that if an array is ordered from the smallest to the largest element, then the element with the smallest value will occupy the first position of the sequence, followed by the element with the smallest value from the rest of the array, and so on. The smallest elements from the unsorted section of the array must be selected and placed at the end of the sorted section.

```
Input  : $\mathbf{x} = \{x_1, x_2, x_3, \ldots, x_{n-1}, x_n\}$
Output: $\mathbf{x}$
for $i \leftarrow 1, 2, 3, \ldots, n-1$ do
    for $j \leftarrow i+1, i+2, i+3, \ldots, n$ do
        if $x_i > x_j$ then
            $c \leftarrow x_i$
            $x_i \leftarrow x_j$
            $x_j \leftarrow c$
        end
    end
end
```

Algorithm 9: The *select sort* algorithm.

The MATLAB code for the sorting calculation performed by Algorithm 9 is given below:

```
function y=selectsort(x);
n=length(x);
for i=1:n-1
  for j=i+1:n
    if x(i)>x(j);
      c=x(i);
      x(i)=x(j);
      x(j)=c;
    end
  end
end
y=x;
```

Comment 5.2. From the point of view of using the `for` loop, we again note that the algorithm contains a double loop with control variables i and j, and that the complexity of the task (in terms of number of operations required) remains approximately n^2.

5.1.3 Insert Sort

Insert sort (abbreviated *insertsort*) is another simple sorting algorithm. The algorithm runs by consecutively processing elements and placing each additional unsorted element into the correct position within the sorted sequence. It is one of the quicker sorting algorithms yet still slower than advanced algorithms such as *quicksort* or *shellsort* (see below). However, it has other advantages:

- simple execution,
- efficient with small sets,
- efficient with partially sorted sets,
- more efficient than other algorithms (select sort, bubble sort),
- stable sorting (does not change the relative order of elements with the same keys),
- an online algorithm, which means it allows data sorting as data arrives.

The algorithm sorts elements from the right hand side of the sequence directly into the left hand side, where an ascending list of values is formed. The only difference to the `selectsort` algorithm is in that this method directly manipulates the individual elements during sorting, not just their indices, therefore the calculation procedure may be used, for example, while loading data from a file.

```
Input  : $\mathbf{x} = \{x_1, x_2, x_3, \ldots, x_{n-1}, x_n\}$
Output: $\mathbf{x}$
for $i \leftarrow 2, 3, \ldots, n-1, n$ do
    $c \leftarrow x_i$
    for $j \leftarrow i-1, i-2, i-3, \ldots, 1$ do
        if $x_j > x_{j+1}$ then
            $x_{j+1} \leftarrow x_j$
            $x_j \leftarrow c$
        end
    end
end
```

Algorithm 10: The *insert sort* algorithm.

The MATLAB code for the calculation performed by Algorithm 10 is given below:

```
function y=insertsort(x);
n=length(x);
for i=2:n
  c=x(i);
  for j=i-1:-1:1
    if x(j)>x(j+1);
      x(j+1)=x(j);
      x(j)=c;
    end
  end
end
y=x;
```

The quicksort and shellsort sorting algorithms introduced in the next section are more advanced. These algorithms are presented in this guide to provide insight into the background of high-quality algorithmic procedures.

5.1.4 Quick Sort (Recursive)

Quick (recursive) sorting into classes, or simply *quicksort*, is one of the fastest current sorting algorithms based on the comparison of elements. The algorithm was devised in 1962 by Sir Charles Antony Richard Hoare. It has the best possible average time complexity $(n \cdot \log n)$ among the sorting algorithms of this type, but in the worst case (which can usually be avoided in practice), it can also have a time complexity of n^2.

Quicksort attempts to divide a sorted sequence of numbers into two approximately equal parts. In one part, the numbers are greater, and in the other, they are smaller than the value of one of the selected sequence elements, named the **pivot**. If the pivot is optimally selected, both parts of the sequence will be approximately the same size. Both parts are then sorted separately.

An important performance aspect of this calculation procedure is the choice of pivot. Generally, the pivot is either a fixed element (the first or final element) or a random element. As a fixed element, the choice is problematic with partially ordered arrays or arrays with an underlying structure (where the problem is not divided optimally and the complexity can increase to n^2).

The calculation procedure of *quicksort* can be described as follows:

```
procedure quicksort(list_of_values)
if length(list_of_values) <= 1
  return
pivot = randomly selected element from list_of_values

Divide list_of_values into 3 parts
  list1 = {elements smaller than the pivot}
  list2 = {pivot}
  list3 = {elements larger than the pivot}
list_of_values = quicksort(list1) + list2 + quicksort(list3)
```

The *QuickSort* algorithm is based on “recursion”.

Recursion is a frequently used technique in mathematics and computer science. The term is probably derived from the Latin verb itself, in which case it is called a recursive function. An integral part of the recursive function is a terminating condition that specifies when the nested loop ends. Because this is a frequent source of error, it must be designed in a sufficiently robust manner, and all possible conditions of its operation must be checked. (For details see, e.g., [14])

With smaller input sizes, *quicksort* is slow since it is very likely that the array will not be split into ideal halves. For this reason, *quicksort* is only used with large arrays. It is much more efficient to use *insertsort* or *shellsort* to sort smaller arrays (see below).

One method of implementing recursive sorting with *quicksort* in Matlab is given below:

```
function y=quicksort(x)
n=length(x);
if(n<=1);y=x;return;end;
if(n==2)
  if(x(1)>x(2))
     x=[x(2); x(1)];
  end
  y=x;
  return;
end
m=fix(n/2);
pivot=x(m);
Smaller=find(x<pivot);
if(isempty(Smaller))
  ind=find(x>pivot);
  if(isempty(ind));y=x;return;end;
  pivot=x(ind(1));
  Smaller=find(x<pivot);
end
Larger=find(x>=pivot);
y=[quicksort(x(Smaller));quicksort(x(Larger))];
```

Comment 5.3. The algorithm assumes that the ordered vector is presented in columns. We therefore call the m-function with a vector in an appropriate form; for example:

```
A=[76 5 44 90 59 63 4 1 28 57];
B=quicksort(A')
```

Comment 5.4. This m-function uses the MATLAB `fix` function for integer division and the `find` function to obtain a vector of elements of the sequence of values x that satisfy the condition described in the input. The logical function `isempty` tests the content of the variable (vector) in the parameter. If the variable is empty, the result of the function is the logical value TRUE (i.e., the value 1). The `return` command does not interrupt the calculation, as in the case of the `break` command, it terminates the called function only (for recursive calculations, this does not end the calculation but returns to the part of the algorithm from which the function was called).

5.1.5 Shell Sort

Shell sort (abbreviated *shellsort*) is decreasing increment sort and similar to *insert-sort*. It was discovered and published in 1959 by Donald Shell. The time complexity of *shellsort* is approximately equal to $n^{\frac{3}{2}}$ and is the most powerful of the algorithms whose complexity is n^2.

A common problem with sorting algorithms is sorting elements into the opposite part of the array from their original location. In normal quadratic algorithms, these elements must gradually “traverse” the entire array. In this respect, *Shellsort* is different because it does not compare adjacent elements but elements at a certain distance at each step (this distance decreases at each step until it is reduced to 1). This ensures that elements moved to the wrong side of the array can be quickly moved to their correct positions.

The algorithm’s main challenge is in selecting the ideal distance for a comparison of individual elements. Donald Shell originally suggested that comparisons should start with a spacing value of $\frac{n}{2}$, where n is the array size, and therefore always halve the distance between compared elements. The disadvantage of this approach is that the elements in even and odd positions are compared only in the algorithm’s final step. Other methods include, for example, selecting a $2 \cdot k - 1$ (Hibbard) sequence with a complexity of $n^{\frac{3}{2}}$, the $9 \cdot 4^i - 9 \cdot 2^i$ (Sedgewick) sequence with a complexity of $n^{\frac{4}{2}}$, or the Fibonacci sequence multiplied twice by the golden ratio. The best results are obtained with the sequence $1, 4, 10, 23, 57, 132, 301, 701, 1750$ or `gap`·2.2, created by Marcin Ciura.

The calculation procedure of the *shellsort* algorithm can be programmed in an m-function, for example, as shown below:

```
function y=shellsort(x)
n=length(x);
gap=floor(n/2);
while gap>0
  for j=gap:n
    for i=j-gap:-gap:1
      if x(i+gap)>=x(i)
        break;
      else
        c=x(i);
        x(i)=x(i+gap);
        x(i+gap)=c;
      end
    end
  end
  gap=floor(gap/2);
end
y=x;
```

Comment 5.5. The m-function uses the `floor` function, which rounds the number to the nearest integer in the direction of $-\infty$; for example, `floor(-0.4)` is -1.

5.2 Working with Text Files

Sorting algorithms operations can be supplemented by loading numerical values from a text file into the vector field in MATLAB. This can then be sorted by one of the described algorithms. Text files can also be used to write and save the results of computations or for data preparation.

Example 5.6. Save the table of values of the function e^x for $x = 0; 0.1; 0.2; \ldots; 1.0$ in a text file named `exp.txt`.

Solution. Data may be written into a text file, for example, by using the MATLAB commands `fopen`, `fprintf` and `fclose`. A solution to the exercise is the following sequence of commands:

```
x=0:.1:1;
y=[x;exp(x)];
fid=fopen('exp.txt','w');
fprintf(fid,'%6.2f %12.8f\n',y);
fclose(fid);
type exp.txt
```

In the `fopen` function, the access mode to the file whose name contains the first parameter must be specified with the second parameter. The basic options include the parameter **"r"** for opening a file for reading and **"w"** for writing to the file (in the latter case, any existing content in the file is overwritten).

The resulting content in the text file named `exp.txt` is determined by the command `type`. The file should contain the data:

```
  0.00   1.00000000
  0.10   1.10517092
  0.20   1.22140276
  0.30   1.34985881
  0.40   1.49182470
  0.50   1.64872127
  0.60   1.82211880
  0.70   2.01375271
  0.80   2.22554093
  0.90   2.45960311
  1.00   2.71828183
```

▲

Example 5.7. Use the `randi` function to randomly generate 10,000 numbers ranging from 0 to 1,000,000 and write these values into a text file named `values.txt`.

Solution. To generate a specified number of random numbers in a predetermined range of values has, the correctly called form of the command is:

```
randi(1000000,1,10000)
```

The correct sequence of commands in combination with writing to a text file is, for example:

```
x=randi(1000000,1,10000);
fid=fopen('values.txt','w');
fprintf(fid,'%7g\n',x);
fclose(fid);
type values.txt
```

▲

Example 5.8. Load the contents of `values.txt` file you have just created into the variable A, using the `dlmread` command.

Solution. This command is very simple and takes the form:

```
A=dlmread('values.txt')
```

However, the values.txt file must contain numerical values only. ▲

Comment 5.9. More generally, the operation from the previous example can be performed as follows:

```
clear;
fid=fopen('values.txt');
k=0;
while feof(fid)==0
    rd=fscanf(fid,'%f',1);
    if ~isempty(rd)
       k=k+1;
       A(k)=rd;
    end
end
fclose(fid);
A'
```

The m-function described above contains the `feof` function, which returns the value TRUE, i.e., 1, if the program reaches the end of the text file while reading (otherwise, it returns the value FALSE, i.e., 0).

Example 5.10. Sort the contents of variable A from the previous example using the sorting algorithms described in this chapter.

Comment 5.11. If it is necessary to monitor the machine's computation time, we can take advantage of the function pair `tic` and `toc`:

```
clc
tic;
B=bubblesort(A);
toc
```

The computation time output may look like the following:

```
Elapsed time is 1.790291 seconds.
```

Example 5.12. Modify the computation procedures of the sorting algorithms so that the result is a list of values sorted in descending order.

Comment 5.13. To check the correctness of an m-function, we can use MATLAB's `sort` (sorts vectors and matrices in ascending and descending order) and `issorted` (returns the value TRUE, i.e., 1, for a sorted vector or matrix, or FALSE, i.e., 0) commands, where the input parameter is simply the respective vector or matrix.

Chapter 6

Systems of Linear Equations

Objectives

This chapter describes:

- algorithms for solving systems of linear equations, using direct methods,
- iterative methods for solving systems of linear equations,
- triple `for` loops,
- matrix calculus.

Many problems in structural mechanics require the solution of a system of linear equations; numerical methods designed for this purpose are therefore commonly found in programming.

In general, a system of n linear equations with n variables can be written as:

$$\begin{array}{ccccccccc} a_{1,1}\cdot x_1 & + & a_{1,2}\cdot x_2 & + & \cdots & + & a_{1,n}\cdot x_n & = & b_1 \\ a_{2,1}\cdot x_1 & + & a_{2,2}\cdot x_2 & + & \cdots & + & a_{2,n}\cdot x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{n,1}\cdot x_1 & + & a_{n,2}\cdot x_2 & + & \cdots & + & a_{n,n}\cdot x_n & = & b_n \end{array}, \tag{6.1}$$

where variables $x_1, \ldots, x_n$, generally x_i for $i = 1, \ldots, n$, are unknown, $a_{i,j}$ for $i, j = 1, \ldots, n$ are the coefficients of the system of equations, and the numbers b_i for $i = 1, \ldots, n$ are the absolute members of the system (or also the right or right-hand side of the system).

Matrix calculus is used to solve the roots of systems of linear equations. The coefficients of the system can be written in the form of a matrix:

$$[A] = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}, \tag{6.2}$$

which is referred to as the matrix of the system.

The unknown and the right-hand side of the system can be expressed as vectors:

$$\{x\} = \begin{Bmatrix} x_1 & x_2 & \cdots & x_n \end{Bmatrix}^T , \tag{6.3}$$

$$\{b\} = \begin{Bmatrix} b_1 & b_2 & \cdots & b_n \end{Bmatrix}^T . \tag{6.4}$$

The entire system of linear equations can then be expressed as a matrix:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \cdot \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix} , \tag{6.5}$$

or abbreviated in matrix form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} , \tag{6.6}$$

where **A** denotes the matrix of the system (i.e., the left(-hand) sides of the equations), **x** denotes the column vector of the unknown roots of the system, and **b** denotes the column vector of the right-hand sides of the equations.

One of the conditions for the unique solution of a system of linear equations is that the matrix **A** must be **regular**.

Comment 6.1. A **Regular matrix** is a square matrix whose determinant is non-zero. The opposite of a regular matrix is a **singular matrix** with a determinant of zero. An important property of a regular matrix is that we can calculate a unique inverse matrix. This is useful, for example, for solving a system of linear equations.

6.1 Direct Methods of Solving Systems of Linear Equations

Methods for solving systems of linear equations that lead to an exact solution (if rounding errors are not taken into account) in a finite number of computational steps are referred to as *direct methods.* Their basic feature is the elimination of unknowns. For full matrices, these methods tend to be the most efficient, but with a large number of equations, the calculation may be limited by the computer's memory.

6.1.1 Solving a Triangular System of Linear Equations

A general upper triangular system of linear equations can be written in the form:

$$\begin{array}{ccccccccc} a_{1,1} \cdot x_1 & + & a_{1,2} \cdot x_2 & + & \cdots & + & a_{1,n} \cdot x_n & = & b_1 \\ & & a_{2,2} \cdot x_2 & + & \cdots & + & a_{2,n} \cdot x_n & = & b_2 \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{n,n} \cdot x_n & = & b_n \end{array} , \tag{6.7}$$

or as a matrix

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ & a_{2,2} & \cdots & a_{2,n} \\ & & \ddots & \vdots \\ & & & a_{n,n} \end{bmatrix} \cdot \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}, \qquad (6.8)$$

The solution, referred to as back substitution (backtracking), is expressed by Algorithm 11.

```
Input  : $n$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Output: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}^T$
for $i \leftarrow n, n-1, \ldots, 2, 1$ do
    $x_i \leftarrow \dfrac{b_i - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j}{a_{i,i}}$
end
```

Algorithm 11: The backward substitution algorithm.

Calculation of the upper triangular system of linear equations can be programmed in MATLAB as follows:

```
n=input('Enter the number of unknowns in the system of eqs:\n n=');
A=zeros(n,n);
fprintf('\n Enter the matrix elements of system A:');
for i=1:n
  for j=i:n
    fprintf('\n A[%d,%d]=',i,j)
    A(i,j)=input('');
  end
end
if det(A)==0
  error('The system of equations is singular! Det(A) is equal to 0!')
end
fprintf('\n Enter the elements of the vector b:');
for i=1:n
    fprintf('\n b[%d]=',i)
    b(i)=input('');
end
if n==1
  x(1)=b(1)/A(1,1);
else
  for i=n:-1:1
    s=0;
    if i<n
      for j=i+1:n
        s=s+A(i,j)*x(j);
      end
    end
    x(i)=(b(i)-s)/A(i,i);
  end
end
fprintf('\n')
disp('System roots')
disp('--------------------------------')
for i=1:n
  fprintf('x[%d]=%16.8f\n',i,x(i))
end
```

Comment 6.2. In the example above, the `input` command is used to enter input data, which allows a description of the entered quantity to be displayed on screen and a value to be assigned to the variable by typing them directly on the keyboard.

Example 6.3. Determine the roots of the triangular system of linear equations of order 4:

$$\begin{array}{ccccccccc} x_1 & + & 2\cdot x_2 & + & 3\cdot x_3 & + & 4\cdot x_4 & = & 2 \\ & & 2\cdot x_2 & + & 6\cdot x_3 & + & 12\cdot x_4 & = & 8 \\ & & & & 6\cdot x_3 & + & 24\cdot x_4 & = & 18 \\ & & & & & & 24\cdot x_4 & = & 24 \end{array} \tag{6.9}$$

Solution. The resulting vector of unknown roots is $x = \begin{Bmatrix} -1 & 1 & -1 & 1 \end{Bmatrix}^T$. ▲

Comment 6.4. We can check the correctness of the solution by re-substituting the roots into the individual equations of the system, or better, by subtracting the left-hand side of the system from the right-hand side, whereby we can then obtain the *residual vector* of the solution **r**:

```
fprintf('\n')
disp('Residual vector')
disp('--------------------------------')
for i=1:n
  r(i)=0;
  for j=i:n
    r(i)=r(i)+A(i,j)*x(j);
  end
  r(i)=r(i)-b(i);
  fprintf('r[%d]=%16.8f\n',i,r(i))
end
```

Individual elements should converge to zero. We obtain a residual vector for the triangular system:

```
r[1] = 0.00000000
r[2] = 0.00000000
r[3] = 0.00000000
r[4] = 0.00000000
```

Comment 6.5. We can also check the accuracy of the solution using the *Euclidean norm of the residual vector* $\mathbf{r}$, given by the expression $\sqrt{\sum_i |r_i|^2}$, which can be called in MATLAB with the command `norm(A*x-b)`.

Comment 6.6. Difficulties arise with the algorithm if the numbers on the diagonal, i.e., $a_{i,i}$, are small. The system matrix $\mathbf{A}$ is then almost singular ($\det \mathbf{A} \approx 0$). A solution is to rearrange the system of linear equations so that the largest matrix elements of the system $\mathbf{A}$ are on the diagonal.

Examples to Practice

!

1. Design an algorithm to solve a general triangular system of linear equations which has a lower matrix of the system $\mathbf{A}$ (zeros above the diagonal).

6.1.2 The Gaussian Elimination Method

Gaussian elimination is one of the oldest numerical methods. It is based on converting the initial matrix $\mathbf{A}$ to an upper triangular matrix. By adjusting rows with *multipliers*, this matrix is fine-tuned into a form where only zeros are located below the main diagonal. The adjusted matrix then corresponds to a system of equations that is equivalent to the original system and can be solved in a manner similar to a triangular system of linear equations by using back substitution (backtracking). The entire calculation procedure is schematically described by Algorithm 12.

The m-function in MATLAB looks as follows:

```
function x=gauss(A,b)
if det(A)==0
  error('The system of equations is singular! Det(A) is equal to 0!')
  return
end
n=length(A);
if n==1
  x(1)=b(1)/A(1,1);
  return
end
```

```
Input  : $n$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Output: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}^T$
for $k \leftarrow 1, 2, \ldots, n-2, n-1$ do
    for $i \leftarrow k+1, k+2, \ldots, n-1, n$ do
        $m \leftarrow -\frac{a_{i,k}}{a_{k,k}}$
        for $j \leftarrow k, k+1, \ldots, n-1, n$ do
            $a_{i,j} \leftarrow a_{i,j} + m \cdot a_{k,j}$
        end
        $b_i \leftarrow b_i + m \cdot b_k$
    end
end
for $i \leftarrow n, n-1, \ldots, 2, 1$ do
    $x_i \leftarrow \frac{b_i - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j}{a_{i,i}}$
end
```

Algorithm 12: The Gaussian elimination algorithm.

```
for k=1:n-1
  if A(k,k)==0
    error('There is a zero on the diagonal!')
    return
  end
  for i=k+1:n
    m=-A(i,k)/A(k,k);
    for j=k:n
      A(i,j)=A(i,j)+m*A(k,j);
    end;
    b(i)=b(i)+m*b(k);
  end
end
for i=n:-1:1
  s=0;
  if i<n
    for j=i+1:n
      s=s+A(i,j)*x(j);
    end
  end
  x(i)=(b(i)-s)/A(i,i);
end
```

Example 6.7. Using the m-function above, solve the roots of the system of four linear equations:

$$\begin{array}{ccccccccc} 2\cdot x_1 & - & x_2 & + & 3\cdot x_3 & - & x_4 & = & 7 \\ x_1 & - & x_2 & + & 4\cdot x_3 & - & 2\cdot x_4 & = & 5 \\ 3\cdot x_1 & + & 2\cdot x_2 & + & x_3 & + & 4\cdot x_4 & = & 31 \\ 4\cdot x_1 & - & 3\cdot x_2 & + & 3\cdot x_3 & - & 3\cdot x_4 & = & -5 \end{array} \tag{6.10}$$

Solution. The solution is the vector of unknown roots $\{x\} = \begin{Bmatrix}1 & 2 & 4 & 5\end{Bmatrix}^T$. This example demonstrates how the Gaussian elimination algorithm works:

```
Original matrix A              Original vector b
----------------------------  ----------------------------
  2.000 -1.000  3.000 -1.000    7.000
  1.000 -1.000  4.000 -2.000    5.000
  3.000  2.000  1.000  4.000   31.000
  4.000 -3.000  3.000 -3.000   -5.000

Modified matrix A             Modified vector b
----------------------------  ----------------------------
  2.000 -1.000  3.000 -1.000    7.000
  0.000 -0.500  2.500 -1.500    1.500
  0.000  0.000 14.000 -5.000   31.000
  0.000  0.000  0.000 -0.857   -4.286

The roots of the system are:
----------------------------
x[ 1] =    1.000
x[ 2] =    2.000
x[ 3] =    4.000
x[ 4] =    5.000

The residual vector is:
----------------------------
r[ 1] =  0.000e+000
r[ 2] =  0.000e+000
r[ 3] = -7.105e-015
r[ 4] = -1.776e-015
```

▲

Example 6.8. Determine the roots of the system of three linear equations with the matrix of the system

$$[A] = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \tag{6.11}$$

and the vector of the right-hand sides:

$$\{b\} = \begin{Bmatrix} 1 & 4 & 1 \end{Bmatrix}^T . \tag{6.12}$$

Solution. The vector of unknown roots is equal to $\{x\} = \begin{Bmatrix} 3 & -5 & 3 \end{Bmatrix}^T$. ▲

Example 6.9. Solve the system of four linear equations given by the matrix of the system

$$[A] = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \tag{6.13}$$

and the vector of the right-hand sides:

$$\{b\} = \begin{Bmatrix} 1 & 2 & 0 & 1 \end{Bmatrix}^T . \tag{6.14}$$

Solution. The resulting vector of the unknown roots of the system of four linear equations is equal to $\{x\} = \begin{Bmatrix} 0.5 & 0.75 & 0.25 & 0.5 \end{Bmatrix}^T$. ▲

Example 6.10. Determine the machine time and accuracy of the solution (the norm of the residual vector) of a randomly generated system of 600 linear equations.

Solution. The example can be solved using the following sequence of commands:

```
clc;
clear;
n=600;
m=1200;
A=randn(n,m);
A=A*A';
b=randn(n,1);
tic, x=gauss(A,b); toc
norm(A*x'-b)
```

▲

Example 6.11. Using the general joint method, solve the reactions and internal forces for the truss illustrated in Figure 6.1. The input parameters are $b = 3$ m, $h = 1.5$ m, $F_1 = 5$ kN and $F_2 = 12$ kN. Then solve the system of linear equations using Gaussian elimination.

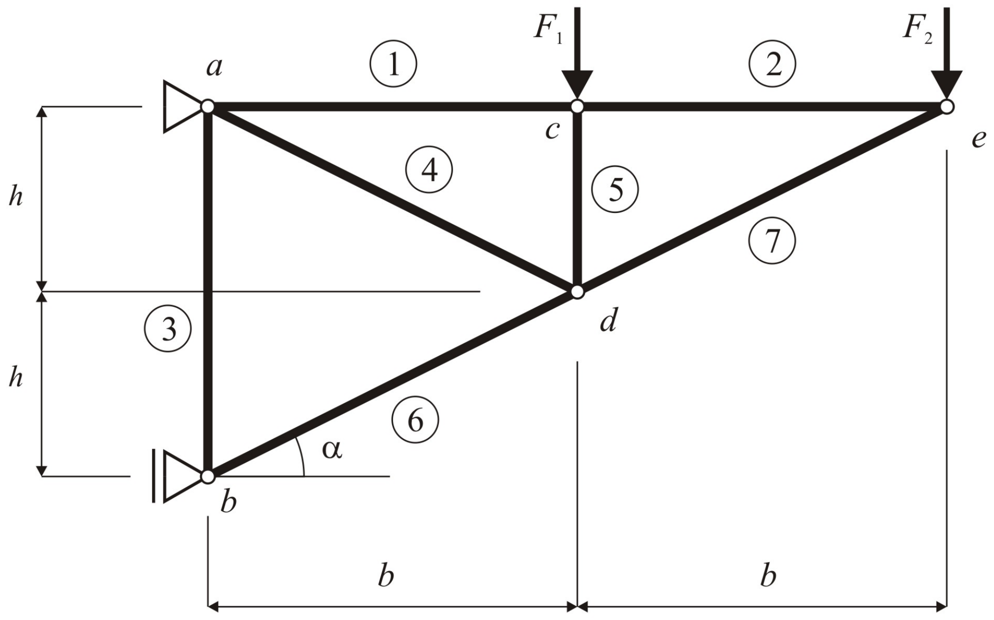


Fig. 6.1 Statics diagram of the truss.

Solution. If the truss in Figure 6.1 is interpreted as a system of material points, from a kinematic point of view it contains $2 \cdot s$ degrees of freedom, where s is the number of material points, i.e. joints. Because the structure consists of five joints ($s = 5$), we obtain $n_v = 2 \cdot s = 10$ degrees of freedom, which are removed by three external ($v_\mathrm{e} = 3$) and seven internal ($v_i = 7$) connections (number of rods). Since $n_v = v_\mathrm{e} + v_i$, it is a statically and kinematically definite construction.

If two equilibrium conditions are specified in each joint, we can obtain a total of ten equilibrium conditions which form a system of linear equations, from which we can then determine ten unknowns, i.e., three reactions (R_{ax}, R_{az} and R_{bx}) and seven internal forces ($N_1, N_1, \ldots, N_7$).

The individual equilibrium conditions are:

- Joint a:
  1. $R_x = 0 : -R_{ax} + N_1 + N_4 \cdot \cos(\alpha) = 0$
  2. $R_z = 0 : -R_{az} + N_3 + N_4 \cdot \sin(\alpha) = 0$
- Joint b:
  3. $R_x = 0 : +R_{bx} + N_6 \cdot \cos(\alpha) = 0$
  4. $R_z = 0 : -N_3 - N_6 \cdot \sin(\alpha) = 0$
- Joint c:
  5. $R_x = 0 : -N_1 + N_2 = 0$

6. $R_z = 0 : +F_1 + N_5 = 0$

- Joint *d*:

7. $R_x = 0 : -N_4 \cdot \cos(\alpha) - N_6 \cdot \cos(\alpha) + N_7 \cdot \cos(\alpha) = 0$
8. $R_z = 0 : -N_4 \cdot \sin(\alpha) - N_5 + N_6 \cdot \sin(\alpha) - N_7 \cdot \sin(\alpha) = 0$

- Joint e:

9. $R_x = 0 : -N_2 - N_7 \cdot \cos(\alpha) = 0$
10. $R_z = 0 : +F_2 + N_7 \cdot \sin(\alpha) = 0$

The entire system of linear equations of order 10 can be clearly written as a matrix:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \ , \tag{6.15}$$

where **A** denotes the matrix of the left-hand sides of the equations, containing the geometry of the structure:

$$\begin{bmatrix} -1 & 0 & 0 & +1 & 0 & 0 & +\cos(\alpha) & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & +1 & +\sin(\alpha) & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 & 0 & 0 & 0 & 0 & +\cos(\alpha) & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -\sin(\alpha) & 0 \\ 0 & 0 & 0 & -1 & +1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\cos(\alpha) & 0 & -\cos(\alpha) & +\cos(\alpha) \\ 0 & 0 & 0 & 0 & 0 & 0 & -\sin(\alpha) & -1 & +\sin(\alpha) & -\sin(\alpha) \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -\cos(\alpha) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & +\sin(\alpha) \end{bmatrix} , \tag{6.16}$$

x represents the column vector of unknown roots, containing the ten unknown reactions and internal forces:

$$\left\{ R_{ax} \quad R_{az} \quad R_{bz} \quad N_1 \quad N_2 \quad N_3 \quad N_4 \quad N_5 \quad N_6 \quad N_7 \right\}^T \tag{6.17}$$

and **b** represents the column vector of the right-hand sides of the equations, containing the nodal load of the truss:

$$\left\{ 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad -F_1 \quad 0 \quad 0 \quad 0 \quad -F_2 \right\}^T \ . \tag{6.18}$$

The trigonometric functions $\cos(\alpha)$ and $\sin(\alpha)$ contained in the matrix **A** can be expressed directly from the truss' dimensions:

$$\cos(\alpha) = \frac{b}{l} \ , \quad \sin(\alpha) = \frac{h}{l} \ , \tag{6.19}$$

where l is the length of rods No. 4, 6 and 7:

$$l = l_4 = l_6 = l_7 = \sqrt{b^2 + h^2} \ . \tag{6.20}$$

Given the condition of the solution of the Gaussian method that the elements on the diagonal of the matrix **A** must not be equal to 0, the matrix **A** and the vector of the right-hand sides **b** must be adjusted by suitably rearranging the order of the junction equations, namely:

- the 5th joint equation is moved to the 4th row,
- the 9th joint equation is moved to the 5th row,
- the 4th joint equation is moved to the 6th row,
- the 6th joint equation is moved to the 8th row,
- the 8th joint equation is moved to the 9th row.

The modified matrix of the left-hand sides **A** therefore has the resulting form:

$$\begin{bmatrix} -1 & 0 & 0 & +1 & 0 & 0 & +\cos(\alpha) & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & +1 & +\sin(\alpha) & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 & 0 & 0 & 0 & 0 & +\cos(\alpha) & 0 \\ 0 & 0 & 0 & -1 & +1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -\cos(\alpha) \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -\sin(\alpha) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\cos(\alpha) & 0 & -\cos(\alpha) & +\cos(\alpha) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\sin(\alpha) & -1 & +\sin(\alpha) & -\sin(\alpha) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & +\sin(\alpha) \end{bmatrix} , \qquad (6.21)$$

and the column vector of the right-hand sides of the equations **b**:

$$\left\{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad -F_1 \quad 0 \quad -F_2\right\}^T . \qquad (6.22)$$

The column vector of unknown roots **x** remains unchanged.

The system of linear equations constructed in this way can be solved for specifically entered input quantities by using the Gaussian elimination method:

```
The roots of the system are:
-----------------------------------
x[ 1] =   29.000 kN
x[ 2] =   17.000 kN
x[ 3] =   29.000 kN
x[ 4] =   24.000 kN
x[ 5] =   24.000 kN
x[ 6] =   14.500 kN
x[ 7] =    5.590 kN
x[ 8] =   -5.000 kN
x[ 9] =  -32.423 kN
x[10] =  -26.833 kN
```

```
The norm of the residual vector is:
-----------------------------------
n =
     0
```

▲

Example 6.12. Solve the reactions and internal forces for the truss depicted in Figure 6.2 using the general joint method. The input parameters are $b = 4$ m, $h = 4$ m, $F_1 = 8$ kN and $F_2 = 14$ kN. Then solve the system of linear equations using the Gaussian elimination method.

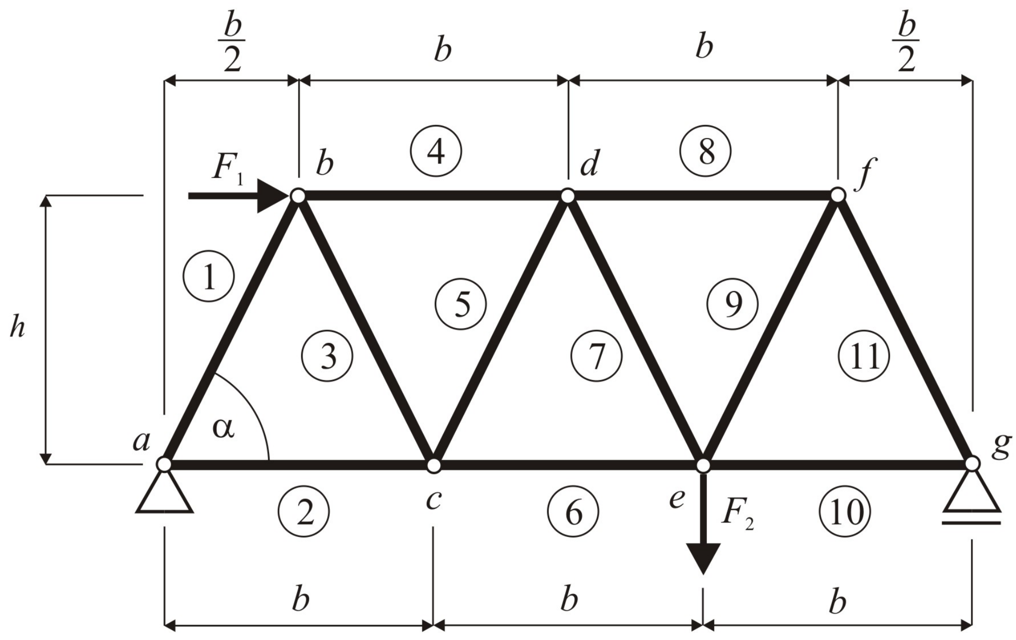


Fig. 6.2 Statics diagram of the truss.

Solution. If the truss in Figure 6.2 is interpreted in the same way as in Example 6.11, i.e., as a system of material points, from a kinematic point of view, it contains $2 \cdot s$ degrees of freedom, where s is the number of material points, or joints. Because the structure contains seven joints ($s = 7$), we obtain $n_v = 2 \cdot s = 14$ degrees of freedom, which are removed by three external ($v_e = 3$) and eleven internal ($v_i = 11$) connections (number of rods). Since $n_v = v_e + v_i$, it is a statically and kinematically definite structure and can only be solved using equilibrium conditions.

If two equilibrium conditions are specified in each joint, we can obtain a total of fourteen equilibrium conditions which form a system of linear equations, from which we can then determine fourteen unknowns, i.e., three reactions (R_{ax}, R_{az} and R_{gz}) and eleven internal forces ($N_1, N_1, \ldots, N_{11}$).

The individual equilibrium conditions are:

- Joint a:
  1. $R_x = 0 : -R_{ax} + N_1 \cdot \cos(\alpha) + N_2 = 0$
  2. $R_z = 0 : -R_{az} - N_1 \cdot \sin(\alpha) = 0$
- Joint b:
  3. $R_x = 0 : +F_1 - N_1 \cdot \cos(\alpha) + N_3 \cdot \cos(\alpha) + N_4 = 0$
  4. $R_z = 0 : +N_1 \cdot \sin(\alpha) + N_3 \cdot \sin(\alpha) = 0$
- Joint c:
  5. $R_x = 0 : -N_2 - N_3 \cdot \cos(\alpha) + N_5 \cdot \cos(\alpha) + N_6 = 0$
  6. $R_z = 0 : +N_3 \cdot \sin(\alpha) + N_5 \cdot \sin(\alpha) = 0$
- Joint d:
  7. $R_x = 0 : -N_4 \cdot \cos(\alpha) - N_5 \cdot \cos(\alpha) + N_7 \cdot \cos(\alpha) + N_8 = 0$
  8. $R_z = 0 : +N_5 \cdot \sin(\alpha) + N_7 \cdot \sin(\alpha) = 0$
- Joint e:
  9. $R_x = 0 : -N_8 - N_9 \cdot \cos(\alpha) + N_{11} \cdot \cos(\alpha) + N_{12} = 0$
  10. $R_z = 0 : +N_9 \cdot \sin(\alpha) + N_{11} \cdot \sin(\alpha) = 0$
- Joint f:
  11. $R_x = 0 : -N_{10} - N_{11} \cdot \cos(\alpha) + N_{13} \cdot \cos(\alpha) = 0$
  12. $R_z = 0 : +N_{11} \cdot \sin(\alpha) + N_{13} \cdot \sin(\alpha) = 0$
- Joint g:
  13. $R_x = 0 : -N_{12} - N_{13} \cdot \cos(\alpha) = 0$
  14. $R_z = 0 : -N_{13} \cdot \sin(\alpha) - R_{gz} = 0$

The entire system of linear equations of order 14 can be clearly written as a matrix, as with Example 6.11:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}\,, \tag{6.23}$$

where **A** denotes the matrix of the left-hand sides of the equations, containing the geometry data of the structure, **x** represents the column vector of unknown roots, containing the fourteen unknown reactions and internal forces (selected order $\mathbf{x} = \left\{R_{ax} \quad R_{az} \quad N_1 \quad N_2 \quad \ldots \quad N_{11} \quad R_{gz}\right\}^T$), and **b** is the column vector of the right-hand sides of the equations, containing the nodal loads of the truss.

Some rows of matrix **A** contain a zero, and therefore it is necessary to rearrange the order of the joint equations:

- the 5th joint equation is moved to the 4th row,
- the 7th joint equation is moved to the 5th row,

- the 9th joint equation is moved to the 8th row,
- the 11th joint equation is moved to the 10th row,
- the 13th joint equation is moved to the 12th row.

The list of the resulting values of the roots of the system is as follows:

```
The roots of the system are:
-----------------------------------
x[ 1] =    8.000 kN
x[ 2] =    2.000 kN
x[ 3] =   -2.236 kN
x[ 4] =    9.000 kN
x[ 5] =    2.236 kN
x[ 6] =  -10.000 kN
x[ 7] =   -2.236 kN
x[ 8] =   11.000 kN
x[ 9] =    2.236 kN
x[10] =  -12.000 kN
x[11] =   13.416 kN
x[12] =    6.000 kN
x[13] =  -13.416 kN
x[14] =   12.000 kN

The residual vector is:
-----------------------------------
r[ 1] =          0
r[ 2] =          0
r[ 3] =          0
r[ 4] =          0
r[ 5] =          0
r[ 6] =          0
r[ 7] =          0
r[ 8] =  -8.882e-16
r[ 9] =  -6.661e-16
r[10] =   8.882e-16
r[11] =   1.776e-15
r[12] =          0
r[13] =  -1.776e-15
r[14] =  -1.776e-15

The norm of the residual vector is:
-----------------------------------
n =
   3.3894e-015
```

▲

Comment 6.13. Because the nodes are labelled appropriately and the system's equations are suitably formulated, the matrix of the system $\mathbf{A}$ in Example 6.12 takes the form of a a band matrix (for details, see Chapter 6.2.3). In the resulting system matrix, the non-zero elements are at most two columns to the left and four columns to the right from the diagonal (the width of the strip is therefore 7). As an extension to the exercise, try to modify the Gaussian elimination algorithm to account for the bandwidth and reduce the number of computations.

6.1.3 The Gauss-Jordan Method

The Gauss-Jordan method is a modified Gaussian elimination method. This computational procedure modifies the system matrix $\mathbf{A}$ into a diagonal or even identity matrix. The calculation procedure is performed by Algorithm 13 and is executed by a MATLAB m-function:

```
function x=gauss_jordan(A,b)
if det(A)==0
  error('The system of equations is singular! Det(A) is equal to 0!')
  return
end
n=length(A);
if n==1
  x(1)=b(1)/A(1,1);
  return
end
for k=1:n
  if A(k,k)==0
    error('There is a zero on the diagonal!')
    return
  end
  for i=1:n
    if ~(i==k)
      m=-A(i,k)/A(k,k);
      for j=k:n
        A(i,j)=A(i,j)+m*A(k,j);
      end;
      b(i)=b(i)+m*b(k);
    end
  end
end
for i=1:n
```

```
  x(i)=b(i)/A(i,i);
end
```

```
Input  : $n$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Output: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}^T$
for $k \leftarrow 1, 2, \ldots, n-2, n$ do
    for $i \leftarrow 1, 2, \ldots, k-1, k+1, \ldots, n$ do
        $m \leftarrow -\frac{a_{i,k}}{a_{k,k}}$
        for $j \leftarrow 1, 2, \ldots, n-1, n$ do
            $a_{i,j} \leftarrow a_{i,j} + m \cdot a_{k,j}$
        end
        $b_i \leftarrow b_i + m \cdot b_k$
    end
end
for $i \leftarrow 1, 2, \ldots, n-1, n$ do
    $x_i \leftarrow \frac{b_i}{a_{i,i}}$
end
```

Algorithm 13: Gauss-Jordan method algorithm.

The Gauss-Jordan method can be used to solve matrix equations, written as follows:

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B}\,, \tag{6.24}$$

where $\mathbf{X}$ is the matrix of the roots of the system, and $\mathbf{B}$ is the matrix of the left--hand sides; both have the general dimension $[n, r]$. The calculation procedure must therefore be modified to work with more right-hand sides, for example, as described by the modified Algorithm 14.

```
Input  : $n$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{B} = [b_{i,j}] = [b_{1,1}, \ldots, b_{n,r}]$
Output: $\mathbf{X} = [x_{i,j}] = [x_{1,1}, \ldots, x_{n,r}]$
for $k \leftarrow 1, 2, \ldots, n-2, n$ do
    for $i \leftarrow 1, 2, \ldots, k-1, k+1, \ldots, n$ do
        $m \leftarrow -\frac{a_{i,k}}{a_{k,k}}$
        for $j \leftarrow 1, 2, \ldots, n-1, n$ do
            $a_{i,j} \leftarrow a_{i,j} + m \cdot a_{k,j}$
        end
        for $j \leftarrow 1, 2, \ldots, r-1, r$ do
            $b_{i,j} \leftarrow b_{i,j} + m \cdot b_{k,j}$
        end
    end
end
for $j \leftarrow 1, 2, \ldots, r-1, r$ do
    for $i \leftarrow 1, 2, \ldots, n-1, n$ do
        $x_{i,j} \leftarrow \frac{b_{i,j}}{a_{i,i}}$
    end
end
```

Algorithm 14: Gauss-Jordan method algorithm for solving matrix equations.

Example 6.14. Calculate the matrix equation using the Gauss-Jordan method:

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}^T = \begin{bmatrix} 1 & 4 & 1 \\ 2 & 2 & 1 \\ 3 & -1 & 2 \end{bmatrix}^T . \tag{6.25}$$

Solution. The solution is the matrix:

$$[X] = \begin{bmatrix} 3 & -5 & 3 \\ 2 & -2 & 1 \\ -2 & 7 & -3 \end{bmatrix}^T . \tag{6.26}$$

▲

The solution of a matrix equation can be used, for example, to solve an **inverse matrix**, when the matrix of the right-hand sides is formed by an identity diagonal matrix, or to calculate a truss with multiple load states, such as the example in Exercise 6.11.

Example 6.15. Use the Gauss-Jordan method to calculate the inverse matrix of the original matrix:

$$\begin{bmatrix} 1 & 2 & 6 \\ 2 & 5 & 15 \\ 6 & 15 & 46 \end{bmatrix} \tag{6.27}$$

Solution. The solution is the inverse matrix:

$$[A]^{-1} = \begin{bmatrix} 5 & -2 & 0 \\ -2 & 10 & -3 \\ 0 & -3 & 1 \end{bmatrix} , \tag{6.28}$$

which can be verified by:

```
A =                         B =                         A*B =
     1     2     6               5    -2     0               1     0     0
     2     5    15              -2    10    -3               0     1     0
     6    15    46               0    -3     1               0     0     1
```

▲

Examples to Practice

1. Determine the reactions and internal forces of the truss in the Example in 6.11, also for the load case, which consists of a pair of vertical nodal loads $F = 10$ kN in joints d and e.

6.1.4 The LU Decomposition

The LU decomposition method is based on the principle that every regular matrix $\mathbf{A}$ can be decomposed into the product of two triangular matrices $\mathbf{L}$ and $\mathbf{U}$ such that:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \, . \tag{6.29}$$

The system of linear equations is then solved in two calculation steps, solving the triangular system of linear equations first:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \, , \tag{6.30}$$

and then similarly

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \, . \tag{6.31}$$

```
Input  : $n$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Output: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}^T$
for $i \leftarrow 1, 2, \ldots, n-1, n$ do
    for $j \leftarrow 1, 2, \ldots, n-1, n$ do
        $u_{i,j} \leftarrow 0$
    end
end
for $i \leftarrow 1, 2, \ldots, n-1, n$ do
    for $j \leftarrow 1, 2, \ldots, n-1, n$ do
        if $i = j$ then
            $l_{i,j} \leftarrow 1$
        else
            $l_{i,j} \leftarrow 0$
        end
    end
end
for $j \leftarrow 1, 2, \ldots, n-1, n$ do
    for $i \leftarrow 1, 2, \ldots, j-1, j$ do
        $u_{i,j} \leftarrow a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} \cdot u_{k,j}$
    end
    for $i \leftarrow j+1, j+2, \ldots, n-1, n$ do
        $l_{i,j} \leftarrow \dfrac{a_{i,j} - \sum\limits_{k=1}^{j-1} l_{i,k} \cdot u_{k,j}}{u_{j,j}}$
    end
end
for $i \leftarrow 1, 2, \ldots, n-1, n$ do
    $y_i \leftarrow b_i - \sum_{j=1}^{i-1} l_{i,j} \cdot y_j$
end
for $i \leftarrow n, n-1, \ldots, 2, 1$ do
    $x_i \leftarrow \dfrac{y_i - \sum\limits_{j=i+1}^{n} u_{i,j} \cdot x_j}{u_{i,i}}$
end
```

Algorithm 15: Algorithm for the LU decomposition method.

In the lower triangular matrix $\mathbf{L}$, the value 1 is selected on the diagonal. The matrix **U** is upper triangular. The advantage of the LU decomposition method is in its use with problems containing multiple systems of linear equations over the same matrix **A**; this matrix is decomposed once only, and then only triangular systems are iteratively solved.

The calculation procedure for solving a system of linear equations using LU decomposition is expressed in Algorithm 15 and can be implemented via a MATLAB m-function, e.g., in the following way:

```
function x=lu_decomposition(A,b)
if det(A)==0
  error('The system of equations is singular! Det(A) is equal to 0!')
  return
end
n=length(A);
if n==1
  x(1)=b(1)/A(1,1);
  return
end
L=eye(n,n);
U=zeros(n,n);
for j=1:n
  for i=1:j
    s=0;
      if i>1
      for k=1:i-1
        s=s+L(i,k)*U(k,j);
      end
    end
    U(i,j)=A(i,j)-s;
  end
  if U(j,j)==0
    error('There is a zero on the diagonal!')
    return
  end
  for i=j+1:n
    s=0;
    if j>1
      for k=1:j-1
        s=s+L(i,k)*U(k,j);
      end
    end
    L(i,j)=(A(i,j)-s)/U(j,j);
  end
```

```
end
for i=1:n
  s=0;
  if i>1
    for j=1:i-1
      s=s+L(i,j)*y(j);
    end
  end
  y(i)=(b(i)-s);
end
for i=n:-1:1
  s=0;
  if i<n
    for j=i+1:n
      s=s+U(i,j)*x(j);
    end
  end
  if U(i,i)==0
    error('There is a zero on the diagonal!')
    return
  end
  x(i)=(y(i)-s)/U(i,i);
end
```

Example 6.16. Use the LU decomposition method to calculate the roots of the system of linear equations given in Exercise 6.7.

Solution. A full listing of interim and final results can represent the solution:

```
Original matrix A             Original vector b
----------------------------  ----------------------------
  2.000 -1.000  3.000 -1.000    7.000
  1.000 -1.000  4.000 -2.000    5.000
  3.000  2.000  1.000  4.000   31.000
  4.000 -3.000  3.000 -3.000   -5.000


Modified matrix L             Modified matrix U
----------------------------  ----------------------------
  1.000  0.000  0.000  0.000    2.000 -1.000  3.000 -1.000
  0.500  1.000  0.000  0.000    0.000 -0.500  2.500 -1.500
  1.500 -7.000  1.000  0.000    0.000  0.000 14.000 -5.000
  2.000  2.000 -0.571  1.000    0.000  0.000  0.000 -0.857
```

```
Matrix L*U
----------------------------
  2.000 -1.000  3.000 -1.000
  1.000 -1.000  4.000 -2.000
  3.000  2.000  1.000  4.000
  4.000 -3.000  3.000 -3.000

The roots of the system are:
-----------------------------------
x[ 1] = 1.000e+000
x[ 2] = 2.000e+000
x[ 3] = 4.000e+000
x[ 4] = 5.000e+000

The residual vector is:
-----------------------------------
r[ 1] =          0
r[ 2] =          0
r[ 3] =  -7.105e-15
r[ 4] =  -1.776e-15

The norm of the residual vector is:
-----------------------------------
norm  =   7.324e-15
```

▲

6.1.5 The Cholesky Method (Decomposition)

The Cholesky method (also called the Cholesky decomposition) is a modification of LU decomposition for solving systems of linear equations with a symmetric, regular, positive definite square system matrix $\mathbf{A}$, which is modified to the product of the lower and upper triangular matrices, where one triangular matrix is the transpose of the matrix of the other:

$$\mathbf{A} = \mathbf{U} \cdot \mathbf{U}^T \; . \tag{6.32}$$

The lower triangular matrix $\mathbf{U}$ from this decomposition is called the Cholesky triangle of the matrix $\mathbf{A}$.

Comment 6.17. A positive definite matrix is a symmetric square matrix whose eigenvalues are greater than zero. The following must be true:

$$\mathbf{x} \cdot \mathbf{A} \cdot \mathbf{x}^T > 0 \; . \tag{6.33}$$

The calculation procedure for solving a system of linear equations using the Cholesky method contains roughly half the number of calculation operations compared to the LU decomposition method and is schematically described by Algorithm 16.

```
Vstup  : $n$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Výstup: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}^T$
for $i \leftarrow 1, 2, \ldots, n-1, n$ do
    for $j \leftarrow 1, 2, \ldots, n-1, n$ do
        if $i = j$ then
            $u_{i,j} \leftarrow 1$
        else
            $u_{i,j} \leftarrow 0$
        end
    end
end
for $j \leftarrow 1, 2, \ldots, n-1, n$ do
    for $i \leftarrow j, j+1, \ldots, n-1, n$ do
        $u_{j,j} \leftarrow \sqrt{a_{j,j} - \sum_{k=1}^{j-1} u_{j,k}^2}$
        $u_{i,j} \leftarrow \dfrac{a_{i,j} - \sum_{k=1}^{j-1} u_{i,k} \cdot u_{j,k}}{u_{j,j}}$
    end
end
for $i \leftarrow 1, 2, \ldots, n-1, n$ do
    $y_i \leftarrow \dfrac{b_i - \sum_{j=1}^{i-1} u_{i,j} \cdot y_j}{u_{i,i}}$
end
for $i \leftarrow n, n-1, \ldots, 2, 1$ do
    $x_i \leftarrow \dfrac{y_i - \sum_{j=1}^{i-1} u_{j,i} \cdot x_j}{u_{i,i}}$
end
```

Algorithm 16: The algorithm for the Cholesky method.

Example 6.18. Use Cholesky decomposition on the matrix:

$$[A] = \begin{bmatrix} 1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 0 & 0 & 0 \\ -1 & 0 & 3 & 1 & 1 \\ -1 & 0 & 1 & 4 & 2 \\ -1 & 0 & 1 & 2 & 5 \end{bmatrix} . \tag{6.34}$$

Solution. The solution is:

$$[U] = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} , \tag{6.35}$$

which can be verified by:

```
A =                                         U =
     1    -1    -1    -1    -1                   1     0     0     0     0
    -1     2     0     0     0                  -1     1     0     0     0
    -1     0     3     1     1                  -1    -1     1     0     0
    -1     0     1     4     2                  -1    -1    -1     1     0
    -1     0     1     2     5                  -1    -1    -1    -1     1
U*U' =
     1    -1    -1    -1    -1
    -1     2     0     0     0
    -1     0     3     1     1
    -1     0     1     4     2
    -1     0     1     2     5
```

▲

The Cholesky method can be executed in an m-function as follows:

```
function x=cholesky_met(A,b)
n=length(A);
U=eye(n,n);
for j=1:n
  for i=j:n
    s=0;
    if i>1
      for k=1:j-1
        s=s+U(j,k)^2;
      end
    end
    U(j,j)=sqrt(A(j,j)-s);
    s=0;
    if i>1
      for k=1:j-1
        s=s+U(i,k)*U(j,k);
      end
    end
    U(i,j)=(A(i,j)-s)/U(j,j);
  end
end
for i=1:n
  s=0;
  if i>1
    for j=1:i-1
      s=s+U(i,j)*y(j);
    end
  end
  y(i)=(b(i)-s)/U(i,i);
end
for i=n:-1:1
  s=0;
  if i<n
    for j=i+1:n
      s=s+U(j,i)*x(j);
    end
  end
  x(i)=(y(i)-s)/U(i,i);
end
```

Example 6.19. Calculate the roots of the system of equations from Example 6.9 using the Cholesky method.

6.2 Iterative Methods of Solving Systems of Linear Equations

Unlike direct methods of calculation, solving systems of linear equations with iterative methods consists in gradually approaching the exact result. There are many ways to create such a sequence of approximations, whose limit is the vector $\mathbf{x}$ of precisely determined roots of the system of linear equations.

A system of linear equations with a real-valued system matrix can be expressed, for example, by the relation (6.6). It is assumed that only one exact solution exists:

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} \; . \tag{6.36}$$

Equation (6.6) can be adjusted to a form suitable for iteration:

$$\mathbf{x} = \mathbf{H} \cdot \mathbf{x} + \mathbf{g} \; , \tag{6.37}$$

where $\mathbf{H}$ represents the *iteration matrix*. The sequence of iterations $\mathbf{x}^{(k)} = \{x_1^k, x_2^k$ až $x_n^k\}^T$ based on the relation (6.37) is constructed according to the recurrence formula:

$$\mathbf{x}^{(k+1)} = \mathbf{H} \cdot \mathbf{x}^{(k)} + \mathbf{g} \; , \tag{6.38}$$

for iteration steps $k = 0, 1, 2, \ldots$.

In addition to the iterative formula, it is necessary to define the choice of zero approximation $\mathbf{x}^{(0)} = \{x_1^0, x_2^0, \ldots, x_n^0\}^T$ and a method to terminate the iteration cycle by specifying:

(a) the exact number of iteration cycles to be performed, for example using a `for` loop,

(b) a terminating condition with a specified accuracy bound $\varepsilon > 0$, for example using the vector norm:

$$\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| < \varepsilon \; . \tag{6.39}$$

Both the iteration matrix $\mathbf{H}$ and the vector $\mathbf{g}$ may change at each iteration step, for example at step k. This refers to a *non-stationary iterative process* and differs from the *stationary* process where the matrix $\mathbf{H}$ and vector $\mathbf{g}$ are independent of the iteration cycle k.

An iterative calculation of a system of linear equations converges to a solution if the system matrix $\mathbf{A}$ is *diagonally dominant* (the values of the elements on the diagonal prevail); it is expressed as:

$$|a_{i,i}| > \sum_{j=1}^{i-1} |a_{i,j}| + \sum_{j=i+1}^{n} |a_{i,j}| \; , \quad i = 1, 2, \ldots, n \; . \tag{6.40}$$

Example 6.20. Construct a function to determine whether the matrix $\mathbf{A}$ is diagonally dominant. Try to use (6.40) on the matrix:

$$[A] = \begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \; . \tag{6.41}$$

6.2.1 The Jacobi Iteration

In a general system of linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, all equations are expressed as:

$$a_{i,1} \cdot x_1 \;+\; a_{i,2} \cdot x_2 \;+\; \cdots \;+\; a_{i,n} \cdot x_n \;=\; b_i \;, \quad i = 1, 2, \ldots, n \;. \tag{6.42}$$

If $a_{i,i} \neq 0$, each equation can be adjusted to:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j}{a_{i,i}} \;, \quad i = 1, 2, \ldots, n \;, \tag{6.43}$$

which means that the i-th unknown root of the system can be determined from the i-th equation.

Jacobi's iterative recurrence formula takes the form:

$$x_i^{(k)} = \frac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j^{(k-1)}}{a_{i,i}} \;, \quad i = 1, 2, \ldots, n \;, \tag{6.44}$$

where k is the iteration cycle number ($k = 1, 2, \ldots, m$).

```
Input  : $m, n, \mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$,
         $\mathbf{x}^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\}^T$
Output: $\mathbf{x}^{(m)} = \{x_1^{(m)}, x_2^{(m)}, \ldots, x_n^{(m)}\}^T$
for $k \leftarrow 1, 2, \ldots, m-1, m$ do
    for $i \leftarrow 1, 2, \ldots, n-1, n$ do
        $x_i^{(k)} = \dfrac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j^{(k-1)}}{a_{i,i}}$
    end
end
```

Algorithm 17: The Jacobi iteration algorithm.

The Jacobi iteration method for m iteration cycles is expressed by Algorithm 17. If the terminating condition 6.39 is used to terminate the iterative calculation, the algorithm is modified slightly (see Algorithm 18).

```
**Input** : $n, \varepsilon, \mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}], \mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T,$
            $\mathbf{x}^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\}^T$
**Output:** $\mathbf{x}^{(m)} = \{x_1^{(m)}, x_2^{(m)}, \ldots, x_n^{(m)}\}^T$
$k \leftarrow 1$
**for** $i \leftarrow 1, 2, \ldots, n-1, n$ **do**
    $x_i^{(1)} = \dfrac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j^{(0)} - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j^{(0)}}{a_{i,i}}$
**end**
**while** $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\| \geqq \varepsilon$ **do**
    $k \leftarrow k + 1$
    **for** $i \leftarrow 1, 2, \ldots, n-1, n$ **do**
        $x_i^{(k)} = \dfrac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j^{(k-1)}}{a_{i,i}}$
    **end**
**end**
```

Algorithm 18: The Jacobi iteration algorithm with a termination condition.

The Jacobi iteration algorithm can be written in MATLAB in different ways. The first m-function contains the calculation procedure for solving a system of linear equations with Jacobi's method and a finite number of cycles, i.e., using a `for` loop. To write this algorithm, we use MATLAB's matrix operation capabilities:

```
function x=jacobi1(A,b,x,m)
n=length(A);
d=diag(A);
r=A-diag(d);
for k=1:m
  x=(b-r*x)./d;
end
```

The second m-function is written in a more generally. The calculation also checks whether the system matrix $\mathbf{A}$ is regular and diagonally dominant:

```
function x=jacobi2(A,b,x,m)
if det(A)==0
  error('The system of equations is singular! Det(A) is equal to 0!')
  return
end
n=length(A);
```

```
for i=1:n
  if A(i,i)==0
    error('There is a zero on the diagonal!')
    return
  end
  s=0;
  for j=1:n
    if ~(i==j)
      s=s+abs(A(i,j));
    end
  end
  if abs(A(i,i))<s
    disp('The matrix A is not diagonally dominant!')
    return
  end
end
for k=1:m
  y=x;
  for i=1:n
    s1=0;
    if i>1
      for j=1:i-1
        s1=s1+A(i,j)*y(j);
      end
    end
    s2=0;
    if i<n
      for j=i+1:n
        s2=s2+A(i,j)*y(j);
      end
    end
    x(i)=(b(i)-s1-s2)/A(i,i);
  end
end
```

The third m-function solves the system of linear equations with Jacobi iteration and a `while` loop, which is terminated by the termination condition (6.39):

```
function x=jacobi3(A,b,x,eps)
y=x;
it_num=1;
for i=1:n
  s1=0;
```

```
    if i>1
      for j=1:i-1
        s1=s1+A(i,j)*y(j);
      end
    end
    s2=0;
    if i<n
      for j=i+1:n
        s2=s2+A(i,j)*y(j);
      end
    end
    if A(i,i)==0
      error('There is a zero on the diagonal!')
      return
    end
    x(i)=(b(i)-s1-s2)/A(i,i);
  end
  while ~(norm(y-x)<eps) & it_num<1000
    y=x;
    it_num=it_num+1;
    for i=1:n
      s1=0;
      if i>1
        for j=1:i-1
          s1=s1+A(i,j)*y(j);
        end
      end
      s2=0;
      if i<n
        for j=i+1:n
          s2=s2+A(i,j)*y(j);
        end
      end
      if A(i,i)==0
        error('There is a zero on the diagonal!')
        return
      end
      x(i)=(b(i)-s1-s2)/A(i,i);
    end
  end
```

All three writing methods assume the zero approximation of the solved roots of the system $\mathbf{x}^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\}^T$.

Example 6.21. Find the solution for the roots of a system of linear equations using Jacobi iterations:

$$\begin{bmatrix} 4 & -1 & -1 & 0 \\ -1 & 4 & 0 & -1 \\ -1 & 0 & 4 & -1 \\ 0 & -1 & -1 & 4 \end{bmatrix} \cdot \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{Bmatrix} . \tag{6.45}$$

First perform the calculations for the finite number of cycles $m = 5$, 10 and 20 and observe the solution's increasing accuracy. Finally, perform the calculation using the m-function with the termination condition (6.39) and an accuracy parameter of $\varepsilon = 1 \cdot 10^{-6}$.

Solution. The correct solution of the roots of the system is equal to the vector:

$$\{x\} = \begin{Bmatrix} 0.5 & 0.75 & 0.25 & 0.5 \end{Bmatrix}^T . \tag{6.46}$$

The course of the iterative calculation for the input $\mathbf{x}^{(0)} = \{1, 1, 1, 1\}^T$ is shown below:

```
Jacobi iteration
------------------------------------------------------
   it#      x[1]         x[2]         x[3]         x[4]
------------------------------------------------------
     0    1.000000     1.000000     1.000000     1.000000
     1    0.750000     1.000000     0.500000     0.750000
     2    0.625000     0.875000     0.375000     0.625000
     3    0.562500     0.812500     0.312500     0.562500
     4    0.531250     0.781250     0.281250     0.531250
     5    0.515625     0.765625     0.265625     0.515625
     6    0.507813     0.757813     0.257813     0.507813
     7    0.503906     0.753906     0.253906     0.503906
     8    0.501953     0.751953     0.251953     0.501953
     9    0.500977     0.750977     0.250977     0.500977
    10    0.500488     0.750488     0.250488     0.500488
    11    0.500244     0.750244     0.250244     0.500244
    12    0.500122     0.750122     0.250122     0.500122
    13    0.500061     0.750061     0.250061     0.500061
    14    0.500031     0.750031     0.250031     0.500031
    15    0.500015     0.750015     0.250015     0.500015
    16    0.500008     0.750008     0.250008     0.500008
    17    0.500004     0.750004     0.250004     0.500004
    18    0.500002     0.750002     0.250002     0.500002
    19    0.500001     0.750001     0.250001     0.500001
    20    0.500000     0.750000     0.250000     0.500000
```

To obtain results with an accuracy parameter of $\varepsilon = 1 \cdot 10^{-6}$, the Jacobi calculation executes 20 iteration cycles.

▲

6.2.2 Gauss-Seidel Iteration Method

The general system of linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ is again expressed by Eq. (6.42), whereas if the condition $a_{i,i} \neq 0$ is met, each equation can be defined according to:

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j}{a_{i,i}} \;, \quad i = 1, 2, \ldots, n \;. \tag{6.47}$$

The Gauss-Seidel iterative recurrence formula is derived from the equation:

$$x_i^{(k)} = \frac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j^{(k-1)}}{a_{i,i}} \;, \quad i = 1, 2, \ldots, n \;, \tag{6.48}$$

which differs from Jacobi's iteration formula in that the calculated roots of the system $x_j^{(k)}$ are used for the next calculation immediately, not in the next iteration cycle. The entire calculation then converges more quickly to the exact solution.

The calculation using the Gauss-Seidel iteration method for a finite number of m iteration cycles is expressed by Algorithm 19. Of course, the procedure may also apply a terminating condition (6.39).

```
Input  : $m, n, \mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$,
         $\mathbf{x}^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\}^T$
Output: $\mathbf{x}^{(m)} = \{x_1^{(m)}, x_2^{(m)}, \ldots, x_n^{(m)}\}^T$
for $k \leftarrow 1, 2, \ldots, m-1, m$ do
    for $i \leftarrow 1, 2, \ldots, n-1, n$ do
        $x_i^{(k)} = \frac{b_i - \sum_{j=1}^{i-1} a_{i,j} \cdot x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j} \cdot x_j^{(k-1)}}{a_{i,i}}$
    end
end
```

Algorithm 19: Gauss-Seidel iteration algorithm.

Example 6.22. Solve the system of linear equations given in Exercise 6.21 using the Gauss-Seidel iteration method. For the calculation, use the accuracy parameter $\varepsilon = 1 \cdot 10^{-6}$ and the zero approximation of the roots of the solved system $\mathbf{x}^{(0)} = \{1, 1, 1, 1\}^T$.

Solution. The course of the iterative calculation is shown below:

```
Gauss-Seidel iteration
--------------------------------------------------
   #it      x[1]       x[2]       x[3]       x[4]
--------------------------------------------------
     0  1.000000   1.000000   1.000000   1.000000
     1  0.750000   0.937500   0.437500   0.593750
     2  0.593750   0.796875   0.296875   0.523438
     3  0.523438   0.761719   0.261719   0.505859
     4  0.505859   0.752930   0.252930   0.501465
     5  0.501465   0.750732   0.250732   0.500366
     6  0.500366   0.750183   0.250183   0.500092
     7  0.500092   0.750046   0.250046   0.500023
     8  0.500023   0.750011   0.250011   0.500006
     9  0.500006   0.750003   0.250003   0.500001
    10  0.500001   0.750001   0.250001   0.500000
    11  0.500000   0.750000   0.250000   0.500000
    12  0.500000   0.750000   0.250000   0.500000
```

To obtain results with an accuracy of $\varepsilon = 1 \cdot 10^{-6}$, the Gauss-Seidel calculation executes 12 iteration cycles.

▲

6.2.3 Sparse and Band Matrix

When solving systems of equations in numerical mathematics, sparse and band system matrices are frequently encountered.

A sparse matrix is a matrix that contains a significant number of zero elements. A band matrix, however, has non-zero elements only in close proximity to the diagonal. The band width of the matrix then represents the maximum number of columns near the diagonal of the matrix with non-zero elements. It is also possible to distinguish half-band widths p, q, which refer to the maximum number of columns with non-zero elements to the left and right of the diagonal.

Example 6.23. Find the solution for the roots of a system of linear equations with the system band matrix:

$$\begin{bmatrix} 3 & -1 & & & \\ -1 & 3 & -1 & & \\ & & \ddots & & \\ & & -1 & 3 & -1 \\ & & & -1 & 3 \end{bmatrix} \cdot \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{Bmatrix} = \begin{Bmatrix} 2 \\ 1 \\ \vdots \\ 1 \\ 2 \end{Bmatrix} \tag{6.49}$$

for $n = 100$ using the Gauss-Seidel method adapted to solving band matrices. Consider the accuracy parameter $\varepsilon = 1 \cdot 10^{-6}$ and zero approximation of the roots of the solved system $\mathbf{x}^{(0)} = \{0, 0, \ldots, 0, 0\}^T$.

Solution. The matrix of the system $\mathbf{A}$ and the vector of the right-hand sides is generated in MATLAB as follows:

```
clc;
clear;
n=1000;
E=ones(n,1);
A=spdiags([-E 3*E -E],-1:1,n,n);
b=ones(n,1);
b(1)=2;
b(n)=2;
x=zeros(n,1);
```

The calculation can then be performed with the m-function, which executes the calculation procedure of the Gauss-Seidel iteration method adapted for solving band matrices. In this case, the half-band widths p, q are equal to 1.

```
function x=gauss_seidel_pas(A,b,x,p,q,eps)
n=length(A);
it_max=1000;
y=x;
it_num=1;
for i=1:n
  s1=0;
  if i>1
    from=i-p;
    if from<1
      from=1;
    end
    for j=from:i-1
      s1=s1+A(i,j)*x(j);
    end
  end
  s2=0;
  if i<n
    to=i+q;
    if to>n
      to=n;
    end
    for j=i+1:to
      s2=s2+A(i,j)*y(j);
    end
  end
  if A(i,i)==0
    error('There is a zero on the diagonal!')
    return
  end
  x(i)=(b(i)-s1-s2)/A(i,i);
end
while ~(norm(y-x)<eps) & it_num<it_max
  y=x;
  it_num=it_num+1;
  for i=1:n
    s1=0;
    if i>1
      from=i-p;
      if from<1
        from=1;
      end
      for j=from:i-1
        s1=s1+A(i,j)*x(j);
      end
    end
    s2=0;
    if i<n
      to=i+q;
      if to>n
        to=n;
      end
      for j=i+1:to
        s2=s2+A(i,j)*y(j);
      end
    end
    if A(i,i)==0
```

```
      error('There is a zero on the diagonal!')
      return
    end
    x(i)=(b(i)-s1-s2)/A(i,i);
  end
end
if it_num==it_max
  disp(['The calculation was not terminated ',...
      'by the termination condition!'])
end
```

The calculation utility also provides a terminating condition where the maximum number of iteration cycles does not exceed 1000. If this condition is fulfilled and the calculation is terminated, it means that the iterative calculation is not converging quickly enough or is diverging. ▲

Example 6.24. Calculate the system of linear equations with the system band matrix given in Exercise 6.23 using the Gauss-Seidel iteration method for $n = 1000$. Consider the accuracy parameter $\varepsilon = 1 \cdot 10^{-6}$ and the zero approximation of the roots of the solved system $\mathbf{x}^{(0)} = \{0, 0, \ldots, 0, 0\}^T$.

Example 6.25. Find the solution for the roots of a system of linear equations with the system band matrix:

$$\begin{bmatrix} 2 & 1 & & & \\ 1 & 2 & 1 & & \\ & & \ddots & & \\ & & 1 & 2 & 1 \\ & & & 1 & 2 \end{bmatrix} \cdot \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{Bmatrix} = \begin{Bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 1 \end{Bmatrix} \tag{6.50}$$

for $n = 100$ using the Gauss-Seidel method, adapted for solving band matrices. Consider the accuracy parameter $\varepsilon = 1 \cdot 10^{-6}$ and the zero approximation of the roots of the solved system $\mathbf{x}^{(0)} = \{0, 0, \ldots, 0, 0\}^T$.

Solution. The correct solution of the roots of the system is equal to the vector:

$$\{x\} = \begin{Bmatrix} 1 & -1 & 1 & -1 & \cdots & 1 & -1 \end{Bmatrix}^T . \tag{6.51}$$

▲

Example 6.26. Calculate the system of linear equations with the system band matrix given in Exercise 6.25 using the Gauss-Seidel iteration method for $n = 1000$. Consider the accuracy parameter $\varepsilon = 1 \cdot 10^{-6}$ and the zero approximation of the roots of the solved system $\mathbf{x}^{(0)} = \{0, 0, \ldots, 0, 0\}^T$.

6.2.4 The Conjugate Gradient Method

The conjugate gradient method is a calculation procedure well suited to solving systems of linear equations for which the system matrix $\mathbf{A}$ is square, symmetric and positive definite.

The combined gradients method consists in selecting an appropriate sequence of vectors $\mathbf{d}^{(k)}$ that determine the direction from $\mathbf{x}^{(k)}$ to the next approximation $\mathbf{x}^{(k+1)}$. The vectors $\mathbf{d}^{(k)}$ are gradually constructed from the sequence of residual vectors $\mathbf{r}^{(k)}$ so that $(\mathbf{d}^{(k)}, \mathbf{A}\cdot\mathbf{d}^{(k-1)}) = 0$ is true for the scalar product $(\mathbf{d}^{(k)}, \mathbf{A}\cdot\mathbf{d}^{(k-1)})$. The vector sequence $\mathbf{d}^{(k)}$ is then $\mathbf{A}$-orthogonal.

The calculation procedure is executed by Algorithm 20, in which the initial approximation $\mathbf{x}^{(0)} = 0$ and the residual vector $\mathbf{r}^{(0)} = \mathbf{b}$ are first created. $\mathbf{r}^{(0)}$ is selected as the first direction of $\mathbf{d}^{(0)}$. The value $\alpha^{(0)}$ is determined for which the function (quadratic form) $F(\mathbf{x}^{(0)} + \alpha \cdot \mathbf{r}^{(0)}) = F(\mathbf{x}) = \frac{1}{2}\cdot(\mathbf{A}\cdot\mathbf{x}, \mathbf{x}) - (\mathbf{b}, \mathbf{x})$ reaches its minimum, so that:

$$\alpha^{(0)} = \frac{(\mathbf{d}^{(0)}, \mathbf{r}^{(0)})}{(\mathbf{d}^{(0)}, \mathbf{A}\cdot\mathbf{d}^{(0)})} = \frac{(\mathbf{r}^{(0)})^T \cdot \mathbf{r}^{(0)}}{(\mathbf{d}^{(0)})^T \cdot \mathbf{A}\cdot\mathbf{d}^{(0)}} \; . \tag{6.52}$$

```
Input  : $m, n, \varepsilon$, $\mathbf{A} = [a_{i,j}] = [a_{1,1}, \ldots, a_{n,n}]$, $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}^T$
Output: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}^T$
$\mathbf{x}^{(0)} \leftarrow 0$
$\mathbf{d}^{(0)} \leftarrow \mathbf{r}^{(0)} \leftarrow \mathbf{b}$
for $k \leftarrow 1, 2, \ldots, m-1, m$ do
    if $\|\mathbf{r}^{(k-1)}\| < \varepsilon$ then
        stop
    else
        $\alpha^{(k-1)} \leftarrow \frac{(\mathbf{r}^{(k-1)})^T \cdot \mathbf{r}^{(k-1)}}{(\mathbf{d}^{(k-1)})^T \cdot \mathbf{A} \cdot \mathbf{d}^{(k-1)}}$
        $\mathbf{x}^{(k)} \leftarrow \mathbf{x}^{(k-1)} + \alpha^{(k-1)} \cdot \mathbf{d}^{(k-1)}$
        $\mathbf{r}^{(k)} \leftarrow \mathbf{r}^{(k-1)} - \alpha^{(k-1)} \cdot \mathbf{A} \cdot \mathbf{d}^{(k-1)}$
        $\beta^{(k-1)} \leftarrow \frac{(\mathbf{r}^{(k)})^T \cdot \mathbf{r}^{(k)}}{(\mathbf{r}^{(k-1)})^T \cdot \mathbf{r}^{(k-1)}}$
        $\mathbf{d}^{(k)} \leftarrow \mathbf{r}^{(k)} + \beta^{(k-1)} \cdot \mathbf{d}^{(k-1)}$
    end
end
```

Algorithm 20: The conjugate gradient algorithm.

Now, the procedure corrects the approximation of the solution of the system $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \alpha^{(0)} \cdot \mathbf{d}^{(0)}$, calculates a new residual vector $\mathbf{r}^{(1)} = \mathbf{r}^{(0)} - \alpha^{(0)} \cdot \mathbf{A} \cdot \mathbf{d}^{(0)}$, and determines a new direction $\mathbf{d}^{(1)}$ in the form $\mathbf{d}^{(1)} = \mathbf{r}^{(1)} + \beta^{(0)} \cdot \mathbf{d}^{(0)}$ so that

$(\mathbf{d}^{(1)}, \mathbf{A} \cdot \mathbf{d}^{(0)}) = 0$. The value of $\beta^{(0)}$ is determined from the equation:

$$\beta^{(0)} = -\frac{(\mathbf{d}^{(0)}, \mathbf{A} \cdot \mathbf{r}^{(1)})}{(\mathbf{d}^{(0)}, \mathbf{A} \cdot \mathbf{d}^{(0)})} = \frac{(\mathbf{r}^{(1)})^T \cdot \mathbf{r}^{(1)}}{(\mathbf{r}^{(0)})^T \cdot \mathbf{r}^{(0)}} \,. \tag{6.53}$$

Next, the procedure determines the value of $\alpha^{(1)}$ so that the function $F(\mathbf{x}^{(1)} + + \alpha \cdot \mathbf{r}^{(1)})$ reaches its minimum. The entire calculation is repeated until the termination condition is satisfied.

The procedure for solving a system of equations using the conjugate gradient method is executed in MATLAB as follows:

```
function x=con_grad(A,b,eps)
n=length(A);
it_max=1000;
x=zeros(n,1);
d=b;
r=b;
for k=1:it_max
  if norm(r)<eps
    return
  end
  alpha=r'*r/(d'*A*r);
  x=x+alpha*d;
  rs=r;
  r=r-alpha*A*d;
  beta=r'*r/(rs'*rs);
  d=r+beta*d;
  if k==it_max
    disp(['The calculation was not terminated ',...
        'by the termination condition!'])
  end
end
```

The values of $\alpha^{(k-1)}$ and $\beta^{(k-1)}$ can also be calculated using an m-function with MATLAB's `dot` command for the scalar product of two vectors:

```
alfa=dot(d,r)/dot(d,A*d);
```

and

```
beta=-dot(d,A*r)/dot(d,A*d);
```

Example 6.27. Calculate the system of linear equations in the system band matrix given in Exercise 6.23 using the conjugate gradient method for $n = 10000$ and accuracy parameters $\varepsilon = 1 \cdot 10^{-6}$ and $\varepsilon = 1 \cdot 10^{-12}$.

Example 6.28. Calculate the system of linear equations in the system band matrix given in Exercise 6.25 using the conjugate gradient method for $n = 1000$ and accuracy parameters $\varepsilon = 1 \cdot 10^{-6}$ and $\varepsilon = 1 \cdot 10^{-12}$.

Example 6.29. Find the solution for the roots of the system of linear equations with the system sparse matrix:

$$\begin{bmatrix}
3 & -1 & & & & & & & & 0.5 \\
-1 & 3 & -1 & & & & & & 0.5 & \\
 & -1 & 3 & -1 & & & & 0.5 & & \\
 & & & \ddots & & & \iddots & & & \\
 & & & -1 & 3 & -1 & & & & \\
 & & & & -1 & 3 & -1 & & & \\
 & & & \iddots & & & \ddots & & & \\
 & & 0.5 & & & & -1 & 3 & -1 & \\
 & 0.5 & & & & & & -1 & 3 & -1 \\
0.5 & & & & & & & & -1 & 3
\end{bmatrix} \cdot \begin{Bmatrix} x_1 \\ x_2 \\ \\ \vdots \\ \\ \\ \vdots \\ \\ x_{n-1} \\ x_n \end{Bmatrix} = \begin{Bmatrix} 2.5 \\ 1.5 \\ \vdots \\ 1.5 \\ 1 \\ 1 \\ 1.5 \\ \vdots \\ 1.5 \\ 2.5 \end{Bmatrix} \qquad (6.54)$$

for $n = 10000$ using the conjugate gradient method with the required accuracy parameter $1 \cdot 10^{-6}$.

Solution. The matrix of the system $\mathbf{A}$ and the vector of the right-hand sides can be generated with MATLAB commands:

```
clc;
clear;
n=100;
E=ones(n,1);
n2=n/2;
A=spdiags([-E 3*E -E],-1:1,n,n);
C=spdiags([E/2],0,n,n);
C=fliplr(C);
A=A+C;
A(n2+1,n2)=-1;
A(n2,n2+1)=-1;
b=zeros(n,1);
b(1)=2.5;
b(n)=2.5;
b(2:n-1)=1.5;
b(n2:n2+1)=1;
```

The correct solution of the roots of the system is equal to the vector:

$$\{x\} = \begin{Bmatrix} 1 & 1 & \cdots & 1 & 1 \end{Bmatrix}^T . \qquad (6.55)$$

▲

Chapter 7

Numerical Integration of a Definite Integral

Objectives

This chapter provides a more detailed introduction to:

- basic algorithms for the approximate calculation of definite integrals,
- advanced computational procedures used for integral calculus.

In numerical integration, we compute an approximate solution of a definite integral:

$$\int_a^b f(x)\,\mathrm{d}x \ , \tag{7.1}$$

where $f(x)$ is a continuous function in the interval $\langle a, b\rangle$ and where a, b represent the limits of the definite integral.

Comment 7.1. For numerical integration, the word *quadrature* is used—mainly associated with one-dimensional integrals. Two-dimensional integration is sometimes referred to as *cubature*.

The interval $\langle a, b\rangle$ is divided into n intervals of the same size:

$$\langle a, b\rangle = \langle a \equiv x_0, x_1\rangle \cup \langle x_1, x_2\rangle \cup \ldots \cup \langle x_{n-1}, x_n \equiv b\rangle \ , \tag{7.2}$$

while the width of all subintervals is the same:

$$h = \frac{b-a}{n} \ . \tag{7.3}$$

In each subinterval, the integrated function $f(x)$ is approximated by a simpler interpolation or approximation function (polynomial of degree m) $\varphi_m(x)$:

$$\int_a^b f(x)\,\mathrm{d}x = \int_a^b \varphi_m(x)\,\mathrm{d}x + R_m(f) \ , \tag{7.4}$$

where $R_m(f)$ is the error of the calculation formula used.

7.1 Rectangle Method

In the case of using the rectangle method of numerical integration, the integrated function $f(x)$ is approximated in each of the subintervals by a zero-degree polynomial, that is, by a constant function $\varphi_0(x) = \text{const}$. It then applies:

$$\int_a^b f(x)\,\mathrm{d}x = h \cdot \sum_{i=0}^{n-1} f(x_i) + R_0(f) \,, \tag{7.5}$$

where $R_0(f)$ is the calculation error that can be minimized by increasing the number of subintervals n.

Example 7.2. Approximate:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.6}$$

using the rectangle method. During the calculation, gradually increase the number of subintervals $n = 5, 10, 20, 100$ and monitor the achieved accuracy of the solution by comparing it with the analytical exact solution of the integral.

Solution. The solution to the problem can be based on the equation (7.5), which can be programmed in MATLAB in the following way, for example:

```
function y=rect(f,a,b,n)
if n<1
  error('The number of intervals n must be > 0 !')
end;
if ~(a<b)
  error('The limits of the integral must be a > b !')
end;
h=(b-a)/n;
y=0;
for x=a:h:b-h
    y=y+f(x);
end
y=y*h;
```

The `rect` function can be called with four input parameters: f is the integrated function, which can be defined in MATLAB with the help of the `inline` command, a, b are the integration limits ($a < b$) and n the number of subintervals into which the interval $\langle a; b\rangle$ was divided.

The calculation can then be called, for example, by the following sequence of commands:

```
clc;
format long;
g=inline('log(x)');
integral=rect(g,1,2,5)
```

The integration result using the rectangle method with five subintervals is then:

```
integral =
   0.315316817512604
```

If the result is compared with the exact value of the analytical solution:

$$\int_1^2 \ln(x)\,\mathrm{d}x = \ln(4) - 1 \approx 0.386294361119891\ . \tag{7.7}$$

for example with the help of commands

```
res=log(4)-1;
fprintf('Deviation from the exact solution = %8.6e\n\n',res-int)
```

the deviation of the achieved approximation from the exact analytical solution is:

```
Deviation from the exact solution = 7.097754e-002
```

For the increased number of subintervals $n = 10, 20, 100$, the resulting value of the approximation of the solved integral, including the deviation from the exact analytical solution, is:

```
integral =
   0.351220577717757

Deviation from the exact solution = 3.507378e-002

integral =
   0.368861530118207

Deviation from the exact solution = 1.743283e-002

integral =
   0.382824458574729

Deviation from the exact solution = 3.469903e-003
```

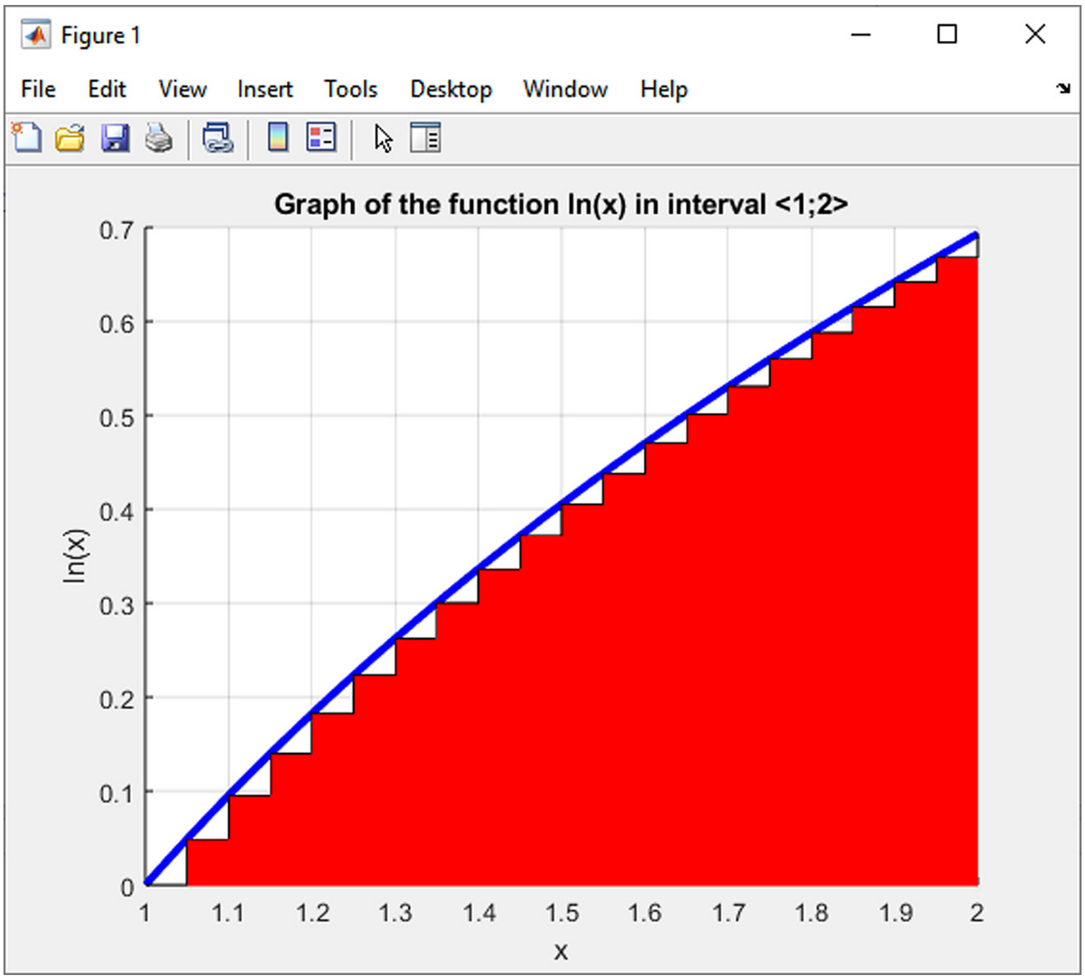


Fig. 7.1 The idea of calculating the integral by the rectangle method with $n = 20$ subintervals.

which testifies to the great inaccuracy and inefficiency of the rectangle method, the idea of which can be shown schematically for $n = 20$ in Fig. 7.1 (the approximation of the solved integral is equal to the red colored area). ▲

Comment 7.3. The analytical solution of the integral (7.6) can be obtained for verification in MATLAB with the command

```
int(log(sym('x')),1,2)
```

which is used to calculate the so-called symbolic integration. The result is then

```
log(4) - 1
```

7.2 Trapezoid Method

If the trapezoid method of numerical integration is used for numerical integration, the integrated function $f(x)$ is approximated by a polynomial of the first degree on individual subintervals, i.e., a linear function $\varphi_1(x) = k \cdot x + q$. Then:

$$\int_a^b f(x)\,\mathrm{d}x = h \cdot \left(\frac{f(a \equiv x_0) + f(x_1)}{2} + \frac{f(x_1) + f(x_2)}{2} + \cdots + \right.$$
$$\left. + \frac{f(x_{n-1}) + f(x_n \equiv b)}{2} \right) + R_1 =$$
$$= h \cdot \left(\frac{1}{2} \cdot f(a \equiv x_0) + f(x_1) + \cdots + f(x_{n-1}) + \frac{1}{2} \cdot f(x_n \equiv b) \right) + R_1 \ , \qquad (7.8)$$

where $R_1(f)$ is the computation error for which we have:

$$R_1(f) = -\frac{b-a}{12} \cdot h^2 \cdot f''(\xi) \ , \qquad (7.9)$$

for $\xi \in \langle a, b \rangle$. If the integrated function has a continuous second derivative, then arbitrarily small calculation errors can be achieved by a suitable choice of the number of subintervals n.

If the limits of the integral are x_0, x_1 and $y_0 = f(x_0), y_1 = f(x_1)$ are their respective functional values ($n = 1$), the so-called trapezoid rule can be defined as follows:

Definition 7.4. Trapezoid Rule:

$$\int_{x_0}^{x_1} f(x)\,\mathrm{d}x = \frac{h}{2} \cdot (y_0 + y_1) - \frac{h^3}{12} \cdot f''(c) \ , \qquad (7.10)$$

where $h = x_1 - x_0$ a c lies between x_0 a x_1.

Example 7.5. Use the trapezoid rule to approximate the integral given in Exercise 7.2:

$$\int_1^2 \ln(x)\,\mathrm{d}x \qquad (7.11)$$

for the $n = 1$ subinterval and determine the maximum deviation of this approximation from the exact solution.

Solution. By applying the trapezoid rule, we can obtain:

$$\int_1^2 \ln(x)\,\mathrm{d}x \approx \frac{h}{2} \cdot (y_0 + y_1) = \frac{1}{2} \cdot (\ln 1 + \ln 2) \approx 0.346573590279973 \ . \qquad (7.12)$$

The calculation error using the trapezoid rule is given for $1 < c < 2$:

$$R_1(f) = -\frac{h^3}{12} \cdot f''(c) \; . \tag{7.13}$$

It holds that:

$$f'(x) = \frac{1}{x} \tag{7.14}$$

and

$$f''(x) = -\frac{1}{x^2} \; , \tag{7.15}$$

therefore the calculation error is:

$$R_1(f) = \frac{1^3}{12 \cdot c^2} \; . \tag{7.16}$$

The greatest inaccuracy of the calculation will therefore be:

$$R_1(f) \leqq \frac{1^3}{12 \cdot 1^2} = \frac{1}{12} = 0.08\overline{3} \; . \tag{7.17}$$

In other words, the trapezoid rule says:

$$\int_1^2 \ln(x)\,\mathrm{d}x = 0.346573590279973 \pm 0.08\overline{3} \; , \tag{7.18}$$

which can be compared to the exact solution of the problem:

$$\int_1^2 \ln(x)\,\mathrm{d}x = \ln(4) - 1 \approx 0.386294361119891 \; . \tag{7.19}$$

▲

Comment 7.6. The analytical solution of derivatives (7.15) and (7.16) can be obtained in MATLAB via the commands

```
diff(log(sym('x')),1)
diff(log(sym('x')),2)
```

which are used to calculate the so-called symbolic derivative (the order of the required derivative is contained in the 2nd input parameter). The resulting statement of both analytical relations then looks looks as follows:

```
1/x
-1/x^2
```

Example 7.7. Determine the approximations of the integral given in Exercise 7.2:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.20}$$

by the trapezoid method for $n = 5, 10, 20, 100$ subintervals and compare the achieved results with the exact analytical solution.

Solution. The calculation of the integral using the trapezoid method can be based on the equation (7.8), which can be applied in MATLAB as follows:

```
function y=trapezoid(f,a,b,n)
if n<1
  error('The number of intervals n must be > 0 !')
end;
if ~(a<b)
  error('The limits of the integral must be a > b !')
end;
h=(b-a)/n;
y=f(a)/2+f(b)/2;
for x=a+h:h:b-h
    y=y+f(x);
end
y=y*h;
```

The `trapezoid` function can be called with the same four input parameters as in the case of the rectangle method: f is the integrated function that can be defined in MATLAB with the help of the `inline` command, a, b are the integration limits ($a < b$) and n is the number of subintervals into which the interval $\langle a; b\rangle$ was divided.

The calculation and comparison of the accuracy with the exact analytical solution can then be called, similar to the case of Exercise 7.2, for example by the following sequence of commands:

```
clc;
format long;
g=inline('log(x)');
int=trapezoid(g,1,2,5)
res=log(4)-1;
fprintf('Deviation from the exact solution = %8.6e\n\n',res-int)
```

The integration result using the rectangle method with five, ten, twenty and one hundred subintervals is then:

```
integral =
   0.384631535568599

Deviation from the exact solution = 1.662826e-003

integral =
   0.385877936745754

Deviation from the exact solution = 4.164244e-004

integral =
   0.386190209632206

Deviation from the exact solution = 1.041515e-004

integral =
   0.386290194477529

Deviation from the exact solution = 4.166642e-006
```

which indicates greater accuracy and efficiency of the solution than when using the rectangle method.

The principle of calculating the solved integral by the trapezoidal rule can be shown schematically for $n = 5$ subintervals in Fig. 7.2 (the approximation of the solved integral is equal to the red colored area). ▲

7.3 Simpson's Method

If polynomials of the second degree, that is, the quadratic function $\varphi_2(x) = a \cdot x^2 + b \cdot x + c$, are chosen for the approximation of the function $f(x)$ on individual subintervals, numerical integration is performed using Simpson's method of numerical integration. However, the number of subintervals n must be even. Then we have:

$$\int_a^b f(x)\,\mathrm{d}x = \frac{h}{3} \cdot \Big(f(a \equiv x_0) + 4 \cdot f(x_1) + 2 \cdot f(x_2) + 4 \cdot f(x_3) + \cdots + \\ + 4 \cdot f(x_{n-3}) + 2 \cdot f(x_{n-2}) + 4 \cdot f(x_{n-1}) + f(x_n \equiv b)\Big) + R_2(f) \,, \quad (7.21)$$

where $R_2(f)$ is the computation error that can be determined:

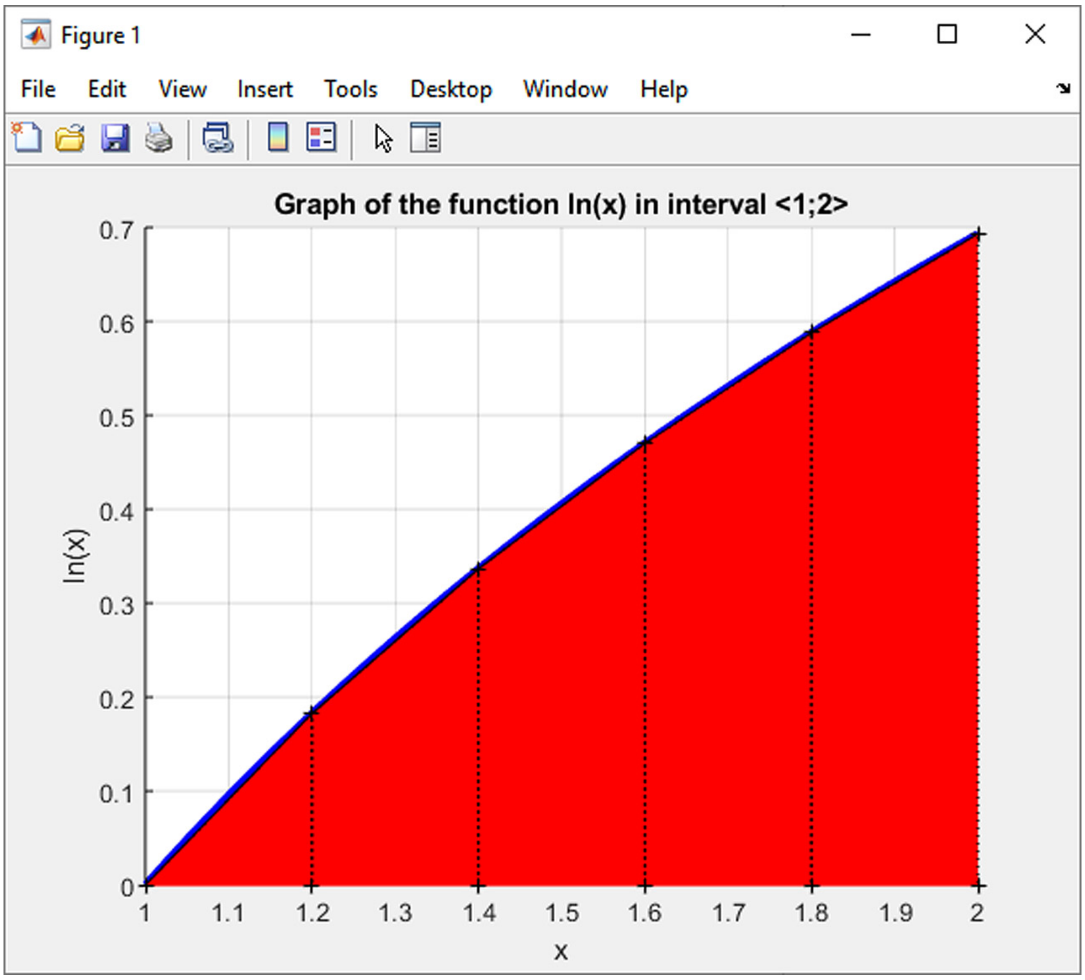


Fig. 7.2 The principle of calculating the integral using the trapezoid method with the number of subintervals $n = 5$.

$$R_2(f) = -\frac{b-a}{180} \cdot h^4 \cdot f''''(\xi) \ , \tag{7.22}$$

for $\xi \in \langle a, b \rangle$. If the integrated function has a continuous fourth derivative, then arbitrarily small calculation errors can be achieved by a suitable choice of the number of subintervals n.

If the limits of the integral are x_0, x_2 and $y_0 = f(x_0)$, $y_2 = f(x_2)$ are their respective functional values ($n = 2$), and if there exists $f'''(x)$ which is continuous, the so-called Simpson's rule can be defined as follows:

Definition 7.8. Simpson's Rule:

$$\int_{x_0}^{x_2} f(x)\,\mathrm{d}x = \frac{h}{3} \cdot (y_0 + 4 \cdot y_1 + y_2) - \frac{h^5}{90} \cdot f''''(c)\ , \tag{7.23}$$

where $h = x_2 - x_1 = x_1 - x_0$, $y_1 = f(x_1)$, and c lies between x_0 and x_2.

Example 7.9. Use Simpson's rule to approximate the integral given in Exercises 7.2 and 7.5:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.24}$$

for $n = 2$ subintervals and determine the maximum deviation of this approximation from the exact solution.

Solution. According to Simpson's rule, the result for $h = 2 - 1.5 = 1.5 - 1 = 0.5$ is:

$$\int_1^2 \ln(x)\,\mathrm{d}x \approx \frac{h}{3} \cdot (y_0 + 4 \cdot y_1 + y_2) = \frac{0.5}{3} \cdot (\ln 1 + 4 \cdot \ln 1.5 + \ln 2) \approx 0.385834602165434\ . \tag{7.25}$$

The calculation error determined by Simpson's rule for $1 < c < 2$:

$$R_2(f) = -\frac{h^5}{90} \cdot f''''(c)\ . \tag{7.26}$$

It holds that:

$$f'''(x) = \frac{2}{x^3} \tag{7.27}$$

and

$$f''''(x) = -\frac{6}{x^4}\ , \tag{7.28}$$

so the greatest calculation inaccuracy will be:

$$R_2(f) \leqq \frac{0.5^5 \cdot 6}{90 \cdot 1^4} = \frac{0.5^5 \cdot 6}{60} = \frac{1}{480} = 0.00208\overline{3}\ . \tag{7.29}$$

The result of Simpson's rule calculation is therefore:

$$\int_1^2 \ln(x)\,\mathrm{d}x = 0.385834602165434 \pm 0.00208\overline{3}\ , \tag{7.30}$$

which is a significantly more accurate value of the resulting approximation of Integral (7.24) than when calculating with the trapezoid rule. ▲

Comment 7.10. The analytical solution of derivations (7.27) and (7.28) can once again be performed in MATLAB via the commands for symbolic differentiation:

```
diff(log(sym('x')),3)
diff(log(sym('x')),4)
```

The resulting statement of both analytical equations then looks as follows:

```
2/x^3
-6/x^4
```

Example 7.11. Use Simpson's method to calculate the approximation of the integral given in Exercise 7.9:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.31}$$

for $n = 4, 8, 16, 32$ subintervals and compare the obtained results with the exact analytical solution.

Solution. The solution of the integral using the Simpson method is based on the relation (7.21), which can be programmed in MATLAB, for example, using the following sequence of commands:

```
function y=simpson(f,a,b,n)
if n<1
  error('The number of intervals n must be > 0 !')
end;
if ~(mod(n,2)==0)
  error('The number of intervals n must be even !')
end;
if ~(a<b)
  error('The limits of the integral must be a > b !')
end;
h=(b-a)/n;
y=f(a)+f(b);
k=1;
for x=a+h:h:b-h
    if mod(k,2)==0
      y=y+2*f(x);
    else
      y=y+4*f(x);
    end
    k=k+1;
end
y=y*h/3;
```

The `simpson` function can be called with the same four input parameters as in the case of the rectangle or trapezoid method: f is the integrated function that can be defined in MATLAB with the help of the `inline command`, a, b are the integration limits ($a < b$) and n is the number of subintervals into which the interval $\langle a; b\rangle$ was divided (it must be an even number, which is also checked in the program via the `mod` command for calculating the remainder of division by an integer).

The calculation and comparison of the accuracy with the exact analytical solution can then be obtained via the commands:

```
clc;
format long;
g=inline('log(x)');
int=simpson(g,1,2,4)
res=log(4)-1;
fprintf('Deviation from the exact solution = %8.6e\n\n',res-int)
```

The integration results using Simpson's method with four, eight, sixteen and thirty-two subintervals are as follows:

```
integral =
   0.386259562814567

Deviation from the exact solution = 3.479831e-005

integral =
   0.386292043466313

Deviation from the exact solution = 2.317654e-006

integral =
   0.386294213675793

Deviation from the exact solution = 1.474441e-007

integral =
   0.386294351862333

Deviation from the exact solution = 9.257558e-009
```

which testifies to the greater accuracy and efficiency of the solution than in the previous two numerical methods of integration. ▲

Example 7.12. Use Simpson's method with division into $n = 32$ subintervals to determine the coordinates of the centroid of the circular arc, the diagram of which is shown in Fig. 7.3. The radius of the circular arc is $r = 8$ m and the central angles are $\varphi_a = -30°$ and $\varphi_b = 22°$.

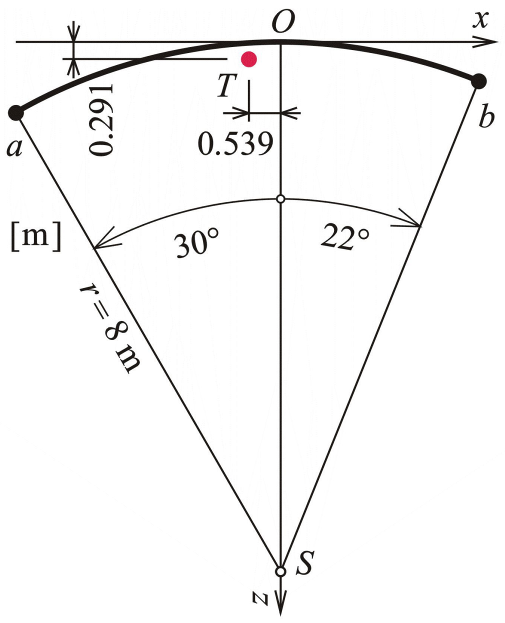


Fig. 7.3 Circular arc diagram.

Solution. The length of the circular arc is given by the equation:

$$s = \int_s \mathrm{d}s \,, \tag{7.32}$$

whereas $\mathrm{d}s = r \cdot \mathrm{d}\varphi$, so that equation (7.32) can be modified:

$$s = \int_{\varphi_a}^{\varphi_b} r \,\mathrm{d}\varphi = r \cdot (\varphi_b - \varphi_a) \,. \tag{7.33}$$

The required static moments are determined as follows:

$$S_x = \int_s z \,\mathrm{d}s = \int_{\varphi_a}^{\varphi_b} r \cdot z \,\mathrm{d}\varphi \tag{7.34}$$

and

$$S_z = \int_s x \,\mathrm{d}s = \int_{\varphi_a}^{\varphi_b} r \cdot x \,\mathrm{d}\varphi \,. \tag{7.35}$$

When converting from Cartesian coordinates to polar coordinates, the equations $x = r \cdot \sin\varphi$ and $z = r \cdot (1 - \cos\varphi)$ can be used, so the relations (7.34) and (7.35) will be modified into the following forms:

$$S_x = \int_{\varphi_a}^{\varphi_b} r^2 \cdot (1 - \cos\varphi) \,\mathrm{d}\varphi \tag{7.36}$$

and

$$S_z = \int_{\varphi_a}^{\varphi_b} r^2 \cdot \sin\varphi \,\mathrm{d}\varphi \,. \tag{7.37}$$

The resulting coordinates of the centroid of the parabolic arc in the given coordinate system are then:

$$x_T = \frac{S_z}{s} \tag{7.38}$$

and

$$z_T = \frac{S_x}{s} \,. \tag{7.39}$$

A specific solution in MATLAB can be done using a sequence of commands:

```
clc;
format long;
fia=-30/180*pi;
fib=22/180*pi;
r=8;
g1=inline('1-cos(fi)');
g2=inline('sin(fi)');
int1=simpson(g1,fia,fib,32);
int2=simpson(g2,fia,fib,32);
s=r*(fib-fia)
xT=int2*r^2/s
zT=int1*r^2/s
```

which yield the following results:

```
s =
   7.260569688296410


xT =
  -0.539095557536041


zT =
   0.290574351201034
```

▲

Example 7.13. Using Simpson's method, determine the coordinates of the centroid of the parabolic arc, the diagram of which is shown in Fig. 7.4. The shape of the median is given by a quadratic parabola with the equation $z(x) = k \cdot x^2$. The horizontal coordinates of both extreme points are $x_a = -2$ m and $x_b = 6$ m, the vertical ordinate of point b is then $z_b = 2$ m.

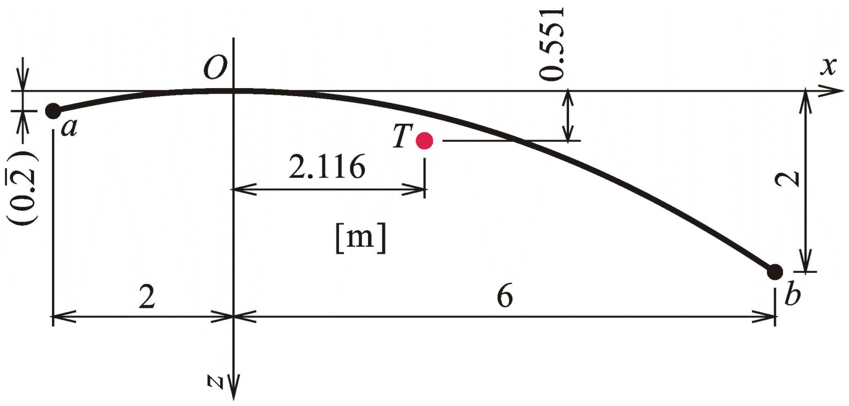


Fig. 7.4 Parabolic arc diagram.

Solution. The parameter k of the parabolic arc is determined from the relation:

$$k = \frac{z_b}{x_b^2} \, . \tag{7.40}$$

The length of the arc is given by the relation:

$$s = \int_s \mathrm{d}s = \int_{x_a}^{x_b} \sqrt{1 + (z')^2} \, \mathrm{d}x = \int_{x_a}^{x_b} \sqrt{1 + 4 \cdot k^2 \cdot x^2} \, \mathrm{d}x \, . \tag{7.41}$$

The required static moments are determined as follows:

$$S_x = \int_s z \, \mathrm{d}s = \int_{x_a}^{x_b} k \cdot x^2 \cdot \sqrt{1 + 4 \cdot k^2 \cdot x^2} \, \mathrm{d}x \tag{7.42}$$

and

$$S_z = \int_s x \, \mathrm{d}s = \int_{x_a}^{x_b} x \cdot \sqrt{1 + 4 \cdot k^2 \cdot x^2} \, \mathrm{d}x \, . \tag{7.43}$$

The resulting coordinates of the center of gravity of the parabolic arc in the given coordinate system are then:

$$x_T = \frac{S_z}{s} \tag{7.44}$$

and

$$z_T = \frac{S_x}{s} \, . \tag{7.45}$$

A specific solution in MATLAB can be obtained by the following sequence of commands:

```
clc;
format long;
xa=-2;
```

```
xb=6;
g1=inline('sqrt(1+(2*(2/(6^2))*x)^2)');
g2=inline('(2/(6^2))*x^2*sqrt(1+(2*(2/(6^2))*x)^2)');
g3=inline('x*sqrt(1+(2*(2/(6^2))*x)^2)');
int1=simpson(g1,xa,xb,32);
int2=simpson(g2,xa,xb,32);
int3=simpson(g3,xa,xb,32);
xT=int3/int1
zT=int2/int1
```

which yield the following results:

```
xT =
   2.115895489649506


zT =
   0.550954275587375
```

▲

Comment 7.14. A certain flaw of the previous calculation is the definition of a trio of inline functions using specific input values for $x_b = 2$ and $z_b = 6$, which enter these functions as the definition of the parabola parameter $k = \frac{2}{6^2}$. In this way, the use of inline functions with two variables $g(k, x)$ could be avoided, for which the `simpson` m-function would also have to be modified. Try to generalize the given calculation.

Examples to Practice

!

1. Use Simpson's method to approximate the following integrals:

   a) $\displaystyle\int_0^1 x^2 \,\mathrm{d}x\,,$

   b) $\displaystyle\int_0^{\pi} \sin^2(x) \,\mathrm{d}x\,,$

   c) $\displaystyle\int_0^{\frac{\pi}{2}} \cos(x) \,\mathrm{d}x\,,$

   d) $\displaystyle\int_0^1 \mathrm{e}^x \,\mathrm{d}x\,,$

   Choose the number of intervals gradually $n = 4, 8, 16, 32$. Compare the results with the exact solution.

7.4 Romberg's Method

Romberg's method of numerical integration is based on an idea related to a more detailed expression of the solution by the trapezoid method:

$$\int_a^b f(x)\,\mathrm{d}x = \frac{h}{2}\cdot\left(f(x_0) + 2\cdot\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right) + c_2\cdot h^2 + c_4\cdot h^4 + c_6\cdot h^6 + \ldots \quad (7.46)$$

where c_i depends only on the derivatives of $f(x)$ in the interval $\langle a; b\rangle$ and not on h.

We introduce the following sequence of differences h_i for $i = 1, 2, \ldots, j$:

$$\begin{aligned} h_1 &= b - a \\ h_2 &= \frac{1}{2}\cdot(b-a) \\ h_3 &= \frac{1}{4}\cdot(b-a) \\ &\vdots \end{aligned} \quad (7.47)$$

$$h_j = \frac{1}{2^{j-1}}\cdot(b-a)\ , \quad (7.48)$$

as well as the respective approximations of the solved integral:

$$\begin{aligned} R_{1,1} &= \frac{h_1}{2}\cdot(f(a) + f(b)) \\ R_{2,1} &= \frac{h_2}{2}\cdot\left(f(a) + f(b) + 2\cdot f\left(\frac{a+b}{2}\right)\right) = \frac{1}{2}\cdot R_{1,1} + h_2\cdot f\left(\frac{a+b}{2}\right) \\ &\vdots \end{aligned} \quad (7.49)$$

$$R_{j,1} = \frac{1}{2}\cdot R_{j-1,1} + h_j\cdot\sum_{i=1}^{2^{j-2}} f(a + (2\cdot i - 1)\cdot h_j) \quad (7.50)$$

for $j = 2, 3, \ldots, n$.

The next step of Romberg's method is the determination of refined approximations of the integrals $R_{j,k}$ for $k = 2, \ldots, j$, which can be determined using the previous values of the approximations $R_{j,k-1}$ and $R_{j-1,k-1}$:

$$R_{j,k} = \frac{4^{k-1}\cdot R_{j,k-1} - R_{j-1,k-1}}{4^{k-1} - 1}\ . \quad (7.51)$$

The most accurate approximation of the solved integral is then $R_{j,j}$, which can be determined using equations (7.48) and (7.51) in the loop, as schematically indicated in Algorithm 21.

```
Input  : $n, a, b$
Output: R
$R_{1,1} = (b-a) \cdot \frac{f(a)+f(b)}{2}$
for $j \leftarrow 2, 3, \ldots, n$ do
    $h_j = \frac{b-a}{2^{j-1}}$
    $R_{j,1} = \frac{1}{2} \cdot R_{j-1,1} + h_j \cdot \sum_{i=1}^{2^{j-2}} f(a + (2 \cdot i - 1) \cdot h_j)$
    for $k \leftarrow 2, 3, \ldots, j$ do
        $R_{j,k} = \frac{4^{k-1} \cdot R_{j,k-1} - R_{j-1,k-1}}{4^{k-1} - 1}$
    end
end
```

Algorithm 21: Algorithm for Romberg's method of numerical integration.

Example 7.15. Use Romberg's method of numerical integration to calculate the approximation of the integral:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.52}$$

for dividing $n = 3$. Compare the resulting approximation with the result of the exact analytical solution.

Solution. The calculation of the integral using Romberg's method can be programmed in MATLAB in the following way, for example:

```
function r=romberg(f,a,b,n)
h=(b-a)./(2.^(0:n-1));
r(1,1)=(b-a)*(f(a)+f(b))/2;
for j=2:n
  s=0;
  for i=1:2^(j-2)
    s=s+f(a+(2*i-1)*h(j));
  end
  r(j,1)=r(j-1,1)/2+h(j)*s;
  for k=2:j
    r(j,k)=(4^(k-1)*r(j,k-1)-r(j-1,k-1))/(4^(k-1)-1);
  end
end
```

The result for $n = 3$ is then:

```
integral =
   0.346573590279973                    0                    0
   0.376019349194069    0.385834602165434                    0
   0.383699509409442    0.386259562814567    0.386287893524509

Deviation from the exact solution = 6.467595e-006
```

▲

7.5 Adaptive Integration

The adaptive method of numerical integration differs from the previous methods of numerical integration by the uneven division of the integration interval $\langle a;b\rangle$. In places where the integrated function is sufficiently smooth and varies slowly, coarser interval divisions can be used. On the other hand, in places where the integrated function changes significantly, it is advisable to use a finer division of intervals.

The method is based on a modification of the trapezoid or Simpson's method. In the case of the trapezoid method it holds that:

$$\int_a^b f(x)\,\mathrm{d}x = S_{a,b} - h^3 \cdot \frac{f''(c_0)}{12}\,, \tag{7.53}$$

where $h = b-a$ and $a < c_0 < b$. If the point c_0 is chosen in the middle of the interval $\langle a;b\rangle$, the relation (7.53) can be modified as follows:

$$\int_a^b f(x)\,\mathrm{d}x = S_{a,c} - \frac{h^3}{8}\cdot\frac{f''(c_1)}{12} + S_{c,b} - \frac{h^3}{8}\cdot\frac{f''(c_2)}{12} = \\ = S_{a,c} + S_{c,b} - \frac{h^3}{4}\cdot\frac{f''(c_3)}{12}\,. \tag{7.54}$$

Using the difference of relations (7.54) from (7.53), it is possible to obtain an estimate of the error of the given calculation operation:

$$S_{a,b} - (S_{a,c} + S_{c,b}) = h^3\cdot\frac{f''(c_0)}{12} - \frac{h^3}{4}\cdot\frac{f''(c_3)}{12} \approx \frac{3}{4}\cdot h^3\cdot\frac{f''(c_3)}{12}\,, \tag{7.55}$$

which is roughly three times the inaccuracy of the calculation of the expression (7.53). When entering the desired accuracy bound ε, it is then possible to define the terminating condition as follows:

$$S_{a,b} - (S_{a,c} + S_{c,b}) < 3\cdot\varepsilon\,. \tag{7.56}$$

If the criterion of the termination condition is not met, both intervals are divided into half. On each part, the terminating condition is then evaluated separately, which leads to an uneven division of the integrated interval $\langle a;b\rangle$ into sections with the same inaccuracy.

Example 7.16. Determine the approximations of the integral:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.57}$$

using the adaptive method of numerical integration, based on the trapezoid method, with the required accuracy bound $1\cdot 10^{-6}$. Compare both calculated approximations with the result of the exact analytical solution.

Solution. The calculation of the integral using the adaptive method, based on the trapezoid rule, can be performed in the MATLAB, for example, with the following m-function:

```
function s=adap_int(f,a0,b0,tol0)
s=0;
n=1;
a(1)=a0;
b(1)=b0;
tol(1)=tol0;
S(1)=trapez(f,a,b);
while n>0
  c=(a(n)+b(n))/2;
  oldS=S(n);
  S(n)=trapez(f,a(n),c);
  S(n+1)=trapez(f,c,b(n));
  if abs(oldS-(S(n)+S(n+1)))<3*tol(n)
     s=s+S(n)+S(n+1);
     n=n-1;
  else
     b(n+1)=b(n);
     b(n)=c;
     a(n+1)=c;
     tol(n)=tol(n)/2;
     tol(n+1)=tol(n);
     n=n+1;
  end
end
```

where the `trapez` function applies the trapezoidal rule:

```
function s=trapez(f,a,b)
s=(f(a)+f(b))*(b-a)/2;
```

After specifying the desired solution inaccuracy tolerance $\varepsilon = 1 \cdot 10^{-6}$, the result of the integral will be:

```
integral =
   0.386293831301211

Deviation from the exact solution = 5.298187e-007
```

The algorithm can be demonstrated by listing individual subintervals of the solved integral. When specifying the desired tolerance of inaccuracy $\varepsilon = 1 \cdot 10^{-3}$, the original interval $\langle a; b \rangle$ is divided into ten subintervals with different widths, on which the trapezoid method is applied (the subintervals are shown within the operations of the calculation algorithm, i.e. in reverse order):

```
   i    ai      bi       hi
------------------------------
   1  1.8750  2.0000  0.125000
   2  1.7500  1.8750  0.125000
   3  1.6250  1.7500  0.125000
   4  1.5000  1.6250  0.125000
   5  1.3750  1.5000  0.125000
   6  1.2500  1.3750  0.125000
   7  1.1875  1.2500  0.062500
   8  1.1250  1.1875  0.062500
   9  1.0625  1.1250  0.062500
  10  1.0000  1.0625  0.062500
```

▲

Comment 7.17. The algorithm of the adaptive numerical integration method can be modified so that the solution is based on Simpson's integration method (e.g. [8]).

7.6 Gaussian Method

The Gaussian method of numerical integration (Gaussian quadrature) is based on the relation:

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx \sum_{i=1}^{n} c_i \cdot f(x_i) \,, \tag{7.58}$$

where the coefficients c_i and the roots x_i for $n = 1, 2, \ldots, 5$ integration points are given in Table 7.1.

n	x_i	c_i
1	0	2
2	$-\sqrt{\frac{1}{3}} \approx -0.5774$	1
	$\sqrt{\frac{1}{3}} \approx 0.5774$	1
3	$-\sqrt{\frac{3}{5}} \approx -0.7746$	$\frac{5}{9} \approx 0.5556$
	0	$\frac{8}{9} \approx 0.8889$
	$\sqrt{\frac{3}{5}} \approx 0.7746$	$\frac{5}{9} \approx 0.5556$
4	$-\sqrt{\frac{15+2\cdot\sqrt{30}}{35}} \approx -0.8611$	$\frac{90-5\cdot\sqrt{30}}{180} \approx 0.3479$
	$-\sqrt{\frac{15-2\cdot\sqrt{30}}{35}} \approx -0.3400$	$\frac{90+5\cdot\sqrt{30}}{180} \approx 0.6521$
	$\sqrt{\frac{15-2\cdot\sqrt{30}}{35}} \approx 0.3400$	$\frac{90+5\cdot\sqrt{30}}{180} \approx 0.6521$
	$\sqrt{\frac{15+2\cdot\sqrt{30}}{35}} \approx 0.8611$	$\frac{90-5\cdot\sqrt{30}}{180} \approx 0.3479$
5	$-\sqrt{\frac{35+2\cdot\sqrt{70}}{63}} \approx -0.9062$	$\frac{322-13\cdot\sqrt{70}}{900} \approx 0.2369$
	$-\sqrt{\frac{35-2\cdot\sqrt{70}}{63}} \approx -0.5385$	$\frac{322+13\cdot\sqrt{70}}{900} \approx 0.4786$
	0	$\frac{128}{225} \approx 0.5689$
	$\sqrt{\frac{35-2\cdot\sqrt{70}}{63}} \approx 0.5385$	$\frac{322+13\cdot\sqrt{70}}{900} \approx 0.4786$
	$\sqrt{\frac{35+2\cdot\sqrt{70}}{63}} \approx 0.9062$	$\frac{322-13\cdot\sqrt{70}}{900} \approx 0.2369$

Tab. 7.1 Roots x_i and coefficients c_i of Gaussian quadrature for $n = 1, 2, \ldots, 5$ points.

When solving the integral with generally specified limits $\langle a; b\rangle$, it is necessary to transform the integral (7.58):

$$\int_a^b f(x)\,\mathrm{d}x = \int_{-1}^1 f\left(\frac{(b-a)\cdot t + b + a}{2}\right)\cdot\frac{b-a}{2}\,\mathrm{d}t \approx \\ \approx \sum_{i=1}^{n} c_i \cdot f\left(\frac{(b-a)\cdot t_i + b + a}{2}\right)\cdot\frac{b-a}{2}\,, \tag{7.59}$$

where t is the result of the substitution

$$t = \frac{2\cdot x - a - b}{b-a}\,, \tag{7.60}$$

and t_i the respective root x_i of the Gaussian quadrature.

Example 7.18. Approximate the integral:

$$\int_1^2 \ln(x)\,\mathrm{d}x \tag{7.61}$$

using the Gaussian method of numerical integration successively for $n = 1, 2, \ldots, 5$ integration points and determine the deviations of these approximations from the exact solution.

Solution. The m-function for calculating the Gaussian quadrature for $n = 1, 2$ až 5 integration points can take the form:

```
function s=gauss_int(f,a,b,n)
if ~((n==1)|(n==2)|(n==3)|(n==4)|(n==5))
  error('The number of intervals n must be 1,2,3,4 or 5 !')
end;
if ~(a<b)
  error('The limits of the integral must be a > b !')
end;
if n==1
  x(1)=0; c(1)=2;
end
if n==2
  x(1)=-sqrt(1/3); x(2)=sqrt(1/3);
  c(1)=1; c(2)=1;
end
if n==3
  x(1)=-sqrt(3/5); x(2)=0; x(3)=sqrt(3/5);
  c(1)=5/9; c(2)=8/9; c(3)=5/9;
end
if n==4
  x(1)=-sqrt((15+2*sqrt(30))/35);
  x(2)=-sqrt((15-2*sqrt(30))/35);
  x(3)=sqrt((15-2*sqrt(30))/35);
  x(4)=sqrt((15+2*sqrt(30))/35);
  c(1)=(90-5*sqrt(30))/180;
  c(2)=(90+5*sqrt(30))/180;
  c(3)=(90+5*sqrt(30))/180;
  c(4)=(90-5*sqrt(30))/180;
end
if n==5
  x(1)=-sqrt((35+2*sqrt(70))/63);
  x(2)=-sqrt((35-2*sqrt(70))/63);
  x(3)=0;
  x(4)=sqrt((35-2*sqrt(70))/63);
  x(5)=sqrt((35+2*sqrt(70))/63);
  c(1)=(322-13*sqrt(70))/900;
  c(2)=(322+13*sqrt(70))/900;
  c(3)=128/225;
  c(4)=(322+13*sqrt(70))/900;
  c(5)=(322-13*sqrt(70))/900;
end
s=0;
for i=1:n
  s=s+(f(((b-a)*x(i)+b+a)/2)*(b-a)/2)*c(i);
end
```

As for a comparison of the accuracy of the solution with increasing number of integration points $n = 1, 2, \ldots, 5$, it is possible to obtain the following results:

```
integral =
   0.405465108108164

Deviation from the exact solution = -1.917075e-002

integral =
   0.386594944116741

Deviation from the exact solution = -3.005830e-004

integral =
   0.386300421584011

Deviation from the exact solution = -6.060464e-006

integral =
   0.386294496938714

Deviation from the exact solution = -1.358188e-007

integral =
   0.386294364348948

Deviation from the exact solution = -3.229058e-009
```

▲

Chapter 8

Numerical Derivation

Objectives

This chapter introduces

- the basic calculation procedures for numerically determining the approximate value of the derivatives of a function,
- an advanced method of numerical derivation,
- the basics of solving partial derivatives of a function of two variables numerically.

Numerical derivation consists in determining the approximate value of the derivative of the function $f(x)$ at a specific point x using the function values at the surrounding points and interpolation polynomials of degree m, for which:

$$f'(x) \approx p'_m(x) \ . \tag{8.1}$$

The derivative of the function $f(x)$ at x gives the direction of the tangent to the function at that point.

8.1 Finite Difference Method

The simplest formulas for calculating the derivative of the function $f(x)$ at x_0 and x_1 ($x_0 < x_1$) take the form:

$$f'(x_0) = \frac{f(x_1) - f(x_0)}{h} - \frac{h}{2} \cdot f''(\xi_0) \tag{8.2}$$

and

$$f'(x_1) = \frac{f(x_1) - f(x_0)}{h} + \frac{h}{2} \cdot f''(\xi_1) \ , \tag{8.3}$$

where $h = x_1 - x_0$ and $\xi_0, \xi_1 \in \langle x_0, x_1 \rangle$. The calculation assumes the existence of the second derivative $f''(x)$ in the solved interval $\langle x_0, x_1 \rangle$. The final terms in

Equations (8.2) and (8.3) denote the errors of the calculated approximation, which is ignored when calculating $f'(x)$.

Both points x_0 and x_1 are therefore separated by a difference of h. Equation (8.2) is generalized as follows:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2} \cdot f''(\xi) \approx \frac{f(x+h) - f(x)}{h} \; , \tag{8.4}$$

where ξ lies in the interval $\langle x, x+h \rangle$. To determine the derivative at x according to (8.4), it is therefore necessary to also know the value of the function $f(x)$ at the second point $x+h$. The equation (8.4) is referred to as the *two-point forward difference formula.*

Definition 8.1. The derivative of the function f at x is defined by the rule:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \; . \tag{8.5}$$

Similarly, Equation (8.3) is generalized as follows:

$$f'(x) = \frac{f(x) - f(x-h)}{h} + \frac{h}{2} \cdot f''(\xi) \approx \frac{f(x) - f(x-h)}{h} \; . \tag{8.6}$$

In this case, we refer to the *two-point backward difference formula*, because to determine the derivative at x, it is also necessary to know the value of the function $f(x)$ at $x - h$.

Example 8.2. Approximate the derivative of the function:

$$f(x) = \frac{1}{x} \tag{8.7}$$

at $x = 2$ with the difference $h = 0.1$ using Equation (8.4).

Solution. Using the two-point forward difference formula, we obtain:

$$f'(x = 2) \approx \frac{f(2 + 0.1) - f(2)}{0.1} = \frac{\frac{1}{2.1} - \frac{1}{2}}{0.1} \approx -0.238095238095238 \; . \tag{8.8}$$

The analytically determined first derivative of the function (8.7) is:

$$f'(x) = -\frac{1}{x^2} \; , \tag{8.9}$$

and for $x = 2$, the exact value of the solution is:

$$f'(2) = -\frac{1}{2^2} = -\frac{1}{4} = -0.25 \; , \tag{8.10}$$

therefore the deviation of the resulting approximation of the exact analytical solution and the error in the numerical calculation of the derivative are:

$$-0.238095238095238-(-0.25)\approx 0.011904761904762\ . \tag{8.11}$$

In (8.4), the calculation error is expressed as:

$$\frac{h}{2}\cdot f''(\xi)\ . \tag{8.12}$$

The second derivative of the function (8.7) is:

$$f''(x)=\frac{2}{x^3}\ , \tag{8.13}$$

therefore (8.12) can be evaluated for $\xi=2$:

$$\frac{0.1}{2}\cdot\frac{2}{2^3}=0.0125\ . \tag{8.14}$$

and $\xi=2.1$:

$$\frac{0.1}{2}\cdot\frac{2}{(2.1)^3}\approx 0.010797969981643\ . \tag{8.15}$$

The error in the calculation of (8.11) in fact lies between the values of (8.14) and (8.15). ▲

Example 8.3. Using Equation (8.6), approximate the derivative of the function:

$$f(x)=\frac{1}{x} \tag{8.16}$$

at $x=2$ with the difference $h=0.1$.

Solution. Using the two-point backward difference formula, we obtain:

$$f'(x=2)\approx\frac{f(2)-f(2-0.1)}{0.1}=\frac{\frac{1}{2}-\frac{1}{1.9}}{0.1}\approx -0.263157894736842\ . \tag{8.17}$$

The error in the calculation is:

$$-0.25-(-0.263157894736842)\approx 0.013157894736842 \tag{8.18}$$

and lies in the range of values:

$$\frac{0.1}{2}\cdot\frac{2}{2^3}=0.0125 \tag{8.19}$$

and

$$\frac{0.1}{2}\cdot\frac{2}{(1.9)^3}\approx 0.014579384749964\ . \tag{8.20}$$

▲

The calculation of the derivative according to (8.2) and (8.3) is based on the derivative of the first degree interpolation polynomial p_1 at the nodes x_0 and $x_1 = = x_0 + h$. Using the second degree polynomial p_2, we obtain:

$$f'(x_0) = \frac{-3 \cdot f(x_0) + 4 \cdot f(x_1) - f(x_2)}{2 \cdot h} + \frac{h^2}{3} \cdot f'''(\xi_0) \;, \tag{8.21}$$

$$f'(x_2) = \frac{f(x_0) - 4 \cdot f(x_1) + 3 \cdot f(x_2)}{2 \cdot h} + \frac{h^2}{3} \cdot f'''(\xi_1) \tag{8.22}$$

and

$$f'(x_1) = \frac{f(x_2) - f(x_0)}{2 \cdot h} - \frac{h^2}{12} \cdot f'''(\xi_0) - \frac{h^2}{12} \cdot f'''(\xi_1) \;, \tag{8.23}$$

where $h = x_1 - x_0 = x_2 - x_1$, $\xi_0 \in \langle x_0, x_1 \rangle$ and $xi_1 \in \rangle x_1, x_2 \rangle$. The calculation assumes that a third derivative $f'''(x)$ exists in the solved interval $\langle x_0, x_2 \rangle$.

Eq. (8.21) is generalized as:

$$\begin{aligned} f'(x) &= \frac{-3 \cdot f(x) + 4 \cdot f(x+h) - f(x+2 \cdot h)}{2 \cdot h} + \frac{h^2}{3} \cdot f'''(\xi_0) \approx \\ &\approx \frac{-3 \cdot f(x) + 4 \cdot f(x+h) - f(x+2 \cdot h)}{2 \cdot h} \;. \end{aligned} \tag{8.24}$$

To determine the derivative at x according to (8.24), it is necessary to also know the value of the function $f(x)$ at the other two points $x + h$ and $x + 2 \cdot h$. Equation (8.24) is referred to as a *three-point forward difference formula.*

Definition 8.4. The derivative of the function f at x is defined as:

$$f'(x) = \lim_{h \to 0} \frac{-3 \cdot f(x) + 4 \cdot f(x+h) - f(x+2 \cdot h)}{2 \cdot h} \;. \tag{8.25}$$

Similarly, Equation (8.22) is generalized as:

$$\begin{aligned} f'(x) &= \frac{f(x - 2 \cdot h) - 4 \cdot f(x-h) + 3 \cdot f(x)}{2 \cdot h} + \frac{h^2}{3} \cdot f'''(\xi_1) \approx \\ &\approx \frac{f(x - 2 \cdot h) - 4 \cdot f(x-h) + 3 \cdot f(x)}{2 \cdot h} \;. \end{aligned} \tag{8.26}$$

In this case, we refer to the *three-point backward difference formula*, because to determine the derivative at x, it is necessary to also know the value of the function $f(x)$ at the two other points $x - h$ and $x - 2 \cdot h$.

The derivative of the function $f(x)$ can also be calculated using the generalized relation (8.23):

$$\begin{aligned} f'(x) &= \frac{f(x+h) - f(x-h)}{2 \cdot h} - \frac{h^2}{12} \cdot f'''(\xi_0) - \frac{h^2}{12} \cdot f'''(\xi_1) \approx \\ &\approx \frac{f(x+h) - f(x-h)}{2 \cdot h} - \frac{h^2}{6} \cdot f'''(\xi) \approx \frac{f(x+h) - f(x-h)}{2 \cdot h} \;, \end{aligned} \tag{8.27}$$

where $\xi \in \rangle x+h, x-h\rangle$. In this case, we refer to the *three-point central difference formula*. The point x lies in the middle of the calculation interval $\langle x+h, x-h\rangle$.

Example 8.5. Approximate the derivative of the function:

$$f(x) = \frac{1}{x} \tag{8.28}$$

at $x = 2$ with the difference $h = 0.1$ using the equation (8.24).

Solution. Using the three-point forward difference formula, we can obtain:

$$\begin{aligned} f'(x=2) &\approx \frac{-3\cdot f(2) + 4\cdot f(2+0.1) - f(2+2\cdot 0.1)}{2\cdot 0.1} = \\ &= \frac{-\frac{3}{2} + \frac{4}{2.1} - \frac{1}{2.2}}{0.2} \approx -0.248917748917749\ . \end{aligned} \tag{8.29}$$

The error of the numerical calculation of the derivative equals:

$$-0.248917748917749 - (-0.25) \approx 0.001082251082251\ . \tag{8.30}$$

In (8.24) the calculation error is expressed by the formula:

$$-\frac{h^2}{3}\cdot f'''(\xi_0)\ . \tag{8.31}$$

The third derivative of the function (8.28) is:

$$f'''(x) = -\frac{6}{x^4}\ , \tag{8.32}$$

therefore (8.31) can be enumerated for $\xi = 2$:

$$-\frac{(0.1)^2}{3}\cdot -\frac{6}{2^4} = 0.00125\ , \tag{8.33}$$

and $\xi = 2.1$:

$$-\frac{(0.1)^2}{3}\cdot -\frac{6}{(2.1)^4} \approx 0.001028378093490\ . \tag{8.34}$$

The error in (8.30) in fact lies between the values of (8.33) and (8.34). ▲

Example 8.6. Using Equation (8.26), approximate the derivative of the function:

$$f(x) = \frac{1}{x} \tag{8.35}$$

at $x = 2$ with the difference $h = 0.1$.

Solution. Using the three-point backward difference formula, we obtain:

$$f'(x=2) \approx \frac{f(2-2\cdot 0.1) - 4\cdot f(2-0.1) + 3\cdot f(2)}{2\cdot 0.1} = \\ = \frac{\frac{1}{1.8} - \frac{4}{1.9} + \frac{3}{2}}{0.2} \approx -0.248538011695906 \; . \quad (8.36)$$

The error of the numerical calculation of the derivative is:

$$-0.248538011695906 - (-0.25) \approx 0.001461988304094 \; . \quad (8.37)$$

The error in (8.26) can be enumerated for $\xi = 2$:

$$-\frac{(0.1)^2}{3} \cdot -\frac{6}{2^4} = 0.00125 \; , \quad (8.38)$$

and $\xi = 1.9$:

$$-\frac{(0.1)^2}{3} \cdot -\frac{6}{(1.9)^4} \approx 0.001534672078944 \; . \quad (8.39)$$

The error in (8.37) lies between the values of (8.38) and (8.39). ▲

Example 8.7. Using (8.27), approximate the derivative of the function:

$$f(x) = \frac{1}{x} \quad (8.40)$$

at $x = 2$ with the difference $h = 0.1$.

Solution. The derivative of the function (8.40) is determined using the three-point central difference formula:

$$f'(x=2) \approx \frac{f(2+0.1) - f(2-0.1)}{2\cdot 0.1} = \frac{\frac{1}{2.1} - \frac{1}{1.9}}{0.2} \approx -0.250626566416040 \; . \quad (8.41)$$

The error in the calculation is:

$$-0.25 - (-0.250626566416040) \approx 6.265664160400863 \cdot 10^{-4} \; . \quad (8.42)$$

In (8.27), the calculation error is expressed by the formula:

$$-\frac{h^2}{6} \cdot f'''(\xi) \; . \quad (8.43)$$

The third derivative of the function (8.40), according to (8.32), is equal to:

$$f'''(x) = -\frac{6}{x^4} \; , \quad (8.44)$$

so that (8.43) can be calculated first for $\xi = 1.9$:

$$-\frac{(0.1)^2}{6} \cdot -\frac{6}{(1.9)^4} \approx 7.673360394717661 \cdot 10^{-4} \; . \tag{8.45}$$

and then for $\xi = 2.1$:

$$-\frac{(0.1)^2}{6} \cdot -\frac{6}{(2.1)^4} \approx 5.141890467449263 \cdot 10^{-4} \; . \tag{8.46}$$

The error in the calculation of (8.42) lies between the values of (8.45) and (8.46). ▲

If the function to be solved has a fourth derivative $f''''(x)$, then the second derivative $f''(x)$ at x_1 can be determined using the polynomial of the second degree p_2:

$$f''(x_1) = \frac{f(x_0) - 2 \cdot f(x_1) + f(x_2)}{h^2} - \frac{h^2}{12} \cdot f''''(\xi) \; , \tag{8.47}$$

for $\xi \in \langle x_0, x_2 \rangle$.

Example 8.8. Using (8.47), approximate the second derivative of the function:

$$f(x) = \frac{1}{x} \tag{8.48}$$

at $x = 2$ with the difference $h = 0.1$.

Solution. The approximation of the second derivative of the function (8.48) is determined by substituting the appropriate values into expression (8.47):

$$f''(2) = \frac{f(2-0.1) - 2 \cdot f(2) + f(2+0.1)}{(0.1)^2} \approx 0.250626566416034 \; . \tag{8.49}$$

The analytically determined second derivative of the function (8.48) is defined by (8.13):

$$f''(x) = \frac{2}{x^3} \; , \tag{8.50}$$

and for $x = 2$, the exact solution is therefore:

$$f''(2) = \frac{2}{2^3} = \frac{2}{8} = 0.25 \; , \tag{8.51}$$

and means that the error in calculating by (8.49) is:

$$0.250626566416034 - 0.25 \approx 6.265664160344797 \cdot 10^{-4} \; . \tag{8.52}$$

In (8.47), the calculation error is expressed by the formula:

$$\frac{h^2}{12} \cdot f''''(\xi) \ . \tag{8.53}$$

The fourth derivative of the function (8.48) is:

$$f''''(x) = \frac{24}{x^5} \ , \tag{8.54}$$

and means that (8.53) can first be calculated for $\xi = 1.9$:

$$\frac{(0.1)^2}{12} \cdot \frac{24}{(1.9)^5} \approx 8.077221468123856 \cdot 10^{-4} \ . \tag{8.55}$$

and then for $\xi = 2.1$:

$$\frac{(0.1)^2}{12} \cdot \frac{24}{(2.1)^5} \approx 4.897038540427868 \cdot 10^{-4} \ . \tag{8.56}$$

The error in the calculation of (8.52) lies between the values of (8.55) and (8.56).
▲

The relation (8.47) can also be obtained as a derivative from the first derivatives:

$$f''(x) \approx \frac{f'(x) - f'(x-h)}{h} =$$

$$= \frac{\dfrac{f(x+h) - f(x)}{h} - \dfrac{f(x) - f(x-h)}{h}}{h} =$$

$$= \frac{f(x+h) - 2 \cdot f(x) + f(x-h)}{h^2} \ , \tag{8.57}$$

where $f'(x)$ is determined using the two-point forward difference formula (8.4).

The third derivative of the function $f(x)$ is similarly determined, for example, by using the three-point central difference formula (8.27):

$$f'''(x) \approx \frac{f''(x+h) - f''(x-h)}{2 \cdot h} =$$

$$= \frac{\dfrac{f(x+2 \cdot h) - 2 \cdot f(x+h) + f(x)}{h^2} - \dfrac{f(x) - 2 \cdot f(x-h) + f(x-2 \cdot h)}{h^2}}{2 \cdot h} =$$

$$= \frac{f(x+2 \cdot h) - 2 \cdot f(x+h) + 2 \cdot f(x-h) - f(x-2 \cdot h)}{2 \cdot h^3} \ , \tag{8.58}$$

or the fourth derivative of the function $f(x)$, for example, by using the difference formula (8.47):

$$f''''(x) \approx \frac{f''(x+h) - 2 \cdot f''(x) + f''(x-h)}{h^2} =$$

$$= \frac{\frac{f(x+2\cdot h) - 2\cdot f(x+h) + f(x)}{h^2} - 2 \cdot \frac{f(x+h) - 2\cdot f(x) + f(x-h)}{h^2} +}{h^2}$$

$$\frac{+\frac{f(x) - 2\cdot f(x-h) + f(x-2\cdot h)}{h^2}}{h^2} =$$

$$= \frac{f(x+2\cdot h) - 4\cdot f(x+h) + 6\cdot f(x) - 4\cdot f(x-h) + f(x-2\cdot h)}{h^4} \ . \quad (8.59)$$

Examples to Practice

1. Use the relationships for numerically determining the first, second, third and fourth derivatives to calculate the slope, bending moment, shear force and load by deriving the deflection curve of the structure from the exercises below:
   a) 3.1
   b) 3.2
   c) 3.3.

   Determine the given quantities by first tabulating them in cross-sections with spacings of 10 cm, and then, display them graphically. Compare with the exact solution.

8.2 Numerical Differentiation with a Variable Difference

In the numerical calculation of the derivative, the question arises of the size of the difference h. The solution lies in Neville's algorithm (similar to Romberg's integration – see Chapter 7.4), which was defined by English mathematician Eric Harold Neville. The calculation procedure is based on the three-point central difference formula 8.27.

The calculation is performed in a cycle with the control variable i for a total n times while the step h is adjusted according to the value:

$$h_i = \frac{h_0}{10^{i-1}} , \tag{8.60}$$

where h_0 is the initial difference value.

The magnitude of the derivative is then:

$$a_{i,1} = \frac{f(x+h_i) - f(x-h_i)}{2 \cdot h_i} , \tag{8.61}$$

which can be further refined to:

$$a_{i,j} = \frac{a_{i,j-1} \cdot 10^{2 \cdot j-2} - a_{i-1,j-1}}{10^{2 \cdot j-2} - 1} , \tag{8.62}$$

for $j = 2, 3, \ldots, n$. The most accurate estimate of the required derivative is contained in the variable $a_{n,n}$ at the end of the calculation. This calculation procedure is executed by Algorithm 22.

```
Input  : $f, x, h_0, n$
Output: a
$h_1 = h_0$
$a_{1,1} = \frac{f(x+h_1) - f(x-h_1)}{2 \cdot h_1}$
for $i \leftarrow 2, 3, \ldots, n$ do
    $h_i = \frac{h_0}{10^{i-1}}$
    $a_{i,1} = \frac{f(x+h_i) - f(x-h_i)}{2 \cdot h_i}$
    for $j \leftarrow 2, 3, \ldots, i$ do
        $a_{i,j} = \frac{a_{i,j-1} \cdot 10^{2 \cdot j-2} - a_{i-1,j-1}}{10^{2 \cdot j-2} - 1}$
    end
end
```

Algorithm 22: Neville's algorithm of numerical derivation.

Example 8.9. Calculate the derivative approximation:

$$f(x) = \frac{1}{x} \tag{8.63}$$

at $x = 2$ using Neville's algorithm of numerical differentiation. Compare the resulting approximation with the result of the exact analytical solution.

Solution. The calculation of the derivative using Neville's algorithm is executed in MATLAB, for example, as follows:

```
function a=neville(f,x,h0,n)
h(1)=h0;
a(1,1)=(f(x+h(1))-f(x-h(1)))/(2*h(1));
for i=2:n
  h(i)=h0/(10^(i-1));
  a(i,1)=(f(x+h(i))-f(x-h(i)))/(2*h(i));
  for j=2:i
    a(i,j)=(a(i,j-1)*10^(2*j-2)-a(i-1,j-1))/(10^(2*j-2)-1);
  end
end
```

The result for $n = 3$ is:

```
deriv =
  -0.250626566416040                   0                   0
  -0.250006250156248  -0.249999984335442                   0
  -0.250000062499978  -0.249999999998399  -0.249999999999966

Deviation from the exact solution = 3.438916e-014
```

▲

8.3 Partial Derivatives

When solving a function of two variables:

$$z = f(x, y) \; , \tag{8.64}$$

partial derivatives can be determined. If y is treated as a constant, the partial derivative with respect to x is defined as:

$$\frac{\partial z}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x} \; . \tag{8.65}$$

Conversely, the partial derivative of the function $f(x, y)$ with respect to y takes the form:

$$\frac{\partial z}{\partial y} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y} \; . \tag{8.66}$$

The second partial derivatives of the function $f(x, y)$ can then be determined using the equations:

$$\frac{\partial^2 z}{\partial x^2} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x, y) - 2 \cdot f(x, y) + f(x - \Delta x, y)}{\Delta x^2} \tag{8.67}$$

and

$$\frac{\partial^2 z}{\partial y^2} = \lim_{\Delta y \to 0} \frac{f(x, y + \Delta y) - 2 \cdot f(x, y) + f(x, y - \Delta y)}{\Delta y^2} \; . \tag{8.68}$$

We can also define the mixed partial derivative of the function $f(x, y)$:

$$\frac{\partial^2 z}{\partial x \partial y} = \lim_{\Delta x, \Delta y \to 0} \frac{f(x + \Delta x, y + \Delta y) - f(x + \Delta x, y - \Delta y) -}{4 \cdot \Delta x \cdot \Delta y} \\ \frac{- f(x - \Delta x, y + \Delta y) + f(x - \Delta x, y - \Delta y)}{4 \cdot \Delta x \cdot \Delta y} \; . \tag{8.69}$$

The third partial derivative of the function $f(x, y)$ is determined in a manner similar to (8.58):

$$\frac{\partial^3 z}{\partial x^3} = \lim_{\Delta x \to 0} \frac{f(x + 2 \cdot \Delta x, y) - 2 \cdot f(x + \Delta x, y) +}{2 \cdot \Delta x^3} \\ \frac{+2 \cdot f(x - \Delta x, y) - f(x - 2 \cdot \Delta x, y)}{2 \cdot \Delta x^3} \tag{8.70}$$

and

$$\frac{\partial^3 z}{\partial y^3} = \lim_{\Delta y \to 0} \frac{f(x, y + 2 \cdot \Delta y) - 2 \cdot f(x, y + \Delta y) +}{2 \cdot \Delta y^3} \\ \frac{+2 \cdot f(x, y - \Delta y) - f(x, y - 2 \cdot \Delta y)}{2 \cdot \Delta y^3} \; . \tag{8.71}$$

The same procedure is followed for the case of the fourth partial derivatives of the function $f(x, y)$, for example, in a manner similar to (8.59):

$$\frac{\partial^4 z}{\partial x^4} = \lim_{\Delta x \to 0} \frac{f(x + 2 \cdot \Delta x, y) - 4 \cdot f(x + \Delta x, y) + 6 \cdot f(x, y) -}{\Delta x^4}$$
$$\frac{-4 \cdot f(x - \Delta x, y) + f(x - 2 \cdot \Delta x, y)}{\Delta x^4} \quad (8.72)$$

and

$$\frac{\partial^4 z}{\partial y^4} = \lim_{\Delta y \to 0} \frac{f(x, y + 2 \cdot \Delta y) - 4 \cdot f(x, y + \Delta y) + 6 \cdot f(x, y) -}{\Delta y^4}$$
$$\frac{-4 \cdot f(x, y - \Delta y) + f(x, y - 2 \cdot \Delta y)}{\Delta y^4} . \quad (8.73)$$

We can also define the mixed fourth partial derivative of the function $f(x, y)$:

$$\frac{\partial^4 z}{\partial x^2 \partial y^2} = \lim_{\Delta x, \Delta y \to 0} \frac{f(x + \Delta x, y + \Delta y) - 2 \cdot f(x + \Delta x, y) + f(x + \Delta x, y - \Delta y) -}{\Delta x^2 \cdot \Delta y^2}$$

$$\frac{-2 \cdot (f(x, y + \Delta y) - 2 \cdot f(x, y) + f(x, y - \Delta y)) +}{\Delta x^2 \cdot \Delta y^2}$$

$$\frac{+f(x - \Delta x, y + \Delta y) - 2 \cdot f(x - \Delta x, y) + f(x - \Delta x, y - \Delta y)}{\Delta x^2 \cdot \Delta y^2} . \quad (8.74)$$

Chapter 9

Solving Differential Equations

Objectives

This chapter:

- introduces students to the numerical solution of simple differential equations,
- highlights how such equations can be applied to solve elementary tasks in construction mechanics.

Differential equations are equations which include derivatives of functions as variables. According to the number of variables and the type of derivatives of functions, differential equations are divided into:

- *ordinary differential equations*, which contain the derivatives of the desired function with respect to only one variable,
- *partial differential equations*, which contain the derivatives of the desired function with respect to several variables, i.e., partial derivatives.

The order of a differential equation is defined according to the highest derivative contained in the given differential equation.

The solution of a differential equation is a function that has appropriate derivatives and satisfies the given differential equation—the integral of the differential equation, of which there are infinitely many. In practical problems, the initial conditions define a unique solution.

9.1 Ordinary Differential Equations of the First Order

First-order ordinary differential equations contain one derivative of a function of one dependent variable $y(x)$. For example, it is possible to numerically solve the function

$y = y(x)$ which in the interval $\langle a, b \rangle$ satisfies the equation:

$$y'(x) = f(x, y(x)) \;, \tag{9.1}$$

where $f(x, y(x))$ is the right-hand side of an ordinary differential equation of the first order. To allow it to be uniquely determined, an initial condition on the form must be satisfied:

$$y(a) = c \;. \tag{9.2}$$

9.1.1 Euler's Method

The simplest computational procedure for the numerical solution of ordinary differential equations with an initial condition was published in 1768 by Swiss mathematician and physicist Leonhard Euler. The solution is based on the approximate calculation of the derivative of the function $y'(x)$ in Equation (9.1) using the finite difference method and the two-point forward difference formula (8.4):

$$y'(x_i) = f(x_i, y(x_i)) \approx \frac{y(x_{i+1}) - y(x_i)}{h} = f(x_i, y_i) \;, \tag{9.3}$$

where h is the step corresponding to the value of $(x_{i+1} - x_i)$. By simple modification of (9.3), we can obtain the recurrence relation of Euler's method:

$$y_{i+1} = y_i + h \cdot f(x_i, y_i) \;, \tag{9.4}$$

for $i = 0, 1, \ldots, n-1$, where n is the number of differences in the solved interval $\langle a, b \rangle$. The value of y_0 must be uniquely determined using the initial condition according to (9.2).

Example 9.1. Use Euler's method to approximate the following ordinary differential equation in the interval $\langle -2; 3 \rangle$:

$$y'(x) = x^2 - 0.2 \cdot y(x) \;, \tag{9.5}$$

with the initial condition $y(-2) = -1$. For the calculation step h, successively select h as $h = 1$, $h = 0.5$, $h = 0.1$ or $h = 0.01$. Compare the resulting approximation with the exact solution:

$$y(x) = 5 \cdot x^2 - 50 \cdot x + 250 - \frac{371}{e^{0.4}} \cdot e^{-0.2 \cdot x} \;. \tag{9.6}$$

Solution. Euler's method is based on the recurrence relation (9.4) and is executed in MATLAB, for example, with the code:

```
f=inline('x^2-0.2*y');
a=-2;
b=3;
c=-1;
h=0.1;
n=(b-a)/h;
x(1)=a;
y(1)=c;
yexact(1)=c;
for i=1:n
  x(i+1)=x(i)+h;
  y(i+1)=y(i)+h*f(x(i),y(i));
  yexact(i+1)=5*x4(i+1)^2-50*x4(i+1)+250-...
      (371/exp(0.4))*exp(-0.2*x4(i+1));
end
[x' y' yexact' (y-yexact)']
plot(x,y,'r',x,yexact,'b');
legend('Aproximation','Exact solution');
title('Euler''s Method');
xlabel('x');
ylabel('y');
```

The resulting solution for the calculation step $h = 0.5$ is:

```
     x          y          yexact   y-yexact
----------------------------------------
   -2.0000    -1.0000    -1.0000     0.0000
   -1.5000     1.1000     0.5553     0.5447
   -1.0000     2.1150     1.2509     0.8641
   -0.5000     2.4035     1.4064     0.9971
    0.0000     2.2882     1.3113     0.9769
    0.5000     2.0593     1.2271     0.8322
    1.0000     1.9784     1.3909     0.5875
    1.5000     2.2806     2.0169     0.2637
    2.0000     3.1775     3.2990    -0.1214
    2.5000     4.8598     5.4127    -0.5529
    3.0000     7.4988     8.5167    -1.0179
```

Figure 9.1 compares the accuracies of the solution for individual values of the calculation steps $h = 1$, $h = 0.5$, $h = 0.1$ and $h = 0.01$.

▲

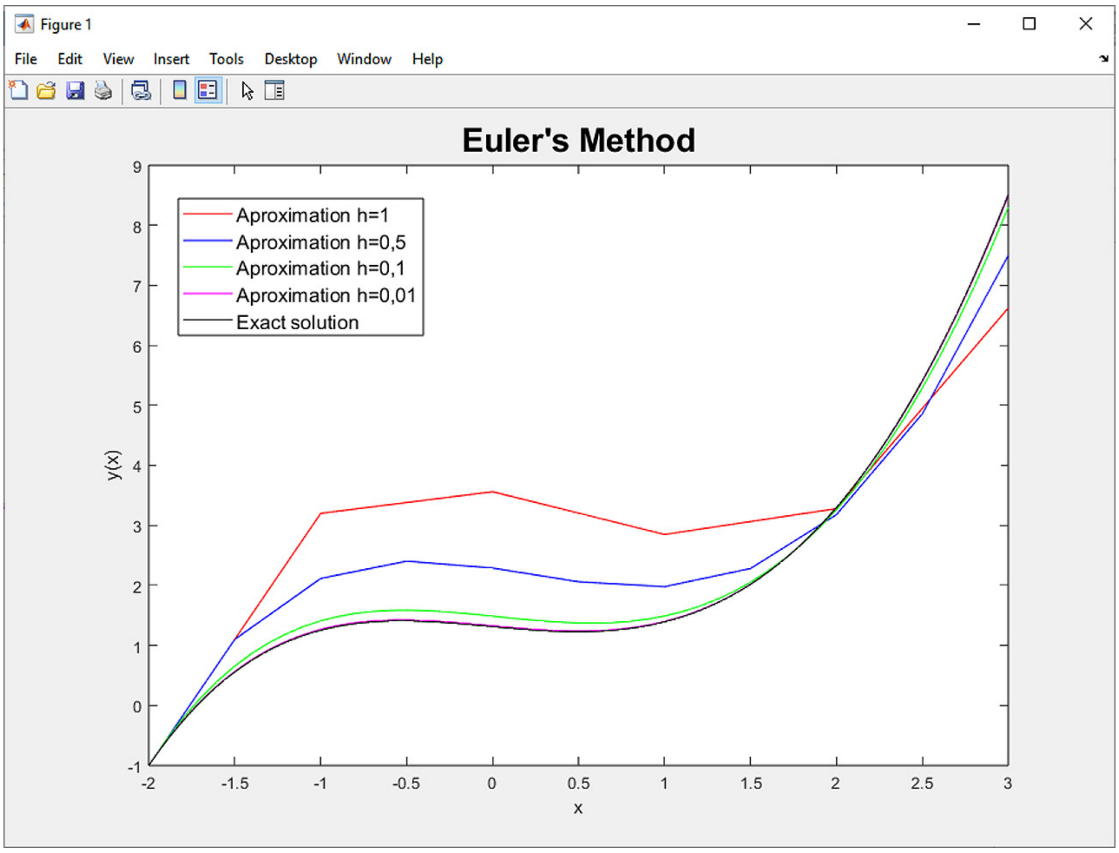


Fig. 9.1 Resulting approximation of the function $y(x)$ in Exercise 9.1 for steps $h = 1$, $h = 0.5$, $h = 0.1$ and $h = 0.01$.

Comment 9.2. The solution to the problem in Exercise 9.1 can also be checked with MATLAB's mathematical tools. One option is the `ode45` function, for example with a solution inaccuracy tolerance of $1 \cdot 10^{-9}$. It is first necessary to enter the solved differential equation using a separate `m-function`:

```
function y=funct(x,y);
y=x^2-0.2*y;
```

which can then be referred to:

```
options=odeset('AbsTol',1e-9);
[x,y]=ode45(@funct,[-2 3],-1,options)
```

The final command lists the values of the resulting function $y(x)$ at x_i. If the command is modified to:

```
ode45(@funct,[-2 3],-1)
```

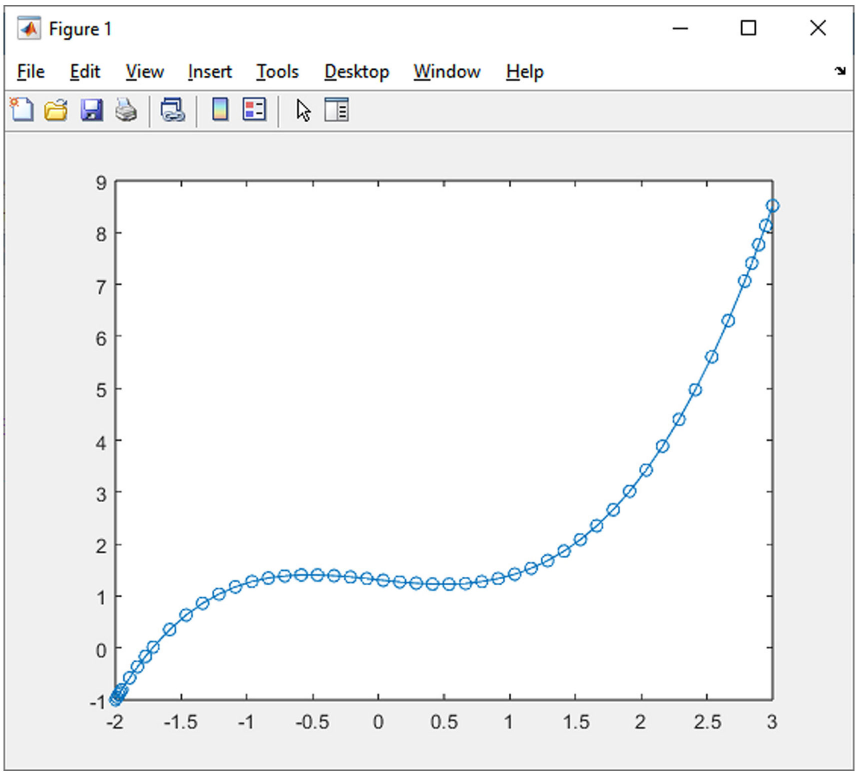


Fig. 9.2 Resulting approximation of the function $y(x)$ in Exercise 9.1 produced by the function `ode45`.

the graph of the solved function $y(x)$ is displayed (see Fig. 9.2).

The second option to solve the problem in Exercise 9.1 using MATLAB commands is the `dsolve` function for the symbolic solution of ordinary differential equations, followed by vectorization of the function $y(x)$, with the sequence of commands:

```
syms y(x) x
eqn = diff(y,x) == x^2-0.2*y;
cond = y(-2)== -1;
y(x) = dsolve(eqn,cond)
x=linspace(-2,3,1000);
z=eval(vectorize(y));
plot(x,z)
```

The resulting expression produced by the `dsolve` function:

```
y =
5*x^2 - 371*exp(-x/5)*exp(-2/5) - 50*x + 250
```

is identical to the exact solution (9.6).

Example 9.3. Using Euler's method, determine the course of the shear force on the cantilever beam shown schematically in Fig. 9.3. Specific input data are given in Table 9.1. Choose h as $h = 1$ or $h = 0.5$. Compare the resulting approximation with the exact solution.

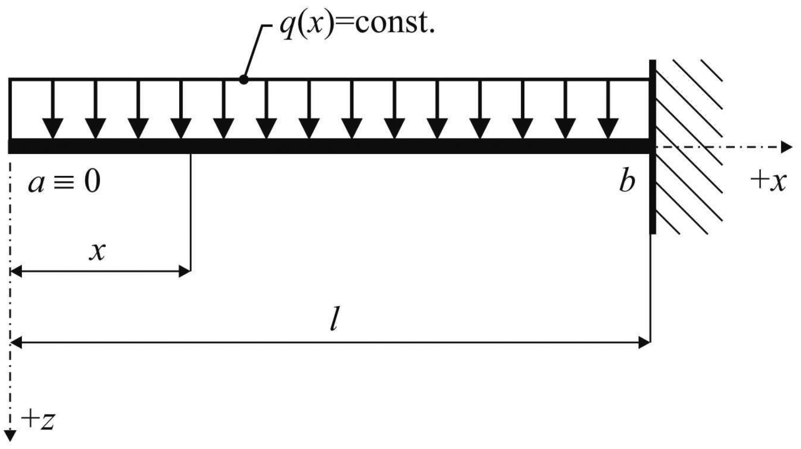


Fig. 9.3 Statics diagram for the solution of the statically determined cantilever.

Continuous force loading q_z :	4 kN/m
Cantilever span l :	6 m

Tab. 9.1 Input data from Exercise 9.3.

Solution. The ordinary differential equation of the first order follows from the well known Schwedler formula:

$$\frac{V_z(x)}{\mathrm{d}x} = -q_z(x) = \text{const} \to y'(x) = -q_z \cdot x^0 \,. \tag{9.7}$$

The initial condition is based on the static boundary condition, which indicates the zero value of the shear force at the free edge of the cantilever, i.e.:

$$V_z(x = 0) = y(x = 0) = 0 \,. \tag{9.8}$$

Approximation of the shear force using Euler's method is based on the recurrence formula (9.4). The exact solution corresponds to the analytically derived equation for the shear force $V_z(x)$:

$$V_z(x) = -q_z \cdot x \,. \tag{9.9}$$

Because the shear force function is linear, in this case, Euler's method yields an exact solution:

```
      x          y          yexac    y-yexac
----------------------------------------
    0.0000     0.0000     0.0000     0.0000
    0.5000    -2.0000    -2.0000     0.0000
    1.0000    -4.0000    -4.0000     0.0000
    1.5000    -6.0000    -6.0000     0.0000
    2.0000    -8.0000    -8.0000     0.0000
    2.5000   -10.0000   -10.0000     0.0000
    3.0000   -12.0000   -12.0000     0.0000
    3.5000   -14.0000   -14.0000     0.0000
    4.0000   -16.0000   -16.0000     0.0000
    4.5000   -18.0000   -18.0000     0.0000
    5.0000   -20.0000   -20.0000     0.0000
    5.5000   -22.0000   -22.0000     0.0000
    6.0000   -24.0000   -24.0000     0.0000
```

▲

Example 9.4. Using Euler's method, determine the course of bending moments on the cantilever described in Exercise 9.3. Select $h = 1$ or $h = 0.5$ for the calculation step h. Compare the resulting approximation with the exact solution.

Solution. The ordinary differential equation of the first order once again follows from the Schwedler formula:

$$\frac{M_y(x)}{\mathrm{d}x} = V_z(x) \rightarrow y'(x) = -q_z \cdot x \; . \tag{9.10}$$

The initial condition is based on the static boundary condition, which indicates the zero value of the bending moment at the free edge of the cantilever:

$$M_y(x = 0) = y(x = 0) = 0 \; . \tag{9.11}$$

Calculation of the approximation of the course of the bending moment using Euler's method is based on the recurrence relation (9.4). The exact solution corresponds to the analytically derived equation for the bending moment $M_y(x)$:

$$M_y(x) = -\frac{q_z \cdot x^2}{2} \; . \tag{9.12}$$

For the calculation step $h = 0.5$, the results obtained by using Euler's method are as follows:

```
        x           y           yexac    y-yexac
----------------------------------------
    0.0000     0.0000     0.0000     0.0000
    0.5000     0.0000    -0.5000     0.5000
    1.0000    -1.0000    -2.0000     1.0000
    1.5000    -3.0000    -4.5000     1.5000
    2.0000    -6.0000    -8.0000     2.0000
    2.5000   -10.0000   -12.5000     2.5000
    3.0000   -15.0000   -18.0000     3.0000
    3.5000   -21.0000   -24.5000     3.5000
    4.0000   -28.0000   -32.0000     4.0000
    4.5000   -36.0000   -40.5000     4.5000
    5.0000   -45.0000   -50.0000     5.0000
    5.5000   -55.0000   -60.5000     5.5000
    6.0000   -66.0000   -72.0000     6.0000
```

Figure 9.4 plots the course of the calculated bending moments.

▲

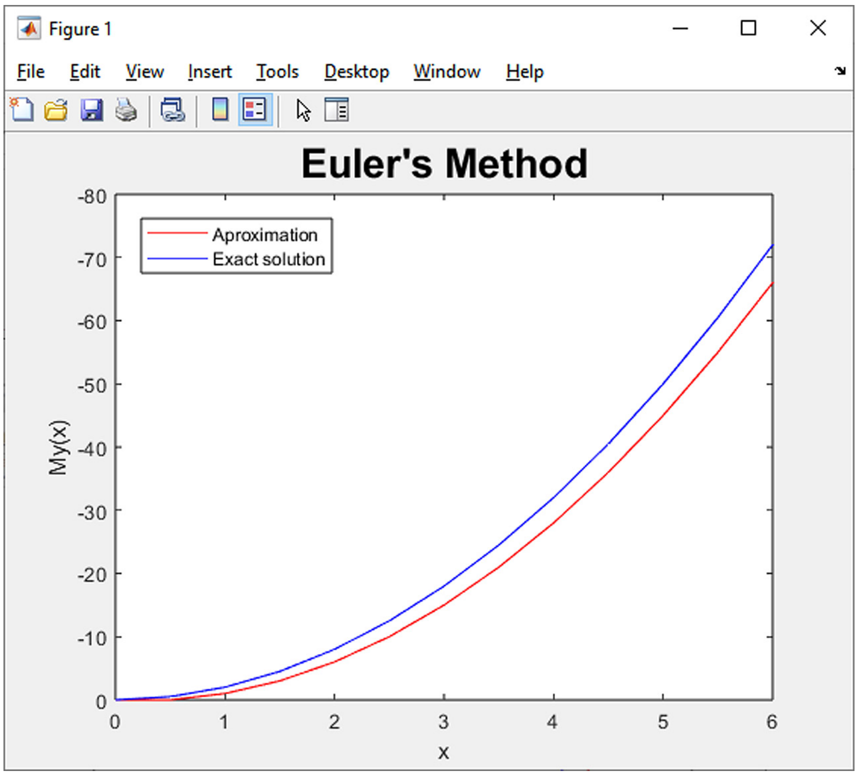


Fig. 9.4 The resulting approximation of the bending moments on the cantilever in Exercise 9.3.

9.1.2 Runge–Kutta Method

Methods based on the Runge–Kutta calculation procedure are suitable for solving ordinary differential equations of the form (9.1). These computing techniques were developed around 1900 by German mathematicians Carl Runge and Martin Wilhelm Kutta. They are expressed by the general recurrence relation:

$$y_{i+1} = y_i + h \cdot \sum_{i=1}^{s} (b_i \cdot k_i) \;, \tag{9.13}$$

where the coefficients k_i are given by the general relation:

$$k_i = f(x_i + c_i \cdot h, y_i + h \cdot \sum_{j=1}^{i-1} (a_{i,j} \cdot k_j)) \;. \tag{9.14}$$

Table 9.2 and Table 9.3 list the coefficients $a_{i,j}$, b_i and c_i for $i, j = 1, \ldots, s$ for Kutta's method of the third order ($s = 3$) and the classic Runge–Kutta method of the fourth order ($s = 4$), respectively.

$c_1 = 0$	$a_{1,1} = 0$	$a_{1,2} = 0$	$a_{1,3} = 0$
$c_2 = \frac{1}{2}$	$a_{2,1} = \frac{1}{2}$	$a_{2,2} = 0$	$a_{2,3} = 0$
$c_3 = 1$	$a_{3,1} = -1$	$a_{3,2} = 2$	$a_{3,3} = 0$
	$b_1 = \frac{1}{6}$	$b_2 = \frac{2}{3}$	$b_3 = \frac{1}{6}$

Tab. 9.2 Coefficients $a_{i,j}$, b_i and c_i of Kutta's method.

$c_1 = 0$	$a_{1,1} = 0$	$a_{1,2} = 0$	$a_{1,3} = 0$	$a_{1,4} = 0$
$c_2 = \frac{1}{2}$	$a_{2,1} = \frac{1}{2}$	$a_{2,2} = 0$	$a_{2,3} = 0$	$a_{2,4} = 0$
$c_3 = \frac{1}{2}$	$a_{3,1} = 0$	$a_{3,2} = \frac{1}{2}$	$a_{3,3} = 0$	$a_{3,4} = 0$
$c_4 = 1$	$a_{4,1} = 0$	$a_{4,2} = 0$	$a_{4,3} = 1$	$a_{4,4} = 0$
	$b_1 = \frac{1}{6}$	$b_2 = \frac{1}{3}$	$b_3 = \frac{1}{3}$	$b_4 = \frac{1}{6}$

Tab. 9.3 Coefficients $a_{i,j}$, b_i and c_i of the classic Runge–Kutta method.

Comment 9.5. Using (9.13) and (9.14), it is possible to express the recurrent expression (9.4) for calculation with Euler's method, which is of order $s = 1$. Table 9.4 lists the corresponding coefficients $a_{i,j}$, b_i and c_i.

$c_1 = 0$	$a_{1,1} = 0$
	$b_1 = 1$

Tab. 9.4 Coefficients $a_{i,j}$, b_i and c_i of Euler's method.

A number of adaptive methods are based on the principle of the Runge–Kutta method, for example the Heun–Euler method ($s = 2$, Table 9.5), the Ralston method ($s = 3$, Table 9.6), and the Bogacki–Shampine method ($s = 4$, Table 9.7).

$c_1 = 0$	$a_{1,1} = 0$	$a_{1,2} = 0$
$c_2 = 1$	$a_{2,1} = 1$	$a_{2,2} = 0$
	$b_1 = 1$	$b_2 = 0$

Tab. 9.5 Coefficients $a_{i,j}$, b_i and c_i of the Heun–Euler method.

$c_1 = 0$	$a_{1,1} = 0$	$a_{1,2} = 0$	$a_{1,3} = 0$
$c_2 = \frac{1}{2}$	$a_{2,1} = \frac{1}{2}$	$a_{2,2} = 0$	$a_{2,3} = 0$
$c_3 = \frac{3}{4}$	$a_{3,1} = 0$	$a_{3,2} = -\frac{3}{4}$	$a_{3,3} = 0$
	$b_1 = \frac{2}{9}$	$b_2 = \frac{3}{9}$	$b_3 = \frac{4}{9}$

Tab. 9.6 Coefficients $a_{i,j}$, b_i and c_i of Ralston's method.

$c_1 = 0$	$a_{1,1} = 0$	$a_{1,2} = 0$	$a_{1,3} = 0$	$a_{1,4} = 0$
$c_2 = \frac{1}{2}$	$a_{2,1} = \frac{1}{2}$	$a_{2,2} = 0$	$a_{2,3} = 0$	$a_{2,4} = 0$
$c_3 = \frac{3}{4}$	$a_{3,1} = 0$	$a_{3,2} = \frac{3}{4}$	$a_{3,3} = 0$	$a_{3,4} = 0$
$c_4 = 1$	$a_{4,1} = \frac{2}{9}$	$a_{4,2} = \frac{1}{3}$	$a_{4,3} = \frac{4}{9}$	$a_{4,4} = 0$
	$b_1 = \frac{7}{24}$	$b_2 = \frac{1}{4}$	$b_3 = \frac{1}{3}$	$b_4 = \frac{1}{8}$

Tab. 9.7 Coefficients $a_{i,j}$, b_i and c_i of the Bogacki–Shampine method.

Example 9.6. Using the Runge–Kutta method, approximate the ordinary differential equation in Exercise 9.1:

$$y'(x) = x^2 - 0.2 \cdot y(x) \ , \tag{9.15}$$

in the interval $\langle -2; 3 \rangle$, with the initial condition $y(-2) = -1$. For h, successively select $h = 1$, $h = 0.5$, $h = 0.1$ or $h = 0.01$. Compare the resulting approximation with the exact solution.

Solution. The relations (9.13) and (9.14), which express the fundamental concept of the Runge–Kutta method, can be applied with the calculation step size $h = 1$ and the values of the coefficients $a_{i,j}$, b_i and c_i from Table 9.3 in the following manner:

```
f=inline('x^2-0.2*y');
a=-2;
b=3;
c=-1;
h=1;
n=(b-a)/h;
x(1)=a; y(1)=c;
for i=1:n
  x(i+1)=x(i)+h;
  K1=h*f(x(i),y(i));
  K2=h*f(x(i)+h/2,y(i)+K1/2);
  K3=h*f(x(i)+h/2,y(i)+K2/2);
  K4=h*f(x(i)+h,y(i)+K3);
  y(i+1)=y(i)+(K1+2*K2+2*K3+K4)/6;
end
[x' y'], plot(x,y,'r');
```

A comparison of the results with the exact solution (Fig. 9.5)

```
     x          y          yexac    y-yexac
----------------------------------------
   -2.0000   -1.0000   -1.0000     0.0000
   -1.0000    1.2508    1.2509    -0.0001
    0.0000    1.3112    1.3113    -0.0001
    1.0000    1.3910    1.3909     0.0001
    2.0000    3.2994    3.2990     0.0004
    3.0000    8.5175    8.5167     0.0008
```

indicates that the accuracy of the calculated approximations is significantly greater than the accuracy achieved with Euler's method.

▲

Example 9.7. Using methods based on the classical Runge–Kutta method, for example Kutta's third-order, Heun–Euler, Ralston or Bogacki–Shampine, approximate the ordinary differential equation given in Exercise 9.1:

$$y'(x) = x^2 - 0.2 \cdot y(x) \, , \tag{9.16}$$

in the interval $\langle -2; 3 \rangle$, with the initial condition $y(-2) = -1$. For the calculation step h, successively select $h = 1$, $h = 0.5$, $h = 0.1$ and $h = 0.01$. Compare the resulting approximation with the exact solution.

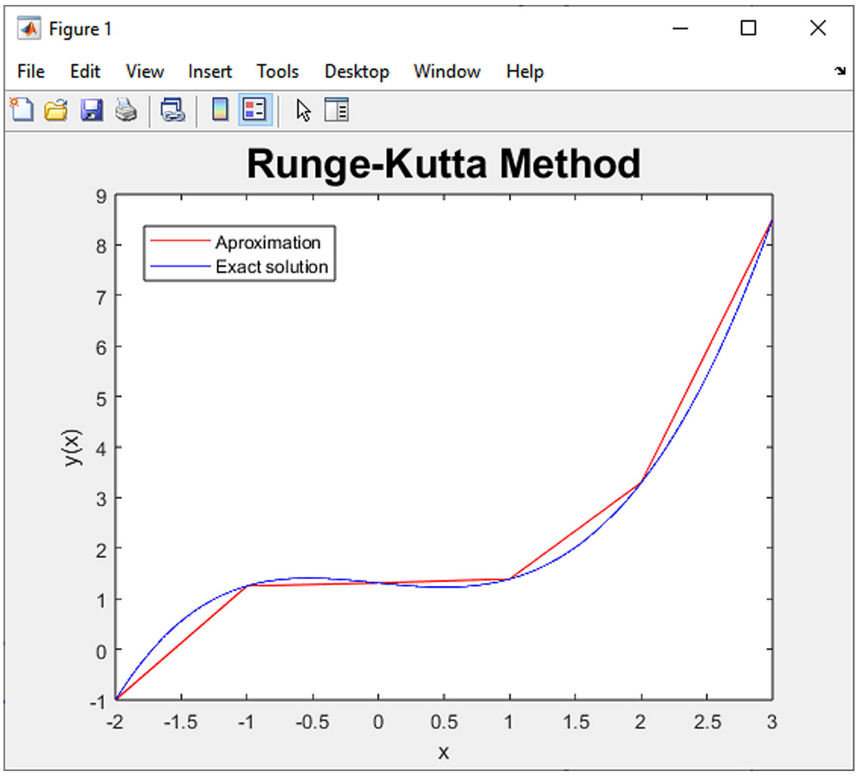


Fig. 9.5 The resulting approximation from the Runge–Kutta method.

Example 9.8. Using the Runge–Kutta method, determine the course of bending moments on the cantilever described in Exercise 9.3. For the calculation step h, select $h = 1$ or $h = 0.5$. Compare the resulting approximation with the exact solution.

Solution. Calculation of the approximation of the course of bending moments using the Runge-Kutta method is based on the recurrence relations (9.13) and (9.14) and the respective values of the coefficients $a_{i,j}$, b_i and c_i listed in Table 9.3.

For the calculation step $h = 0.5$, the Runge–Kutta method generates significantly more accurate results than Euler's method:

```
       x          y         yexac    y-yexac
----------------------------------------
    0.0000     0.0000     0.0000     0.0000
    0.5000    -0.5000    -0.5000     0.0000
    1.0000    -2.0000    -2.0000     0.0000
    1.5000    -4.5000    -4.5000     0.0000
    2.0000    -8.0000    -8.0000     0.0000
    2.5000   -12.5000   -12.5000     0.0000
```

```
   3.0000  -18.0000  -18.0000    0.0000
   3.5000  -24.5000  -24.5000    0.0000
   4.0000  -32.0000  -32.0000    0.0000
   4.5000  -40.5000  -40.5000    0.0000
   5.0000  -50.0000  -50.0000    0.0000
   5.5000  -60.5000  -60.5000    0.0000
   6.0000  -72.0000  -72.0000    0.0000
```

Figure 9.6 plots the results for the calculated bending moments.

▲

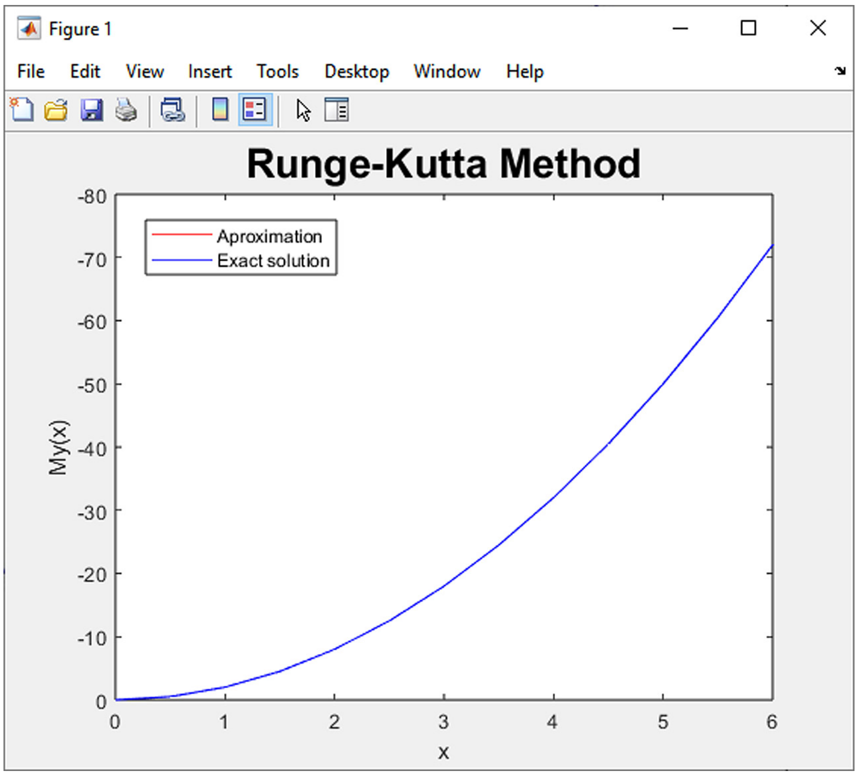


Fig. 9.6 Resulting approximation of the bending moments on the cantilever described in Exercise 9.8.

Example 9.9. Determine the course of bending moments on the cantilever beam from Exercise 9.3 using the other methods based on the classic Runge–Kutta method, i.e., Kutta's third-order method, the Heun–Euler method, the Ralston method and the Bogacki–Shampine method. For the calculation step h, successively select $h = 1$, $h = 0.5$, $h = 0.1$ and $h = 0.01$. Compare the resulting approximation with the exact solution.

9.1.3 Leapfrog Method

The leapfrog method is an example of a two-step method and is based on a recurrent formula:

$$y_{i+1} = y_{i-1} + 2 \cdot h \cdot f(x_i, y_i) \; . \tag{9.17}$$

To calculate the value of y_{i+1}, the values of the function y_i and y_{i-1} at the two previous points must be known. When the calculation begins, both values in the initial section y_0 and y_1 are determined from the initial condition using one of the one-step methods.

The leapfrog method is named thus because it describes the value of the calculated approximation which oscillates around the exact solution.

Example 9.10. Using the leapfrog method, approximate the ordinary differential equation given in Exercise 9.1:

$$y'(x) = x^2 - 0.2 \cdot y(x) \; , \tag{9.18}$$

in the interval $\langle -2; 3 \rangle$, with the initial condition $y(-2) = -1$. To determine the value of the function at the second point of the calculation of $y(-2+h)$, use Euler's methods. For the calculation step h, successively select $h = 1$, $h = 0.5$, $h = 0.1$ and $h = 0.01$. Compare the resulting approximation with the exact solution.

Solution. The leapfrog method is based on the recurrence relation (9.17). An example of a program's code for this method and a calculation step size of $h = 0.5$ is given below:

```
f=inline('x^2-0.2*y');
a=-2; b=3;
c=-1;
h=0.5;
n=(b-a)/h;
x(1)=a; y(1)=c;
x(2)=x(1)+h; y(2)=y(1)+h*f(x(1),y(1));;
for i=2:n
  x(i+1)=x(i)+h;
  y(i+1)=y(i-1)+2*h*f(x(i),y(i));
end
[x' y'], plot(x,y,'r');
```

Figure 9.7 plots the calculated approximation resulting from the leapfrog method for a calculation step of $h = 0.5$. The figure also illustrates the leapfrog method's characteristic calculation procedure, namely oscillation around the exact solution.

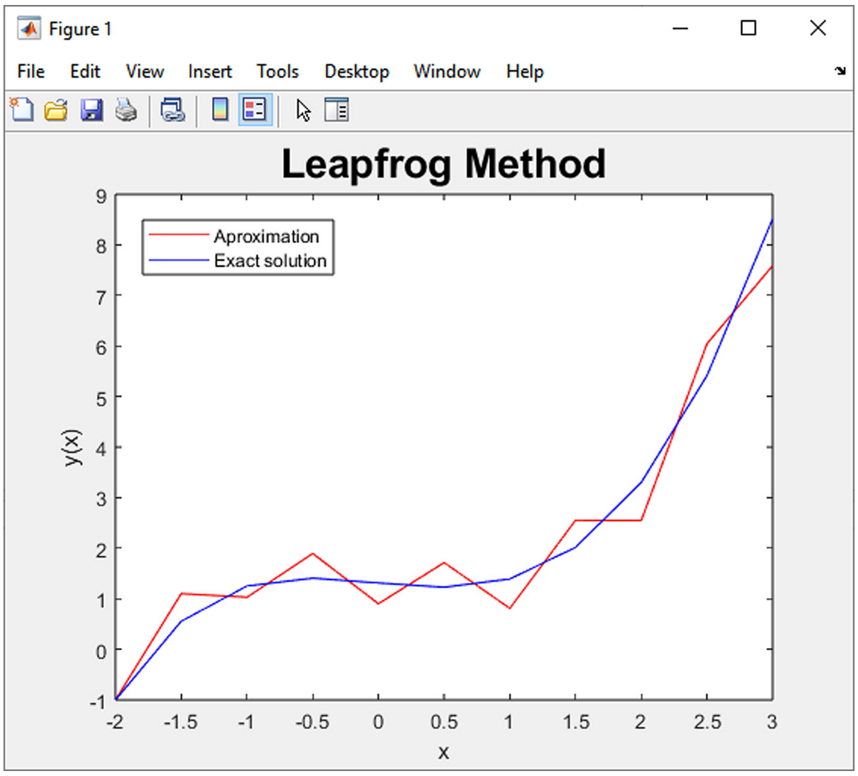


Fig. 9.7 Resulting approximation of the leapfrog method with a computational step of $h = 0.5$.

The numerical results and deviations from the exact solution are summarized below:

```
     x          y          yexac    y-yexac
----------------------------------------
   -2.0000    -1.0000    -1.0000     0.0000
   -1.5000     1.1000     0.5553     0.5447
   -1.0000     1.0300     1.2509    -0.2209
   -0.5000     1.8940     1.4064     0.4876
    0.0000     0.9012     1.3113    -0.4101
    0.5000     1.7138     1.2271     0.4866
    1.0000     0.8084     1.3909    -0.5824
    1.5000     2.5521     2.0169     0.5352
    2.0000     2.5480     3.2990    -0.7509
    2.5000     6.0425     5.4127     0.6298
    3.0000     7.5895     8.5167    -0.9272
```

▲

9.2 Ordinary Differential Equations of the Second Order

As in the case of first-order ordinary differential equations (9.1), we can also solve second-order ordinary differential equations:

$$y''(x) = f(x, y(x), y'(x)) \; . \tag{9.19}$$

There are many types of second-order differential equation; for example, we can solve ordinary differential equations with constant coefficients a,b and c, expressed in the form:

$$a \cdot y''(x) + b \cdot y'(x) + c \cdot y(x) = f(x) \; . \tag{9.20}$$

The numerical solution of equations such as (9.20) is based on conversion into a system of two differential equations:

$$z(x) = y'(x) \tag{9.21}$$

and

$$z'(x) = \frac{f(x) - b \cdot z(x) - c \cdot y(x)}{a} \; . \tag{9.22}$$

Solving this case then requires two initial conditions:

$$y(a) = c \tag{9.23}$$

and

$$z(b) = d \; . \tag{9.24}$$

Example 9.11. Determine the course of the bending moments on the cantilever described in Exercise 9.3 by solving an ordinary differential equation of the second order based on Schwedler's relations:

$$\frac{M_y(x)}{\mathrm{d}x^2} = -q_z(x) = \text{const} \rightarrow y''(x) = -q_z \cdot x^0 \; . \tag{9.25}$$

The initial conditions are based on static boundary conditions, which indicate a zero value of the shear force and the bending moment at the free edge of the cantilever beam, i.e.:

$$V_z(x = 0) = y'(x = 0) = 0 \tag{9.26}$$

and

$$M_y(x = 0) = y(x = 0) = 0 \; . \tag{9.27}$$

Use Euler's method for the numerical solution. As for the calculation step h, choose $h = 1$ or $h = 0.5$. Compare the resulting approximation with the exact solution.

Solution. The entire calculation procedure is clear from the code of the program with the calculation step $h = 0.5$ in a MATLAB `m-file`:

```
f=inline('-4*x^0');
q=4;
a=0;
b=6;
c=0;
d=0;
h=0.5;
n=(b-a)/h;
x(1)=a;
y(1)=c;
y2(1)=d;
for i=1:n
  x(i+1)=x(i)+h;
  y(i+1)=y(i)+h*f(x(i));
  y2(i+1)=y2(i)+h*(y(i));
end
[x' y2']
plot(x,y2,'r');
```

The coding can be improved in the following way (the solved function $y(x)$ and its derivative $y'(x)$ are stored in one variable):

```
f=inline('-4*x^0');
q=4;
a=0;
b=6;
c=0;
d=0;
h=0.5;
x=a:h:b;
f=inline('-4*x^0');
y(1,:)=[d c];
for i=2:length(x)
   K1=[y(i-1,2) -q];
   y(i,:)=y(i-1,:)+K1*h;
end
[x' y(:,1)]
plot(x,y(:,1),'r');
```

For the calculation step $h = 0.5$, the following results can be obtained using Euler's method:

```
      x          y          yexac    y-yexac
----------------------------------------
    0.0000     0.0000     0.0000     0.0000
    0.5000     0.0000    -0.5000     0.5000
    1.0000    -1.0000    -2.0000     1.0000
    1.5000    -3.0000    -4.5000     1.5000
    2.0000    -6.0000    -8.0000     2.0000
    2.5000   -10.0000   -12.5000     2.5000
    3.0000   -15.0000   -18.0000     3.0000
    3.5000   -21.0000   -24.5000     3.5000
    4.0000   -28.0000   -32.0000     4.0000
    4.5000   -36.0000   -40.5000     4.5000
    5.0000   -45.0000   -50.0000     5.0000
    5.5000   -55.0000   -60.5000     5.5000
    6.0000   -66.0000   -72.0000     6.0000
```

▲

Example 9.12. Solve the differential equation (9.25) from Exercise 9.11 to determine the course of bending moments using the classic Runge–Kutta method.

Solution. The entire calculation procedure is again evident from the code of the program implementing a calculation step of $h = 0.5$ in a MATLAB `m-file`:

```
f=inline('-4*x^0');
q=4;
a=0;
b=6;
c=0;
d=0;
h=0.5;
n=(b-a)/h;
x(1)=a;
y(1)=c;
y2(1)=d;
for i=1:n
  K1=f(x(i));
  K2=f(x(i)+h/2);
  K3=f(x(i)+h/2);
  K4=f(x(i)+h);
  y(i+1)=y(i)+h*((K1+K4)/6+(K2+K3)/3);
  K1=y(i);
  K2=(y(i)+y(i+1))/2;
  K3=(y(i)+y(i+1))/2;
  K4=y(i+1);
  y2(i+1)=y2(i)+h*((K1+K4)/6+(K2+K3)/3);
  x(i+1)=x(i)+h;
end
[x' y2']
plot(x,y2,'r');
```

Even in this case, the coding can be improved by storing the solved function $y(x)$ and its derivative $y'(x)$ in a single variable:

```
f=inline('-4*x^0');
q=4;
a=0;
b=6;
c=0;
d=0;
h=0.5;
x=a:h:b;
f=inline('-4*x^0');
y(1,:)=[d c];
for i=2:length(x)
   K1=f(x(i-1));
   K2=f(x(i-1)+h/2);
   K3=f(x(i-1)+h/2);
   K4=f(x(i-1)+h);
   y(i,2)=y(i-1,2)+h*((K1+K4)/6+(K2+K3)/3);
   K1=y(i-1,2);
   K2=(y(i-1,2)+y(i,2))/2;
   K3=(y(i-1,2)+y(i,2))/2;
   K4=y(i,2);
   y(i,1)=y(i-1,1)+h*((K1+K4)/6+(K2+K3)/3);
end
[x' y(:,1)]
plot(x,y(:,1),'r');
```

For the calculation step $h = 0.5$, the Runge–Kutta method can obtain significantly more accurate results than Euler's method:

```
     x          y         yexac    y-yexac
----------------------------------------
    0.0000     0.0000     0.0000    0.0000
    0.5000    -0.5000    -0.5000    0.0000
    1.0000    -2.0000    -2.0000    0.0000
    1.5000    -4.5000    -4.5000    0.0000
    2.0000    -8.0000    -8.0000    0.0000
    2.5000   -12.5000   -12.5000    0.0000
    3.0000   -18.0000   -18.0000    0.0000
    3.5000   -24.5000   -24.5000    0.0000
    4.0000   -32.0000   -32.0000    0.0000
    4.5000   -40.5000   -40.5000    0.0000
    5.0000   -50.0000   -50.0000    0.0000
    5.5000   -60.5000   -60.5000    0.0000
    6.0000   -72.0000   -72.0000    0.0000
```

▲

Example 9.13. Solve the differential equation (9.25) from Exercise 9.11 for determining the course of bending moments using other methods that are based on the classic Runge–Kutta method, i.e. the Kutta method of the third order, the Heun–Euler method, and the Ralston and Bogacki–Shampine methods.

Example 9.14. Determine the shape of the deflection curve of the cantilever beam schematically shown in Fig. 9.8. Specific input data are given in Table 9.8. Use Euler's method for the numerical solution. For the calculation step h, choose $h = 0.5$, $h = 0.25$, $h = 0.1$, or $h = 0.01$. Compare the resulting approximation with the exact solution.

Continuous force loading q_z :	4 kN/m
Cantilever beam span l :	3 m
Width of the rectangular cross-section b :	0.02 m
Height of the rectangular cross-section h :	0.15 m
Moment of inertia I_y :	$\frac{1}{12} \cdot 0.02 \cdot 0.15^3 = 5.625 \cdot 10^{-6}$ m^4
Modulus of elasticity in tension and compression E :	$2.1 \cdot 10^{11}$ Pa

Tab. 9.8 Input data from Exercise 9.14.

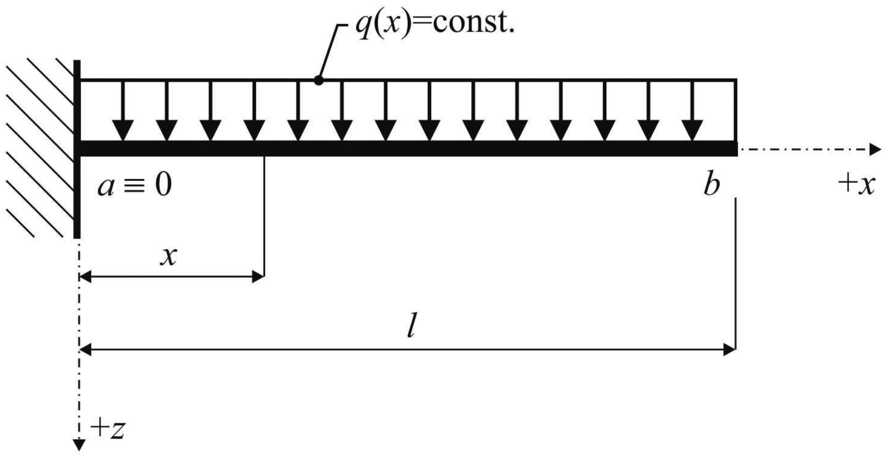


Fig. 9.8 Static diagram of solved statically determined cantilever beam.

Solution. A second-order ordinary differential equation has the following form:

$$EI_y w_z(x)'' = -M_y(x)\ , \tag{9.28}$$

where EI_y is the bending stiffness of the beam (constant and non-zero).

The initial conditions are based on the deformation boundary conditions, which indicate a zero value of deflection and rotation in the cantilever beam, i.e.:

$$\varphi_y(x=0) = w_z'(x=0) = 0 \tag{9.29}$$

and

$$w_z(x=0) = y(x=0) = 0\ . \tag{9.30}$$

From the equilibrium conditions, the size of the force reaction in the cantilever can be determined first:

$$R_{a,z} = q_z \cdot l\,(\uparrow)\ , \tag{9.31}$$

moment reaction in the cantilever beam:

$$M_{a,y} = \frac{q_z \cdot l^2}{2}\,(\circlearrowleft)\ , \tag{9.32}$$

and finally the bending moment equation itself:

$$M_y(x) = -\frac{q_z \cdot l^2}{2} + q_z \cdot l \cdot x - \frac{q_z \cdot x^2}{2} = q_z \cdot \left(-\frac{l^2}{2} + l \cdot x - \frac{x^2}{2}\right)\ , \tag{9.33}$$

which appears in the solved differential equation (9.28).

The calculation via the Euler method can then be performed for a step of $h = 0.25$ using the `horner` function (see chapter 3.1) and the following sequence of commands:

```
qz=4000;
l=3;
E=2.1*10^11;
width=0.02;
height=0.15;
Iy=1/12*width*height^3;
M=[-qz/2*l^2 qz*l -qz/2];
a=0;
b=l;
c=0;
d=0;
h=0.25;
n=(b-a)/h;
x(1)=a;
y(1)=c;
y2(1)=d;
for i=1:n
  x(i+1)=x(i)+h;
  y(i+1)=y(i)-h*horner(2,M,x(i+1))/(E*Iy);
  y2(i+1)=y2(i)+h*y(i)*1000;
end
[x' y2']
plot(x,y2,'r');
```

The achieved results can be compared with the exact value based on the analytically determined equation of the deflection curve:

$$w_z(x) = \frac{q_z}{EI_y} \cdot \left(\frac{l^2 \cdot x^2}{4} - \frac{l \cdot x^3}{6} + \frac{x^4}{24} \right) . \tag{9.34}$$

For the calculation step $h = 0.25$, the following numerical values of the resulting solution are obtained:

```
        x          y1        y1exac      y[mm]   yexac[mm]   y-yexac
------------------------------------------------------------------------
     0.0000     0.0000     0.0000     0.0000     0.0000     0.0000
     0.2500     0.0032     0.0035     0.0000     0.4503    -0.4503
     0.5000     0.0058     0.0064     0.8003     1.7019    -0.9017
     0.7500     0.0080     0.0088     2.2619     3.6161    -1.3542
     1.0000     0.0097     0.0107     4.2593     6.0670    -1.8078
     1.2500     0.0110     0.0122     6.6799     8.9424    -2.2625
     1.5000     0.0119     0.0133     9.4246    12.1429    -2.7183
```

```
    1.7500    0.0126    0.0141   12.4074   15.5826   -3.1752
    2.0000    0.0130    0.0147   15.5556   19.1887   -3.6332
    2.2500    0.0133    0.0150   18.8095   22.9018   -4.0923
    2.5000    0.0134    0.0152   22.1230   26.6755   -4.5525
    2.7500    0.0134    0.0152   25.4630   30.4767   -5.0138
    3.0000    0.0134    0.0152   28.8095   34.2857   -5.4762
```

Figure 9.9 shows the calculated approximation of the deflection curve of the cantilever beam determined by Euler's method for a calculation step of $h = 0.25$.

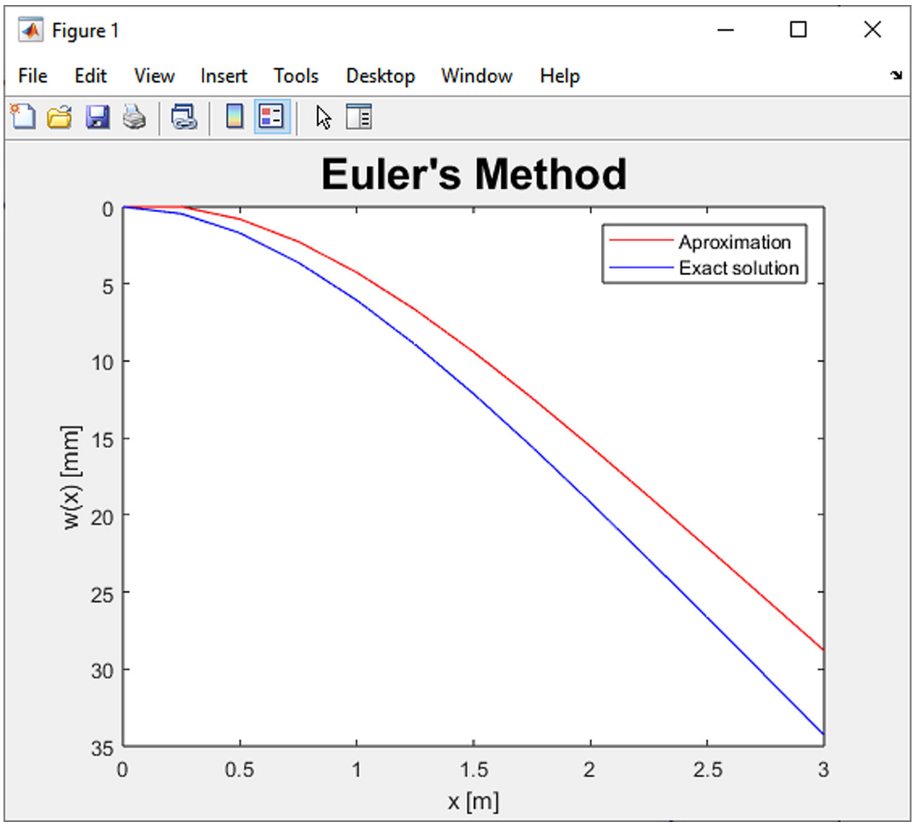


Fig. 9.9 The resulting approximation of the deflection curve of the cantilever beam with a calculation step of $h = 0.25$.

▲

Example 9.15. Approximate the deflection curve of the cantilever beam from Exercise 9.14 using the Runge–Kutta method.

Solution. The calculation of the deflection curve approximation can be done by the following `m-script`:

```
qz=4000;
l=3;
E=2.1*10^11;
width=0.02;
height=0.15;
Iy=1/12*width*height^3;
M=[-qz/2*l^2 qz*l -qz/2];
a=0;
b=l;
h=0.25;
n=(b-a)/h;
c=0;
d=0;
x(1)=a;
y(1)=c;
y2(1)=d;
for i=1:n
  x(i+1)=x(i)+h;
  K1=horner(2,M,x(i))/(E*Iy);
  K2=horner(2,M,x(i)+h/2)/(E*Iy);
  K3=horner(2,M,x(i)+h/2)/(E*Iy);
  K4=horner(2,M,x(i)+h)/(E*Iy);
  y(i+1)=y(i)-h*((K1+K4)/6+(K2+K3)/3);
  K1=y(i)*1000;
  K2=(y(i)+y(i+1))*1000/2;
  K3=(y(i)+y(i+1))*1000/2;
  K4=y(i+1)*1000;
  y2(i+1)=y2(i)+h*((K1+K4)/6+(K2+K3)/3);
end
[x' y2']
plot(x,y2,'r');
```

For the calculation step $h = 0.5$, the Runge–Kutta method can obtain significantly more accurate results than the Euler's method:

```
     x         y1       y1exac     y[mm]  yexac[mm]  y-yexac
------------------------------------------------------------
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
    0.2500    0.0035    0.0035    0.4376    0.4503   -0.0127
    0.5000    0.0064    0.0064    1.6777    1.7019   -0.0243
    0.7500    0.0088    0.0088    3.5813    3.6161   -0.0347
```

```
   1.0000    0.0107    0.0107    6.0229    6.0670   -0.0441
   1.2500    0.0122    0.0122    8.8900    8.9424   -0.0524
   1.5000    0.0133    0.0133   12.0833   12.1429   -0.0595
   1.7500    0.0141    0.0141   15.5170   15.5826   -0.0656
   2.0000    0.0147    0.0147   19.1182   19.1887   -0.0705
   2.2500    0.0150    0.0150   22.8274   22.9018   -0.0744
   2.5000    0.0152    0.0152   26.5983   26.6755   -0.0772
   2.7500    0.0152    0.0152   30.3979   30.4767   -0.0788
   3.0000    0.0152    0.0152   34.2063   34.2857   -0.0794
```

▲

Example 9.16. Approximate the deflection curve of the cantilever beam from Exercise 9.14 using the other methods based on the classic Runge–Kutta method, i.e. Kutt's third-order method, the Heun–Euler method, the Ralston method and the Bogacki–Shampine method.

Example 9.17. Approximate the following ordinary differential equation of the second order:

$$y''(t) + 100 \cdot y'(t) + 10000 \cdot y(t) = 10000 \cdot |\sin(377 \cdot t)| \tag{9.35}$$

in the interval $\langle 0; 0.08\rangle$ with initial conditions $y(0) = 0$ and $y'(0) = 0$. For the numerical solution, use Euler's method and the classical Runge–Kutta method of the 4th order. As for the calculation step h, successively choose $h = 0.01$, $h = 0.0025$ and $h = 0.0001$. Compare the resulting approximation with the solution using the `ode45` function of MATLAB.

Solution. MATLAB does not have functions for solving ordinary differential equations of the second order. The solution is based on converting the problem into a system of two differential equations described by the relations (9.21) and (9.22). In the case of the solved problem, the given decomposition is performed using a separate `m-function`, such as:

```
function y=funct(t,y);
y=[y(2);-100*y(2)-10000*y(1)+10000*abs(sin(377*t))];
```

which in `m-script` can be referred to for example as follows:

```
t0=[0 0];
interval=[0,0.08];
options = odeset('AbsTol',1e-9);
[t,y]=ode45(@funct,interval,t0,options)
plot(t,y(:,1))
```

A program that performs the calculation of the solved differential equation of the 2nd order by Euler's method, the classic Runge–Kutta method of the 4th order and the `ode45` function of MATLAB for the calculation step $h = 0.0025$ could for instance look as follows:

```
a=0;
b=0.08;
c=0;
d=0;
h=0.0025;
n=(b-a)/h;
x(1)=a;
ye(1)=c;
yrk(1)=c;
y2e(1)=d;
y2rk(1)=d;
for i=1:n
  x(i+1)=x(i)+h;
  ye(i+1)=ye(i)+h*(1E4*abs(sin(377*x(i)))-100*ye(i)-1E4*y2e(i));
  y2e(i+1)=y2e(i)+h*(ye(i));
  K1=h*(1E4*abs(sin(377*x(i)))-100*yrk(i)-1E4*y2rk(i));
  K2=h*(1E4*abs(sin(377*(x(i)+h/2)))-100*(yrk(i)+K1/2)-1E4*y2rk(i));
  K3=h*(1E4*abs(sin(377*(x(i)+h/2)))-100*(yrk(i)+K2/2)-1E4*y2rk(i));
  K4=h*(1E4*abs(sin(377*(x(i)+h)))-100*(yrk(i)+K3)-1E4*y2rk(i));
  yrk(i+1)=yrk(i)+(K1+2*K2+2*K3+K4)/6;
  K1=yrk(i);
  K2=(yrk(i)+yrk(i+1))/2;
  K3=(yrk(i)+yrk(i+1))/2;
  K4=yrk(i+1);
  y2rk(i+1)=y2rk(i)+h*((K1+K4)/6+(K2+K3)/3);
end
t0=[0 0];
interval=[0,0.08];
options=odeset('AbsTol',1e-9);
[xm,y2m]=ode45(@funct,interval,t0,options);
[x' y2e' y2rk']
plot(x,y2e,'r',x,y2rk,'b',xm,y2m(:,1),'k:');
```

The numerical results obtained by Euler's method and the Runge–Kutta method for a calculation step of $h = 0.0025$ are as follows:

```
     x          y_eul     y_rk
------------------------------
    0.0000     0.0000     0.0000
    0.0025     0.0000     0.0126
    0.0050     0.0000     0.0610
    0.0075     0.0506     0.1415
    0.0100     0.1479     0.2242
    0.0125     0.2371     0.3098
    0.0150     0.3315     0.4091
    0.0175     0.4499     0.4969
    0.0200     0.5548     0.5663
    0.0225     0.6246     0.6364
    0.0250     0.7018     0.6956
    0.0275     0.7712     0.7291
    0.0300     0.7794     0.7547
    0.0325     0.7879     0.7785
    0.0350     0.8050     0.7803
    0.0375     0.7879     0.7698
    0.0400     0.7615     0.7662
    0.0425     0.7549     0.7513
    0.0450     0.7392     0.7238
    0.0475     0.6995     0.7071
    0.0500     0.6830     0.6923
    0.0525     0.6774     0.6663
    0.0550     0.6306     0.6475
    0.0575     0.6037     0.6413
    0.0600     0.6036     0.6266
    0.0625     0.5850     0.6112
    0.0650     0.5702     0.6125
    0.0675     0.5849     0.6100
    0.0700     0.5971     0.6003
    0.0725     0.5890     0.6048
    0.0750     0.6051     0.6128
    0.0775     0.6308     0.6095
    0.0800     0.6124     0.6122
```

Figure 9.10 shows approximations calculated by Euler's method, the Runge–Kutta method and the `ode45` function of MATLAB for a calculation step of $h =$ $= 0.0025$.

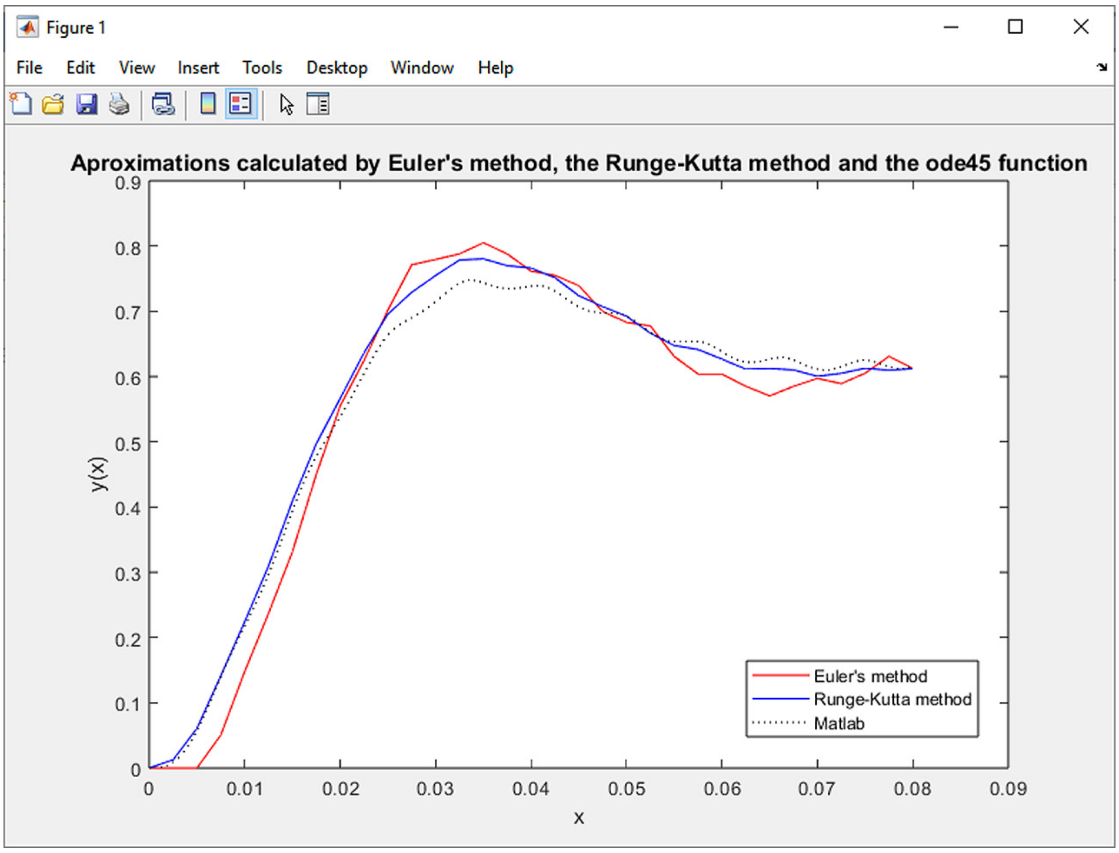


Fig. 9.10 Comparison of the approximations achieved by the Euler's method, the Runge–Kutta method and the `ode45` function for a calculation step of $h = 0.0025$.

▲

Chapter 10

Interpolation and Approximation

Objectives

The aim of this chapter is to:

- explain the concepts of interpolation and approximation,
- apply basic algorithms for interpolation and approximation in engineering tasks.

Examples of **interpolation** tasks include:

- given a function f_x, find the polynomial $\Phi_n(x)$ of the nth degree which equals x_k for $n+1$ arguments, where $k = 0, 1, \ldots, n$ is receives the same values as the function f_x.

- given the function table of f_x compiled for $x = x_k$, approximate the values of f_x using the polynomial $\Phi_n(x)$ for points x which are different from the nodal points x_i.

If the given values of the function y_i $(i = 0, 1, \ldots, n)$ at the nodal points x_0, x_1 až x_n are not given exactly (and are obtained, e.g., by measurements which are always imprecise to some degree), it is not important that the sought function coincides with the function exactly at the nodal points, as in the case of interpolation. The task of **approximation** is therefore to find a simpler and mathematically precisely defined continuous approximation function F_x in the interval $\langle a, b\rangle$ which would best fit the empirical points $x_0, x_1, \ldots, x_n$.

10.1 Linear Interpolation

Linear interpolation makes it possible to replace the course of a function between two points with coordinates x_k, y_k and x_{k+1}, y_{k+1} by a segment defined by the equation of a straight line:

$$\frac{y(x) - y_k}{x - x_k} = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} . \qquad (10.1)$$

After adjusting (10.1), an equation which defines $y(x)$ based on the parameter x can be obtained:

$$y(x) = \frac{y_k \cdot (x - x_{k+1}) - y_{k+1} \cdot (x - x_k)}{x_k - x_{k+1}} . \qquad (10.2)$$

Example 10.1. Using linear interpolation for two points with coordinates $[x_0, y_0] = [1, 1.8]$ and $[x_1, y_1] = [2, 2.27]$, determine the value of the interpolation function $y(x = 1.5)$.

Solution. A function for linear interpolation in the file `lin_interpol.m` could look, e.g., as follows:

```
function y=lin_interpol(x,xy2)
y=(xy2(1,2)*(x-xy2(2,1))-xy2(2,2)*(x-xy2(1,1)))/(xy2(1,1)-xy2(2,1));
```

When calling the function `y=lin_interpol(x,coords_xy_2p)` with parameters `x=1.5` and `coords_xy_2p=[1 1.8; 2 2.27]`, we obtain the following result:

```
y =
    2.0350
```

This can also be represented graphically – see Fig. 10.1.

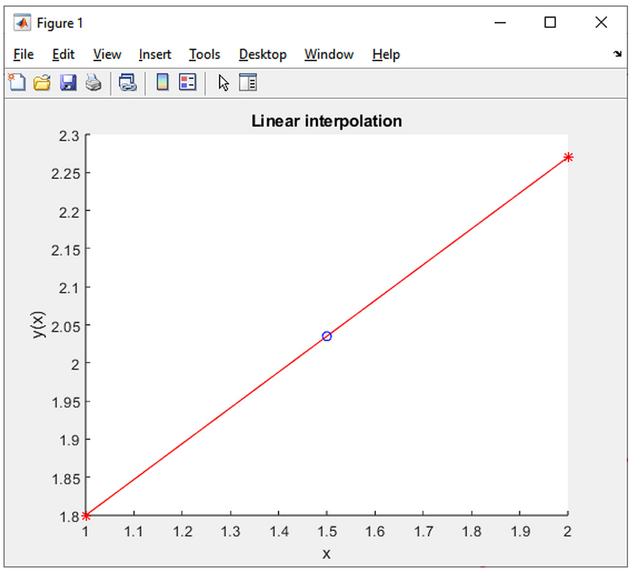


Fig. 10.1 Resulting linear interpolation for the point with coordinate $x = 1.5$, marked with a circle.

▲

10.2 Lagrange Interpolation

If $f(x)$ is a real function defined for the interval $\langle a, b\rangle$, we can also consider the following function:

$$\Phi(x) = a_0 \cdot \varphi_0(x) + a_1 \cdot \varphi_1(x) + a_2 \cdot \varphi_2(x) + \ldots + a_i \cdot \varphi_i(x) + \ldots + a_n \cdot \varphi_n(x) \ , \tag{10.3}$$

where a_i are real coefficients and $\varphi_i(x)$ is equal to x^i for $i = 0, 1, \ldots, n$. The solution is then the interpolation polynomial $\Phi(x)$ for which it holds:

$$\Phi(x_i) = f(x_i) \ , \tag{10.4}$$

where x_i is in the interval $\langle a, b\rangle$ for $i = 0, 1, 2, \ldots, n$. This means that the sought function of the interpolation polynomial $\Phi(x)$ should attain identical values as the given function $f(x)$ for $n + 1$ input parameters $x_0, x_1, x_2, \ldots, x_n$.

This problem can be solved, for example, by successively substituting $x = x_i$, $i = 0, 1, 2, \ldots, n$ into the equation (10.4), thus obtaining a system of $n + 1$ linear equations with unknown coefficients a_i:

$$\begin{array}{ccccccccc} a_0 \cdot \varphi(x_0) & + & a_1 \cdot \varphi(x_0) & + & \ldots & + & a_n \cdot \varphi(x_0) & = & f(x_0) \\ a_0 \cdot \varphi(x_1) & + & a_1 \cdot \varphi(x_1) & + & \ldots & + & a_n \cdot \varphi(x_1) & = & f(x_1) \\ & & & & \vdots & & & & \\ a_0 \cdot \varphi(x_n) & + & a_1 \cdot \varphi(x_n) & + & \ldots & + & a_n \cdot \varphi(x_n) & = & f(x_n) \end{array} \ . \tag{10.5}$$

One of the ways to avoid solving the above-mentioned system of linear equations (10.5) when determining the interpolation polynomial $\Phi(x_i)$ is the Lagrange method.

Comment 10.2. Although the method is named after Joseph Louis Lagrange, who published it in 1795, it was first discovered in 1779 by Edward Waring and its implications partially published in 1783 by Leonhard Euler.

Given $n+1$ different nodal points $x_0, x_1, x_2, \ldots, x_n$ in the interval $\langle a, b\rangle$ and the function values $y_i = f(x_i)$ for $i = 0, 1, \ldots, n$, then an interpolation polynomial of degree at most n can be constructed, for which it will hold:

$$\Phi_n(x) = P_0(x) + P_1(x) + \ldots + P_n(x) = y_0 \cdot l_0(x) + y_1 \cdot l_1(x) + \ldots y_n \cdot l_n(x) \ . \tag{10.6}$$

For $l_i(x)$ it holds:

$$l_i(x_j) = \begin{cases} 1 & \text{pro } i = j \\ 0 & \text{pro } i \neq j \end{cases} \ . \tag{10.7}$$

The polynomial satisfies this condition:

$$\begin{aligned} l_i(x) = &\prod_{\substack{j=0\\ j\neq i}}^{n} \frac{x-x_j}{x_i-x_j} = \\ &= \frac{x-x_0}{x_i-x_0}\cdot\frac{x-x_1}{x_i-x_1}\cdot\ldots\cdot\frac{x-x_{i-1}}{x_i-x_{i-1}}\cdot\frac{x-x_{i+1}}{x_i-x_{i+1}}\cdot\ldots\cdot\frac{x-x_n}{x_i-x_n}\,. \end{aligned} \tag{10.8}$$

The resulting form of the Lagrange interpolation polynomial is then:

$$\begin{aligned} L_n(x) = &\, y_0\cdot\left(\frac{x-x_1}{x_0-x_1}\cdot\frac{x-x_2}{x_0-x_2}\cdot\ldots\cdot\frac{x-x_n}{x_0-x_n}\right)+ \\ &+ y_1\cdot\left(\frac{x-x_0}{x_1-x_0}\cdot\frac{x-x_2}{x_1-x_2}\cdot\ldots\cdot\frac{x-x_n}{x_1-x_n}\right)+\ldots \\ &+ y_i\cdot\left(\frac{x-x_0}{x_i-x_0}\cdot\frac{x-x_1}{x_i-x_1}\cdot\ldots\cdot\frac{x-x_{i-1}}{x_i-x_{i-1}}\cdot\frac{x-x_{i+1}}{x_i-x_{i+1}}\cdot\ldots\cdot\frac{x-x_n}{x_i-x_n}\right)+\ldots \\ &+ y_n\cdot\left(\frac{x-x_0}{x_n-x_0}\cdot\frac{x-x_1}{x_n-x_1}\cdot\ldots\cdot\frac{x-x_{n-1}}{x_n-x_{n-1}}\right)\,. \end{aligned} \tag{10.9}$$

A function that determines for the specified point with coordinate x in the input parameter `par` the value of the Lagrange interpolation polynomial, compiled for the specified set of points with coordinates x and y stored in the input parameters with vectors `x` and `y`, can be programmed in MATLAB, e.g., using the `lagrange.m` script:

```
function s=lagrange(x,y,par)
n=length(x);
s=0;
for i=1:n
  m=y(i);
  for j=1:n
    if ~(i==j)
      m=m*(par-x(j))/(x(i)-x(j));
    end
  end
  s=s+m;
end
```

Example 10.3. Use the Lagrange interpolation polynomial for three points with coordinates $[x_0, y_0] = [0, 1]$, $[x_1, y_1] = [2, 2]$ and $[x_2, y_2] = [3, 4]$ to determine the equation of the interpolation function $y(x)$.

Solution. The given example can be solved in general by substituting the specified coordinates of three points into the general equation of the interpolation polynomial (10.9):

$$\begin{aligned} L_2(x) &= y_0 \cdot \frac{(x-x_1)\cdot(x-x_2)}{(x_0-x_1)\cdot(x_0-x_2)} + y_1 \cdot \frac{(x-x_0)\cdot(x-x_2)}{(x_1-x_0)\cdot(x_1-x_2)} + \\ &+ y_2 \cdot \frac{(x-x_0)\cdot(x-x_1)}{(x_2-x_0)\cdot(x_2-x_1)} = \\ &= 1 \cdot \frac{(x-2)\cdot(x-3)}{(0-2)\cdot(0-3)} + 2 \cdot \frac{(x-0)\cdot(x-3)}{(2-0)\cdot(2-3)} + 4 \cdot \frac{(x-0)\cdot(x-2)}{(3-0)\cdot(3-2)} = \\ &= \frac{1}{6}\cdot(x^2-5\cdot x+6) + 2 \cdot -\frac{1}{2}\cdot(x^2-3\cdot x) + 4 \cdot \frac{1}{3}\cdot(x^2-2\cdot x) = \\ &= \frac{1}{2}\cdot x^2 - \frac{1}{2}\cdot x + 1 \,. \end{aligned} \tag{10.10}$$

The correctness of the derived interpolation polynomial can be verified by substituting the coordinates of the specified points:

$$L_2(x_0) = \frac{1}{2}\cdot x_0^2 - \frac{1}{2}\cdot x_0 + 1 = \frac{1}{2}\cdot 0^2 - \frac{1}{2}\cdot 0 + 1 = 1 \,, \tag{10.11}$$

$$L_2(x_1) = \frac{1}{2}\cdot x_1^2 - \frac{1}{2}\cdot x_1 + 1 = \frac{1}{2}\cdot 2^2 - \frac{1}{2}\cdot 2 + 1 = 2 \,, \tag{10.12}$$

$$L_2(x_2) = \frac{1}{2}\cdot x_2^2 - \frac{1}{2}\cdot x_2 + 1 = \frac{1}{2}\cdot 3^2 - \frac{1}{2}\cdot 3 + 1 = 4 \,. \tag{10.13}$$

The constructed Lagrange interpolation polynomial can also be displayed graphically – see Fig. 10.2.

▲

Example 10.4. Using the Lagrange interpolation polynomial, determine the value of the bending moment of the structure described in Exercise 3.1 for the point at the coordinate $x = l/5 = 1.2$ m. To construct the Lagrange interpolation polynomial, use the values of the actual bending moments at three points with coordinates $[x_0, x_1, x_2] = [0, l/2, l] = [0, 3, 6]$ m.

Solution. First, of course, it is necessary to determine the values of the actual bending moments at the specified points, which for the given specification take on the values $M_y(x_0 = 0) = 0$ kNm, $M_y(x_1 = 3) = 9$ kNm and $M_y(x_2 = 6) = -18$ kNm. The creation of the Lagrange interpolation polynomial is possible with the already created and previously described script `lagrange.m`. The entire calculation can be performed, for example, by the following sequence of commands:

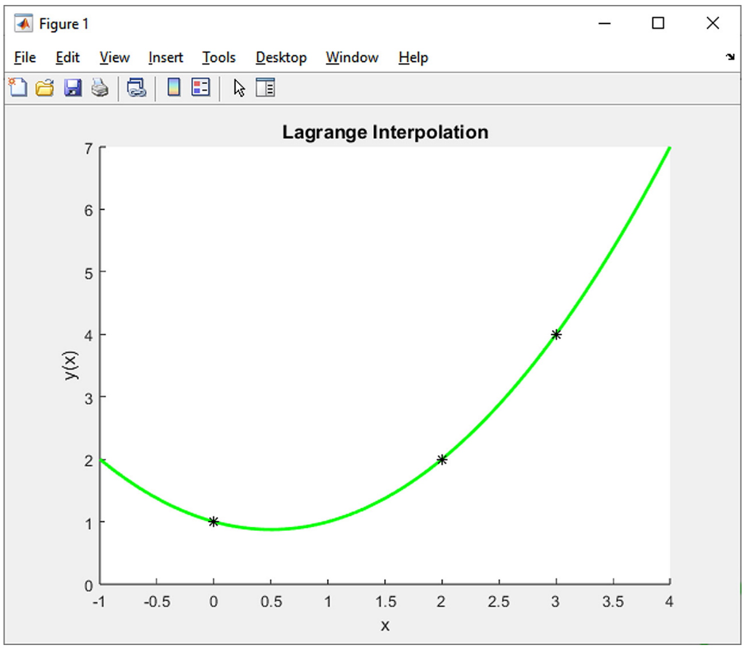


Fig. 10.2 The resulting interpolation using the Lagrange interpolation polynomial for points on coordinates $[x_0, y_0] = [0, 1]$, $[x_1, y_1] = [2, 2]$ and $[x_2, y_2] = [3, 4]$.

```
clc;
clear;
format short;
qz=4000;
l=6;
M=[0 3/8*qz*l -qz/2]/1000;
x=[0 l/2 l]; % 0 m, 3 m, 6 m
y=[horner(2,M,x(1)) horner(2,M,x(2)) horner(2,M,x(3))];
par=l/5;
res=lagrange(x,y,par)
```

The result of the entire solution is then the value at the point with the coordinate $x = l/5 = 1.2$ m:

```
res =
    7.9200
```

Due to the fact that the resulting Lagrange interpolation polynomial forms a 2nd degree polynomial just like the course of the bending moments, one can observe a complete match between this pair of functions – see Fig. 10.3.

▲

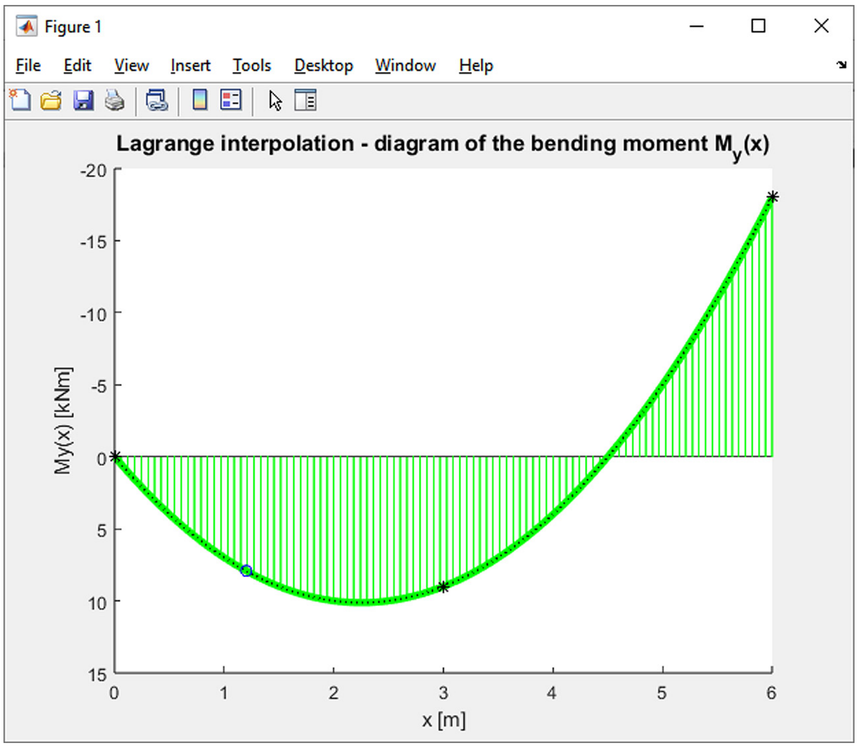


Fig. 10.3 The resulting interpolation of the course of bending moments on the beam from Exercise 3.1 using the Lagrange interpolation polynomial for the points $M_y(x_0 = 0) = 0$ kNm, $M_y(x_1 = 3) = 9$ kNm and $M_y(x_2 = 6) = -18$ kNm marked with an asterisk. The value of the bending moment at the specified point with the coordinate $x = l/5 = 1.2$ m is then indicated by a circle.

10.3 Newton's Interpolation

The interpolation polynomial $\Phi_n(x)$ can also be expressed as a function containing differences:

$$\begin{aligned}\Phi_n(x) = a_0 + a_1 \cdot (x - x_0) + a_2 \cdot (x - x_0) \cdot (x - x_1) + \\ + \ldots + a_n \cdot (x - x_0) \cdot (x - x_1) \cdot \ \ldots \ \cdot (x - x_n)\ .\end{aligned} \tag{10.14}$$

The desired interpolation polynomial $\Phi_n(x)$ must take values at points x_i:

$$\Phi_n(x_i) = f(x_i) \text{ pro } i = 0, 1, \ldots, n\ . \tag{10.15}$$

The solution to the problem is to determine the unknown coefficients a_k for $k = = 0, 1, \ldots, n$, which are contained in the relation (10.14). By successively substituting $x = x_i$ for $i = 0, 1, \ldots, n$ into the polynomial $\Phi_n(x)$ from (10.14) the following conditions can be obtained:

$$\begin{aligned}
\Phi_0(x_0) &= f(x_0) = a_0 \\
\Phi_1(x_1) &= f(x_1) = a_0 + a_1 \cdot (x_1 - x_0) \\
\Phi_2(x_2) &= f(x_2) = a_0 + a_1 \cdot (x_1 - x_0) + a_2 \cdot (x_1 - x_0) \cdot (x_2 - x_0) \\
&\ \vdots \\
\Phi_n(x_n) &= f(x_n) = a_0 + a_1 \cdot (x_1 - x_0) + a_2 \cdot (x_1 - x_0) \cdot (x_2 - x_0) + \\
&\quad + \ldots + a_n \cdot (x_1 - x_0) \cdot (x_2 - x_0) \cdot \ \ldots \ \cdot (x_n - x_0) \ .
\end{aligned} \tag{10.16}$$

The sought coefficients a_k can be gradually determined from the given equations, e.g.:

$$\begin{aligned}
a_0 &= f(x_0) \\
a_1 &= \frac{f(x_1) - a_0}{x_1 - x_0} \\
a_2 &= \frac{f(x_2) - a_0 - a_1 \cdot (x_1 - x_0)}{(x_1 - x_0) \cdot (x_2 - x_0)} \\
&\ \vdots \\
a_n &= \frac{f(x_n) - a_0 - a_1 \cdot (x_1 - x_0) - \ldots - a_{n-1} \cdot (x_1 - x_0) \cdot \ \ldots \ \cdot (x_{n-1} - x_0)}{(x_1 - x_0) \cdot (x_2 - x_0) \cdot \ \ldots \ \cdot (x_n - x_0)} \ .
\end{aligned} \tag{10.17}$$

The equation (10.16) for $k = 1, \ldots, n$ can alternatively be expressed using the following relation:

$$\Phi_k(x_k) = f(x_k) = \Phi_{k-1}(x_k) + a_k \cdot (x_1 - x_0) \cdot (x_2 - x_0) \cdot \ \ldots \ \cdot (x_k - x_0) \ , \tag{10.18}$$

which can also be used to generalize the calculation of unknown coefficients a_k:

$$a_k = \frac{f(x_k) - \Phi_{k-1}(x_k)}{(x_1 - x_0) \cdot (x_2 - x_0) \cdot \ \ldots \ \cdot (x_k - x_0)} \ . \tag{10.19}$$

This problem can also be described using so-called divided differences:

$$\begin{aligned}
f[\ x_k\ ] &= f(x_k) \\
f[\ x_k \quad x_{k+1}\ ] &= \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k} \\
f[\ x_k \quad x_{k+1} \quad x_{k+2}\ ] &= \frac{f[\ x_{k+1} \quad x_{k+2}\ ] - f[\ x_k \quad x_{k+1}\ ]}{x_{k+2} - x_k} \\
f[\ x_k \quad x_{k+1} \quad x_{k+2} \quad x_{k+3}\ ] &= \frac{f[\ x_{k+1} \quad x_{k+2} \quad x_{k+3}\ ] - f[\ x_k \quad x_{k+1} \quad x_{k+2}\ ]}{x_{k+3} - x_k} \\
&\ \vdots \ .
\end{aligned} \tag{10.20}$$

These numbers correspond to the coefficients a_k for $k = 0, 1, \dots, n$ of Newton's interpolation polynomial, which can be defined in the final form:

$$\begin{aligned} N_n(x) = f[\ x_1\ ] &+ f[\ x_1 \quad x_2\ ] \cdot (x - x_1) \\ &+ f[\ x_1 \quad x_2 \quad x_3\ ] \cdot (x - x_1) \cdot (x - x_2) \\ &+ f[\ x_1 \quad x_2 \quad x_3 \quad x_4\ ] \cdot (x - x_1) \cdot (x - x_2) \cdot (x - x_3) \\ &+ \dots + \\ &+ f[\ x_1 \quad \cdots \quad x_n\ ] \cdot (x - x_1) \cdot \ \dots\ \cdot (x - x_{n-1})\ , \end{aligned} \tag{10.21}$$

or for $k = 1, \dots, n$:

$$N_k(x) = N_{k-1}(x) + f[\ x_1 \quad \cdots \quad x_k\ ] \cdot (x - x_1) \cdot \ \dots\ \cdot (x - x_{k-1})\ . \tag{10.22}$$

The calculation of Newton's interpolation polynomial can be expressed algorithmically, e.g., using Algorithm 23.

```
Input  : $x = [\ x_1 \quad \cdots \quad x_n\ ]$, $y = [\ y_1 \quad \cdots \quad y_n\ ]$, $z$
Output: $N_n(z)$
for $j \leftarrow 1, 2, \dots, n$ do
    $f[x_j] \leftarrow y_j$
end
for $i \leftarrow 2, 3, \dots, n$ do
    for $j \leftarrow 1, 2, \dots, n + 1 - i$ do
        $f[\ x_j \quad \cdots \quad x_{j+i-1}\ ] \leftarrow \dfrac{f[\ x_{j+1} \quad \cdots \quad x_{j+i-1}\ ] - f[\ x_j \quad \cdots \quad x_{j+i-2}\ ]}{x_{j+i-1} - x_j}$
    end
end
$N_n(z) \leftarrow \sum_{i=1}^{n} f[\ x_1 \quad \cdots \quad x_i\ ] \cdot (x - x_1) \cdot \ \dots\ \cdot (x - x_{i-1})$
```

Algorithm 23: Determining the value of Newton's interpolation polynomial $N_n(x)$.

x_1	$f[\ x_1\ ]$		
		$f[\ x_1 \quad x_2\ ]$	
x_2	$f[\ x_2\ ]$		$f[\ x_1 \quad x_2 \quad x_3\ ]$
		$f[\ x_2 \quad x_3\ ]$	
x_3	$f[\ x_3\ ]$		

Tab. 10.1 Divided differences of Newton's interpolation polynomial for three points.

For the recursive expression of the divided differences of Newton's interpolation polynomial, a tabular expression is used (for three points, see Table 10.1). The coefficients of Newton's interpolation polynomial (10.22) can then be subtracted from the upper edge of the displayed triangle.

Example 10.5. Using Newton's interpolation polynomial, determine the equation of the interpolation function $y(x)$ for the three points from Exercise 10.3 with coordinates $[x_0, y_0] = [0, 1]$, $[x_1, y_1] = [2, 2]$ and $[x_2, y_2] = [3, 4]$.

Solution. Using the procedure (10.21) for constructing Newton's interpolation polynomial, we can compile Table 10.2 whose individual terms are determined using Eq. (10.20):

$$f[\ x_1 \quad x_2\ ] = \frac{f[x_2] - f[x_1]}{x_2 - x_1} = \frac{2-1}{2-0} = \frac{1}{2}\ , \tag{10.23}$$

$$f[\ x_1 \quad x_2 \quad x_3\ ] = \frac{f[\ x_2 \quad x_3\ ] - f[\ x_1 \quad x_2\ ]}{x_3 - x_1} = \frac{2 - \dfrac{1}{2}}{3-0} = \frac{1}{2}\ , \tag{10.24}$$

$$f[\ x_2 \quad x_3\ ] = \frac{f[x_3] - f[x_2]}{x_3 - x_2} = \frac{4-2}{3-2} = 2\ . \tag{10.25}$$

$x_1 = 0$	$f[\ x_1\ ] = 1$		
		$f[\ x_1 \quad x_2\ ] = \frac{1}{2}$	
$x_2 = 2$	$f[\ x_2\ ] = 2$		$f[\ x_1 \quad x_2 \quad x_3\ ] = \frac{1}{2}$
		$f[\ x_2 \quad x_3\ ] = 2$	
$x_3 = 3$	$f[\ x_3\ ] = 4$		

Tab. 10.2 Divided differences of Newton's interpolation polynomial for three points from Exercise 10.3.

As already said, the coefficients of the sought Newton's interpolation polynomial according to (10.22) can then be subtracted from Table 10.2 from the upper edge of the displayed triangle:

$$\begin{aligned} N_3(x) &= f[\ x_1\ ] + f[\ x_1 \quad x_2\ ] \cdot (x - x_1) + f[\ x_1 \quad x_2 \quad x_3\ ] \cdot (x - x_1) \cdot (x - x_2) = \\ &= 1 + \frac{1}{2} \cdot (x - 0) + \frac{1}{2} \cdot (x - 0) \cdot (x - 2) = \frac{1}{2} \cdot x^2 - \frac{1}{2} \cdot x + 1\ . \end{aligned} \tag{10.26}$$

From the resulting equation of the relation of Newton's interpolation polynomial (10.26) it is clear that the same second-order polynomial as in the case of Exercise 10.3 was achieved. ▲

A function that determines for the specified point with the coordinate x in the input parameter `par` the value of Newton's interpolation polynomial, compiled for the specified set of points with coordinates x and y stored in the input parameters with the vectors `x` and `y`, can be coded in MATLAB, e.g., via the script `newton.m`:

```
function s=newton(x,y,par)
n=length(x);
for j=1:n
  tab(j,1)=y(j);
end
for i=2:n
  for j=1:n+1-i
    tab(j,i)=(tab(j+1,i-1)-tab(j,i-1))/(x(j+i-1)-x(j));
  end
end
s=tab(1,1);
for i=2:n
  m=tab(1,i);
  for j=1:i-1
      m=m*(par-x(j));
  end
  s=s+m;
end
```

The script can be slightly modified so that it is possible to effectively determine the values of Newton's interpolation polynomial even for a vector containing the x coordinates of several points in the input parameter `par` (the script is also functional for one coordinate).

```
function s=newton(x,y,par)
n=length(x);
for j=1:n
  tab(j,1)=y(j);
end
for i=2:n
  for j=1:n+1-i
    tab(j,i)=(tab(j+1,i-1)-tab(j,i-1))/(x(j+i-1)-x(j));
  end
end
num=length(par);
for k=1:num
  tot=tab(1,1);
  for i=2:n
    m=tab(1,i);
```

```
      for j=1:i-1
        m=m*(par(k)-x(j));
      end
      tot=tot+m;
    end
    s(k)=tot;
  end
```

Example 10.6. Using Newton's interpolation polynomial, determine the value of the bending moment according to the instructions in Exercise 10.6.

Comment 10.7. It is also possible to construct Newton's interpolation polynomial via a rather interesting script – shown below – which allows you to enter the individual points needed to construct the interpolation polynomial directly from the graph by clicking the left mouse button. One can observe how the order of the interpolation polynomial increases with increasing points. The procedure is terminated by a right click of the mouse.

```
xmin=-3;
xmax=3;
x_p=xmin:.01:xmax;
ymin=-3;
ymax=3;
plot([xmin xmax],[0 0],'k',[0 0],[ymin ymax],'k');
grid on;
x=[];
y=[];
button=1;
k=0;
title('\fontsize{12}Newton''s Interpolation');
while ~(button==3)
  [x_new,y_new,button]=ginput(1);
  if button==1
    k=k+1;
    x(k)=x_new;
    y(k)=y_new;
    y_p=newton(x,y,x_p);
    plot(x,y,'o',x_p,y_p,[xmin xmax],[0,0],'k',[0 0],...
        [ymin ymax],'k');
    title('\fontsize{12}Newton''s Interpolation');
    axis([xmin xmax ymin ymax]);
    grid on;
  end
end
```

10.4 Approximation by the Method of Least Squares

When interpolating by one of the previous methods, it was assumed that the interpolated function is specified by a table with values x_i and $f(x_i) = y_i$, where $i = 0, 1, \ldots, n$. In the case of approximation, the task is not to find a function that coincides at the specified points with the sought-after function, but rather to identify an approximation function $F(x)$ that would best fit the $n+1$ specified empirical points $[x_0, y_0]$, $[x_1, y_1]$ to $[x_n, y_n]$.

In the method of least squares, the sum of the squares of the differences between the values of the approximation function $F(x_i)$ and the measured values y_i is used as the goodness-of-fit criterion:

$$Q = \sum_{i=0}^{n} (F(x_i) - y_i)^2 \,. \tag{10.27}$$

The function $F(x)$ can generally be defined as:

$$F(x) = a_0 \cdot f_0(x) + a_1 \cdot f_1(x) + \ldots + a_m \cdot f_m(x) \;, \tag{10.28}$$

where $f_0, f_1, \ldots, f_m$ are suitably chosen linearly independent functions and a_0, a_1 až a_m are unknown real coefficients, which are determined so that the value of Q in the relation (10.27) is minimal. It must therefore hold that:

$$\frac{\partial Q}{\partial a_k} = 2 \cdot \sum_{i=0}^{n} (F(x_i) - y_i) \cdot \frac{\partial F(x_i)}{\partial a_k} = 0 \;, \tag{10.29}$$

where $k = 0, 1, \ldots, m$.

When choosing

$$\frac{\partial F(x_i)}{\partial a_k} = f_i(x_i) \;, \tag{10.30}$$

it must hold that:

$$\frac{\partial Q}{\partial a_k} = 2 \cdot \sum_{i=0}^{n} [a_0 \cdot f_0(x_i) + a_1 \cdot f_1(x_i) + \ldots + a_m \cdot f_m(x_i) - y_i] \cdot f_k(x_i) = 0 \,. \tag{10.31}$$

Relation (10.31) can be further modified as follows:

$$\begin{aligned} \sum_{i=0}^{n} [a_0 \cdot f_k(x_i) \cdot f_0(x_i) + a_1 \cdot f_k(x_i) \cdot f_1(x_i) + \ldots + a_m \cdot f_k(x_i) \cdot f_m(x_i)] = \\ = \sum_{i=0}^{n} f_k(x_i) \cdot y_i \;, \end{aligned} \tag{10.32}$$

i.e.,

$$a_0 \cdot \sum_{i=0}^{n} f_k(x_i) \cdot f_0(x_i) + a_1 \cdot \sum_{i=0}^{n} f_k(x_i) \cdot f_1(x_i) + \ldots + a_m \cdot \sum_{i=0}^{n} f_k(x_i) \cdot f_m(x_i) = \\ = \sum_{i=0}^{n} f_k(x_i) \cdot y_i \ , \tag{10.33}$$

where $k = 0, 1, \ldots, m$.

Relation (10.33) can also be expressed in matrix form:

$$\begin{bmatrix} \sum\limits_{i=0}^{n} f_0^2(x_i) & \sum\limits_{i=0}^{n} f_0(x_i) \cdot f_1(x_i) & \ldots & \sum\limits_{i=0}^{n} f_0(x_i) \cdot f_m(x_i) \\ \sum\limits_{i=0}^{n} f_1(x_i) \cdot f_0(x_i) & \sum\limits_{i=0}^{n} f_1^2(x_i) & \ldots & \sum\limits_{i=0}^{n} f_1(x_i) \cdot f_m(x_i) \\ \vdots & \vdots & \ddots & \vdots \\ \sum\limits_{i=0}^{n} f_m(x_i) \cdot f_0(x_i) & \sum\limits_{i=0}^{n} f_m(x_i) \cdot f_1(x_i) & \ldots & \sum\limits_{i=0}^{n} f_m^2(x_i) \end{bmatrix} \cdot \begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{Bmatrix} = \\ = \begin{Bmatrix} \sum\limits_{i=0}^{n} f_0(x_i) \cdot y_i \\ \sum\limits_{i=0}^{n} f_1(x_i) \cdot y_i \\ \vdots \\ \sum\limits_{i=0}^{n} f_m(x_i) \cdot y_i \end{Bmatrix} . \tag{10.34}$$

10.4.1 Linear Approximation

In linear approximation, the relationship between the variables x and y is:

$$F(x) = a \cdot x + b \ , \tag{10.35}$$

where a, b are unknown parameters that can be determined from the condition according to (10.27):

$$Q = \sum_{i=0}^{n} (a \cdot x_i + b - y_i)^2 = \min \ . \tag{10.36}$$

Solving the problem given by (10.28) leads to a system of two equations:

$$\frac{\partial Q}{\partial a} = 0 \tag{10.37}$$

and

$$\frac{\partial Q}{\partial b} = 0 \ . \tag{10.38}$$

After adjusting both equations according to (10.31) to (10.33), their resulting form can be obtained:

$$\begin{aligned} n \cdot b + \left(\sum_{i=0}^{n} x_i\right) \cdot a &= \sum_{i=0}^{n} y_i \\ \left(\sum_{i=0}^{n} x_i\right) \cdot b + \left(\sum_{i=0}^{n} x_i^2\right) \cdot a &= \sum_{i=0}^{n} x_i \cdot y_i \ , \end{aligned} \tag{10.39}$$

which can be expressed as a matrix:

$$\begin{bmatrix} n & \sum\limits_{i=0}^{n} x_i \\ \sum\limits_{i=0}^{n} x_i & \sum\limits_{i=0}^{n} x_i^2 \end{bmatrix} \cdot \begin{Bmatrix} b \\ a \end{Bmatrix} = \begin{Bmatrix} \sum\limits_{i=0}^{n} y_i \\ \sum\limits_{i=0}^{n} x_i \cdot y_i \end{Bmatrix} . \tag{10.40}$$

10.4.2 Approximation by m-th Degree Polynomial

If a polynomial of the m-th degree is chosen as the approximation function:

$$F_m(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2 + \ldots + a_m \cdot x^m \ , \tag{10.41}$$

after simplifying (10.29) to (10.33), a system of $m+1$ equations can be obtained by substituting $k = 0, 1, \ldots, m$ in $f_k(x) = x^k$:

$$\begin{bmatrix} \sum\limits_{i=0}^{n} (x_i^0)^2 & \sum\limits_{i=0}^{n} x_i^0 \cdot x_i^1 & \ldots & \sum\limits_{i=0}^{n} x_i^0 \cdot x_i^m \\ \sum\limits_{i=0}^{n} x_i^1 \cdot x_i^0 & \sum\limits_{i=0}^{n} (x_i^1)^2 & \ldots & \sum\limits_{i=0}^{n} x_i^1 \cdot x_i^m \\ \vdots & \vdots & \ddots & \vdots \\ \sum\limits_{i=0}^{n} x_i^m \cdot x_i^0 & \sum\limits_{i=0}^{n} x_i^m \cdot x_i^1 & \ldots & \sum\limits_{i=0}^{n} (x_i^m)^2 \end{bmatrix} \cdot \begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{Bmatrix} = \begin{Bmatrix} \sum\limits_{i=0}^{n} x_i^0 \cdot y_i \\ \sum\limits_{i=0}^{n} x_i^1 \cdot y_i \\ \vdots \\ \sum\limits_{i=0}^{n} x_i^m \cdot y_i \end{Bmatrix} . \tag{10.42}$$

The system of equations (10.42) with unknown coefficients $a_0, a_1, \ldots, a_m$ can then be further adjusted to the form:

$$\begin{bmatrix} n+1 & \sum\limits_{i=0}^{n} x_i & \sum\limits_{i=0}^{n} x_i^2 & \ldots & \sum\limits_{i=0}^{n} x_i^m \\ \sum\limits_{i=0}^{n} x_i & \sum\limits_{i=0}^{n} x_i^2 & \sum\limits_{i=0}^{n} x_i^3 & \ldots & \sum\limits_{i=0}^{n} x_i^{m+1} \\ \vdots & \vdots & \ddots & \vdots & \\ \sum\limits_{i=0}^{n} x_i^m & \sum\limits_{i=0}^{n} x_i^{m+1} & \sum\limits_{i=0}^{n} x_i^{m+2} & \ldots & \sum\limits_{i=0}^{n} x_i^{2 \cdot m} \end{bmatrix} \cdot \begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{Bmatrix} = \begin{Bmatrix} \sum\limits_{i=0}^{n} y_i \\ \sum\limits_{i=0}^{n} x_i \cdot y_i \\ \vdots \\ \sum\limits_{i=0}^{n} x_i^m \cdot y_i \end{Bmatrix} . \tag{10.43}$$

Example 10.8. Perform a linear approximation as well as a 2nd degree polynomial approximation for the data contained in Table 10.3. For both cases, determine the sum of the squares (squares) of the differences between the values of the approximation function $F(x_i)$ and the measured values y_i.

x	1	2	3	4	5
y	0	2	2	5	4

Tab. 10.3 Input data for calculating the approximation in Exercise 10.8.

Solution. The computation of the linear approximation with the m-th degree polynomial including the sum of the squares of the differences between the values of the appropriate approximation function $F(x_i)$ and the measured values of y_i can be performed via the following sequence of commands:

```
clear;
clc;
x0=[1 2 3 4 5];
y0=[0 2 2 5 4];
m=2;
for i=1:m+1
    for j=i:m+1
        A(i,j)=sum(x0.^((i-1)+(j-1)));
        if ~(i==j)
            A(j,i)=A(i,j);
        end
    end
    b(i)=sum((x0.^(i-1)).*y0);
end
c=A\b';
x=0:.1:6;
for j=1:length(x)
  s=c(1);
  for i=1:m
    s=s+c(i+1)*x(j)^(i);
  end
  y(j)=s;
end
hold on
plot(x,y,'g-','LineWidth',2);
plot(x0,y0,'k*');
```

```
title('\fontsize{12}Approximation by the Method of Least Squares');
xlabel('x');
ylabel('y(x)');
hold off
sum_squares=0;
for j=1:length(x0)
  s=c(1);
  for i=1:m
    s=s+c(i+1)*x0(j)^(i);
  end
  sum_squares=sum_squares+(s-y0(j))^2;
end
sum_squares
```

For the case of linear approximation, it is possible to obtain a straight line – as shown in Fig. 10.4 – with the sum of the squares of the differences between the values of the respective approximation function $F(x_i)$ and the measured values of y_i being equal to 3.1. In the case of approximation by a polynomial of the 2nd degree (see Fig. 10.5) the sum of the squares of the differences between the values of the respective approximation function $F(x_i)$ and the measured values y_i is equal to 2.4571.

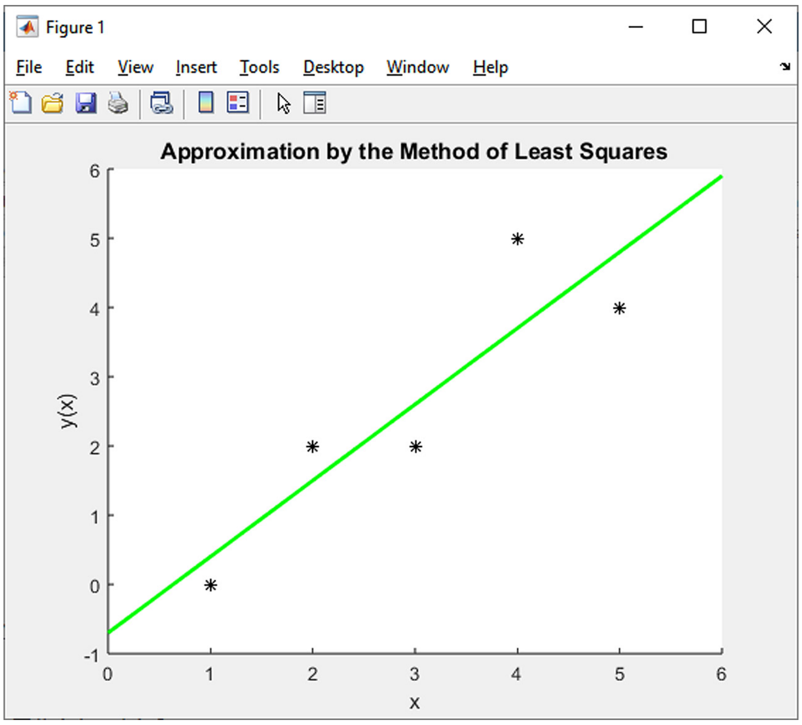


Fig. 10.4 Linear approximation for the points specified in Exercise 10.8.

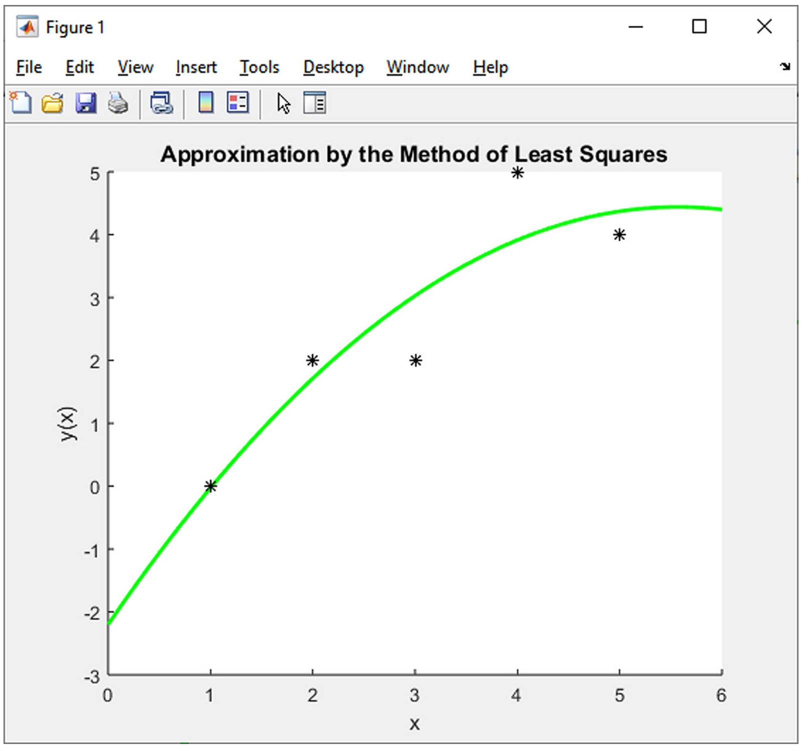


Fig. 10.5 Approximation by a polynomial of the 2nd degree for the points specified in Exercise 10.8.

▲

Example 10.9. Select the most appropriate degree of polynomial to approximate the measured values of cubic compressive strength of concrete depending on the days of maturation of the concrete mixture, which are shown in Table 10.4. Use the sum of squares of the differences between the values of the approximation function $F(x_i)$ and the measured values y_i as the goodness-of-fit criterion.

x [dny]	0	0	0	7	7	7	14	14	14	28	28	28
y [MPa]	0	0	0	21.5	22.2	21.2	30.7	31.4	30.5	40.1	43.4	41.5

Tab. 10.4 Input data for calculating the approximation function in Exercise 10.9.

Literature

[1] *Algorithm.* Website focused on the creation of algorithms. [on-line]. <https://www.programming-algorithms.net>. (Cited on page 50.)

[2] Eaton, J.W. *Octave.* Programming software for performing mathematical calculations. Freeware, verze 4.2.1. [on-line]. <http://www.gnu.org/software/octave>. University of Wisconsin, Department of Chemical Engineering, 1998-2017. (Cited on page 1.)

[3] Kučera, R. *Numerical methods.* Study textbook. VSB – Technical University of Ostrava. (152 p). ISBN 80-248-1198-7. (in Czech)

[4] Materna, A. — Štěpánek, P. — Teplý, B. *Automation of engineering tasks.* Textbook. Brno University of Technology, 1985. (132 p). (in Czech)

[5] MATLAB. Programming system for performing mathematical calculations. Commercial software, version R2023b. [on-line]. <http://www.mathworks.com>. The MathWorks, January 2024. (Cited on page 1.)

[6] Mika, S. *Numerical methods of algebra.* Math for Technical Universities. 2nd edition. SNTL – Publishing company of technical literature, Prague, 1985. (176 p). (in Czech) (Cited on page 35.)

[7] *OctaveOnline.* Programming software for performing mathematical calculations – on-line version. [on-line]. <https://octave-online.net>. (Cited on page 1.)

[8] Olehla, M. — Tišer, J. *Practical use of Fortran.* 2nd edited edition. Publishing company of transport and connections, Prague, 1979. (432 p). (in Czech) (Cited on page 47 and 120.)

[9] Ralston, A. *Fundamentals of numerical mathematics.* 1st edition. Academia, Prague, 1973. (635 p). (in Czech)

[10] Rektorys, K. *An overview of applied mathematics.* 4th edition. SNTL – Publishing company of technical literature, Prague, 1981. (1140 p). (in Czech)

[11] Sauer T. *Numerical Analysis*. George Mason University. Pearson Education, Inc., 2006. (669 p). ISBN 0-321-26898-9. (Cited on page 16.)

[12] Sigmon, K. MATLAB *Primer*. Electronic textbook of MATLAB with demonstration examples. 2nd edition. [on-line]. <https://web.archive.org/web/20050315101013/http://artax.karlin.mff.cuni.cz/%7ebeda/cz/matlab/primer/matlab-primer.html>. Department of Mathematics, University of Florida, 1989, 1992. (Cited on page 12.)

[13] *Wikipedia*. Free-content online encyclopedia. Website. [on-line]. <http://en.wikipedia.org>. (Cited on page 13.)

[14] Wirth N. *Algorithms and data structures*. 1st edition. Alfa, publishing company of technical and economic literature, Bratislava, 1988. (488 p). (in Slovak) (Cited on page 55.)