

# Software Development – Numerical Programming

Hans-Joachim Bungartz

October 14, 2003



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Numerical Programming . . . . .	7
1.2	Discretization . . . . .	8
1.3	A Small Example of a Numerical Simulation: Population Dynamics . . . . .	9
1.4	Contents of the Course . . . . .	9
1.5	References . . . . .	10
<b>2</b>	<b>Some Informatical Basics</b>	<b>11</b>
2.1	Computers . . . . .	11
2.2	Operating Systems . . . . .	13
2.3	Algorithms . . . . .	13
2.4	Data Structures . . . . .	16
2.5	Languages . . . . .	17
<b>3</b>	<b>Principles of Programming</b>	<b>19</b>
3.1	General Remarks . . . . .	19
3.2	Constants, Types, Variables, Names, Expressions . . . . .	20
3.3	Functions . . . . .	21
3.4	Data Structures 1: Sequences, Lists, Sets . . . . .	22
3.5	Control Structures: Loops and IF-Clauses . . . . .	23
3.6	Procedures . . . . .	23
3.7	Data Structures 2: Tables, Arrays, Trees, Pointers . . . . .	25
3.8	In- and Output . . . . .	28
<b>4</b>	<b>Basics of Numerical Analysis</b>	<b>29</b>
4.1	Floating Point Numbers . . . . .	29
4.2	Floating Point Arithmetic, Round-off Errors . . . . .	31
4.3	Round-off Error Analysis . . . . .	33
4.4	Condition . . . . .	34

4.5	Stability . . . . .	36
4.6	Summary . . . . .	37
<b>5</b>	<b>Direct Solution of Systems of Linear Equations</b>	<b>39</b>
5.1	Preparatory Remarks . . . . .	39
5.2	Gaussian Elimination . . . . .	42
5.3	Pivoting . . . . .	44
5.4	Cholesky Decomposition . . . . .	45
<b>6</b>	<b>Interpolation</b>	<b>47</b>
6.1	Preparatory Remarks . . . . .	47
6.2	Interpolation with Polynomials . . . . .	48
6.3	Aitken and Neville's Scheme . . . . .	49
6.4	Divided Differences and Error . . . . .	49
6.5	Condition of Polynomial Interpolation . . . . .	50
6.6	Trigonometric Interpolation . . . . .	51
6.7	Polynomial Splines . . . . .	51
6.8	Interpolation with Cubic Splines . . . . .	52
<b>7</b>	<b>Numerical Quadrature</b>	<b>55</b>
7.1	Simple and Composite Rules . . . . .	56
7.2	The Principle of Extrapolation . . . . .	58
7.3	Gauß Quadrature . . . . .	58
7.4	Monte-Carlo Quadrature . . . . .	59
<b>8</b>	<b>Iterative Solution of Systems of Linear Equations</b>	<b>61</b>
8.1	Classical Relaxation Techniques . . . . .	61
8.2	The Multigrid Principle . . . . .	64
8.3	Nonlinear Equations . . . . .	66
8.4	The Method of Conjugate Gradients . . . . .	68
<b>9</b>	<b>Ordinary Differential Equations</b>	<b>73</b>
9.1	Preparatory Remarks . . . . .	73
9.2	Approximation by Finite Differences . . . . .	74
9.3	Consistency and Convergence . . . . .	75
9.4	Multistep Methods . . . . .	76
9.5	Stiffness, Implicit Methods . . . . .	77

<i>CONTENTS</i>	5
<b>10 Partial Differential Equations</b>	<b>79</b>
10.1 Preliminary Remarks . . . . .	79
10.2 Finite Differences (FD) . . . . .	80
10.3 Finite Elements (FE) . . . . .	81
10.4 Finite Volumes . . . . .	85
10.5 Unsteady Problems . . . . .	86
<b>11 Grid Generation</b>	<b>87</b>
11.1 Structured Grids . . . . .	88
11.1.1 Composite Grids . . . . .	88
11.1.2 Block-Structured Grids . . . . .	88
11.1.3 Elliptic Generators . . . . .	89
11.1.4 Hyperbolic Generators . . . . .	90
11.1.5 Algebraic Generators . . . . .	90
11.1.6 Adaptive Grids . . . . .	91
11.2 Unstructured Grids . . . . .	92
11.2.1 Delaunay Triangulations, Voronoi Diagrams . . . . .	92
11.2.2 Point Creation . . . . .	92
11.2.3 Other Techniques . . . . .	93
11.2.4 Adaptive Grids . . . . .	94
11.3 Methods for Varying Geometries . . . . .	94
11.3.1 Front Tracking Methods . . . . .	95
11.3.2 Front Capturing Methods . . . . .	95
11.3.3 Clicking and Sliding Mesh Techniques . . . . .	95
<b>12 Parallel Computing</b>	<b>97</b>



# Chapter 1

## Introduction

### 1.1 Numerical Programming

What is **numerical programming**?

The combination of numerical methods and algorithms and their implementation!

Some important characteristics of the numerical approach:

- **constructive:**  
not dealing with topics like existence or uniqueness of solutions, but with their construction  
⇒ more pragmatic than pure mathematics or theoretical informatics
- **approximate:**  
due to their complexity, many practical problems can not be solved exactly (analytically)  
⇒ therefore: if accuracy can be obtained by investing computational effort (longer computing times lead to better results), then approximations are satisfying;  
furthermore: often, input data (resulting from measurements) are not exact neither – why processing them exactly?
- **computer-based:**  
due to problem size, the approximate solutions are determined with the help of computers  
⇒ somewhere between mathematics and informatics
- **value-oriented:**  
basically, only floating point values and elementary operations (ADD, SUB, MULT, DIV, SQRT, and comparisons) are used – everything else is based on these:
  - the sine function is computed via some small and fast program for approximating the sine by a suitable polynomial every time it is called for some given argument
  - formula manipulation is not done neither: no symbolic differentiation or integration, just approximate values for  $f'(a)$  oder  $F(a)$  for a given  $a$⇒ more primitive than human beings (as often in a computer)
- **subject to round-off errors:**  
all arithmetic operations are executed with a fixed number of digits and, hence, not exactly
  - the resulting *round-off errors* can accumulate – and, suddenly, wrong results can occur
  - therefore, the influence of round-off errors has to be studied carefully and estimated a priori

- **application-oriented:**

- numerical problems typically stem from application domains like science or engineering disciplines
- there, certain phenomena or processes shall be studied by *numerical simulation* (instead of or in addition to experiments) – and for that, sophisticated numerical methods are necessary

⇒ significant importance especially for the applications – like mechanics in the case of COMMAS!

- **implementation-oriented:**

the algorithm is not sufficient – we also need efficient implementations (data structures, programming methods) and readable and maintainable codes (software)

## 1.2 Discretization

the crucial notion of numerical programming is **discretization**:

- a fixed number of digits (fixed length of number representation in the computer) ⇒ discrete values (no  $\pi$ , just something like 3.1415927!)
- hence, everything has to be *discretized*:
  - **numbers**: there are only discrete values, i.e. integers, for example; for real numbers, the so-called *floating point arithmetic* has been developed (a fixed number of relevant digits with a, however, variable location: 12345.0, 123.45, or 0.0012345, e.g.)
  - **functions**: the above-mentioned  $\sin(x)$  is available only at discrete abscissas (elsewhere, it has to be *interpolated*), and precision is limited to a certain number of digits; instead of on  $\mathbb{R}$ , functions of one variable live on a discrete *grid of grid points*; an *equidistant* grid has a constant distance between neighbouring grid points (*resolution, mesh width  $h$* ), but there exist grids of variable mesh width, too (*adaptive* grids); obviously, a grid's resolution determines the number of grid points and, thus, influences memory and run time requirements
  - **operators**: differential quotients (derivatives) are typically approximated by difference quotients:

$$\frac{f(x+h) - f(x)}{h}, \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \quad \text{for} \quad f', f'';$$

herewith, the limit process known from analysis is imitated or approximated:

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

this is typical for numerical methods: limit processes, asymptotic behaviour etc. are the starting point for numerical approximations; this also shows the basic principle of discretization: a higher resolution entails higher cost, but, on the other hand, ensures a better approximation to the continuous quantity

- **domains**: the integral of a function  $f(x, y)$  over a circle (for example, take  $f \equiv 1$  in order to determine its area) is computed numerically not via the circle, but with the help of a sequence of polygons, e.g.

## 1.3 A Small Example of a Numerical Simulation: Population Dynamics

- problem: how does the strength  $p(t)$  of some population  $P$  develop in time?
- possible mathematical model (ordinary differential equation):

$$\dot{p}(t) = a \cdot p(t) - b \cdot p^2(t), \quad p(0) = p_0, \quad a \gg b > 0;$$

- justification: growth (derivative) is related to the current strength; first, a big population helps (active reproduction); later, i.e. for very big  $p(t)$ , it begins to disturb (competition)
- the above model describes the situation in the USA between 1790 and 1950 quite well (with  $a = 0.03134$  and  $b = 1.5587 \cdot 10^{-10}$ );  
S-shaped development, saturation (no more changes, i.e.  $\dot{p}(t) = 0$ ) for  $p(t) \equiv a/b$ , hence

$$\lim_{t \rightarrow \infty} p(t) = \frac{a}{b}$$

- analysis provides the following solution:

$$p(t) = \frac{a \cdot p_0}{b \cdot p_0 + (a - bp_0) \cdot e^{-a \cdot t}}, \quad p(0) = p_0;$$

- obviously: coarse model (no specific external influence, no enemies, no prey etc.)  
⇒ in most situations more complicated equations, for which in general no explicit solution can be given  
⇒ need for numerical solution; to discretize means to advance in small time steps  $\delta t$ , for example following the rule

$$p_{k+1} \approx p_k + \delta t \cdot \dot{p}_k \approx p_k + \delta t \cdot (a \cdot p_k - b \cdot p_k^2), \quad p_k \approx p(k \cdot \delta t)$$

- questions: influence of  $\delta t$  (a big time step is cheap, but possibly inaccurate)? are there better numerical schemes?

## 1.4 Contents of the Course

in this course (in contrast to several other COMMAS courses): the underlying mathematical or physical model isn't a topic at all, it is supposed to be given  
⇒ we need numerical algorithms and programs for the approximate solution of the model equations – that's numerical programming!

hence, the course contains

- some introduction to numerical methods (the basic notions, simple methods for standard problems)
- an introduction to numerical prototyping (development and testing of numerical algorithms) with Maple
- an introduction to real-life numerical programming as well as to the development of numerical software with C/C++

## 1.5 References

- Maple Manuals
- Cornil, *Testud*: An Introduction to Maple V, Springer, 2001
- Betounes, Redfern: *Mathematical Computing – An Introduction to Programming Using Maple*, Springer, 2001
- Gander, Hrebicek: *Solving Problems in Scientific Computing Using Maple and MATLAB*, Springer, 1997
- Stoer, Bulirsch: *Introduction to Numerical Analysis*, Springer, 1996
- Kernighan, Ritchie: *The C Programming Language – ANSI C Version*, Prentice Hall
- Eckel: *Thinking in C++*, Prentice Hall

# Chapter 2

## Some Informatical Basics

In the following, some informatical basics are discussed which are especially important for (numerical) programming.

### 2.1 Computers

First of all, we turn to our most important tool, the computer itself:

- most of today's computers (PCs, work stations) still follow the classical *von-Neumann-model* and consist of
  - *CPU (Central Processing Unit)* with a control unit (processing the program) and an arithmetic logical unit (performing the computations)
  - *main memory* (contains both programs and data)
  - *I/O unit* (responsible for the input and output of data and for communication with peripherals)
- the heart of modern computers are *micro processors* (basically the CPU); for a survey of the state of the art, see <http://bwrc.eecs.berkeley.edu/CIC/summary/>

processor	computer class	clock rate (GHz)	transistors (Mill.)	word length (Bit)
intel Pentium 4	PC	≤ 2.5	55	32
intel Itanium 2	PC	≤ 1.0	221	64
AMD Athlon	PC	≤ 1.8	37	32
Power4	IBM	≤ 1.3	680	64
Sparc Ultra III	SUN	≤ 1.1	30	64
Alpha 21364	DEC	≤ 1.2	152	64
Mako	HP	≤ 1.0	325	64

explanation:

- clock rate: frequency, decisive for the processor's performance, obviously physically bounded by the speed of signal (speed of light); 1 GHz corresponds a cycle time of 1 ns!
- number of transistors on a chip: the level of integration is further increasing; hence, the possible on-chip functionality increases, too
- word length: number of bits interpreted together (one *word*); originally 8 (one byte), then 16, today 32 or 64 or even more (VLIW)

- important performance characteristics are *MIPS (Millions of Instructions Per Second)* and – for our purposes more important – *FLOPS (FLoating point Operations Per Second)*: measures performance for non-integer computations); there are often differences between integer and floating point performance; sometimes, there are floating point co-processors for tuning floating point operations
- besides the classical von-Neumann-model several non-standard principles of computer architecture have been developed (vector processors, array processors, data flow machines etc.), but only few of them have been or are successful
- each (micro-) processor has a *machine code* or *assembler* (sequence of simple commands) which he can process directly; since its programming is far from being comfortable, the programmer uses a *higher-level language* together with a so-called *compiler* to translate the program to machine code or an *interpreter* to directly execute single commands; in course of time, machine commands became more and more complicated and required a small *micro program* for their processing; the *RISC (Reduced Instruction Set Computers)* technology, today state-of-the-art, tried to return to small machine instruction sets with fixed formats
- there are several classes of machine commands: read, write or store, arithmetic commands, jump commands etc.
- how is the processing of a machine instruction realized?
  - the instruction is loaded to the control unit and analyzed
  - if necessary, operands are loaded from memory
  - the instruction is executed
  - if required, the result is written to memory
- today, machine commands are subdivided in small steps (fetch command, decode command, fetch operands, execute instruction etc.; in the Alpha 21164, the division of two floats takes 61 cycles and is executed with a strong overlap of the different steps (*Pipelining*), which leads to an accelerated execution of the whole program
- concerning memory, there is a whole *memory hierarchy*: from ultra fast memory on the chip (registers) via different levels of a *cache memory* (quickly available section of the main memory) and the main memory itself to remote memory; the rule: the faster, the more expensive; typical *slow-down-factors* (the access to a register shall cost one unit of time) and sizes:

register	1	0.5 kByte
level-1-cache	2 – 3	8 kByte
level-2-cache	6 – 12	96 kByte
level-3-cache	14 – 20	4 MByte
near main memory	60 – 100	1..2 GByte
far main memory	100 – 250	2..? GByte
background memory	O(100)	TByte
remote (net, MPI)	O(1000) – O(10000)	TByte

- besides microprocessors in PCs and work stations, *supercomputers* are of great importance; for both classes, *Moore's law* holds, which says that we can see an increase of performance by a factor of 10 every 5 years, with an advantage of supercomputers of roughly three orders of magnitude (or 15 years, respectively)
- today, there are mainly four classes of supercomputers (to a large extent used for numerical simulations):
  - *vector computers*: the classical representative, starting with the CRAY-1 in 1976; principle: a special processor, extreme pipelining

- *parallel computers*: several or many (standard-) processors in one computer; central problem: synchronization and communication
- *vector-parallel computers*: hybrid form, a few special processors
- *clusters*: several or many physically separated computers are connected and used together;
- the worldwide biggest supercomputers can be found in the American National Laboratories; one of Germany's national supercomputer centres is located in Stuttgart (HLRS)

## 2.2 Operating Systems

- the crucial program that always runs; the link between the programs of the user and the hardware
- several tasks:
  - resource management: which application program is allowed to use which service (print, ...), what has to be done for that (activate a driver etc.)?
  - interruption management: processing of user input (mouse click), control of error management (division by zero), ...
  - user login and logout, provide programming environment and windows system
  - management of multi-user-mode or multitasking (several windows of one user, background processes (clock))
  - ...
- prominent examples:
  - Bill's world: DOS, Windows{95/98/2000/NT}: dominant male for PCs (private and office)
  - UNIX-derivates (UNIX, AIX, IRIX, Solaris, ...): classical dominant male for work stations and, hence, in the academic field
  - LINUX: to some extent a link between the worlds ("UNIX for PCs"), heavily increasing in importance

## 2.3 Algorithms

*the* central term in informatics: precise sequence of unique instructions; sometimes properties like termination are required, too

⇒ basically nothing else but a finite list of clear statements of what to do

- we always speak of algorithms:
  - algorithms for sorting numbers
  - algorithms for the solution of a numerical problem
  - algorithms for inserting data into a data base
  - ...
- in spite of the above definition (precise, unique), algorithms can be rather different:

- *deterministic* algorithm: here, everything is determined in detail  
example: division with rest
- *non-deterministic* algorithm: choice of possible alternatives, with statements like “choose an element from the set”  
example: find a set’s minimum via successive comparisons
- *probabilistic* or *randomized* algorithm: the concrete step depends on a *probability distribution*, with statements like “roll the dice and behave according to the result”  
example: look for the highest point in the Alps; the one who goes upward only may reach the Zugspitze but miss the Mont Blanc; better: go downward from time to time
- desired properties of algorithms:
  - the algorithm shall be *effective*, i.e. do for what it has been designed for each input
  - it shall be *efficient*, i.e. do the job as fast or cheap as possible
  - for numerical (i.e. generally approximative) algorithms there is the additional aspect of the quality of the result (accuracy)
- typically, the efficiency (or – negatively formulated – the complexity) of an algorithm is quantified based on the number of necessary elementary arithmetic operations (run time complexity) or on the amount of memory (memory complexity) as a function of the number of input data;  
we use the *Landau notation* and say that the algorithm has the complexity  $O(n^k)$ , if for each problem size  $n > n_0$  the result can be obtained with an effort  $\leq C \cdot n^k$ , where  $C$  and  $n_0$  are constants
- as an example, consider the problem to sort a set  $S$  of  $n$  natural numbers (*the* classical informatical problem); what are the possibilities?
  - naive strategy: look for the minimum, put it on rank 1, and remove it from the set; proceed in the same way with the remaining set of  $n - 1$  numbers  
 $\Rightarrow$  obviously effective  
 $\Rightarrow$  requires  $(n - 1) + (n - 2) + (n - 3) + \dots + 1 = \frac{n(n-1)}{2} = O(n^2)$  elementary operations (here: comparisons)
  - better: follow the *divide & conquer* paradigm:  
subdivide the set into two subsets of (roughly) the same size, sort both of them (following recursively the same principle), and, afterwards, merge both sorted parts together (*Mergesort*)  
 $\Rightarrow$  example of a *recursive* algorithms: a problem is tackled by a reduction to problems of the same type, but of smaller size

obviously effective; the figure illustrates the efficiency: on each horizontal level  $O(n)$  comparisons are necessary; since the size of the subsets is reduced by a factor of roughly 2 from level to level, there are  $\log_2(n)$  levels  
 $\Rightarrow$  the overall complexity is  $O(n \cdot \log_2(n))$ ;  
 this is much better than  $n^2$  (take  $n = 10^6$ :  $10^{12}$  vs.  $2 \cdot 10^7!$ ), Mergesort is better and more efficient than the first method

we write Mergesort as an algorithm in some kind of *pseudo code* (not taking care of a special syntax):

```

procedure mergesort(S);
  begin
     $n := |S|$ ;
    if  $n > 1$  then
       $S =: S_1 \cup S_2, S_1 \cap S_2 = \emptyset, || S_1 | - | S_2 | \leq 1$ ;
  end

```

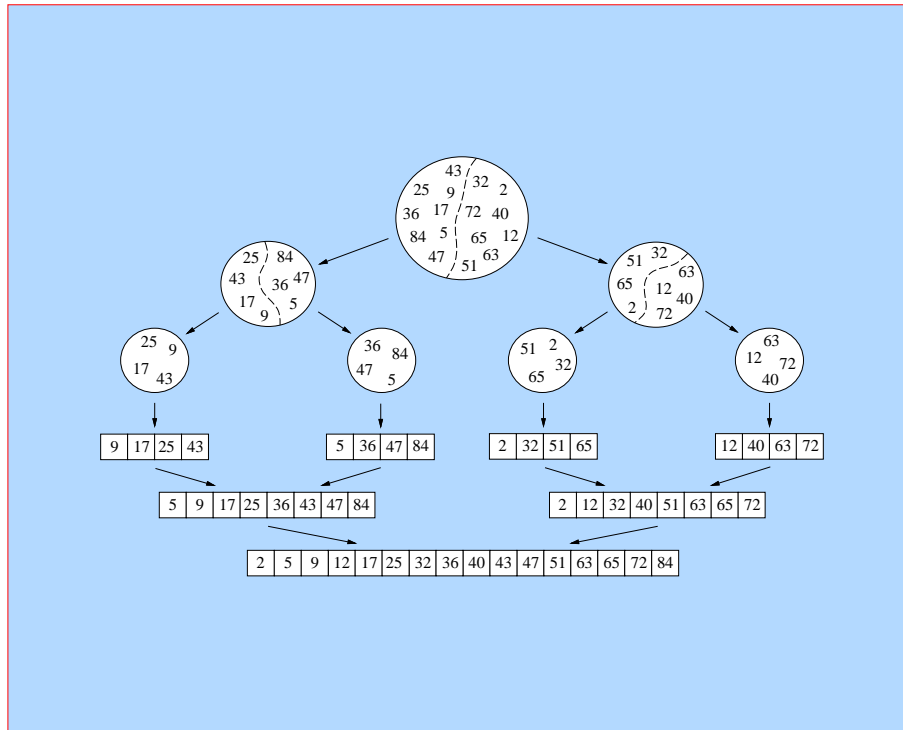


Figure 2.1: Mergesort

```

mergesort( $S_1$ );
mergesort( $S_2$ );
merge both resulting sorted subsequences;
end

```

the *recursion* can be seen clearly: if Mergesort is called for a set of more than one element, this entails two more calls of Mergesort with smaller sets as arguments (divide and conquer); an important thing that must not be forgotten is the termination condition of the recursion ( $n \leq 1$ ) – otherwise, you risk to compute eternally! recursion is a crucial notion of informatics: recursive functions are computable functions

- one more recursive sorting algorithm: *Quicksort*  
start with an arbitrarily sorted vector  $S$ ;  
via  $S[k]$ , we get access to the  $k$ -th component of  $S$ ;  
here is the pseudo code:

```

procedure quicksort( $S,l,r$ );
begin
   $i := l$ ;
   $k := r+1$ ;
   $x := S[l]$ ;
  while  $i < k$ 
    repeat  $i += 1$  until  $S[i] \geq x$ ;
    repeat  $k -= 1$  until  $S[k] \leq x$ ;
    if  $k > i$  then exchange ( $S[k],S[i]$ );
  exchange ( $S[l],S[k]$ );
  if  $l < k-1$  then quicksort( $l,k-1$ );
  if  $k+1 < r$  then quicksort( $k+1,r$ );
end

```

the initial call is `quicksort(S,1,n)`

the underlying principle is to select a key element (here the first one) and reorder the vector such that all smaller elements are located to the left and all bigger elements are located to the right of the key element; then, we proceed in the same way for both (non-sorted) parts

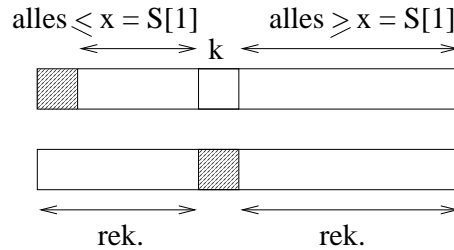


Figure 2.2: Quicksort

the analysis of Quicksort's complexity is more complicated than before: we have to distinguish the worst case and the average case;

$\Rightarrow$  *worst case* (the array is already sorted in reverse order):  $O(n^2)$

$\Rightarrow$  *average case* (mean of all possible arrangements):  $O(n \cdot \log(n))$

nevertheless, Quicksort is very widespread in applications, since the constant factor multiplied with  $n \cdot \log(n)$  is (in average) smaller than for Mergesort

- there are many more sorting algorithms: Bubblesort, Heapsort, Bucketsort, ...
- it can be shown that comparison-based sorting algorithms need at least a complexity of  $O(n \cdot \log(n))$

## 2.4 Data Structures

- another core topic of informatics
- we are especially interested in how to represent and store sets, vectors, matrices, and other structures that occur in numerical applications or algorithms
- important aspects:
  - do we have *static* (fixed and a priori known size) or *dynamic* (varying size) structures? examples of dynamic structures: a data base, an adaptively refined grid, a list of active grid points in an iterative algorithm
  - how fast is access to the different components of a structure?
  - how simple are operations like the insertion or deletion of components?
  - typically, there is more than one possibility of representation; example: store a set of numbers as a vector, as a simply chained list, as a double chained list, or as a binary tree

## 2.5 Languages

- languages like PASCAL, FORTRAN, or C need a *compiler* (translation program) in order to transform the source code into executable code (in the processor's machine language) that can be directly processed by the computer
- languages like BASIC, Perl, or other script languages do not use a compiler; here, the programs are *interpreted*; i.e., another program, the so-called *interpreter*, goes through the source code line by line and arranges the commands to be executed
- advantages and drawbacks: prototyping (test a new idea) can be done faster with interpreter languages (no need to wait for the compiled code), compilers allow more efficient runs (the compiler sees the whole program and can realize some optimizations)
  - ⇒ development of algorithms with interpreter languages, production codes with compiler languages
- among the user or high languages, very different concepts can be distinguished:
  - *procedural* or *imperative* languages:
    - a program is a sequence of single commands which are to be executed sequentially in the programmed order (possibly modified by jumps etc.)
    - ⇒ the focus is put on the underlying algorithm;
    - examples: FORTRAN, PASCAL, C – the standard programming languages
  - *declarative* languages:
    - here, the problem to be solved is in the centre of interest – it is more or less described by the code
    - examples: *logical* languages (PROLOG; are based on a set of rules) and *functional* languages (LISP; description of functional dependencies in a mathematical sense; a result is computed as soon as all input data are available – not when some command gives the order)
  - *object oriented* languages:
    - here, the focus is put on the object, i.e. on data in a broader sense; objects have *properties*, can execute some spectrum of *methods*, and communicate;
    - examples: SMALLTALK, EIFFEL, (in a restricted sense also) C++
- in this course, we concentrate on imperative languages
  - pseudo code for the presentation of algorithms
  - Maple's interpreter language for prototyping and small examples
  - C/C++ for real programs
- knowing the syntax of some specific programming language is important; however, the crucial thing is to understand the basic concepts, structures, and principles – which are more or less the same for most imperative languages
- If you have learned to program in one language, you will be able to write programs in some other language soon!
- “Do you speak English?” makes sense, “Do you speak C” is not that reasonable!



## Chapter 3

# Principles of Programming

- each programming language has its *syntax* defined in a *grammar* and its *semantics* assigning a specific meaning to the syntactic constructs
- the syntax is often complicated, and it takes some time to get used to it; however, we nearly always find the same or similar constructs
- languages like Maple’s internal programming language do have a small *syntactical overhead* only – their syntax is intuitive and “close to the formula”
- objective of this chapter: learn the basic programming constructs that are relevant for numerical purposes, and do this Maple-based
- a principal remark in advance: Maple itself is *no programming language*, but a software package for symbolic and numerical calculations
  - *symbolic* and coming first: algebraic formula manipulation, i.e. closed differentiation and integration, closed representation of roots etc. (the so-called *computer algebra*)
  - *numerically* and coming second: computing with values only

hence, and first of all, working with Maple means to use the various built-in functions; additionally, Maple offers a simple script language that allows to write your own programs – especially during the development or testing of algorithms – and, thus, to extend its functionality; this makes Maple attractive for us – the big codes for the number crunchers won’t be written in Maple

- note: basically everything we know from standard imperative languages is available in Maple, too!

### 3.1 General Remarks

- characters: small and capital letters, the 10 digits and 32 special characters; characters are combined to *symbols*:
  - *key words* or *reserved* words like ‘if’, ‘for’, ‘proc’, ‘read’, ‘and’, or ‘mod’; predefined functions (‘sin’), commands (‘expand’), or type names (‘integer’), however, are not reserved
  - operators like ‘+’, ‘and’, ‘<’, or ‘:=’ (assignment)

- strings are combinations of letters, digits, and `_`, if the first character is not a digit; predefined functions like `'sin'` are strings as well
- numbers: `45`, `0.36`
- punctuation characters structure a command or a program, respectively; the most important ones are the colon and the semicolon: both terminate a command – in case of a colon, the result is not shown
- the programming language of Maple knows – as other programming languages do – different kinds of commands: assignments, control structures (branchings, repetitions), commands for input and output, and others; as a speciality, expressions (see below) are commands, too (as its result, the value is indicated)

## 3.2 Constants, Types, Variables, Names, Expressions

- *constants* of a certain *data type* (integers, floats, logical values or Booleans, strings) have a fixed value that can not be changed: the number `45` or `'true'`, e.g.
- *variables* are of a certain type as well and consist of a *name* and of a current *value* that can be changed with the help of an *assignment* (use `:=` in Maple; equality as a relation (does  $x$  have the value 4?) is expressed by `x = 4`): `x := 4`;  
attention: *Pi* as a name for  $\pi$  is predefined, but not a constant (can be changed!)
- in some languages a variable's type has to be *declared* explicitly (*variable declaration*); Maple does not require that (avoids work, but may lead to some confusion sometimes: `x := 4` is supposed to be an integer, whereas `x := 4.` is interpreted as a float); with the command

```
type(x, integer);
```

one can check whether a variable  $x$  is an integer

- *arithmetic expression (formula)*:
  - (correct) construct of variables, constants, and expressions (recursive definition);  
examples: `37 * 14` or `7 * x^4 + 3 * x^3 - 5 * x + 2` (represented internally as a *syntax tree*; “correct” means correct with respect to the well-known mathematical formula syntax)
  - expressions are the simplest case of Maple commands; at the end, there is a colon (no output) or a semicolon (show result)
  - with an assignment, an expression can be given a new name: `f := 7*x^4+3*x^3-5*x+2`, for example; after that, an input of `'f'` will be answered by the expression on the right-hand side of the assignment
- by *substitution*, we can insert an expression in another one; as an example, we can give  $x$  in  $f$  some value

```
subs(x = 5, f);
```

or replace  $x$  by an expression itself:

```
subs(x = 2 + h, f);
```

with that, we can formulate difference quotients of  $f$  in  $x = 2$ , for example:

$$m := (\text{subs}(x = 2 + h, f) - \text{subs}(x = 2, f))/h;$$

trying to obtain the derivative by calling

```
subs(h = 0, m);
```

won't be successful: Maple does not accept the division by zero; however, Maple understands

$$\text{limit}(m, h=0);$$

and accepts

$$m1 := \text{simplify}(m); \quad \text{subs}(h = 0, m1);$$

here, 'simplify' results in cancelling down the  $h$  (note that the result of 'simplify' is not always considered by the programmer to be a simplification!);

fixing the abscissa  $x = 2$  was rash: everything shown above can be done for an arbitrary  $a$ , too; then, at the end,  $a$  can be substituted in the final expression, or we can perform the assignment

$$a := 2;$$

and let Maple show us the final expression once more;

(of course the whole story is superfluous, since Maple is able to differentiate)

- besides the *arithmetic expressions* studied so far, there are also *Boolean expressions* which can take the logical values 'true' or 'false':

$$5 \bmod 2 = 0;$$

- since Maple calculates symbolically, it tries to keep the (exact) formulas as long as possible; only if there is no alternative or if there is an explicit command, it will use (real and thus generally approximate) values – for Maple, 'evaluation' means symbolic simplification only:
  - 'sin(1);' makes Maple just write 'sin(1)' once more; 'sin(pi/6);' results in the output '1/2'
  - $5/3$  produces  $\frac{5}{3}$  and not – if you might expect – the non-precise 1.6666667;  $5.0/3$ , however, is evaluated in the usual sense (the point indicates that there are real numbers anyway)
  - $5 \bmod 2 = 0$  produces  $1 = 0$  and not 'false'
  - the concrete evaluation can be enforced via 'evalf' (arithmetic) or 'evalb' (Boolean)

### 3.3 Functions

- the operator  $- >$  is used to define a *function* (i.e. a mapping that assigns a value to each argument):

$$x - > x^2;$$

if we want to get the function's value at some given point, this can be computed via

$$(x - > x^2)(4);$$

or the function is given a name first

$$f := x - > x^2;$$

followed by a standard evaluation as we know it from mathematics:

$$f(4);$$

an alternative: make a simple substitution ( $f := x^2$ ;  $\text{subs}(x = 4, f)$ ); however, then, we would not have defined a function (with implications on differentiation etc.)

- functions of several variables are treated in the same way:

$$g := (x, y) \rightarrow x^2 - y^2; \quad g(2, 4);$$

- the function ‘if’ is a predefined one; there are three call parameters: a condition in the first argument (i.e. a Boolean expression) producing – after evaluation – a ‘true’ or a ‘false’), and two result expressions (the first for the case of ‘true’, the second for the case of ‘false’):

$$\text{power} := n \rightarrow \text{if}(n = 0, 1, 2 * \text{power}(n - 1));$$

it is important to note two things:

- this is already a recursive function with a condition of termination (although there is no need for a recursive formulation in that case, obviously) – hence, Maple can handle recursions!
- the apostrophes around ‘if’ are necessary, since ‘if’ is a *reserved word* and is used as a control structure, too (see below)

### 3.4 Data Structures 1: Sequences, Lists, Sets

- a *sequence* consists of several objects (generally, but not necessarily of the same type), separated by commas:

$$3, 4, 5;$$

with the function ‘seq’, sequences can be constructed:

$$s1 := \text{seq}(i^2, i = 1..6); \quad \text{or} \quad s2 := \text{seq}(\sin(Pi * i / 6), i = 0..6);$$

single elements or subsequences (index numbering starts with 1) can be accessed via

$$s2[3]; \quad \text{or} \quad s2[2..4];$$

concatenation leads to *one single* sequence:

$$s3 := \text{seq}(\text{seq}(j, j = 0..i), i = 0..3); \quad \text{produces} \quad 0, 0, 1, 0, 1, 2, 0, 1, 2, 3$$

- a *list* or a *vector* is a ordered sequence in angular brackets:

$$v1 := [s1]; \quad \text{or} \quad v1 := [1, 4, 9, 16, 25, 35];$$

again, assignments are possible, and again, type conformity is not enforced:

$$v1[3] := 6; \quad \text{or} \quad v1[3] := \text{blubber};$$

in contrast to sequences, now, the hierarchical structure survives a concatenation:

$$v3 := [\text{seq}([\text{seq}(j, j = 0..i)], i = 0..3)]; \quad \text{provides} \quad [[0], [0, 1], [0, 1, 2], [0, 1, 2, 3]]$$

access results in sublists or single components:

$$v3[3]; \quad \text{provides} \quad [0, 1, 2], \quad v3[3, 3]; \quad \text{and} \quad v3[3][3]; \quad \text{result in} \quad 2$$

lists can be both result of a function and call parameter:

$$f1 := x \rightarrow [x, 4 * x]; \quad \text{or} \quad f2 := v \rightarrow v[1] * v[2];$$

with the help of lists, vectors and matrices for linear algebra purposes can be constructed

- *sets* are not ordered, there are no multiple occurrences of elements; in Maple, sets are represented as sequences in set brackets:

$$m3 := \{s3\}; \quad \text{produces the set} \quad \{0, 1, 2, 3\}$$

there are the classical set operations ‘union’, ‘intersect’, and ‘minus’ (set difference)

## 3.5 Control Structures: Loops and IF-Clauses

*Control structures* control the order of the executed commands, they are in particular responsible for any deviation from the strict sequential order of execution; we consider *loops* and *if-clauses*:

- *loop*: repeated execution of a command; typically, a programming language offers three variants (at the end, ‘od’ or ‘enddo’ can be found):
  - *for-loop*: ‘for <counter> from <initial value> to <terminal value> do ... od (enddo)’
  - *while-loop*: ‘while <counter within range> do ... od (enddo)’
  - *repeat-loop*: ‘repeat ... until <counter outside range>’

Maple knows the first two only:

- *for-loop*:

```
for i from 100 to 102 do i, ithprime(i) od ;
```

shows the couples (100, 541), (101, 547), and (102, 557); after the execution of the do-loop, the counter has the first value outside the range (here 103); of course, this sequence can also be constructed via seq:

```
seq(ithprime(i), i = 100..102);
```

- *while-loop*:

now we compute the sum of the first 9 prime numbers:

```
s := 0; i := 1; while i < 10 do s := s + ithprime(i); i := i + 1 od ;
```

here, it is obvious that  $i = 10$  afterwards!

- *if-clause*: bifurcation or branching; there are two possibilities how to continue, depending on a condition; such a construct exists in all (useful) procedural languages (at the end, ‘fi’ or ‘endif’ can be found):  
‘if <condition> then <do this> else <do that> fi (endif)’  
in Maple, ‘if’ comes in two versions:

- *functional* ‘if’, i.e. ‘if’ as a function with three arguments: already introduced

```
‘if’(x < 0, 0, sqrt(x));
```

- *if-clause*:

```
if x < 0 then 0 else sqrt(x) fi;
```

- for more than two alternatives, ‘elif’ can be used as an abbreviation, in order to avoid too many nested if-clauses:  
‘if <condition 1> then <statement 1> elif <condition 2> then <statement 2> elif ... else <condition n+1> fi’
- loops and if-clauses can be nested in an arbitrary way

## 3.6 Procedures

- essence of a procedural language, allow for a hierarchical structuring (parts of the program are “outsourced” to separate *subroutines*) and, hence, the clear design of even large program systems

- simplest possibility has already been mentioned: from a formula (an expression), we get a function via *formal parameters* and via the operator `->`:

$$f := (x, y) -> x^2 + \sin(y);$$

afterwards, the (complicated) expression can be replaced by the (concise) *function call* `f(3, a)` etc. (i.e. a *nameless* procedure)

- for more than one expression (when reading a set of numbers, e.g., as a sorting routine's first step), a *procedure* is defined; for that, we need
  - *proc*-operator: denotes the begin of a procedure (i.e. of an independent block within a program, which is entered by an explicit call only)
  - list of *formal parameters*: the quantities that are *handed over* during the call; they can (but need not) be *declared*:

$$sum := proc(a :: integer, b :: integer)$$

in this case, Maple makes a type check (do the values handed over and the declared types fit together?) and communicates possible inconsistencies (error message)

- possibly list of *local variables*: variables visible *within* the procedure only (typically auxiliary variables like loop counters)
  - possibly list of *global variables*: variables visible *outside* the procedure, too (attention: the procedure can change global variables (*side effects*))
  - list of *options*: for example 'option trace', in order to print everything computed in the procedure (would be omitted otherwise); or 'option remember' (call parameters and result are stored in a table; in case of a new call, the table is checked first before any call is executed)
  - the *procedure's body*: a sequence of commands – the things that have to be done by the procedure
  - an 'end' at the end
- a procedure call's result is *the result of the last executed formula in the procedure* (as long as no 'RETURN(...)' prescribes something else explicitly)

- a small example:

```
count := proc(animals)
local i;
global MUH, MAEH;

for i in animals do
if i="cow" then MUH:=MUH+1
elif i="sheep" then MAEH:=MAEH+1
fi
od;
MUH, MAEH
end;
```

now, in the *main program* (or in the procedure, from where 'count' is called), some initial values are given

```
MUH :=0;    MAEH := 0;    i := 0;
```

and the procedure 'count' is called:

```
erg := count(["cow", "sheep", "hen", "cow"]);
```

we get the correct result

```
erg := 2, 1
```

- some remarks concerning the example:
  - *MUH* and *MAEH* are *global* variables – their values were changed within the procedure, and this is now visible from outside, too (check the values *after* the call of ‘count’)
  - *i*, however, is defined as a local variable within the procedure only – hence, the *i* in the procedure has no influence on the outer one carrying the value zero (again, check the value immediately after the function call: the value of *i* has remained zero from the outside point of view)
- assigning values to formal parameters in the procedure is prohibited in general (since the call parameters are evaluated during the call and, hence, afterwards no longer assignable); nevertheless, it would be nice to be allowed to do this sometimes: a *result parameter* allows to avoid the possibly dangerous use of global variables; the solution: use apostrophes to prevent the evaluation of call parameters:

```
primsum := proc(n,m,list)
local i,s;
list := [seq(ithprime(i),i=n..m)];
s := 0;
for i in op(list) do s:=s+i od;
end;
```

with an empty list (defined outside),

```
erg := [];
```

an error message is produced by

```
primsum(25, 30, erg);
```

whereas

```
primsum(25, 30, 'erg');
```

provides the desired sum (for the meaning of ‘op()’, see the next section; with ‘for i in list’ only the final sum is returned as a result)!

- with the help of the command ‘RETURN(...)’; the immediate return to the calling part of the program is enforced; here, the argument is a sequence, which is the result of the function call statement (this is helpful, for example, if there are if-clauses in the procedure)

## 3.7 Data Structures 2: Tables, Arrays, Trees, Pointers

in the following, we present some more sophisticated data structures

- *tables*:
  - data type ‘table’ for the representation of data in tables
  - could be done via nested lists, too:

```
w1:=["London", "cloudy", 6]; w2:=["Paris", "sunny", 8]; cities:=[w1, w2];
for city in cities do if city[3]<7 then print(city[1]) fi od;
```

the additional (and not necessary) index should be avoided – it does not contribute anything to the associative identification of the data records

- for that, there are *tables* (*associative arrays*, *hash tables*): here, one field (component) of the table is directly used as an index (in the example, the name of the city, for example):

```
weather:=table(); temp:=table();
weather["London"]:= "cloudy"; temp["London"]:=6;
weather["Paris"]:= "sunny"; temp["Paris"]:=8;
```

occupied or free entries can be accessed:

```
weather["Paris"]; weather["Rome"];
```

one gets

```
"sunny"
```

and the not really helpful *weather<sub>Rome</sub>*;

tables have special evaluation rules: a name denoting a table is not evaluated automatically, but only via the command ‘op()’:

*weather*; results in *weather*,      op(*weather*); provides the entries explicitly

- list entries can be turned into table entries:

```
for city in cities do
  weather[city[1]]:=city[2];
  temp[city[1]]:=city[3];
od;
```

- via ‘indices(temp);’ one gets a list of all indices occurring in the table (each of which packed into an own list):

```
["London"], ["Paris"]
```

‘op()’ provides this without the extra packing:

```
for i in indices(temp) do print(op(i)) od;
```

- tables as a standard data structure are more often the exception in programming languages (Perl and Java support them as well)

- *arrays*:

here, we have the Maple data type ‘array’, a special case of tables with integer sequences as indices;

differences with respect to lists:

- arrays have a fix *dimension*, the number of integers in the index (*components*) is the same for each data record or entry (for nested lists, this is not necessarily true)
- access to an element of a multi-dimensional array can be done via ‘A[4,5]’, but not via ‘A[4][5]’
- due to their bigger homogeneity, arrays are implemented in a much more efficient way than lists are

during initialization, the index range must be declared for each dimension:

```
v := array(1..3);    A := array(1..3, 1..2);
```

with the help of the ‘print’-command, arrays can be displayed (before values have been assigned, the array is displayed with variable elements  $A_{1,1}$  etc.);

the lower bounds of the index ranges don't have to be 1; if they are, a 2-dimensional array is accepted as a matrix (type 'matrix') and available for the operations of linear algebra (most of them in the package 'linalg', to be loaded with 'with(linalg);'):

```
evalm(v & * A);
```

for example, evaluates the matrix-vector product  $v^T A$ ; if one tries  $Av$ , an error message is given (the index ranges don't fit together)

- *trees*:

sometimes, data structures with specific properties are needed:

- *dynamic*: the form and size of the single objects of some type may change (arrays, however, are *static*!)
- *recursive*: often helpful (compact representation, better suited for recursive algorithms); example list: a list is either empty, or it consists of a first element and a list (the rest)

as an important example, we consider *trees*:

- a tree is either empty (name 'EMPTY') or a list with two elements: an arbitrary expression (the contents of the *root* of the tree) and a list of trees (the so-called *sons* or *son nodes*)

- some examples:

```
b1 := [node1, []];
b2 := [node2, []];
b3 := [node3, [b1, b2]];
b4 := [node4, [b3, EMPTY, [node5, []]]];
```

- a routine to print a tree ('printf' is a variant of 'print'):

```
print_tree := proc(b, depth::nonnegint)
local i, j;
for j from 1 to depth do printf("| ") od;
if b=EMPTY then printf("EMPTY\n")
else
  printf("contents: %a\n", b[1]);
  for i in b[2] do print_tree(i, depth+1) od
fi
end;
```

the call 'print\_tree(b4,0)' produces the output

```
contents: node4
| contents: node3
| | contents: node1
| | contents: node2
| EMPTY
| contents: node5
```

- Where do we need and meet tree structures?

- \* in the description of a grammar's syntax: arithmetic expressions have – as already mentioned – a tree structure
- \* in *search trees*: all elements to the root's left have a smaller value than the root has, those to the root's right have a bigger one
- \* ...

- *pointers (references)*:

with dynamic data structures, the problem may arise that the consecutive memory reserved during definition or declaration turns out to be not sufficient later (the component ‘EMPTY’ in b4 is replaced by a bigger tree, for example): the allocation and storing have to be reorganized, or parts of the structure are stored elsewhere; *pointers* can avoid this:

- if we use pointers, a tree is either empty or a list of two elements: the node’s value (as before) and a list of two pointers to trees (de facto a list of addresses of the places in memory where these son tree’s roots are stored);  
advantage: pointers don’t change their size – no matter, how big the structures are to which they point (full tree or empty tree)
- hence, one has to distinguish the object itself (the tree) and a pointer to an object; of course, this proceeding can be iterated (“pointers to pointers to objects”)

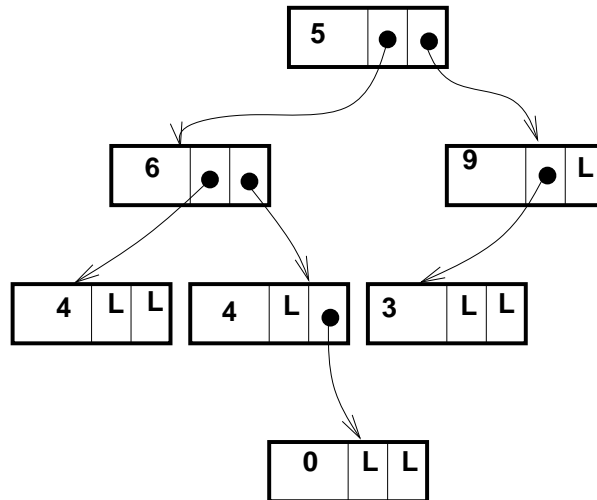


Figure 3.1: Example of a pointered binary tree (a tree with exactly two (possibly empty) son nodes)

- with pointers, not only lists or trees, but also much more complicated data structures (graphs etc.) can be described
- with pointers, working with dynamic and complicated objects often becomes simpler: instead of applying operations to whole objects, often the pointers have to be manipulated or redirected

Maple does not know pointers explicitly; internally, however, trees are realized via pointers

## 3.8 In- and Output

see exercises!

# Chapter 4

## Basics of Numerical Analysis

Let us start with some principal remarks concerning phenomena and notation in numerical programming.

### 4.1 Floating Point Numbers

To calculate numerically means to work with concrete numbers – hence, we need these first:

- the set  $\mathbb{R}$  of the real numbers is *not bounded* (there is no biggest or smallest number) and *dense* (between two different real numbers, there is always another one); the number of real numbers is infinite, even *non-countable*
- the set  $\mathbb{Z}$  of the integer numbers is discrete with a constant distance between two neighbours, but not bounded and of infinite cardinality as well
- the set of numbers that are *representable exactly* is necessarily finite, discrete, and bounded:
  - an *integer arithmetic* works with integer numbers only, typically in a range  $[-N, N]$  or  $[-N + 1, N]$
  - a *fixed point arithmetic* works with decimal point numbers of a fixed number of digits *in front of* and *behind* the decimal point, typically in a range  $[-999.9999, 999.9999]$ , within which there are fixed distances between neighbouring numbers (as with  $\mathbb{Z}$ )  
 $\Rightarrow$  obvious drawback: fixed domain of numbers, frequent overflow
  - a *floating point arithmetic* works with decimal point numbers, too, but the relative position of the decimal point and, hence, the size and position of the representable domain of numbers are variable
- *normalized  $t$ -digit floating point numbers to basis  $B$*  ( $B \in \mathbb{N} \setminus \{1\}$ ,  $t \in \mathbb{N}$ ):

$$\mathbb{F}_{B,t} := \{M \cdot B^E : M = 0 \text{ or } B^{t-1} \leq |M| < B^t, \quad M, E \in \mathbb{Z}\}$$

- *mantissa  $M$ , exponent  $E$*
- normalization (no leading zero) provides uniqueness of representation
- limitation of exponent leads to *machine numbers*:

$$\mathbb{F}_{B,t,\alpha,\beta} := \{f \in \mathbb{F}_{B,t} : \alpha \leq E \leq \beta\}$$

- the quadruple  $(B, t, \alpha, \beta)$  characterizes the system of machine numbers completely, in computers such a system is used almost always (and is, hence, fixed for a machine or for some program run); all that has to be stored of some number is  $M$  and  $E$
- often, floating point numbers and machine numbers are used equivalently; usually,  $B$  and  $t$  are clear due to the context; therefore we will only write  $\mathbb{F}$  in the following

some examples:

$$126880 \cdot 10^{-34} : \quad B = 10, t = 6, M = 126880 \in [10^5, 10^6[, E = -34$$

$$40001 \cdot 2^3 : \quad B = 2, t = 16, M = 40001 \in [2^{15}, 2^{16}[, E = 3$$

$$-54410 \cdot 4^0 : \quad B = 4, t = 8, |M| = 54410 \in [4^7, 4^8[, E = 0$$

( $54410_{10}$  corresponds to  $31102022_4$ )

- *resolution:*

maximum relative distance between two neighbouring floating point numbers:

$$\frac{(|M| + 1) \cdot B^E - |M| \cdot B^E}{|M| \cdot B^E} = \frac{1 \cdot B^E}{|M| \cdot B^E} = \frac{1}{|M|} \leq B^{1-t} =: \varrho$$

note that the maximum *absolute* distance is not constant: with increasing absolute values of the numbers, the “mesh width” of the discrete grid of floating point numbers (logarithmic scale!) also increases  $\Rightarrow$  the domain of representable numbers is much larger!

- *smallest positive machine number:*  $\sigma := B^{t-1} \cdot B^\alpha$
- *largest machine number:*  $\lambda := (B^t - 1) \cdot B^\beta$
- example: *IEEE format* (Institute of Electrical and Electronics Engineers, determined in the US norm ANSI/IEEE-Std-754-1985):

level	$B$	$t$	$\alpha$	$\beta$	$\varrho$	$\sigma$	$\lambda$
single precision	2	24	-149	104	$2^{-23}$	$2^{-126}$	$\doteq 2^{128}$
double precision	2	53	-1074	971	$2^{-52}$	$2^{-1022}$	$\doteq 2^{1024}$
extended precision	2	64	-16445	16320	$2^{-63}$	$2^{-16382}$	$\doteq 2^{16384}$

single precision corresponds to about 6-7 decimal digits, with double precision, about 13 digits are sure

- apart from that, there are several special cases:
  - *NaN* (not-a-number): non-defined value, as *quiet* (is inherited with alert) or *signaling* (causes alert or error message)
  - *exponent overflow*: absolute value of the number is larger than  $\lambda$
  - *exponent underflow*: absolute value of the number is smaller than  $\sigma$

against these types of leaving the allowed range of numbers, explicit measures can not be taken in every numerical program: in these cases, the arithmetic of the computer is supposed to inform about their occurrence (and not to go on calculating until the rocket explodes)

Although floating point numbers are a sophisticated construct to extend the domain of representable numbers, the latter is still discrete – we make a *representation*, *approximation*, or *round-off error*. For a theoretical analysis, we look at the process of rounding a bit more in detail:

- consider an arbitrary  $x \in \mathbb{R}$ ;  $x$  has in  $\mathbb{F}$  exactly one left and one right neighbour:

$$f_l(x) := \max\{f \in \mathbb{F} : f \leq x\}, \quad f_r(x) := \min\{f \in \mathbb{F} : f \geq x\}$$

for the special case of  $x \in \mathbb{F}$ , we have  $f_l(x) = f_r(x) = x$

- an explicit formula for the neighbours of  $x > 0$  can be obtained via the notation

$$x =: (M + \delta) \cdot B^E, \quad 0 \leq \delta < 1;$$

then,

$$f_l(x) := M \cdot B^E \quad \text{and} \quad f_r(x) := f_l(x) \text{ if } \delta = 0 \text{ and } f_r(x) := (M + 1) \cdot B^E \text{ otherwise}$$

- concerning the round-off mapping  $\text{rd}: \mathbb{R} \rightarrow \mathbb{F}$  itself, useful requirements are:

- surjective:  $\forall f \in \mathbb{F} \exists x \in \mathbb{R} : \text{rd}(x) = f$
- idempotent:  $\text{rd}(f) = f \quad \forall f \in \mathbb{F}$
- monotone:  $x \leq y \Rightarrow \text{rd}(x) \leq \text{rd}(y) \quad \forall x, y \in \mathbb{R}$

widespread round-off strategies:

- *round down*:  $\text{rd}_-(x) := f_l(x)$
- *round up*:  $\text{rd}_+(x) := f_r(x)$
- (*correct*) *rounding*:  $\text{rd}_*(x) := \begin{cases} f_l(x) & \text{if } x \leq (f_l(x) + f_r(x))/2 \\ f_r(x) & \text{if } x \geq (f_l(x) + f_r(x))/2 \end{cases}$   
plus an additional rule for the case in the midpoint (for example, let the resulting mantissa get even)
- *truncation*:  $\text{rd}_0(x) := \begin{cases} f_l(x) & \text{if } x \geq 0 \\ f_r(x) & \text{if } x \leq 0 \end{cases}$

all four round-off rules presented here are surjective, idempotent, and monotone!

- rounding down, rounding up, and truncation are also called *directed* rounding
- a bound for the *relative round-off error*:

$$\forall x \in \mathbb{R} \text{ holds } \text{rd}(x) = x(1 + \varepsilon),$$

where the *relative error*  $\varepsilon$  fulfils

$$|\varepsilon| \leq \begin{cases} \frac{1}{2}\varrho & \text{(correct rounding)} \\ \varrho & \text{(directed rounding)} \end{cases}$$

## 4.2 Floating Point Arithmetic, Round-off Errors

when rounding a simple number, the exact value is known; in nested computations, the situation is different: starting from the first step, we only work with approximations

$\Rightarrow$  important: prevent the fast increase of accumulated errors, use a “clean” floating point arithmetic

- exact execution of the arithmetic elementary operations  $* \in \{+, -, \cdot, /\}$  in the system  $\mathbb{F}$  of floating point numbers is generally not possible (even for arguments from  $\mathbb{F}$ : how shall the sum  $1234 + 0.1234$  be represented exactly if we compute with four digits only?)
- notation:

- $a * b \in \mathbb{R}$  and generally  $\notin \mathbb{F}$  for the *exact* result in  $\mathbb{R}$
- $a \dot{*} b \in \mathbb{F}$  for the *actually computed* result of the addition

as already mentioned, we do not care about exponent overflow or underflow

- what is optimal in the sense of floating point arithmetic? of course, if the *computed* result is just the *rounded exact* result (since we would anyway make this error when forcing the exact result – if known – into the frame of  $\mathbb{F}$ ; nothing better is possible):

$$a \dot{*} b = \text{rd}(a * b) \quad \forall a, b \in \mathbb{F}, \quad \forall * \in \{+, -, \cdot, /\}$$

such an *ideal arithmetic* is possible (the IEEE standard requires it for the binary floating point arithmetic for the four elementary operations and even for the square root, and with respect to all three precision levels and to all four round-off strategies)!

- for the ideal arithmetic, we get bounds for the round-off error of our arithmetic operations:

$$\forall a, b \in \mathbb{F} : \quad a \dot{*} b = \text{rd}(a * b) = (a * b)(1 + \varepsilon(a, b))$$

with

$$|\varepsilon(a, b)| \leq \bar{\varepsilon} = \frac{1}{2}\varrho \text{ or } \varrho \quad (\text{depending on the rounding})$$

$\bar{\varepsilon}$  is called *machine accuracy* or *computing accuracy* and depends only on the parameters  $B$  and  $t$  of the floating point arithmetic

- though it is technically possible, many, but not all computers have an ideal arithmetic; there are compromises:
  - *strong hypothesis*: there is a  $\tilde{\varepsilon} = O(\varrho)$ , describing the relative error in every case (holds for most machines):

$$\forall a, b \in \mathbb{F} : \quad a \dot{*} b = (a * b)(1 + \varepsilon(a, b))$$

with

$$|\varepsilon(a, b)| \leq \tilde{\varepsilon}$$

- *weak hypothesis*: with the above  $\tilde{\varepsilon}$  only

$$a \dot{*} b = (a(1 + \varepsilon_1)) * (b(1 + \varepsilon_2)) \quad \text{with } |\varepsilon_1|, |\varepsilon_2| \leq \tilde{\varepsilon}$$

still holds (no functional dependence of the computed result on the exact one: at least this holds for almost all computers)

- if the square root is hardware-implemented, its calculation can be done as fast and precise as the division's
- so far, we did not care about exponent overflow and underflow; the *extended precision* aims at providing more security: so-called *protection digits* prevent the danger of leaving the domain of numbers in many cases (think of  $\sqrt{a^2 + b^2}$ , for example)
- also take care of the implementation of comparisons: if  $a < b$  is realized via a comparison of the difference  $a - b$  with 0, exponent underflow threatens!

now, we can try to analyze the round-off errors for complicated numerical computations; for such a *round-off error analysis*, typically, a strong hypothesis is supposed

### 4.3 Round-off Error Analysis

- a numerical algorithm is a finite sequence of elementary operations with a fixed order of execution; floating point arithmetic provides an additional source of error; therefore, we have the following objectives of an algorithm:

- it shall provide a precise approximation (small negative influence of the discretization)
- it shall provide this fast (efficiency)
- no catastrophes during computations (small influence of round-off errors)

the last aim requires some *a-priori analysis*: which bounds can be given (beforehand, for all possible concrete numbers) for the overall error, if a certain quality of the elementary operations is assumed (worst case)?

- two straightforward strategies:
  - *forward analysis*: interpret the computed result as a disturbed exact result, i.e. comparison with the latter (nice, since this leads directly to the relative error, but in general very difficult due to error correlations)
  - *backward analysis*: interpret the computed result as an *exact* result for modified input data (simpler, the usual approach, with a nice possibility of evaluation: if the disturbances of the input data derived from error analysis are of the order of magnitude of their uncertainty (generally realistic anyway – think of measurements), then there are no problems with this algorithms (with respect to this aspect)
- the weak hypothesis only allows for a backward analysis, the strong one (as well as the ideal arithmetic) allows for both a backward interpretation

$$a \dot{+} b = a(1 + \varepsilon) + b(1 + \varepsilon) \quad \text{or} \quad a \cdot b = (a\sqrt{1 + \varepsilon}) \cdot (b\sqrt{1 + \varepsilon})$$

and a forward interpretation:

$$\text{the relative error of the computed result is always } \leq \tilde{\varepsilon}$$

- although ideal arithmetic and strong hypothesis are looking equivalent, the first one is better (the actually occurring errors (not the bounds!) are often significantly better or smaller)

we study a simple example of a round-off error analysis:

- problem: compute the polynomial value  $y := \sum_{i=0}^n a_i x^i$  for a  $x$
- algorithm: Horner scheme

$$y := (\dots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3}) \dots + a_1)x + a_0$$

or

```
y := a[n];
for i:=n-1 downto 0 do y:=y*x+a[i] od;
```

- for each step, the strong hypothesis provides

$$\tilde{y} := (\tilde{y} \cdot x \cdot (1 + \mu_i) + a_i) \cdot (1 + \alpha_i)$$

with  $\alpha_i$  and  $\mu_i$  bounded by  $\tilde{\varepsilon}$ ;  
a transformation provides

$$\tilde{y} = \sum_{i=0}^n \tilde{a}_i x^i$$

with

$$\tilde{a}_i := a_i \cdot (1 + \alpha_i) \cdot (1 + \mu_{i-1}) \cdot \dots \cdot (1 + \alpha_0), \quad \alpha_n := 0$$

i.e.: the *computed* value  $\tilde{y}$  can be interpreted as *exact* evaluation of a polynomial with slightly modified coefficients

## 4.4 Condition

crucial, but, mostly, only qualitatively defined numerical term: how important is the result's sensitivity to changes of the input data?

$\Rightarrow$  very important and often misunderstood: the *condition* is a property of the *problem*, not of the *algorithm*!

but one step after the other:

- numerical problem  $p$  maps input data  $x$  to result data  $y$  (both  $x$  and  $y$  are vectors, in general);  
examples:
  - solve system of linear equations  $Ax = b$ :  
input data are  $A \in \mathbb{R}^{n,n}$  and  $b \in \mathbb{R}^n$ , result is  $x \in \mathbb{R}^n$
  - compute zeros of a polynomial of degree  $n$  (real coefficients):  
input data are the coefficients  $a_0, \dots, a_n$ , result are the  $n$  (complex) zeros
  - compute eigenvalues  $A \in \mathbb{R}^{n,n}$ , i.e.  $\lambda$  complex with  $Ax = \lambda x$  for an eigenvector  $x \neq 0$ :  
input is the matrix  $A$ , output are all  $\lambda$  with the above property; from linear algebra, we know that the eigenvalues of  $A$  are roots of  $A$ 's *characteristic polynomial* of degree  $n$   
 $\Rightarrow$  thus, the problem consists of two natural subproblems: first, calculate the coefficients of the polynomial, and then, compute the polynomial's roots; hence, we have a concatenation of mappings (however, here the frontiers between problem and algorithm are not that clear any more: the structuring of the problem is already an algorithm!)
- why do we deal with *disturbed input data*  $\delta x$  at all? since these are typically only available with some uncertainty, and since, consequently, disturbances of the result  $\delta y$  can not be avoided  
 $\Rightarrow$  obviously:
  - for a *well-conditioned problem* (small  $\delta x$  lead to small  $\delta y$  only), small  $\delta x$  don't have a severe impact on the result; hence, we can look for a suitable algorithm without big restrictions
  - for a *badly-conditioned problem* (even small  $\delta x$  can lead to large  $\delta y$ ), which reacts in a very sensitive way to disturbances  $\delta x$ : here, the choice of an appropriate algorithm is much more important
- the simplest case: the condition of arithmetic operations (obviously problems of numerics and, thus, carrying a condition);  
for that, we introduce condition as the difference of the exactly computed result for exact input data and the exactly computed result for disturbed input data (note that these considerations have nothing to do with inexact computations ( $*$  instead of  $*$ )):

$$\delta(a * b) := (a + \delta a) * (b + \delta b) - a * b$$

furthermore, we consider the relative condition

$$\delta(a * b) / (a * b);$$

in the following table: study the order (higher order terms in the disturbances  $\delta a$  and  $\delta b$  are neglected)!

	absolute	relative
$\delta(a \pm b)$	$= \delta a \pm \delta b$	$(\delta a \pm \delta b)/(a \pm b)$
$\delta(a \cdot b)$	$\approx b \cdot \delta a + a \cdot \delta b$	$\delta a/a + \delta b/b$
$\delta(a/b)$	$\approx \delta a/b - a \cdot \delta b/b^2$	$\delta a/a - \delta b/b$
$\delta(\sqrt{a})$	$\approx \delta a/(2\sqrt{a})$	$\delta a/(2a)$

one can see: for multiplication, division, and square root, the relative condition (right column) stays within the range of the relative disturbances of the input data, thus nothing serious happens; the picture is different for addition and subtraction: if  $a$  and  $b$  have different signs in the addition with  $a + b$  close to zero, we get some *cancellation* (leading non-zeros disappear), and the relative error may get large:

$$1000000 - 999999 = 1, \quad (1000000 + 1) - (999999 - 1) = 3$$

hence, the relative error (here called relative condition) is

$$\frac{\delta(a - b)}{a - b} = \frac{3 - 1}{1} = 2,$$

although the relative disturbances of the input are only  $O(10^{-6})$ ; in the case of perfect cancellation (exact result is zero): then, the relative error becomes arbitrarily large!!

$\Rightarrow$  in this sense, cancellation is a problem, and real subtractions should be avoided in algorithms – if possible

- ill-conditioned problems are very difficult to treat numerically – sometimes, it is not possible at all: each error in the input data, each disturbance accumulated so far (due to round-off errors in preceding calculations) may result in a completely false result!
- in most cases, condition is defined not in the simple way we did it above (via a simple difference or the relative error, resp.), but via the partial derivative of the result with respect to the input data:

$$\text{condition}(p(x)) := \frac{\partial p(x)}{\partial x};$$

partitioning problem  $p(x)$  into two (or more) subproblems, the chain rule leads to

$$\text{condition}(p(x)) = \text{condition}(r(q(x))) = \frac{\partial r(z)}{\partial z} \Big|_{z=q(x)} \cdot \frac{\partial q(x)}{\partial x};$$

of course, the overall condition of  $p(x)$  does not depend on the substructuring, but the condition of the subproblems depends on the respective substructuring;

problem case: a well-conditioned  $p$  with excellently-conditioned first part  $q$  and poorly-conditioned second part  $r$ ; the (even if small) errors from the first may cause a catastrophe in the second step;

not yet clear? consider

$$\frac{\partial p}{\partial x} = O(10^{-10}), \quad \frac{\partial r}{\partial z} = O(10^{10}), \quad \frac{\partial q}{\partial x} = O(10^{-20});$$

despite the very good condition of the first part  $q$ , a numerical solution of  $q$  will entail some round-off errors, which will cause the result  $z = q(x)$  to have an accuracy of  $O(10^{-14})$  only

(double precision); now,  $r$  will let explode these errors to  $O(10^{-4})$ , and suddenly we have only four digits, although  $p$  is well-conditioned!

$\Rightarrow$  the combination of *condition* of the problem and *input* or *round-off errors* during the computations is responsible for troubles!

- example: eigenvalues of a real symmetric matrix  $A = A^T \in \mathbb{R}^{n,n}$ :
  - the overall problem is well-conditioned: small errors in the matrix entries lead to small errors in the eigenvalues (a very pleasant situation)
  - now, we decompose the problem into the two steps (see above) “construct the characteristic polynomial” and “calculate its roots” (a strategy suggested by linear algebra: that’s the way eigenvalues are defined)
  - the first part is perfectly-conditioned, the second miserably: already errors in the last significant digit of the coefficients of the polynomial may lead to a completely different shape and, hence, a completely different set of roots; the resulting overall result can not be used for anything (a numerical disaster)
    - $\Rightarrow$  the eigenvalues of  $A$  must not be computed this way!
  - this holds generally: avoid the determination of roots as a subtask in a numerical problem!
  - by the way, and fortunately, there are alternatives for the computation of eigenvalues, without ill-conditioned subproblems!

## 4.5 Stability

Now, we turn to the characterization of numerical algorithms and of the results delivered by them:

- *acceptable result*:
  - we’ve learned already: input data can be disturbed, they may be – mathematically spoken – determined only up to some tolerance, i.e. are located in some environment  $\{\tilde{x} : \|\tilde{x} - x\| < \varepsilon\}$  of the exact input  $x$ ; each such  $\tilde{x}$  must be considered to be equivalent to  $x$ ; thus, it is natural to regard each such  $p(\tilde{x})$  as an *acceptable result* for the desired exact  $p(x)$ ;
  - an approximation  $\tilde{y}$  for  $y = p(x)$  is therefore called *acceptable*, if it is the exact solution for one of the above  $\tilde{x}$  (often formulated a bit weaker)
  - the error  $\tilde{y} - y$  has various sources: besides round-off errors, we are confronted with *discretization*, *truncation* or *iteration* errors (series and integrals are approximated by sums, derivatives by difference quotients, iterations are stopped after some steps etc.)
  - the proof of acceptability can be done via some so-called *control calculus* – similar to backward analysis for round-off errors
- *stable algorithm*:
  - a numerical algorithm is called (*numerically*) *stable*, if it produces acceptable results for all allowed inputs that are disturbed of machine accuracy  $\bar{\varepsilon}$  only, including all kinds of occurring errors
  - a stable algorithm can produce large errors (for example, if the problem to be solved is ill-conditioned – this shows the interdependences of the different terms)

- the elementary arithmetic operations are numerically stable (weak hypothesis supposed); for concatenations, this is not necessarily true (otherwise, everything would be numerically stable)
- a simple example of an unstable algorithm:  
compute the positive root of

$$x^2 + 2px - q$$

for the input  $p = 500, q = 1$ ; the well-known formula

$$\sqrt{p^2 + q} - p$$

provides  $\sqrt{250001} - 500 = 0.00099999900\dots$ , with 5 digits only, however, we get zero (check with Maple!); but, the only chance for zero to be a root of  $p$  is to remove the ‘-1’, which is, unfortunately, much more than a modification of the order of machine accuracy  $-1 \cdot 10^{-5}$ ; hence, the computed result is not acceptable, and the algorithm is not numerically stable (although there are no problems at all for  $p = q = 1$  – even if we use 5 digits only!);

a simple transformation of the above formula for the positive root into

$$\frac{q}{\sqrt{p^2 + q} + p}$$

leads to a stable algorithm!

Note that we will basically look for numerically stable algorithms in the following sections!

## 4.6 Summary

Since the things discussed so far are essential for a deep understanding of numerical algorithms, we repeat the most important points:

- *floating point numbers* are the numbers we are using as a replacement of real numbers on computers
- *floating point arithmetic* denotes the calculus with these numbers; many basic properties are determined by the hardware (machine accuracy, IEEE standard, ideal/strong/weak arithmetic) or to be fixed by the user (single/double precision etc.)
- the discrete frame results in the necessity to *round*; rounding entails *round-off errors*
- in order to be able to estimate the influence of round-off errors a priori, some *round-off error analysis* has to be applied:
  - *forward interpretation*: consider the computed result as a function of the exact one; the relative error of the computed result shall be small (strong hypothesis, possible only in care situations)
  - *backward interpretation*: consider the computed result as the exact one for slightly disturbed input (weaker, usual proceeding)
- *condition*: property of the problem, indicates the result’s sensitivity with respect to modifications of the input; ill-posed problems are the numerical nightmare
- concerning elementary operations, only the real subtraction (different signs) is critical: *cancellation of digits* may produce large relative errors
- in a sequence of (sub)problems, the ill-otherwise, round-off errors from previous tasks might explode)

- since there is often some uncertainty about the input, a result provided by some numerical algorithm is called *acceptable*, if it is exact for some slightly disturbed input data; algorithms that produce acceptable approximations are called *numerically stable*
- the elementary operations are numerically stable, if – at least – the weak hypothesis is valid; nothing general can be said about concatenations (otherwise, everything would be stable)

## Chapter 5

# Direct Solution of Systems of Linear Equations

### 5.1 Preparatory Remarks

- *Numerical linear algebra* was at the origin of numerical algorithms; this covers tasks like matrix-vector product, eigenvalues, solution of systems of linear equations etc.) – all of them appear as modules in a huge variety of numerical problems. As an important example, we study the direct solution of systems of linear equations,

$$\text{for } A = (a_{i,j})_{1 \leq i,j \leq n} \in \mathbb{R}^{n,n}, \quad b = (b_i)_{1 \leq i \leq n} \in \mathbb{R}^n, \quad \text{find } x \in \mathbb{R}^n \text{ with } A \cdot x = b,$$

which play a crucial part in numerics:

- interpolation (see Section 6)
- ordinary differential equations (see Section 9)
- partial differential equations (see Section 10)
- ...
- concerning the given problem (i.e. the matrix), we distinguish
  - *full* matrices: the number of non-zeros in  $A$  is of the order of the number of matrix entries, i.e.  $O(n^2)$
  - *sparse* matrices: here, the zeros clearly dominate the non-zeros (typically  $O(n)$  or  $O(n \log(n))$  non-zeros); often, sparse matrices have some *structure*: diagonal matrices, tridiagonal matrices ( $a_{i,j} = 0$  for  $|i - j| > 1$ ), general band structure ( $a_{i,j} = 0$  for  $|i - j| > c$ ) etc.
- concerning the solution techniques, we distinguish
  - *direct* solvers: provide the exact solution  $x$  (modulo round-off errors)
  - *indirect* solvers: provide, starting from some initial guess  $x^{(0)}$ , *iteratively* a sequence of (hopefully better and better) approximations  $x^{(i)}$ , without reaching  $x$  in general
- For the following, it is reasonable to restrict ourselves to *invertible* or *non-singular* matrices  $A$ , i.e.  $\det(A) \neq 0$  or  $\text{Rang}(A) = n$  or  $Ax = 0 \Leftrightarrow x = 0$ . Two straightforward approaches must never be chosen for numerical purposes:
  - $x := A^{-1}b$ , i.e. constructing the inverse of  $A$  explicitly

- use of *Cramer's rule* via the determinants of  $A$  and the  $n$  matrices which we get from  $A$  by replacing one column by the right-hand side  $b$
- As always, something is especially important in numerical linear algebra: Look carefully at the problem before computing, since things can be simplified significantly often. Already the simple term

$$y := A \cdot B \cdot C \cdot D \cdot x, \quad A, B, C, D \in \mathbb{R}^{n,n},$$

can be calculated in a rather stupid way via

$$y := (((A \cdot B) \cdot C) \cdot D) \cdot x$$

with  $O(n^3)$  operations (matrix-matrix product!) or in a smarter way via

$$y := A \cdot (B \cdot (C \cdot (D \cdot x)))$$

with  $O(n^2)$  operations (only matrix-vector products!)

This gives us a general rule: To *apply* a linear mapping in the form of a matrix (i.e. to evaluate its effect on an arbitrary vector) can, in general, be done in much cheaper ways than by explicitly constructing the matrix!

In this section, we deal with direct methods, and we start with some preliminaries:

- a *vector norm* is a mapping  $\|\cdot\| : \mathbb{R}^n \rightarrow \mathbb{R}$  with the three properties
  - *positive*:  $\|x\| > 0 \quad \forall x \neq 0$
  - *homogeneous*:  $\|\alpha x\| = |\alpha| \cdot \|x\|$  for arbitrary  $\alpha \in \mathbb{R}$
  - *triangle inequality*:  $\|x + y\| \leq \|x\| + \|y\|$
- the set  $\{x \in \mathbb{R}^n : \|x\| \leq 1\}$  is called *norm ball*
- examples of vector norms:
  - *sum norm*:  $\|x\|_1 := \sum_{i=1}^n |x_i|$
  - *Euclidean norm*:  $\|x\|_2 := \sqrt{\sum_{i=1}^n |x_i|^2}$  (the usual vector length)
  - *maximum norm*:  $\|x\|_\infty := \max_{1 \leq i \leq n} |x_i|$
- from a vector norm, a *matrix norm* can be defined or *induced* according to

$$\|A\| := \max_{\|x\|=1} \|Ax\|;$$

besides the (accordingly adapted) three above properties, such a matrix norm is

- *sub-multiplicative*:  $\|AB\| \leq \|A\| \cdot \|B\|$
- *consistent*:  $\|Ax\| \leq \|A\| \cdot \|x\|$
- the *condition number*  $\kappa(A)$  is defined as

$$\kappa(A) := \frac{\max_{\|x\|=1} \|Ax\|}{\min_{\|x\|=1} \|Ax\|};$$

roughly speaking,  $\kappa(A)$  indicates, how much a norm ball is distorted by the matrix  $A$  or by the respective linear mapping (for  $A = Id$  (identity matrix: 1 in all diagonal elements, zeros elsewhere) and for certain classes of matrices there is no distortion at all, but we have  $\kappa(A) = 1$ );

for  $A$  non-singular, it holds

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|$$

- the *Frobenius norm*, which treats a matrix as a vector of length  $n^2$  and calculates the Euclidean norm for this vector, is no matrix norm induced from some vector norm (nevertheless, it fulfils the three norm properties positivity, homogeneity, triangle inequality)

Now, we can tackle the determination of the condition of the problem to solve a system of linear equations (in order to see the terms from the previous section in “real life”):

- for that, we have to conclude in

$$(A + \delta A)(x + \delta x) = b + \delta b$$

from disturbances  $\delta A, \delta b$  of the input data  $A, b$  to the disturbance  $\delta x$  of the result  $x$ ; of course,  $\delta A$  has to stay so small that the disturbed matrix remains invertible (sure for  $\|\delta A\| < \|A^{-1}\|^{-1}$ , for example)

- get a formula for  $\delta x$  from the above relation and derive an estimate based on the properties ‘sub-multiplicative’ and ‘consistent’ of an induced matrix norm (for that, we have introduced this concept!):

$$\|\delta x\| \leq \frac{\|A^{-1}\|}{1 - \|A^{-1}\| \cdot \|\delta A\|} \cdot (\|\delta b\| + \|\delta A\| \cdot \|x\|) ;$$

divide both sides by  $\|x\|$  and note that the relative disturbances of the input  $\|\delta a\|/\|A\|$  and  $\|\delta b\|/\|b\|$  shall be bounded by  $\varepsilon$  (for our considerations concerning the condition, we assume small disturbances of the input); it follows

$$\begin{aligned} \frac{\|\delta x\|}{\|x\|} &\leq \frac{\varepsilon \kappa(A)}{1 - \varepsilon \kappa(A)} \cdot \left( \frac{\|b\|}{\|A\| \cdot \|x\|} + 1 \right) \\ &\leq \frac{2\varepsilon \kappa(A)}{1 - \varepsilon \kappa(A)}, \end{aligned}$$

because  $\|b\| = \|Ax\| \leq \|A\| \cdot \|x\|$

- since it is that nice and important, our result once more:

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{2\varepsilon \kappa(A)}{1 - \varepsilon \kappa(A)} ;$$

the bigger the condition number  $\kappa(A)$  is, the bigger our upper bound (right) of the effect on the result becomes, and hence, the worse the problem “solve  $Ax = b$ ” is conditioned (this explains the name ‘condition number’); only if  $\varepsilon \kappa(A) \ll 1$ , a numerical solution of the problem is useful (note once more: this is due to the problem only, not due to round-off errors etc.!).

Another important quantity is the *residual*  $r$ . For an approximation  $\tilde{x}$  to  $x$ ,  $r$  is defined as

$$r := b - A\tilde{x} = A(x - \tilde{x}) =: Ae$$

with the *error*  $e$ ;

- attention: error and residual can have different orders of magnitude; in particular,  $r = O(\bar{\varepsilon})$  does not at all imply  $e = O(\bar{\varepsilon})$ ; the correlation is also depending on the condition number  $\kappa(A)$
- nevertheless, the residual is helpful:

$$r = b - A\tilde{x} \quad \Leftrightarrow \quad A\tilde{x} = b - r$$

shows that  $\tilde{x}$ , for small residual, can be interpreted as *exact* result for slightly disturbed input data ( $A$  original, instead of  $b$  now  $b - r$ ); therefore,  $\tilde{x}$  is acceptable (w.r.t. the results from the previous section)

## 5.2 Gaussian Elimination

The classical solution method for systems of linear equations known from linear algebra is *Gaussian elimination*, the natural generalization of solving two equations in two unknowns:

- solve one of the  $n$  equations (the first, e.g.) with respect to one unknown (for example,  $x_1$ )
- insert the resulting term for  $x_1$  (depending on  $x_2, \dots, x_n$ ) into the other  $n - 1$  equations; from these,  $x_1$  is now *eliminated*
- solve the resulting system of  $n - 1$  equations in  $n - 1$  unknowns analogously and continue, until  $x_n$  is fixed explicitly
- $x_n$  is now inserted into the elimination equation for  $x_{n-1}$ , which results in  $x_{n-1}$  explicitly
- continue, until, finally, the elimination equation for  $x_1$  provides the value for  $x_1$  by inserting the (now available) values of  $x_2, \dots, x_n$

Actually, elimination means some suitable modification of  $A$  and  $b$ , such that the first column contains only zeros below  $a_{1,1}$ ; here, the new system (consisting of the first equation and the other equations without  $x_1$ ) is solved by the same vector  $x$  as the old system! This leads us to the following algorithm:

### Gaussian elimination:

```

for j from 1 to n do
  for k from j to n do r[j,k]:=a[j,k] od;
  y[j]:=b[j];
  for i from j+1 to n do
    l[i,j]:=a[i,j]/r[j,j];
    for k from j+1 to n do a[i,k]:=a[i,k]-l[i,j]*r[j,k] od;
    b[i]:=b[i]-l[i,j]*y[j]
  od
od;
for i from n downto 1 do
  x[i]:=y[i];
  for j from i+1 to n do x[i]:=x[i]-r[i,j]*x[j] od;
  x[i]:=x[i]/r[i,i]
od;

```

### Discussion:

- outer  $j$ -loop: eliminate variables (i.e., produce zeros in the subdiagonal columns) step by step
- first inner  $k$ -loop: store the part of row  $j$  lying in the upper triangle into the auxiliary matrix  $R$  (we eliminate only *below* the diagonal); store the (modified) right-hand side  $b_j$  to  $y_j$ ; the newly created  $r_{j,k}$  and  $y_j$  won't be changed afterwards!
- inner  $i$ -loop:
  - determine the factors necessary for the elimination of entries below the diagonal in column  $j$  and store them into some auxiliary matrix  $L$  (here, for the moment,  $r_{j,j} \neq 0$  is supposed to be true)
  - subtract the corresponding multiple of row  $j$  from row  $i$
  - modify the right-hand side, too (in order to avoid changes in the solution  $x$ )

- finally: insert backward (i.e., starting with  $x_n$ ) the resulting components of  $x$  into the equations stored in  $R$  and  $y$  (again,  $r_{i,i} \neq 0$  is supposed to hold)
- a careful count of the elementary arithmetic operations results in an overall amount of

$$\frac{1}{3}n^3 + O(n^2)$$

elementary arithmetic operations

Besides the matrix  $A$ , the above algorithm of Gaussian elimination needs the auxiliary matrices  $L$  and  $R$ . We get (verify with the algorithm):

- in  $R$ , only the upper triangular part (including the diagonal) is filled
- in  $L$ , only the strictly lower triangular part (excluding the diagonal) is filled
- if we fill the diagonal of  $L$  with a 1 everywhere, we get the fundamental relation

$$A = L \cdot R;$$

such a *decomposition* or *factorization* of a given matrix  $A$  in factors of specific properties (here: triangular form) is a fundamental technique in numerical linear algebra

For us, the statement above means: Instead of applying classical Gaussian elimination,  $Ax = b$  can also be solved via the triangular decomposition  $A = LR$  – with the algorithm based upon

$$Ax = LRx = L(Rx) = Ly = b$$

- 1. step: *triangular decomposition*: decompose  $A$  in factors  $L$  (lower triangular matrix with 1 in all diagonal places) and  $R$  (upper triangular matrix; the decomposition is unique for this definition of diagonal values)
- 2. step: *forward substitution*: solve  $Ly = b$  (by insertion, from  $y_1$  to  $y_n$ )
- 3. step: *backward transformation*: solve  $Rx = y$  (by insertion, from  $x_n$  to  $x_1$ )

Solution via the  $LR$ -decomposition:

```

for i from 1 to n do
  for k from 1 to i-1 do
    l[i,k]:=a[i,k];
    for j from 1 to k-1 do l[i,k]:=l[i,k]-l[i,j]*r[j,k] od;
    l[i,k]:=l[i,k]/r[k,k]
  od;
  for k from i to n do
    r[i,k]:=a[i,k];
    for j from 1 to i-1 do r[i,k]:=r[i,k]-l[i,j]*r[j,k] od
  od
od;
for i from 1 to n do
  y[i]:=b[i];
  for j from 1 to i-1 do y[i]:=y[i]-l[i,j]*y[j] od
od;
for i from n downto 1 do
  x[i]:=y[i];
  for j from i+1 to n do x[i]:=x[i]-r[i,j]*x[j] od;
  x[i]:=x[i]/r[i,i]
od;

```

Discussion:

- in order to verify the first part (decomposition into the factors  $L$  and  $R$ ), take the general formula for the product of two matrices:

$$a_{i,k} = \sum_{j=1}^n l_{i,j} \cdot r_{j,k};$$

in contrast to the usual situation, now,  $A$  is known, whereas  $L$  and  $R$  have to be determined (it is due to the triangular form of the factors that this works)!

- in the case of  $i > k$ , one has to sum up until  $j = k$  only and can determine  $l_{i,k}$  (solve for  $l_{i,k}$ : everything remaining on the right-hand side is already known):

$$a_{i,k} = \sum_{j=1}^{k-1} l_{i,j} \cdot r_{j,k} + l_{i,k} \cdot r_{k,k},$$

$$l_{i,k} := \left( a_{i,k} - \sum_{j=1}^{k-1} l_{i,j} \cdot r_{j,k} \right) / r_{k,k},$$

- in the case of  $i \leq k$ , one has to sum up until  $j = i$  only can determine  $r_{i,k}$  (solve for  $r_{i,k}$ : as before, everything on the right-hand side is already known; note that  $l_{i,i} = 1$ ):

$$a_{i,k} = \sum_{j=1}^{i-1} l_{i,j} \cdot r_{j,k} + l_{i,i} \cdot r_{i,k},$$

$$r_{i,k} := a_{i,k} - \sum_{j=1}^{i-1} l_{i,j} \cdot r_{j,k};$$

you see: working through this unknowns with increasing row and column indices (as done in the algorithm, starting with  $i = k = 1$ ), one can successively compute all  $l_{i,k}$  and  $r_{i,k}$  – on the right-hand side, there are only variables that have already been computed!

- afterwards, there is the forward substitution (second  $i$ -loop) and – as before in Gaussian elimination – the backward substitution (third and last  $i$ -loop)

It can be shown that the two variants “Gaussian elimination” and “ $LR$ -decomposition” are identical in the sense that they apply the same operations (i.e., in particular, we have the same number of operations); only the order of the operations differs!

### 5.3 Pivoting

In the above algorithm of Gaussian elimination, we supposed that the divisions  $a_{i,j}/r_{j,j}$  and  $x_i/r_{i,i}$  do not cause any troubles, i.e. that there are especially no zeros created in the diagonal of  $R$ . Since these diagonal values are crucial, they are called *pivots*. The condition  $r_{j,j} \neq 0$  for all  $j$ , however, does not necessarily hold. If a zero occurs, the algorithm has to be modified (which is possible, as long as  $A$  is non-singular!):

- *partial pivoting* or *column pivoting*:  
if  $r_{j,j} = 0$ , look in column  $j$  below line  $j$  for an entry  $a_{i,j} \neq 0$ ,  $i = j + 1, \dots, n$ :
  - there is no such  $a_{i,j} \neq 0$ : the remaining submatrix has a completely vanishing column, which means  $\det(A) = 0$  (error situation)

- there are non-zeros: usually, choose the one of maximum absolute value (let it be  $a_{i,j}$ ) as pivot and exchange the rows  $i$  and  $j$  in both the matrix and the right-hand side
- large pivots are favourable, since they lead to small elimination factors  $l_{i,j}$  and do not produce too large entries in  $L$  and  $R$
- of course, such an exchange of rows does not change the linear system or its solution at all
- *total pivoting*: here, not only column  $j$  of the remaining submatrix is searched for a suitable pivot, but the whole remaining submatrix; apart from the exchange of rows, also an exchange of columns has to be done (re-ordering of the unknowns  $x_k$ ); therefore, total pivoting is more expensive

## 5.4 Cholesky Decomposition

In the special situation of *positive definite matrices*, a cheaper proceeding is possible than with the two algorithms presented above.

- A quadratic symmetric matrix  $A \in \mathbb{R}^{n,n}$  is called *positive definite*, if all of its eigenvalues are positive or if

$$x^T A x > 0 \quad \forall x \neq 0.$$

- It can be shown that pivoting is not necessary for positive definite matrices.
- We further decompose the factor  $R$  in  $A = LR$  into a diagonal matrix  $D$  and an upper triangular matrix  $\tilde{R}$  with a '1' in all diagonal entries (can be done always):

$$A = L \cdot R = L \cdot D \cdot \tilde{R} \quad \text{with } D = \text{diag}(r_{1,1}, \dots, r_{n,n});$$

then, symmetry of  $A$  provides

$$A^T = (L \cdot D \cdot \tilde{R})^T = \tilde{R}^T \cdot D \cdot L^T = L \cdot D \cdot \tilde{R} = A,$$

and the decomposition's uniqueness enforces

$$L = \tilde{R}^T$$

and, hence,

$$A = L \cdot D \cdot L^T =: \tilde{L} \cdot \tilde{L}^T,$$

if the diagonal factor  $D$  is equally distributed among both triangular factors ( $\sqrt{r_{i,i}}$  in both diagonals of  $\tilde{L}$ ; the  $r_{i,i}$  are all positive, since  $A$  is positive definite); thus, we have the nice form (attention: from now on,  $L$  has this new meaning – we skip the tilde for reasons of simplicity of the notation!)

$$A = L \cdot L^T,$$

which we will use in the following.

- Consequently, in the  $LR$ -decomposition, the computation of the  $r_{i,k}$ ,  $i \neq k$  can be avoided, and, thus, about half of the memory requirements and computational cost can be saved. This variant is called *Cholesky decomposition*.
- For the construction of the Cholesky algorithm, we again start from the formula of the matrix product:

$$a_{i,k} = \sum_{j=1}^n l_{i,j} \cdot l_{j,k}^T = \sum_{j=1}^k l_{i,j} \cdot l_{j,k}^T = \sum_{j=1}^k l_{i,j} \cdot l_{k,j}, \quad i \geq k;$$

from that, we compute  $L$  (the lower triangle) column by column, starting with the diagonal element in each column,

$$a_{k,k} = \sum_{j=1}^k l_{k,j}^2,$$

$$l_{k,k} := \sqrt{a_{k,k} - \sum_{j=1}^{k-1} l_{k,j}^2},$$

and, afterwards, processing the other rows  $i > k$ :

$$a_{i,k} = \sum_{j=1}^k l_{i,j} \cdot l_{k,j},$$

$$l_{i,k} := \left( a_{i,k} - \sum_{j=1}^{k-1} l_{i,j} \cdot l_{k,j} \right) / l_{k,k};$$

#### Cholesky decomposition:

```

for k from 1 to n do
  l[k,k]:=a[k,k];
  for j from 1 to k-1 do l[k,k]:=l[k,k]-l[k,j]^2 od;
  l[k,k]:=(l[k,k])^0.5;
  for i from k+1 to n do
    l[i,k]:=a[i,k];
    for j from 1 to k-1 do l[i,k]:=l[i,k]-l[i,j]*l[k,j] od;
    l[i,k]:=l[i,k]/l[k,k]
  od
od;
```

Of course, the above algorithm does the decomposition only. As before, forward and backward substitution have to be added in order to solve  $Ax = b$ .

#### Discussion:

- As already supposed, the number of computations decreases by about 50%, i.e.

$$\frac{1}{6}n^3 + O(n^2);$$

- The positivity of the pivots is used in the arguments of the square root; this positivity is ensured since  $A$  is positive definite – as already mentioned.

For the moment, this is enough information for systems of linear equations. We will return to this topic in Section 8.

# Chapter 6

## Interpolation

### 6.1 Preparatory Remarks

approximation and interpolation:

- *approximation*: approximate some (totally or partially unknown or too complicated) function  $f(x)$  by some  $p(x)$  which is easy to handle (construct, evaluate, differentiate, integrate, etc.), following certain rules
- *interpolation*: special case of the approximation, where the rule is that the function values of the function to be approximated are prescribed in so-called *nodes*  $x_i$ ,  $i = 0, \dots, n$ ; thereby, the approximant becomes an *interpolant*; there are two principal classes of interpolation problems:
  - interpolation of a given function  $f(x)$ : in the nodes, the values of  $f$  are inherited, and in-between, something else is defined (here, it makes sense to speak of an *interpolation error* between the nodes, because generally  $f(x) \neq p(x)$  here)
  - interpolation of discrete data  $(x_i, y_i)$ ,  $i = 0, \dots, n$ : frequent numerical task (think of CAD/CAGD (*Computer-Aided (Geometric) Design*), for example: use single measured of control points to construct smooth free form curve or surface); here, at the beginning, we don't have anything between the nodes – the interpolant is expected to provide exactly this connection (here, an interpolation error makes no sense, and other criteria of quality must be developed)

in the following, we write in both cases  $y_i$  for the prescribed function values; if we have an underlying  $f : [a, b] \rightarrow \mathbb{R}$ , it is supposed to be sufficiently smooth (all necessary derivatives shall exist)

basics and some notation:

- *nodes*: the abscissas  $x_i$  where the interpolant is to be “fixed”
- *nodal points*: pairs  $(x_i, y_i)$ , i.e. nodes  $x_i$  with corresponding *nodal values*  $y_i$
- polynomials are widespread interpolants, since their handling is very simple:
  - $\mathbb{P}_n$ : vector space of all polynomials with real coefficients of degree smaller or equal  $n$  in one variable
  - it holds  $\dim(\mathbb{P}_n) = n + 1$ , an example of a basis is  $\{x^i, i = 0, \dots, n\}$
  - with the differential operator  $D^k$  (for  $\frac{\partial^k}{\partial x^k}$ ), we know that  $D^{n+1}p = 0 \forall p \in \mathbb{P}_n$

- nevertheless, polynomial interpolation is not the only possibility: interpolation is also possible with piecewise polynomials (*polynomial splines*) or with trigonometric, rational, or exponential functions; the resulting interpolants are also characterized by simple ways of construction and manipulation
- for various tasks (for example, see numerical quadrature in the next section), complicated functions are therefore replaced by well-suited interpolants, and the task is then solved for the interpolant
- different variants of the interpolation problem:
  - *single nodes*: for  $x_i$  prescribe  $p(x_i)$  (*Lagrange interpolation*, studied in this section); example: find the quadratic polynomial  $p(x)$  with  $p(-1) = p(1) = 1$  and  $p(0) = 0$
  - *multiple nodes*: in  $x_i$ , function value and derivative(s) are prescribed (*Hermite interpolation*); example: find the quadratic polynomial  $q(x)$  with  $q(x) = q'(x) = 0$  and  $q''(x) = 2$

## 6.2 Interpolation with Polynomials

- $p \in \mathbb{P}_n$  is called *polynomial interpolant* of  $f$  with respect to the nodes  $a = x_0 < x_1 < \dots < x_n = b$ , if  $p(x_i) = f(x_i) =: y_i$  for all  $i = 0, \dots, n$
- analogous definition, if instead of a function  $f$  only discrete data  $y_0, y_1, \dots, y_n$  are prescribed
- hence, the number of nodes directly determines the degree of the interpolation polynomial (in practice, this often means high degrees, which – as we will see later – may be a problem)
- existence and uniqueness of the solution of this interpolation problem are known from analysis
- the difference  $f(x) - p(x)$  is called *remainder* or *error term* and a multiple of  $\omega(x) := \prod_{i=0}^n (x - x_i)$ ; we want to be able to estimate or give a bound for the absolute value of this term

The simplest approach for  $p(x)$  is well-known: take  $p(x)$  with general coefficients,

$$p(x) := \sum_{i=0}^n a_i \cdot x^i,$$

make the incidental test for each node, and solve the resulting system of  $n + 1$  linear equations in the  $n + 1$  unknowns  $a_0, \dots, a_n$ ; an alternative is provided by the so-called *Lagrange polynomials*  $L_k(x)$ ,

$$L_k(x) = \prod_{i:i \neq k} \frac{x - x_i}{x_k - x_i},$$

by defining the interpolant as

$$p(x) := \sum_{k=0}^n y_k \cdot L_k(x)$$

and by verifying easily:

- $L_k(x_i) = \delta_{i,k}$  (Kronecker symbol: 1 for  $i = k$ , 0 elsewhere)
- the Lagrange polynomials are linearly independent and form a basis of  $\mathbb{P}_n$
- indeed, the polynomial  $p(x)$  just defined is the desired interpolant

### 6.3 Aitken and Neville's Scheme

Often, we are not interested in an explicit or closed form of the interpolant, but only want to *evaluate*  $p(x)$  at one or several points  $x$ . It is important and interesting to note that such an evaluation does not require to know  $p(x)$  explicitly before. An elegant *recursive* algorithm was given by Aitken and Neville:

- define auxiliary polynomials  $p_{i,k}(x)$  of degree  $k$ , which interpolate the nodes  $(x_l, y_l)$  for  $l = i, \dots, i+k$
- do this recursively with the help of the auxiliary polynomials  $p_{i,k-1}(x)$  and  $p_{i+1,k-1}(x)$  of degree  $k-1$ , which interpolate in  $x_i, \dots, x_{i+k-1}$  or  $x_{i+1}, \dots, x_{i+k}$ , respectively:

$$p_{i,k}(x) := \frac{x_{i+k} - x}{x_{i+k} - x_i} \cdot p_{i,k-1}(x) + \frac{x - x_i}{x_{i+1} - x_i} \cdot p_{i+1,k-1}(x)$$

(verification by checking the interpolation properties)

- this leads to the algorithm

```

for i=0 to n do p[i,0] := y[i] od;
for k=1 to n do
  for i=0 to n-k do
    p[i,k] := (x[i+k]-x)/(x[i+k]-x[i])*p[i,k-1] +
              (x-x[i])/((x[i+k]-x[i])*p[i+1,k-1]);
  od;
od;

```

Although the algorithm uses a concrete value  $x$ , the scheme of Aitken and Neville also allows for detecting the interpolating polynomial recursively: In Maple, for example,  $x$  can be treated as a variable, and the polynomial can, hence, be determined explicitly with this recursion. The resulting  $p(x)$  is, of course, the same as the one introduced above (due to the uniqueness of the interpolation task). Only the way of construction of  $p(x)$  is different.

### 6.4 Divided Differences and Error

We study a fourth way of representation of the polynomial interpolant. The highest coefficient of  $p_{i,k}(x)$  is denoted by

$$[x_i, \dots, x_{i+k}]f$$

and called *divided differences of  $f$  of order  $k$  w. r. t.  $x_i, \dots, x_{i+k}$* , i.e.

$$p_{i,k}(x) = [x_i, \dots, x_{i+k}]f \cdot x^k + \langle \text{component in } \mathbb{P}_{k-1} \rangle .$$

The scheme of Aitken and Neville can be used to derive a recursion formula for the divided differences, too:

$$[x_i, \dots, x_{i+k}]f := \frac{[x_{i+1}, \dots, x_{i+k}]f - [x_i, \dots, x_{i+k-1}]f}{x_{i+k} - x_i} .$$

Furthermore, we have:

- $[x_i]f = f(x_i) = y_i$
- $[x_i, x_{i+1}]f = (y_{i+1} - y_i)/(x_{i+1} - x_i)$
- neither for the scheme of Aitken and Neville nor for the divided differences, the order of the nodes is important (the order can be arbitrary!)

With the divided differences, we get a second closed form for  $p(x)$ , *Newton's interpolation formula*:

$$\begin{aligned} p(x) &:= [x_0]f + \\ &\quad [x_0, x_1]f \cdot (x - x_0) + \\ &\quad [x_0, x_1, x_2]f \cdot (x - x_0) \cdot (x - x_1) + \\ &\quad [x_0, \dots, x_3]f \cdot (x - x_0) \cdot (x - x_1) \cdot (x - x_2) + \\ &\quad \dots \\ &\quad [x_0, \dots, x_n]f \cdot \prod_{i=0}^{n-1} (x - x_i) \end{aligned}$$

Newton's representation is attractive, since it provides a simple way to add a new node  $x_{n+1}$ . The current interpolant with respect to  $x_0, \dots, x_n$  just has to be completed by an additional summand – everything calculated so far does not get lost, the construction is *incremental*.

Furthermore, divided differences help to estimate the interpolation error  $f(x) - p(x)$ , which is the crucial quantity for the analysis of every interpolation method:

- let  $f \in \mathcal{C}^n([x_0, x_n])$ , i.e.  $n$  times continuously differentiable on  $[x_0, x_n]$ ; then, there exists a  $\xi \in [x_0, x_n]$  with

$$[x_0, \dots, x_n]f = \frac{D^n f(\xi)}{n!}$$

(proof: cf. mean value theorem)

- now, we consider the error in some  $\bar{x} \in [x_0, x_n]$ , which is not a node (if it is, the interpolation error is zero!), let  $P(x)$  be the polynomial of degree  $n + 1$ , which, in addition to the interpolation properties of  $p(x)$ , interpolates  $f$  (now supposed to be  $n + 1$  times continuously differentiable) also in  $\bar{x}$ ; then, we get:

$$\begin{aligned} f(\bar{x}) - p(\bar{x}) &= P(\bar{x}) - p(\bar{x}) = [x_0, \dots, x_n, \bar{x}]f \cdot \prod_{i=0}^n (\bar{x} - x_i) \\ &= \frac{D^{n+1} f(\xi)}{(n+1)!} \cdot \prod_{i=0}^n (\bar{x} - x_i); \end{aligned}$$

for equidistant nodes with mesh width  $h := x_{i+1} - x_i$ , the product can be estimated: in the worst case, the distance between  $\bar{x}$  and the farthest node is  $nh$ ; to the second farthest, the distance is  $(n-1)h$  etc.; from that, it follows

$$|f(\bar{x}) - p(\bar{x})| \leq \frac{\max_{[a,b]} |D^{n+1} f(x)|}{n+1} \cdot h^{n+1} = O(h^{n+1})$$

## 6.5 Condition of Polynomial Interpolation

we study the condition of the problem of polynomial interpolation:

- input data:
  - nodes  $x_i$
  - nodal values  $y_i$
  - inner point  $x$  (here, the function value shall be (re-)constructed)
- the result is the value  $y := p(x)$

The sensitivity of  $p(x)$  with respect to disturbances in the point of evaluation  $x$  is simple – it is described by  $p'(x)$ ; this derivative is always bounded for polynomials on  $[x_0, x_n]$ , but it can nevertheless get very big (especially for higher degree  $n$ ). In practice, the sensitivity with respect to disturbances in the nodal points is more important; we consider small changes in the nodal values  $y_i$ . From

$$y = p(x) = \sum_{k=0}^n y_k \cdot L_k(x),$$

we get immediately

$$\frac{\partial y}{\partial y_k} = L_k(x);$$

hence, the Lagrange polynomials are important!

An example:

- $x_i := i, i = 0, \dots, 40 =: n$
- $k = 20$ , i.e.  $x_k = 20$
- $x \in ]x_0, x_1[ = ]0, 1[$

$$\begin{aligned} |L_{20}(x)| &= \left| \prod_{i \neq 20} \frac{x - x_i}{x_{20} - x_i} \right| = \prod_{i \neq 20} \frac{|x - x_i|}{|20 - x_i|} = \frac{x}{20} \cdot \frac{1-x}{19} \cdot \prod_{i=2}^{19} \frac{i-x}{20-i} \cdot \prod_{i=21}^{40} \frac{i-x}{i-20} \\ &\geq \frac{x-x^2}{380} \cdot \prod_{i=2}^{19} \frac{i-1}{20-i} \cdot \prod_{i=21}^{40} \frac{i-1}{i-20} \\ &= \frac{x-x^2}{380} \cdot \frac{18!}{18!} \cdot \frac{39!}{19! \cdot 20!} \\ &\geq 1.8 \cdot 10^8 \cdot (x-x^2) \end{aligned}$$

especially, for example, we have

$$|L_{20}(0.5)| \geq 4.5 \cdot 10^7$$

This is a principal result: small errors in the central nodal values are increased drastically near the interval's boundary by polynomial interpolation.

⇒ For large degrees  $n$  (starting from 7 or 8), polynomial interpolation is extremely badly conditioned and, hence, practically useless!

⇒ We have to look for something better!

## 6.6 Trigonometric Interpolation

Very important and widespread (signal processing, image processing, encoding, compression, JPEG, ...); from a numerical point of view, especially the Fast Fourier Transform (FFT) should be discussed. However, for mechanical purposes, this topic is not of primary relevance and will be skipped.

## 6.7 Polynomial Splines

Summary of polynomial interpolation:

- number of nodes and degree are strictly connected

- for higher degree (and, hence, for a larger number of nodes – which is typical for practically relevant problems), polynomial interpolation is useless for reasons of the condition

Therefore, the idea to connect piecewise polynomials of a lower degree in order to get a global interpolant also for large numbers of nodes seems to be promising. This is exactly what *polynomial splines* or shortly *splines* do. Their definition, first, given without a specific interpolation task in mind:

- let  $a = x_0 < x_1 < \dots < x_n = b$  and  $m \in \mathbb{N}$ ; the  $x_i$  are called *nodes* (again, we restrict ourselves to the *simple* case  $x_i \neq x_j$  for  $i \neq j$ )
- $s : [a, b] \rightarrow \mathbb{R}$  is called *spline* of order  $m$  or of degree  $m - 1$ , if the following is fulfilled:
  - $s(x) = p_i(x)$  on  $[x_i, x_{i+1}]$  with  $p_i \in \mathbb{P}_{m-1}$ ,  $i = 0, 1, \dots, n - 1$
  - $s \in \mathcal{C}^{m-2}([a, b])$ <sup>1</sup>
- i.e.: between every two neighbouring nodes,  $s(x)$  is a polynomial of degree  $m - 1$ , and globally (i.e. especially in the nodes),  $s(x)$  is  $m - 2$  times continuously differentiable; this means for concrete scenarios:
  - $m = 1$ : staircase functions (piecewise constant, no continuity in the  $x_i$ )
  - $m = 2$ : polylines (piecewise linear, continuous in the  $x_i$ )
  - $m = 3$ : quadratic splines (locally quadratic, globally once continuously differentiable)
  - $m = 4$ : cubic splines (locally cubic, in the  $x_i$  twice continuously differentiable)
- for fixed nodes and fixed degrees, with  $s_1$  and  $s_2$ ,  $\alpha s_1 + s_2$  is a spline too – hence, they form a vector space; its dimension is  $n + m - 1$  ( $p_0 \in \mathbb{P}_{m-1}$  fixes  $m$  degrees of freedom; in addition to that, there is one degree of freedom for each inner node  $x_1, \dots, x_{n-1}$  for a possible jump in the  $m - 1$ -st derivative (all lower derivatives are continuous due to the above definition))

main areas of application of splines:

- the interpolation (our topic here): a spline shall be designed in a way that it fulfils  $n + m - 1$  interpolation conditions – corresponding to the number of degrees of freedom; attention: the *nodes* of the interpolation problem do not need to be identical with the *nodes* of the spline's definition!
- CAD, where *free form curves* are used for the modelling of smooth and curved forms (in this case, typically, no interpolation conditions are prescribed, but the desired form is indicated with the help of so-called *control points*)

## 6.8 Interpolation with Cubic Splines

*Cubic splines* (i.e.  $m = 4$ ) are widespread for purposes of interpolation; we discuss the following special case:

- single nodes  $a = x_0 < x_1 < \dots < x_n = b$
- local: spline  $s \in \mathbb{P}_3$  on each  $[x_i, x_{i+1}]$ ,  $i = 0, \dots, n - 1$
- global: spline  $s \in \mathcal{C}^2([a, b])$
- nodes are the points of interpolation (not necessarily), i.e.

$$s(x_i) = y_i, \quad i = 0, 1, \dots, n$$

<sup>1</sup>The notation  $\mathcal{C}^k([a, b])$  denotes the space of functions that are  $k$  times differentiable with a continuous  $k$ -th derivative, all on the interval  $[a, b]$ .

thus, we have:

- we have  $n + m - 1 = n + 3$  degrees of freedom (dimension of the spline vector space, see above)
- on the other hand, there are  $n + 1$  interpolation condition
- thus, two conditions to fix degrees of freedom are left

or, from a different point of view:

- define on each subinterval  $[x_i, x_{i+1}]$  the cubic polynomial as a function of  $y_i, y_{i+1}$  (function values in both nodes involved) and of  $y'_i, y'_{i+1}$  (first derivatives there):

$$\begin{aligned} s(x) &= p_i \left( \frac{x - x_i}{x_{i+1} - x_i} \right) \\ &= p_i(t) \\ &=: y_i \cdot \alpha_1(t) + y_{i+1} \cdot \alpha_2(t) + (x_{i+1} - x_i) \cdot (y'_i \cdot \alpha_3(t) + y'_{i+1} \cdot \alpha_4(t)) \end{aligned}$$

mit  $\alpha_1, \alpha_2, \alpha_3, \alpha_4 \in \mathbb{P}_3$ ,  $t \in [0, 1]$  und

$$\begin{aligned} \alpha_1(0) &= 1, & \alpha_1(1) &= \alpha'_1(0) = \alpha'_1(1) = 0 & \Rightarrow & \alpha_1(t) &:= 1 - 3t^2 + 2t^3 \\ \alpha_2(1) &= 1, & \alpha_2(0) &= \alpha'_2(0) = \alpha'_2(1) = 0 & \Rightarrow & \alpha_2(t) &:= 3t^2 - 2t^3 \\ \alpha'_3(0) &= 1, & \alpha_3(0) &= \alpha_3(1) = \alpha'_3(1) = 0 & \Rightarrow & \alpha_3(t) &:= t - 2t^2 + t^3 \\ \alpha'_4(1) &= 1, & \alpha_4(0) &= \alpha_4(1) = \alpha'_4(0) = 0 & \Rightarrow & \alpha_4(t) &:= -t^2 + t^3 \end{aligned}$$

one can easily see:

$$\begin{aligned} s(x_i) &= p_i(0) = y_i, & s(x_{i+1}) &= p_i(1) = y_{i+1}, \\ \frac{ds}{dx}(x_i) &= \frac{dp_i}{dt}(0) \cdot \frac{dt}{dx}(x_i) = y'_i, & \frac{ds}{dx}(x_{i+1}) &= \frac{dp_i}{dt}(1) \cdot \frac{dt}{dx}(x_{i+1}) = y'_{i+1}; \end{aligned}$$

Note that this approach automatically provides continuity and continuous differentiability of  $s(x)$  in the nodes!

- Since the  $y_i$  are determined by the interpolation conditions,  $n + 1$  degrees of freedom are left and need to be fixed – the first derivatives  $y'_i$ ,  $i = 0, \dots, n$ .
- The continuity of the second derivative of  $s(x)$  must be ensured explicitly by  $n - 1$  conditions at the inner nodes  $x_1, \dots, x_{n-1}$ :

$$y'_{i-1} \frac{1}{h_{i-1}} + y'_i \left( \frac{2}{h_{i-1}} + \frac{2}{h_i} \right) + y'_{i+1} \frac{1}{h_i} = 3 \left( \frac{y_i - y_{i-1}}{h_{i-1}^2} + \frac{y_{i+1} - y_i}{h_i^2} \right), \quad i = 1, \dots, n - 1$$

(derive  $s(x)$  twice with respect to  $x$  and set the resulting expression for  $p_{i-1}$  in  $t = 1$  (second derivative in  $x_i$  from the left) to be identical with  $p_i$  in  $t = 0$  (second derivative in  $x_i$  from the right));

obviously, this is a tridiagonal system of linear equations of dimension  $n - 1$

- hence, this point of view also leads to two still open degrees of freedom

What to do with the resulting two free degrees of freedom? Usually, *boundary conditions* are given:

- the derivative in the boundary nodes can be given, i.e.  $y'_0$  and  $y'_n$
- or, the second derivatives of  $s(x)$  at the boundary nodes can be prescribed (the special case of zeros is called *natural boundary condition*)
- or, *periodic boundary conditions* are given, i.e., the values of the first two derivatives in  $x_0$  are the same as in  $x_n$

In all three cases, solving a system of linear equations does not become significantly more expensive – even if tridiagonality may be lost.

This leads us to performance:

- spline interpolation is very cheap: it requires  $O(n)$  arithmetic operations; compare this with the  $O(n^3)$  operations during solving the full system of equations of polynomial interpolation (incidental check!)
- concerning accuracy, we have (again supposing the equidistant case of a constant mesh width  $h$ ):

$$|f(x) - s(x)| = O(h^4)$$

- furthermore, any kind of troubles with high polynomial degree (work, bad condition, ...) is avoided

Finally, some remarks concerning the origin of the word *spline*: It stems from naval construction (bendable rod made from wood or steel); if it is fixed on one side (with metal strips), it takes the form of minimum under the boundary conditions. Such a minimum property can be shown for our spline, too.

## Chapter 7

# Numerical Quadrature

*Numerical quadrature* means the numerical calculation of a (here one-dimensional) integral

$$I(f) := \int_a^b f(x) \, dx.$$

The numerical approach comes only at the end of all other efforts (closed integration with the help of standard techniques like integration by parts or substitution etc., decomposition into a sum of integrals at points where  $f$  or some derivative of  $f$  is discontinuous etc.). First, this shall save computing time, and second, numerical problems shall be circumvented where possible (for example, numerical quadrature rules typically require sufficient smoothness of the integrand  $f$ ).

Most *quadrature rules*, i.e. methods for the numerical quadrature of functions, can be written as *weighted sums of function values (samples)*, i.e.

$$I(f) \approx Q(f) := \sum_{i=1}^n g_i f(x_i) = \sum_{i=1}^n g_i y_i$$

with *weights*  $g_i$  and pairwise different *nodes*  $x_i$ ,  $a \leq x_1 < x_2 < \dots < x_{n-1} < x_n \leq b$ . Since the evaluation of the integrand at some point  $x$  is often quite expensive (especially, if  $f$  is given only implicitly; in some applications, even a differential equation has to be solved in order to be able to get  $f(x)$  for some  $x$ ), we look for rules that lead to small errors with moderate  $n$  already.

How can we derive suitable quadrature rules? The standard approach is to replace the integrand  $f$  by an (easily to construct and to evaluate) approximation  $\tilde{f}$ , and to integrate the latter *exactly*:

$$Q(f) := \int_a^b \tilde{f}(x) \, dx.$$

As  $\tilde{f}$ , frequently a polynomial interpolant  $p(x)$  of  $f(x)$  with respect to the nodes  $x_i$  is chosen. In this case, the representation of  $p(x)$  via Lagrange polynomials (cf. the previous section)  $L_i(x)$  provides the weights basically for free:

$$Q(f) := \int_a^b p(x) \, dx = \int_a^b \sum_{i=1}^n y_i L_i(x) \, dx = \sum_{i=1}^n y_i \cdot \int_a^b L_i(x) \, dx,$$

which defines the weights as

$$g_i := \int_a^b L_i(x) \, dx.$$

Because of  $\sum_{i=1}^n L_i(x) = 1$  for all  $x \in [0, 1]$  (easy to prove – do it as an exercise, starting from the definition in Sect. 6.2), the sum of all weights is always  $b - a$ , if a polynomial interpolant is used for integration. This is reasonable, since otherwise, even a constant function would not be integrated exactly! In every case, all weights  $g_i$  should be *positive*, in order to prevent small errors  $|\delta y_i| \leq \varepsilon$  from reducing the result's accuracy significantly:

$$|\delta Q(f)| = \left| \sum_{i=1}^n g_i \delta y_i \right| \leq \varepsilon \cdot \sum_{i=1}^n |g_i|.$$

If all weights are positive, then the sum on the right-hand side takes its minimum value  $b - a$ ; for negative weights, both the sum and, hence, the upper error bound increase – the disturbance  $\delta Q(f)$  in the computed result could be significant!

## 7.1 Simple and Composite Rules

Next, we look at some concrete quadrature rules. These are called *simple*, if the integration interval is treated as a whole, and *composed* or *composite*, if the integration interval is subdivided in subintervals to apply simple rules on each of them. This is similar to the difference of polynomial and spline interpolation.

First, some simple rules; let  $h := b - a$  denote the length of the integration interval:

- **quadrilateral rule:**

$$Q_R(f) := h \cdot f\left(\frac{a+b}{2}\right) = I(p_0),$$

where  $p_0$  is the polynomial interpolant of degree 0 for the one and only node  $x_0 := (a+b)/2$  – obviously the simplest possible strategy. For the *remainder*  $R_R(f) := Q_R(f) - I(f)$ , the relation

$$R_R(f) = -h^3 \cdot \frac{f''(\xi)}{24}$$

for some intermediate position  $\xi \in ]a, b[$  can be shown, if  $f$  is twice continuously differentiable. From this result, we learn the following:

- Polynomials of degree smaller than or equal 1 are integrated exactly (here, the second derivative vanishes). This may be surprising at first glance, since we only use a *constant* interpolant. However, the integral value is, obviously, correct for linear functions, too: the amount of area missing to the left of the one and only node is added too much on its right side, and vice versa.
- The rule's accuracy is of the order  $O(h^3)$ : If we halve the mesh width  $h$ , the integration error is reduced by a factor of 8.

- **trapezoidal rule:**

$$Q_T(f) := h \cdot \frac{f(a) + f(b)}{2} = I(p_1),$$

where  $p_1$  is the polynomial interpolant of degree 1 with respect to the two nodes  $x_0 := a$  and  $x_1 := b$ . Here, the remainder  $R_T(f)$  fulfils

$$R_T(f) = h^3 \cdot \frac{f''(\xi)}{12}$$

(hence, of third order as well and exact for polynomials up to degree 1, too). Thus, we don't gain anything from the improved interpolant  $p_1$ .

- **barrel rule:**

$$Q_F(f) := h \cdot \frac{f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)}{6} = I(p_2),$$

where  $p_2$  is the polynomial interpolant of degree 2 with respect to the three nodes  $x_0 := a$ ,  $x_1 := (a+b)/2$ , and  $x_2 := b$ . Here, we have

$$R_F(f) = h^5 \cdot \frac{f^{(4)}(\xi)}{2880},$$

i.e. an improved situation, if the integrand  $f$  is four times continuously differentiable.

- **Newton-Cotes formulas:**

The natural generalization of the three upper methods to larger  $n$ :

$$Q_{NC(n)}(f) := I(p_n),$$

where  $p_n$  is the polynomial interpolant of degree  $n$  with respect to the  $n+1$  equidistant nodes  $x_i := a + h \cdot i/n$ ,  $i = 0, \dots, n$ .

Attention: For  $n = 8$  as well as for  $n \geq 10$ , we get negative weights. Hence, Newton-Cotes formulas for these  $n$  are not useful (cf. polynomial interpolation with equidistant nodes for higher degree)!

- **Clenshaw-Curtis formulas:**

The problem of negative nodes for higher degree can be avoided, if we subdivide the semi-circular angle instead of the interval from  $a$  to  $b$ , i.e.

$$x_i := a + h \cdot \frac{1 - \cos(i\pi/n)}{2}, \quad i = 0, \dots, n.$$

Now, the nodes are no longer equidistant, but are denser at the boundary points than in the centre, and all weights are always positive. Thus, Clenshaw-Curtis formulas are principally suited for larger  $N$ , too;

Now for two famous composite rules; here, increasing the number of nodes does not automatically mean an increase of the polynomial degree, but a reduction of the subintervals' length. For the following, note the new meaning of  $h$ : so far, it was the *global* length  $b-a$  of the integration interval, whereas from now on, it will denote the *local* distance between two nodes *within* the interval  $[a, b]$ !

- **trapezoidal sum:**

Here, the integration interval  $[a, b]$  is subdivided into  $n$  subintervals of length  $h := (b-a)/n$ . The equidistant connection points  $x_i := a + ih$ ,  $i = 0, \dots, n$  are used as nodes. Now, the trapezoidal rule is applied to each subinterval, and the resulting values are accumulated:

$$Q_{TS}(f) := h \cdot \left( \frac{f_0}{2} + f_1 + f_2 + \dots + f_{n-1} + \frac{f_n}{2} \right),$$

where  $f_i := f(x_i)$ . For the remainder, we have

$$R_{TS}(f) = h^2 \cdot (b-a) \cdot \frac{f''(\xi)}{12}$$

(summation of the single remainders and intermediate value theorem).

- **Simpson sum:**

The integration interval is subdivided as before, but we apply the barrel rule to each pair of neighbouring nodes instead:

$$Q_{SS} := \frac{h}{3} \cdot (f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \dots + 2f_{n-2} + 4f_{n-1} + f_n).$$

It holds

$$R_{SS}(f) = h^4 \cdot (b - a) \cdot \frac{f^{(4)}(\xi)}{180}.$$

Composite rules are frequently used, because they allow for simple algorithms. Another advantage are their asymptotic properties, which can be exploited for *extrapolation*. We won't go into detail, but present the basic principle.

## 7.2 The Principle of Extrapolation

Let us look at the barrel rule once more. Its formula can also be obtained via the linear combination of three trapezoidal rules (one with mesh width  $h$  and two with mesh width  $h/2$ ):

$$\begin{aligned} Q_F(f) &= h \cdot \frac{f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)}{6} \\ &= \frac{4}{3} \left( \frac{h}{2} \cdot \frac{f(a) + f\left(\frac{a+b}{2}\right)}{2} \right) + \frac{4}{3} \left( \frac{h}{2} \cdot \frac{f\left(\frac{a+b}{2}\right) + f(b)}{2} \right) \\ &\quad - \frac{1}{3} \left( h \cdot \frac{f(a) + f(b)}{2} \right). \end{aligned}$$

This fact shows that an appropriate combination of approximate values of *low* order of approximation may lead to an approximation of *higher* order, without having to derive and use methods of higher order explicitly. For the trapezoidal rule, this principle is not used frequently. For the trapezoidal sum, however, this is a very interesting starting point. Compute trapezoidal sums  $T(h_i)$  for decreasing mesh widths  $h_i$ ,  $i = 1, \dots, p$ , i.e.  $h_i := h/2^i$ , for example. Then, construct a polynomial interpolant  $P(h)$  of degree  $p$  through the points  $(h_i, T(h_i))$ ,  $i = 1, \dots, p$ , and take the value  $P(0)$  as an improved approximation  $Q_{EX}(f)$  to  $I(f)$  – the computed values  $T(h_i)$  are *extrapolated* to the “ideal mesh width  $h = 0$ ”. This strategy goes back to Richardson and Romberg. For sufficient smoothness of  $f$  – it must now be  $2p$  times continuously differentiable – it can be shown that the integration error fulfils

$$Q_{EX}(f) - I(f) = O(h_1^2 h_2^2 \dots h_p^2)$$

(hence a *significantly* higher order of approximation). Extrapolation, thus, is advantageous in all cases of an integration of very smooth functions  $f$ .

## 7.3 Gauß Quadrature

For reasons of simplicity and without loss of generality, we consider the integration interval  $[-1, 1]$  in this section.

Gauß quadrature starts from the above-mentioned approach

$$\int_{-1}^{+1} f(x) dx \approx \sum_{i=1}^n g_i f(x_i)$$

with weights  $g_i$  and nodes  $x_i$ , but, as for the Clenshaw-Curtis rules, we leave the *equidistance* of the nodes. The  $x_i$  are rather parameters of the method themselves and can be chosen in such a way that they fulfil certain additional conditions. Here, we strive for *exact integration*

$$\int_a^b p(x) dx = \sum_{i=1}^n g_i p(x_i)$$

of all polynomials  $p \in \mathbb{P}_k$  with a  $k$  as large as possible. This is driven by the idea of a Taylor expansion of the integrand  $f$ , where as many as possible of the leading terms shall be integrated exactly.

How large can  $k$  be? The degree  $2n$  can surely not be reached, as the simple example of the polynomial  $q \in \mathbb{P}_{2n}$ ,

$$q(x) := \prod_{i=1}^n (x - x_i)^2,$$

shows: The exact integral of  $q$  must be positive, the weighted sum, however, vanishes for each choice of weights  $g_i$ , since the nodes of our integration rule are just the roots of the example polynomial  $q(x)$ .

But,  $k = 2n - 1$  is possible (compare this value with the techniques presented and discussed so far); the number  $k$  is called the *polynomial degree of exactness* of the Gauß quadrature with  $n$  nodes. We skip both derivation and proof – however, it is neither long nor difficult:

- There is exactly one set of nodes  $x_i$  that allows for the maximum polynomial degree of exactness  $2n - 1$ , and these are the pairwise different roots of the *Legendre polynomials*

$$\omega_n(x) := \frac{n!}{(2n)!} D^n ((x^2 - 1)^n).$$

Here, the operator  $D$  stands for derivation. The roots are located in a denser way close to the boundary points of the integration interval than around its centre.

- There is exactly one set of weights  $g_i$  that allows for  $k = 2n - 1$ , and these are

$$g_i := \int_{-1}^{+1} L_i(x) dx$$

with the *Lagrange polynomials*  $L_i \in \mathbb{P}_{n-1}$ . It is obvious that this is at least a necessary condition. The Lagrange polynomials of degree  $n - 1$  must be integrated exactly by Gauß quadrature. In the weighted sum, due to the definition of the  $L_i$ , all summands apart from  $g_i L_i(x_i) = g_i$  are zero. Since the  $L_i^2(x)$  are integrated exactly, too, it follows also that all weights are positive.

## 7.4 Monte-Carlo Quadrature

For integrals of one- or low dimensional functions, the techniques presented so far (or their multidimensional counterparts, respectively) can do the job. This situation changes for really high dimensional integrals:

- In physics, especially in quantum mechanics, we encounter six-, nine-, twelve-, or higher dimensional integrals (three spatial dimensions for each particle).
- In a statistical context, the computation of expectation values is a quite frequent task. For continuous random variables, these are integrals. A large number of characteristics (i.e. dimensions) is quite typical. A prominent and up-to-date example is finance or financial mathematics and here especially *option pricing*, where the current market price of an option shall be determined. Of course, this price should depend on the future price development of the respective shares. Typically, each day's price is treated as a separate dimension – which leads to 365-dimensional integrals for a one-year option.

High dimensional integrals are very problematic, because any conventional discretization turns out to be not feasible. Even the simple trapezoidal rule in each dimension needs  $2^d$  nodes – with  $d = 365$  obviously not realistic for the next decades!

Therefore, especially in physics and statistics, *Monte-Carlo methods*, i.e. *random-based* methods, are very popular. The underlying principle is very simple: Choose the nodes as realizations of a random variable, sample the function here, and form the arithmetic mean:

$$Q_n(f) := \frac{1}{n} \sum_{i=1}^n f(x_i).$$

The *law of large numbers* ensures convergence towards the exact value of the integral (in a probabilistic sense).

We state (without derivation):

- For the quality of the approximation (as before only in a probabilistic sense, i.e. with probability 1 only and not sure – the nodes could “by chance” all be roots of the integrand), it holds

$$|Q_n(f) - I(f)| = O(n^{-1/2}),$$

i.e. better for increasing  $n$  (fortunately).

- This is independent of the integrand’s shape and independent of the dimension. This means that 127 nodes (in the sense of the order of magnitude) always perform the same – in a one- or 100-dimensional integral! This property explains the attractivity of Monte-Carlo methods in the higher dimensional case.
- On the other hand, in the one-dimensional and equidistant case,  $n^{-1/2}$  is just  $\sqrt{n}$  – definitely not a supreme rate of convergence!

## Chapter 8

# Iterative Solution of Systems of Linear Equations

Systems of equations to be solved numerically often stem from the discretization of ordinary and partial differential equations. This holds for the linear as well as for the general non-linear case. For at least two reasons, the *direct* methods studied in Sect. 5 are in general not appropriate:

- First,  $n$  is typically that big (in general,  $n$  is directly correlated with the number of grid points, which – especially for unsteady partial differential equations (three spatial variables plus time) – leads to large  $n$ ), that a computational effort of the order  $O(n^3)$  is not acceptable.
- Second, those matrices are generally *sparse* and have a certain *structure* (tridiagonal, band structure, block structure etc.), which reduces storage requirements and run time; elimination methods may destroy this structure and, hence, the advantages.

Therefore, for such large and sparse matrices or systems of linear equations, *iterative* methods are the state of the art. They start with an *initial guess*  $x^{(0)}$  and generate from that a sequence of approximates  $x^{(i)}$ ,  $i = 1, 2, \dots$ , which converge (convergence assumed) to the exact solution  $x$ . One step of iteration typically costs about  $O(n)$  operations for a sparse matrix (less is hardly possible, if each vector component  $x^{(i)}$  shall be updated in each step). Hence, the number of necessary steps until getting a certain prescribed accuracy will be the crucial criterion during the design of iterative algorithms.

First, we study the classical representative of iterative solvers for linear systems, the so-called *relaxation techniques*. Afterwards, the multigrid principle is briefly presented, which can improve the convergence behaviour of relaxation techniques significantly. Finally, some methods for nonlinear equations and the method of conjugate gradients are discussed.

### 8.1 Classical Relaxation Techniques

The probably oldest iterative schemes for the solution of systems of linear equations  $Ax = b$  with  $A \in \mathbb{R}^{n,n}$  and  $x, b \in \mathbb{R}^n$  are the so-called *relaxation* or *smoothing methods*: *Richardson's* method, *Jacobi's* method, the *Gauß-Seidel* method, and *SOR*, the *Successive Over Relaxation*. The name relaxation or smoothing stems from a certain property we will study later. We start with the algorithms, each followed by a short derivation and convergence analysis.

For all methods mentioned above, the starting point is the *residual*  $r^{(i)}$ ,

$$r^{(i)} := b - Ax^{(i)} = Ax - Ax^{(i)} = A(x - x^{(i)}) = -Ae^{(i)}, \quad (8.1)$$

where  $x^{(i)}$  is the current approximation to the exact solution  $x$  after  $i$  iteration steps and  $e^{(i)}$  denotes the respective error. Since  $e^{(i)}$  is not available (the error can not be given without knowing the exact solution  $x$ ), it is reasonable to consider the vector  $r^{(i)}$  as *direction* in which we want to look for an improvement of  $x^{(i)}$ . The method of Richardson suggests the simplest way and takes the residual directly as a correction to  $x^{(i)}$ . More sophisticated are the Jacobi and the Gauß-Seidel method: For the update of the  $k$ -th component of  $x^{(i)}$ , both determine the step size in the given direction  $r^{(i)}$  such that  $r_k^{(i)}$  vanishes (at least for the moment). The SOR method and its counterpart, the *damped* relaxation, take into account, additionally, that such a correction often goes too far or not far enough, respectively.

Richardson Iteration:

```
for i = 0, 1, ...
  for k = 1, ..., n:     $x_k^{(i+1)} := x_k^{(i)} + r_k^{(i)}$ 
```

Here, the residual  $r^{(i)}$  is directly taken componentwise as a correction for the current approximation  $x^{(i)}$ .

Jacobi Iteration:

```
for i = 0, 1, ...
  for k = 1, ..., n:     $y_k := \frac{1}{a_{kk}} \cdot r_k^{(i)}$ 
  for k = 1, ..., n:     $x_k^{(i+1)} := x_k^{(i)} + y_k$ 
```

That is, in each substep  $k$  of one step  $i$  some correction  $y_k$  is computed and stored. If applied immediately, this correction would lead to a vanishing  $k$ -component of the residual  $r^{(i)}$  (verify by simply inserting). Equation  $k$  would be exactly solved for the current approximation to  $x$  – which, however, can be destroyed immediately during the following substep  $k + 1$ . Note that the corrections are not added immediately, but only at the end of the iteration step (second  $k$ -loop).

Gauß-Seidel Iteration:

```
for i = 0, 1, ...
  for k = 1, ..., n:     $y_k := \frac{1}{a_{kk}} \cdot r_k^{(i)}, \quad x_k^{(i+1)} := x_k^{(i)} + y_k$ 
```

Here, the same correction as with Jacobi is applied. But now, the update is realized immediately and not at the end of the step.

Sometimes, in each of the three sketched methods, a *smoothing* (multiplication of the correction with a factor  $0 < \alpha < 1$ ) or a *over relaxation* (factor  $1 < \alpha < 2$ ) lead to an improved convergence behaviour:

$$x_k^{(i+1)} := x_k^{(i)} + \alpha y_k. \quad (8.2)$$

For Gauß-Seidel, the version with over relaxation is called *SOR* (*Successive Over Relaxation*).

For a short convergence analysis of the above methods, we need an algebraic formulation instead of the algorithmic one. They all are based on the simple idea to write the matrix  $A$  as a sum  $A = M + (A - M)$ , where  $Mx = b$  should be very easy to solve and where the difference  $A - M$  should not be too big with respect to some matrix norm. With the help of such a suitable  $M$ , the Richardson, Jacobi, Gauß-Seidel, and SOR method can be written as

$$Mx^{(i+1)} + (A - M)x^{(i)} = b \quad (8.3)$$

or, solved for  $x^{(i+1)}$ ,

$$x^{(i+1)} := M^{-1}b - M^{-1}(A - M)x^{(i)} = M^{-1}b - (M^{-1}A - I)x^{(i)} = x^{(i)} + M^{-1}r^{(i)}. \quad (8.4)$$

Furthermore, we decompose  $A$  in an additive way into its diagonal part  $D_A$ , its strictly lower triangular part  $L_A$ , and its strictly upper triangular part  $U_A$ :

$$A =: L_A + D_A + U_A. \quad (8.5)$$

With that, the following relations can be shown:

$$\begin{aligned} \text{Richardson : } M &:= I, \\ \text{Jacobi : } M &:= D_A, \\ \text{Gau\ss-Seidel : } M &:= D_A + L_A, \\ \text{SOR : } M &:= \frac{1}{\alpha}D_A + L_A. \end{aligned} \quad (8.6)$$

Looking at the algorithmic formulations of the Richardson and the Jacobi scheme, the first two lines are obvious. Since the Gau\ss-Seidel iteration is a special case of SOR ( $\alpha = 1$ ), it is sufficient to show (8.6) for the general SOR case. From the algorithm, it follows

$$\begin{aligned} x_k^{(i+1)} &:= x_k^{(i)} + \alpha \left( b_k - \sum_{j=1}^{k-1} a_{kj} x_j^{(i+1)} - \sum_{j=k}^n a_{kj} x_j^{(i)} \right) / a_{kk} \\ \Leftrightarrow x^{(i+1)} &:= x^{(i)} + \alpha D_A^{-1} \left( b - L_A x^{(i+1)} - (D_A + U_A) x^{(i)} \right) \\ \Leftrightarrow \frac{1}{\alpha} D_A x^{(i+1)} &= \frac{1}{\alpha} D_A x^{(i)} + b - L_A x^{(i+1)} - (D_A + U_A) x^{(i)} \\ \Leftrightarrow \left( \frac{1}{\alpha} D_A + L_A \right) x^{(i+1)} &+ \left( \left( 1 - \frac{1}{\alpha} \right) D_A + U_A \right) x^{(i)} = b \\ \Leftrightarrow M x^{(i+1)} + (A - M) x^{(i)} &= b, \end{aligned}$$

which shows (8.6) for SOR, too.

Concerning convergence, there are two direct consequences from (8.3):

- If the sequence  $(x^{(i)})$  converges, the limit will be the exact solution  $x$  of our system  $Ax = b$ .
- The *spectral radius*  $\rho$  (i.e. the largest absolute value of all eigenvalues) of the iteration matrix  $-M^{-1}(A - M)$  (i.e. the matrix that is applied to  $x^{(i)}$  in order to get  $x^{(i+1)}$ ) is the decisive quantity for the convergence behaviour:

$$\lim_{i \rightarrow \infty} x^{(i)} = x = A^{-1}b \quad \Leftrightarrow \quad \rho < 1. \quad (8.7)$$

To see that, subtract  $Mx + (A - M)x = b$  from (8.3):

$$M e^{(i+1)} + (A - M) e^{(i)} = 0 \quad \Leftrightarrow \quad e^{(i+1)} = -M^{-1}(A - M) e^{(i)}. \quad (8.8)$$

If all eigenvalues are absolutely smaller than 1 and, hence,  $\rho < 1$  holds, all error components are reduced in each step of iteration.

Concerning the methods' convergence, there are several results, out of which some shall be mentioned:

- A necessary condition for SOR's convergence is  $0 < \alpha < 2$ .
- If  $A$  is positive definite, both SOR (for  $0 < \alpha < 2$ ) and Gau\ss-Seidel converge.
- If  $A$  and  $2D_A - A$  are both positive definite, then Jacobi converges.
- If  $A$  is strictly diagonally dominant (i.e.  $a_{ii} > \sum_{j \neq i} |a_{ij}|$  for all  $i$ ), then Jacobi and Gau\ss-Seidel converge.

- In certain cases, the optimum parameter  $\alpha$  can be determined (minimum  $\rho$ , such that error reduction per step of iteration becomes maximal).

The Gauß-Seidel iteration is not generally better than Jacobi (as one might think due to the immediately executed update). There are examples where the first converges and the latter diverges, and vice versa. In many cases, Gauß-Seidel can do with 50% of the iteration steps Jacobi needs.

Obviously,  $\rho$  is not only crucial for the question whether an iterative method converges at all, but also for its quality, i.e. its speed of convergence: The smaller  $\rho$  is, the faster all components of the error  $e^{(i)}$  are reduced in each iteration step. In practice, the above results concerning convergence are of theoretical value only, often, since  $\rho$  is typically very close to 1, such that – in spite of convergence – the number of necessary steps of iteration to obtain a certain sufficient accuracy is much too large. For partial differential equations, for example, we have the typical scenario that  $\rho$  depends on the problem size  $n$  and, hence, of the resolution of the underlying grid, i.e. for a concrete example

$$\rho = O(1 - h_l^2) = O\left(1 - \frac{1}{4^l}\right) \quad (8.9)$$

for some mesh width  $h_l = 2^{-l}$ . This is a tremendous drawback: The finer and more precise our grid is, the poorer convergence of our iterative scheme becomes. Consequently, better iterative solvers are a must!

## 8.2 The Multigrid Principle

Here, so-called *multigrid methods* can help, which are one of the most fruitful algorithmic developments of modern numerical analysis. The basic idea is simple, but requires some at least superficial knowledge of Fourier analysis. Imagine the error  $e^{(i)}$  in some relaxation scheme to be decomposed into components of different frequencies, as in a Fourier transform. Roughly speaking, this is an expansion where not monomials  $x^k$ , but trigonometric terms  $\sin(kx)$  and  $\cos(kx)$  form the elements of the series;  $k$  gives the frequency. Typically, our relaxation schemes reduce those error components which are highly oscillating with respect to the underlying grid very fast, whereas the relatively low-frequency components are only reduced slightly – they *smooth* the error (therefore the name smoother). Figure 8.1 shows this phenomenon for a very simple example of a one-dimensional Laplace equation (i.e.  $u''(x) = 0$ ) to be solved of an equidistant grid of mesh width  $h = 2^{-6}$ . Left, the (randomly chosen) initial error  $e^{(0)}$  is shown, whereas right, we can see the error after 100 steps if iteration (actually very much for such a simple example) of a damped Jacobi iteration of damping parameter  $\alpha = 0.5$ . As we can see clearly, the error is smooth now: the highly oscillating have nearly been removed, whereas the others are still present. The multigrid algorithm presented in the following tackles all frequencies successfully after two(!) steps only – as we can learn from the hardly visible curve oscillating around zero in the right part of the figure.

The consequence? Obviously, a (too) fine grid has problems with the removal of long-wave error components. The short-wave ones, however (as long as they are representable on the grid at all, of course) are reduced efficiently. Since the terms ‘long-wave’ etc. are relative ones, it seems straightforward to switch to a coarser grid for reducing the low-frequency components. This is also an economic decision, since coarser grids entail a smaller computational effort. A recursive extension of this two-grid approach (the components still oscillating on the coarse grid will be tackled on even coarser grids etc.) leads us to *multigrid methods*.

The simplest strategy, i.e. to discretize a given PDE on several grids of different resolution in an independent way and to solve the resulting systems of linear equations via relaxation techniques afterwards does not help: In each “solution”, only part of the job has been done – how shall the information be put together? A better and successful strategy are *correction schemes*, where the computations on the coarser grid are considered as a correction of the fine grid solution. For reasons of simplicity, we restrict ourselves to the case of two regular grids. Take an approximation

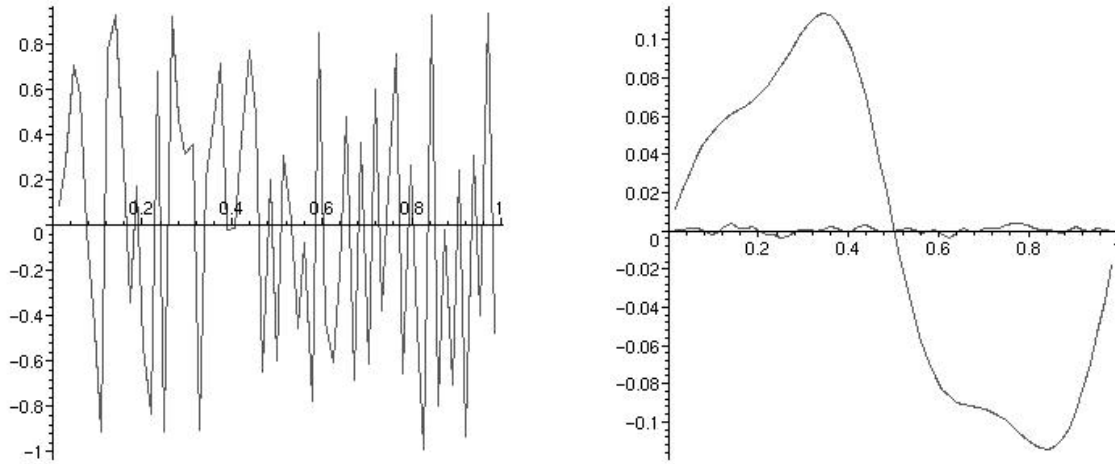


Figure 8.1: Error smoothing with relaxation techniques

$x_l$  for the mesh width  $h_l = 2^{-l}$ , and let  $A_l$ ,  $b_l$ , and  $\Omega_l$  denote the corresponding matrix, right-hand side, and underlying grid, resp. In most cases, coarsening means switching to double mesh width  $h_{l-1} = 2^{-(l-1)}$ , i.e. to a grid  $\Omega_{l-1}$  with a number of grid points reduced by a factor of  $2^d$  for a  $d$ -dimensional problem. Now, we can formulate the following algorithm for *coarse grid correction*:

- smooth the current approximation  $x_l$ ;
  - form the residual  $b_l - A_l x_l$ ;
  - restrict  $r_l$  to the coarse grid  $\Omega_{l-1}$ ;
  - compute the solution  $e_{l-1}$  for  $A_{l-1} e_{l-1} = r_{l-1}$ ;
  - prolongate  $e_{l-1}$  to the fine grid  $\Omega_l$ ;
  - add the resulting correction to  $x_l$  (possibly, smooth again);
- (8.10)

Some remarks on the single steps:

- **presmoothing**: smoothes the error (i.e. reduces the high-frequency error components); typically, a few damped Jacobi or Gauß-Seidel steps
- **restriction**: simply adopting the values in the coarse grid points (injection) or appropriate averaging process via values in neighbouring fine grid points (*full weighting*), see Fig. 8.2
- **coarse grid equation**: solved on the fine grid, correction equation would lead us in one step to the solution:

$$A_l(x_l + e_l) = Ax_l + Ae_l = b_l - r_l + r_l = b_l; \quad (8.11)$$

on the coarse grid (i.e.  $A_{l-1}e_{l-1} = r_{l-1}$ ), we get some correction only; the coarse grid equation can be solved directly (if  $\Omega_{l-1}$  is already coarse enough), via relaxation, or recursively via another coarse grid correction (then, we have a real multigrid method)

- **prolongation**: the computed coarse grid correction  $e_{l-1}$  must be interpolated back to  $\Omega_l$  again (e.g. directly adopting the value in the coarse grid points and averaging in the fine grid points, see Fig. 8.2)

- **postsMOOTHING**: sometimes, some smoothing steps *after* the coarse grid correction are helpful

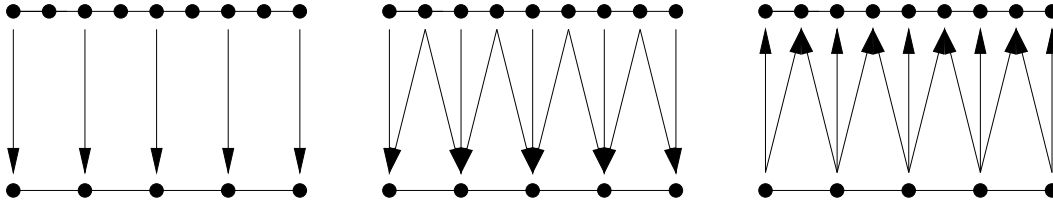


Figure 8.2: Restriction (injection (left) and full weighting (centre)) and prolongation (right) between two grids  $\Omega_l$  and  $\Omega_{l-1}$  in 1D

If the coarse grid equation itself is solved by a coarse grid correction step, we get from a 2-grid to a multigrid algorithm with a hierarchy of grids  $\Omega_k$ ,  $k = l, \dots, 1$ . On the coarsest grid  $\Omega_1$ , we can often solve directly (only a few unknowns). Depending on the order of the visits of the different grids in the recursion, we get different algorithmic variants, see Fig. 8.3.

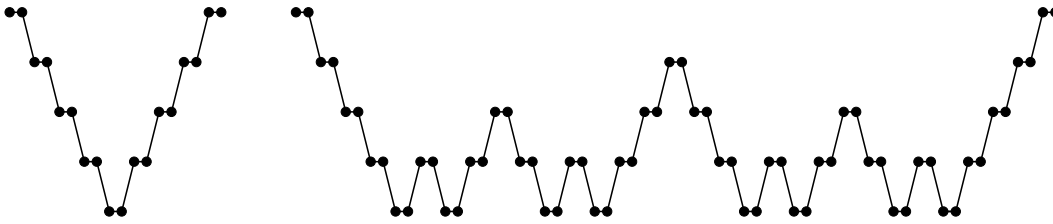


Figure 8.3: V-cycle (left) and W-cycle (right; fine above, coarse below)

What is the gain of multigrid methods compared with simple relaxations?

- First, the overall work is dominated by the finest grid. If  $C$  denotes the number of arithmetic operations for one smoothing step on  $\Omega_l$ ,  $C/2$  or  $C/4$  or  $C/8$  operations have to be performed on  $\Omega_{l-1}$  in 1D, 2D, or 3D, resp. A corresponding result holds for restriction and prolongation. The overall cost of a V-cycle, hence, is of the same order as the cost of a single relaxation step on the finest occurring grid (geometric series), i.e.  $kC$  with a small  $k$ . The same holds for memory requirements. In this sense, a multigrid step does not cost more than a simple relaxation step.
- Concerning speed of convergence, the spectral radius  $\rho$  of the multigrid iteration matrix does not depend on the number  $n$  of unknowns and, thus, on the resolution of the discretization in many cases. Reduction rates of 0.2 and smaller per step of iteration are quite frequent.

These two properties show that multigrid methods are optimal iterative methods. Therefore, today, the multigrid principle is used in a very general sense and in various variants or sophistications of the basic scheme presented here.

### 8.3 Nonlinear Equations

Now, we know how systems of linear equations can be solved – with direct or iterative methods. The solution of  $Ax = b$  can also be interpreted as the process of looking for a root of the linear function  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ ,  $F(x) := b - Ax$ . Unfortunately, the world is not linear but nonlinear. Therefore, we have to tackle the solution of *nonlinear equations*, too. In general, this can also be

done in an iterative way. Since we want to do without calculus of several variables, we restrict ourselves in the following to the simple case  $n = 1$ , i.e. *one* nonlinear equation.

- consider one continuously differentiable (nonlinear) function  $f : \mathbb{R} \rightarrow \mathbb{R}$ , with a root in  $\bar{x} \in ]a, b[$
- we start from some (still to be determined) iterative scheme:
  - we get a sequence of approximate values  $(x^{(i)})$ ,  $i = 0, 1, \dots$ , which (hopefully) converges to one or the root  $\bar{x}$  of  $f(x)$
  - as a measure for the *speed of convergence*, we consider the reduction of the error in each step, and, in the case of convergence,

$$|x^{(i+1)} - \bar{x}| \leq c \cdot |x^{(i)} - \bar{x}|^\alpha,$$

depending on the maximum possible value of  $\alpha$ , we speak of *linear* ( $\alpha = 1$  and, additionally,  $0 < c < 1$ ) or *quadratic* ( $\alpha = 2$ ) convergence etc.

- there are *conditionally* or *locally* convergent methods, for which convergence is only sure for sufficiently good initial values  $x^{(0)}$ , and *unconditionally* or *globally* convergent methods, where the iteration leads to a root of  $f$  for each starting point
- the *method of bisection*:  
we start from  $[c, d] \subseteq [a, b]$  with  $f(c) \cdot f(d) < 0$ ; continuity of  $f$  guarantees the existence of (at least) one root in  $[c, d]$ ; halvening the interval  $[c, d]$  allows – depending on the sign of  $f((c+d)/2)$  – to restrict search to one of the subintervals  $[c, (c+d)/2]$  or  $(c+d)/2, d]$ , etc.; we stop if we reach a root ( $f((c+d)/2) = 0$ ) or when  $d - c$  falls below some tolerance  $\varepsilon$
- *regula falsi*:  
a variant of bisection, where not the interval's centre, but the root of the connection of  $(c, f(c))$  and  $(d, f(d))$  is chosen as one boundary point of the new and smaller interval (such that  $f$  again has different sign at both boundary points); note that regula falsi is not necessarily faster than simple bisection!
- *secant method*:  
start with *two* initial guesses  $x^{(0)}$  and  $x^{(1)}$ ; in the following,  $x^{(i+1)}$  is determined based upon  $x^{(i-1)}$  and  $x^{(i)}$ , just by looking for the root of the line through  $(x^{(i-1)}, f(x^{(i-1)}))$  and  $(x^{(i)}, f(x^{(i)}))$  (the *secant*  $s(x)$ ):

$$\begin{aligned} s(x) &:= f(x^{(i)}) + \frac{f(x^{(i)}) - f(x^{(i-1)})}{x^{(i)} - x^{(i-1)}} \cdot (x - x^{(i)}), \\ 0 = s(x^{(i+1)}) &= f(x^{(i)}) + \frac{f(x^{(i)}) - f(x^{(i-1)})}{x^{(i)} - x^{(i-1)}} \cdot (x^{(i+1)} - x^{(i)}), \\ x^{(i+1)} &:= x^{(i)} - (x^{(i)} - x^{(i-1)}) \cdot \frac{f(x^{(i)})}{f(x^{(i)}) - f(x^{(i-1)})} \end{aligned}$$

- *Newton's method*:  
here, one starts with *one* initial guess  $x^{(0)}$  and gets in the following  $x^{(i+1)}$  from  $x^{(i)}$  by determining the root of the *tangential line*  $t(x)$  to  $f(x)$  in  $x^{(i)}$  (*linearization*: replace  $f$  by its tangential line or by its Taylor polynomial of first degree):

$$\begin{aligned} t(x) &:= f(x^{(i)}) + f'(x^{(i)}) \cdot (x - x^{(i)}), \\ 0 = t(x^{(i+1)}) &= f(x^{(i)}) + f'(x^{(i)}) \cdot (x^{(i+1)} - x^{(i)}), \\ x^{(i+1)} &:= x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})} \end{aligned}$$

(Newton's method corresponds to the secant method in the limit case of  $x^{(i-1)} = x^{(i)}$ )

- speed of convergence: globally linear for bisection and regula falsi, locally quadratic for Newton and locally 1.618 for the secant method; since one Newton step requires one evaluation of  $f$  and  $f'$ , it has to be compared with *two* steps of the other methods  
 $\Rightarrow$  the secant method looks quite good!
- for roots of polynomials of higher degree (often an extremely badly-conditioned problem) there are special techniques

Note that in most applications in practice  $n \gg 1$  (unknowns are functional values in grid points!) holds, and that we must deal with very large nonlinear systems of equations. In more than one dimension, the simple derivative  $f'(x)$  must be replaced by the so-called *Jacobian matrix*  $F'(x)$ , the matrix of first partial derivatives of all vector components of  $F$  with respect to all variables. Hence, Newton's formula reads as

$$x^{(i+1)} := x^{(i)} - F'(x^{(i)})^{-1} F(x^{(i)}),$$

where, of course, the matrix  $F'(x)$  must not be inverted. Rather, the corresponding system of linear equations with the right-hand side  $-F(x^{(i)})$  is solved directly:

```

compute  $F'(x)$ ;
decompose  $F'(x) := LR$ ;
solve  $LRs = -F(x)$ ;
update  $x := x + s$ ;
evaluate  $F(x)$ ;

```

For large  $n$ , the repeated calculation of the Jacobian becomes very costly and may be even possible only approximately, since numerical differentiation is quite frequent. Furthermore, the direct solution of a linear system in each Newton step is not cheap. Therefore, in the case  $n >$ , Newton's method has been a starting point for many algorithmic developments:

- the *Newton-chord* and the *Shamanskii method*:  
 here, the Jacobian is not calculated and inverted in each Newton step, but we always use  $F'(x^{(0)})$  (chord) or use some  $F'(x^{(i)})$  for several successive steps (Shamanskii)
- the *inexact Newton method*:  
 here, the linear system in each Newton step is not solved directly (i.e. exactly, for example via  $LR$ -decomposition), but iteratively; we speak of an *inner* iteration within the (*outer*) Newton iteration
- *quasi-Newton methods*:  
 here, a sequence  $B^{(i)}$  of approximations to  $F'(\bar{x})$  is generated – not via expensive recalculations, but via cheap *updates*; we profit from the fact that a *rank-1 update* ( $B + uv^T$ ) with two arbitrary vectors  $u, v \in \mathbb{R}^n$  (invertible if and only if  $1 + v^T B^{-1} u \neq 0$ ) is easy to invert:

$$(B + uv^T)^{-1} = \left( I - \frac{(B^{-1}u)v^T}{1 + v^T B^{-1}u} \right) B^{-1};$$

Broyden suggested a suitable choice for  $u$  and  $v$  ( $s$  as in Newton's algorithm):

$$B^{(i+1)} := B^{(i)} + \frac{F(x^{(i+1)})s^T}{s^T s};$$

Note that the multigrid principle from the last section can be and is applied successfully to nonlinear scenarios, too.

## 8.4 The Method of Conjugate Gradients

Once more back to linear equations: One of today's most important methods, the method of *conjugate gradients* (*cg*), is based upon another principle than relaxation or smoothing. To see

this, we approach the solution of linear equations via some detour.

In the following, let  $A \in \mathbb{R}^{n,n}$  be a symmetric and positive definite matrix, i.e.  $A = A^T$  and  $x^T A x > 0 \forall x \neq 0$ . In this case, solving  $Ax = b$  is equivalent to minimizing the quadratic form

$$f(x) := \frac{1}{2}x^T A x - b^T x + c \quad (8.12)$$

for an arbitrary scalar  $c \in \mathbb{R}$ . Since  $A$  is positive definite, the hypersurface defined by  $z := f(x)$  forms a paraboloidal in  $\mathbb{R}^{n+1}$  with  $n$ -dimensional ellipsoidals as isosurfaces  $f(x) = \text{const.}$ , and  $f$  has one global minimum  $x$ . The problems' equivalence is obvious:

$$f'(x) = \frac{1}{2}A^T x + \frac{1}{2}Ax - b = Ax - b = -r(x) = 0 \Leftrightarrow Ax = b. \quad (8.13)$$

This means: Each disturbance  $e \neq 0$  of the solution  $x$  of  $Ax = b$  increases the value of  $f$  due to (8.12), the point  $x$ , hence, denotes indeed a minimum:

$$\begin{aligned} f(x+e) &= \frac{1}{2}x^T A x + e^T A x + \frac{1}{2}e^T A e - b^T x - b^T e + c \\ &= f(x) + e^T (Ax - b) + \frac{1}{2}e^T A e \\ &= f(x) + \frac{1}{2}e^T A e > f(x). \end{aligned} \quad (8.14)$$

Consequently, we must now look for strategies for finding minima. The so-called *steepest descent* provides a straightforward approach for that. For  $i = 0, 1, \dots$ , repeat

$$\begin{aligned} r^{(i)} &:= b - Ax^{(i)}, \\ \alpha_i &:= \frac{r^{(i)T} r^{(i)}}{r^{(i)T} A r^{(i)}}, \\ x^{(i+1)} &:= x^{(i)} + \alpha_i r^{(i)}, \end{aligned} \quad (8.15)$$

or begin with  $r^{(0)} := b - Ax^{(0)}$  and repeat for  $i = 0, 1, \dots$

$$\begin{aligned} \alpha_i &:= \frac{r^{(i)T} r^{(i)}}{r^{(i)T} A r^{(i)}}, \\ x^{(i+1)} &:= x^{(i)} + \alpha_i r^{(i)}, \\ r^{(i+1)} &:= r^{(i)} - \alpha_i A r^{(i)}, \end{aligned} \quad (8.16)$$

which saves one of the matrix-vector products. The method of steepest descent looks for an improvement in the direction of the negative gradient  $-f'(x^{(i)}) = r^{(i)}$ , which, in fact, indicates the direction of  $f$ 's steepest descent. Of course, it would be more natural to search in the direction of the error  $e^{(i)} := x^{(i)} - x$ , but the error is not known, unfortunately. Apart from the direction, we also need a suitable step width. For that, we look for the minimum of  $f(x^{(i)} + \alpha_i r^{(i)})$  as a function of  $\alpha_i$  (set partial derivative with respect to  $\alpha_i$  to zero), which provides the above algorithm after a short calculation.

If we use the unit vectors in the coordinate directions as search directions instead of the residuals  $r^{(i)}$  and look for a best improvement in these directions, we get the Gauß-Seidel iteration, by the way. Hence, despite the different approach, there are relations between relaxation and minimization! However, steepest descent has no smoothing property: high-frequency components can increase during the iteration.

The convergence behaviour of steepest descent is rather poor. One of the rare trivial special cases is the identity matrix: Here, the isosurfaces are spheres, the negative gradient always points to the

sphere's centre (the minimum), and we reach the minimum in one step of iteration only! Generally, we come arbitrarily close to the minimum, but it can take an arbitrarily long time (since some of the progress achieved so far can be destroyed again afterwards).

Therefore, we look for better search directions in our minimization approach. If all of them were orthogonal, and if the error after  $i$  steps were orthogonal to all of the  $i$  search directions visited so far, then, a progress achieved could never be lost again, and we would be in the exact minimum after at most  $n$  steps – as for some direct solver. Due to this property, cg and its derivatives are called *semi-iterative methods*.

Instead of the residuals or negative gradients  $r^{(i)}$ , we now look for alternative directions  $d^{(i)}$ ,  $i = 0, 1, \dots$ :

$$x^{(i+1)} := x^{(i)} + \alpha_i d^{(i)}. \quad (8.17)$$

Suppose for a moment the above-mentioned ideal case: The current error  $e^{(i)}$  shall be orthogonal to the subspace spanned by  $d^{(0)}, \dots, d^{(i-1)}$ . Then, the new correction  $\alpha_i d^{(i)}$  should be chosen such that  $d^{(i)}$  is orthogonal to  $d^{(0)}, \dots, d^{(i-1)}$  and that the new error  $e^{(i+1)}$  is orthogonal to  $d^{(i)}$ , i.e.

$$0 = d^{(i)T} e^{(i+1)} = d^{(i)T} (e^{(i)} + \alpha_i d^{(i)}) \Rightarrow \alpha_i := -\frac{d^{(i)T} e^{(i)}}{d^{(i)T} d^{(i)}}. \quad (8.18)$$

Unfortunately, we do not know  $e^{(i)}$ . Therefore, we replace errors by residuals and construct *A-orthogonal* or *conjugate* search directions instead of orthogonal ones. Two vectors  $u \neq 0$  and  $v \neq 0$  are called *A-orthogonal* or *conjugate*, if  $u^T A v = 0$ . The corresponding requirement to  $e^{(i+1)}$  now reads

$$0 = d^{(i)T} A e^{(i+1)} = d^{(i)T} A (e^{(i)} + \alpha_i d^{(i)}) \Rightarrow \alpha_i := -\frac{d^{(i)T} A e^{(i)}}{d^{(i)T} A d^{(i)}} = \frac{d^{(i)T} r^{(i)}}{d^{(i)T} A d^{(i)}}. \quad (8.19)$$

It is interesting to note that the above condition is equivalent to searching the minimum in direction  $d^{(i)}$ , as we proceeded during steepest descent:

$$0 = \frac{\partial f}{\partial \alpha_i}(x^{(i+1)}) = f'(x^{(i+1)}) \frac{\partial x^{(i+1)}}{\partial \alpha_i} = -r^{(i+1)T} d^{(i)} = d^{(i)T} A e^{(i+1)}.$$

In contrast to (8.18), all expressions in (8.19) are known. Hence, we get the following algorithm – however still without a concrete rule how we can construct conjugate directions. One starts with  $d^{(0)} := r^{(0)} = b - Ax^{(0)}$  and iterates for  $i = 0, 1, \dots$ :

$$\begin{aligned} \alpha_i &:= \frac{d^{(i)T} r^{(i)}}{d^{(i)T} A d^{(i)}}, \\ x^{(i+1)} &:= x^{(i)} + \alpha_i d^{(i)}, \\ r^{(i+1)} &:= r^{(i)} - \alpha_i A d^{(i)}. \end{aligned} \quad (8.20)$$

A comparison of (8.16) and (8.20) shows the close relations of this methods to steepest descent.

Now, we must construct conjugate directions. This can be done via a *Gram-Schmidt A-orthogonalization*, which is defined in an analogous way to standard Gram-Schmidt orthogonalization well-known from linear algebra (start from a basis  $u^{(0)}, \dots, u^{(n-1)}$  and subtract from  $u^{(i)}$  all components that are not *A-orthogonal* to  $d^{(0)}, \dots, d^{(i-1)}$ ). It can be shown that the resulting algorithm is equivalent to a standard Gauß elimination, if we choose the unit vectors as the underlying basis. However, this is of course not what we were looking for – we must look for a better performance.

If the unit vectors do not help, try the residuals as the underlying basis  $r^{(i)}$ . An analysis of this strategy reveals interesting and surprising properties:

- the residuals are orthogonal:  $r^{(i)T} r^{(j)} = 0$  for  $i \neq j$

- $r^{(i+1)}$  is orthogonal to  $d^{(0)}, \dots, d^{(i)}$
- $r^{(i+1)}$  is conjugate to  $d^{(0)}, \dots, d^{(i-1)}$

Hence, we get most for free. An explicit conjugation of  $r^{(i+1)}$  is necessary only with respect to  $d^{(i)}$ . These results provide all ingredients for the cg algorithm. Start with  $d^{(0)} := r^{(0)} = b - Ax^{(0)}$  and iterate for  $i = 0, 1, \dots$ :

$$\begin{aligned}
 \alpha_i &:= \frac{r^{(i)T} r^{(i)}}{d^{(i)T} A d^{(i)}}, \\
 x^{(i+1)} &:= x^{(i)} + \alpha_i d^{(i)}, \\
 r^{(i+1)} &:= r^{(i)} - \alpha_i A d^{(i)}, \\
 \beta_{i+1} &:= \frac{r^{(i+1)T} r^{(i+1)}}{r^{(i)T} r^{(i)}}, \\
 d^{(i+1)} &:= r^{(i+1)} + \beta_{i+1} d^{(i)}.
 \end{aligned} \tag{8.21}$$

Two remarks: First, we get  $d^{(i)T} r^{(i)} = r^{(i)T} r^{(i)}$ . Therefore, the scalar product with  $d$  is not necessary for computing  $\alpha_i$ . Second, the parameter  $\beta_{i+1}$  contains the conjugation of  $r^{(i+1)}$  with respect to  $d^{(i)}$ .

In principle, the cg algorithm is a direct solver. However, numerical problems (round-off errors and cancellations, which, in particular, might disturb the  $A$ -orthogonality of the search directions) and the size of  $n$  in realistic applications are responsible for the fact that cg is used as an iterative solver only (nobody could await  $n$  steps of iteration). Again, as for relaxation methods, the speed of convergence depends on the mesh width for discretized differential equations: the finer the grid's resolution is, the slower the process converges. Fortunately, there is a remedy for cg, too: *preconditioning*: Instead of  $Ax = b$ , solve  $M^{-1}Ax = M^{-1}b$ , where the *preconditioner*  $M$  provides an improved convergence behaviour. The ideal choice of  $M^{-1} := A^{-1}$  is not feasible, since computing the preconditioner would cost more than the original problem. Thus, the critical point is to construct a both simple and effective preconditioner. Again, the multigrid or multilevel principle plays an important part.



# Chapter 9

## Ordinary Differential Equations

### 9.1 Preparatory Remarks

The probably most important field of application for numerical methods are *differential equations*, i.e. equations that describe relations of functions and their derivatives. We distinguish

- *ordinary* differential equations (ODE), where only one variable occurs (typically time); examples of simple application scenarios are
  - the oscillation of a pendulum  $\ddot{y}(t) = -y(t)$  with the solution  $y(t) = c_1 \cdot \sin(t) + c_2 \cdot \cos(t)$ ;
  - the exponential growth  $\dot{y}(t) = y(t)$  with the solution  $y(t) = c \cdot e^t$ ;
- *partial* differential equations (PDE), where several variables occur (several spatial coordinates and, possibly, time); simple examples:
  - the *Poisson equation* in 2D describes, for example, the form of a membrane fixed at the boundary and subject to an external load  $f$ :

$$\Delta u(x, y) := u_{xx}(x, y) + u_{yy}(x, y) = f(x, y) \quad \text{on } [0, 1]^2$$

(here, finding a closed form of the solution is already much harder!)

- the *heat equation* in 1D describes, for example, the temperature distribution in a wire for given temperature at the wire's end points:

$$u_t(x, t) = u_{xx}(x, t) \quad \text{in } [0, 1]^2$$

unsteady equation (time-dependence!)

The differential equation does not uniquely define the solution (see the constants in the solutions of the ODE given above) – additional conditions must be given, as mentioned above in words: the membrane is fixed at the boundary, the pendulum has an initial position and speed, etc.). That is, we look for the function  $u$  that fulfils the differential equation *and* the respective condition. Furthermore, single equations are the exception – typically, we encounter systems of differential equations.

In the following, we discuss some basics concerning the numerical solution of ODE. For remarks on PDE, see the next section.

- To enforce uniqueness of the solution, we can prescribe *initial conditions* (concerning the strength of some population at the beginning of the period of observation in population

dynamics) or *boundary conditions* (a space shuttle shall be launched and land at defined positions) – depending on the problem. Accordingly, we speak of *initial value problems (IVP)* or *boundary value problems (BVP)*.

- In simple cases, ODE can be solved analytically:
  - For the above example of  $\dot{y}(t) = y(t)$ , the solution is obvious.
  - Sometimes, we can apply special techniques like the *separation of variables*:

$$\begin{aligned}\dot{y}(t) &= t \cdot y(t) \\ \frac{1}{y(t)} \cdot \frac{dy}{dt} &= t \\ \frac{1}{y} \cdot dy &= t dt \\ \int_{y_0}^y \frac{1}{\eta} \cdot d\eta &= \int_{t_0}^t \tau d\tau \\ \ln(y) - \ln(y_0) &= \frac{t^2}{2} - \frac{t_0^2}{2} \\ y(t) &= y_0 \cdot e^{t^2/2} \cdot e^{-t_0^2/2}.\end{aligned}$$

Obviously, this function  $y(t)$  solves the given ODE and fulfils  $y(t_0) = y_0$ ! Attention! Our proceeding a bit naive: Are we allowed to divide by  $y(t)$ , is the logarithm of  $y(t)$  defined at all etc.?

- In many cases, statements about the existence of solutions are possible; for example, the so-called *Lipschitz condition*

$$\|f(t, y_1) - f(t, y_2)\| \leq L \cdot \|y_1 - y_2\|$$

for the IVP

$$\dot{y}(t) = f(t, y(t)), \quad y(a) = y_a, \quad t \in [a, b]$$

guarantees both existence and uniqueness of its solution.

But let us turn to numerical issues. The worst case are – as always – ill-conditioned problems; here, the use of numerical techniques is critical. We study one small example to see the problem:

- IVP:
  - ODE:  $\ddot{y}(t) - N\dot{y}(t) - (N+1)y(t) = 0, \quad t \geq 0$
  - initial conditions (two due to second derivative):  $y(0) = 1, \dot{y}(0) = -1$
- solution:  $y(t) = e^{-t}$
- now the disturbed initial condition:  $y_\varepsilon(0) = 1 + \varepsilon$ , everything else as before
- new solution:  $y_\varepsilon(t) = (1 + \frac{N+1}{N+2}\varepsilon)e^{-t} + \frac{\varepsilon}{N+2}e^{(N+1)t}$
- one can see:  $y(t)$  and  $y_\varepsilon(t)$  are completely different; in particular,  $y(t)$  tends to zero with  $t \rightarrow \infty$ , whereas  $y_\varepsilon(t)$  grows without a bound – for arbitrary  $\varepsilon > 0$ !
- i.e.: smallest disturbances can have disastrous consequences!

## 9.2 Approximation by Finite Differences

In the following, we consider the general IVP of first order (first derivative only)

$$\dot{y}(t) = f(t, y(t)), \quad y(a) = y_a, \quad t \in [a, b]$$

and assume the existence of a unique solution.

- If  $f$  does not depend on its second argument  $y$ , the task becomes a simple integration problem!
- We start with the discretization, i.e. here: replace derivatives or *differential quotients* by *difference quotients* or *finite differences*, for example

$$\frac{y(t + \delta t) - y(t)}{\delta t} \quad \text{or} \quad \frac{y(t) - y(t - \delta t)}{\delta t} \quad \text{or} \quad \frac{y(t + \delta t) - y(t - \delta t)}{2 \cdot \delta t}$$

for  $\dot{y}(t)$  or, for IVP of higher order,

$$\frac{\frac{y(t + \delta t) - y(t)}{\delta t} - \frac{y(t) - y(t - \delta t)}{\delta t}}{\delta t} = \frac{y(t + \delta t) - 2 \cdot y(t) + y(t - \delta t)}{(\delta t)^2}$$

for  $\ddot{y}(t)$ .

- The first of the above approximations to  $\dot{y}(t)$  leads to

$$\begin{aligned} y(a + \delta t) &\approx y(a) + \delta t \cdot f(t, y(a)), \quad \text{i.e.} \\ y_{k+1} &:= y_k + \delta t \cdot f(t_k, y_k), \quad t_k = a + k\delta t, \quad k = 0, 1, \dots, N, \quad a + N \cdot \delta t = b \end{aligned}$$

as simplest scheme for the generation of discrete approximations  $y_k$  to  $y(t_k)$ . This method is called *Euler's method*.

- Besides Euler's method, there are several other schemes, for example
  - the method of *Heun*:

$$y_{k+1} := y_k + \frac{\delta t}{2} (f(t_k, y_k) + f(t_{k+1}, y_k + \delta t f(t_k, y_k)))$$

(here,  $f(t_k, y_k)$  is replaced by a “better” approximation) or

- the method of *Runge* and *Kutta* that goes even one step further in this direction:

$$y_{k+1} := y_k + \frac{\delta t}{6} (T_1 + 2T_2 + 2T_3 + T_4)$$

with

$$\begin{aligned} T_1 &:= f(t_k, y_k), \\ T_2 &:= f\left(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} T_1\right), \\ T_3 &:= f\left(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} T_2\right), \\ T_4 &:= f(t_{k+1}, y_k + \delta t T_3). \end{aligned}$$

Obviously, both are discretizations of our IVP; what is the advantage of these more complicated rules? Of course an higher accuracy of the produced approximation to  $y(t)$ !

## 9.3 Consistency and Convergence

Now, we want to clarify the term of a method's *accuracy*. Two things have to be distinguished:

- the error that occurs *locally* in each point just by replacing derivatives by differences – even if we use the exact  $y(t)$ , i.e. without considering approximations;

- the error accumulated *globally* during the solution process.

According to that, we distinguish two kinds of a discretization error:

- *local discretization error*:

- the maximum error that exists just by the transition from differential to difference quotients – even if both are based on the exact solution  $y(t)$ ; in the Euler case, this means

$$l(\delta t) := \max_{a \leq t \leq b - \delta t} \left\{ \frac{y(t + \delta t) - y(t)}{\delta t} - f(t, y(t)) \right\} .$$

- If  $l(\delta t) \rightarrow 0$  for  $\delta t \rightarrow 0$ , the discretization scheme is called *consistent*.
- Obviously, consistency is the minimum we have to require – without consistency, a method is useless.

- *global discretization error*:

- the maximum error between the calculated approximations  $y_k$  and the corresponding exact values  $y(t_k)$ :

$$e(\delta t) := \max_{k=0, \dots, N} \{|y_k - y(t_k)|\} .$$

- This is the quantity we are finally interested in – the global discretization error indicates the quality of the approximation produced by our method!
- If  $e(\delta t) \rightarrow 0$  for  $\delta t \rightarrow 0$ , the discretization scheme is called *convergent*; note: convergence is more than consistency!

- All three methods introduced so far are consistent and convergent:

- Euler: method of *first order*, i.e.  $l(\delta t) = O(\delta t)$  and  $e(\delta t) = O(\delta t)$
- Heun: method of *second order*, i.e.  $l(\delta t) = O((\delta t)^2)$  and  $e(\delta t) = O((\delta t)^2)$
- Runge-Kutta: method of *fourth order*, i.e.  $l(\delta t) = O((\delta t)^4)$  and  $e(\delta t) = O((\delta t)^4)$

This shows the difference in quality: The higher the order, the more benefit can be taken from investing more effort (i.e. halving the step size  $\delta t$ , for example: doing this, the error is reduced asymptotically by a factor of 2 (Euler), 4 (Heun), or 16 (Runge-Kutta))

- Of course, we are not yet pleased. The number of *evaluations* of  $f$  for different arguments did significantly increase (cf. the Runge-Kutta formulas:  $T_2$ ,  $T_3$ , and  $T_4$  each require an additional evaluation of  $f$ ). In numerical practice,  $f$  is typically very complicated, and an evaluation of  $f$  entails high computational work. Therefore, we will look for alternatives.

## 9.4 Multistep Methods

- All methods discussed so far are so-called *one-step methods*: For getting  $y_{k+1}$ , no other previous points of time than  $t_k$  are used (but – as mentioned – new positions in-between).
- This does not hold for *multistep methods*: Here, no additional points of evaluation of  $f$  are chosen, but older ones are recycled, for example in the *Adams-Bashforth method of second order*:

$$y_{k+1} := y_k + \frac{\delta t}{2} (3f(t_k, y_k) - f(t_{k-1}, y_{k-1}))$$

(second order can be shown easily).

- In analogy to that, methods of still higher order can be derived, if still older ancestors are taken into account (principle: replace  $f$  by a polynomial  $p$  of suitable degree that interpolates  $f$  in the  $(t_i, y_i)$  considered, and use this  $p$  according to

$$y_{k+1} := y_k + \int_{t_k}^{t_{k+1}} \dot{y}(t) dt = y_k + \int_{t_k}^{t_{k+1}} f(t, y(t)) dt \approx y_k + \int_{t_k}^{t_{k+1}} p(t) dt,$$

to get  $y_{k+1}$ ).

- At the beginning (as long as not enough "old" points are available), some suitable one-step method is used.
- Multistep methods show that consistency and convergence are not equivalent. The above Adams-Bashforth methods are both consistent and convergent, the midpoint rule, however, produces a counterexample:

$$y_{k+1} := y_{k-1} + 2\delta t f_k;$$

a consistent 2-step method of second order which is not generally convergent. To see this, apply the midpoint rule to the following IVP:

$$\dot{y}(t) = -2y(t) + 1, \quad y(0) = 1, \quad t \geq 0, \quad \text{solution: } y(t) = \frac{1}{2}(e^{-2t} + 1).$$

We get the rule

$$y_{k+1} = y_{k-1} + 2\delta t(-2y_k + 1) = y_{k-1} - 4\delta t y_k + 2\delta t, \quad y_0 = 1.$$

With the exact values  $y(0)$  and  $y(\delta t)$  as initial values, the following results are computed:

$\delta t$	$y_9$	$y_{10}$	$y_{79}$	$y_{80}$	$y_{999}$	$y_{1000}$
1.0	-4945.9	20953.9				
0.1	0.5820	0.5704	-1725.3	2105.7		
0.01	0.9176	0.9094	0.6030	0.6010	-154.6	158.7

With other words: For each arbitrarily small step size  $\delta t$ , the sequence of the computed  $y_k$  oscillates with non-bounded absolute values for  $k \rightarrow \infty$ , instead of converging towards the exact solution  $1/2$ . Hence, we have consistency, but *no convergence!*

By the way: The reason for that is that the midpoint rule is not *stable*, but we won't discuss stability here.

## 9.5 Stiffness, Implicit Methods

As a last phenomenon, we consider *stiff* differential equations.

- again, a simple IVP as an example:
  - equation:  $\dot{y} = -1000y + 1000, \quad t \geq 0$
  - initial condition:  $y(0) = y_0 = 2$
  - This IVP is well-conditioned: The disturbed initial condition  $y(0) = 1 + \varepsilon$ , e.g., produces a solution only disturbed by  $\varepsilon \cdot e^{-2t}$ ; hence, condition is no excuse!
- solution:  $y(t) = e^{-1000t} + 1$
- discretization scheme: Euler's method (consistent and convergent of first order and numerically stable – thus, no problems can be expected)

Hence, let us program the Euler scheme:

$$y_{k+1} = y_k + \delta t(-1000y_k + 1000) = (1 - 1000\delta t)y_k + 1000\delta t$$

or (complete induction!)

$$y_{k+1} = (1 - 1000\delta t)^{k+1} + 1,$$

and we realize:

Although the exact solution  $y(t)$  converges towards its limit 1 extremely fast, the above sequence of  $y_k$  oscillates and diverges, if  $\delta t > 0.002$ . I.e., we are forced to use an extremely small step size, though the shape of the exact solution  $y(t)$  suggests that is – if necessary at all – should also be necessary for small values of  $t$  close to zero.

The answer is stiffness:

- The terms consistency, convergence, and stability are asymptotic ones:  $\delta t \rightarrow 0$  and  $O(\delta t)$  always mean “for sufficiently small  $\delta t$ ”.
- The above phenomenon is called *stiffness*: The exact solution contains a rather unimportant term (here the exponential term), which, nevertheless, enforces a very fine resolution on the whole domain.
- *Implicit methods* provide a possible remedy:
  - *explicit*: all learned so far; the rule is of an explicit form with  $y_{k+1}$  only on the left-hand side
  - *implicit*: the unknown  $y_{k+1}$  appears on the rule’s right-hand side, too (possibly, we need an iterative scheme for that)
- The simplest implicit scheme is the so-called *implicit Euler* or *backward Euler*:

$$y_{k+1} = y_k + \delta t f(t_{k+1}, y_{k+1}).$$

- In our above example, this leads to

$$y_{k+1} = \frac{y_k + 1000\delta t}{1 + 1000\delta t} = \frac{1}{(1 + 1000\delta t)^{k+1}} + 1.$$

As one can see,  $\delta t > 0$  can now be chosen without restrictions, convergence is always ensured!

- note: explicit methods approximate an IVP’s solution with polynomials, implicit methods do this with rational functions; polynomials can not approximate  $e^{-t}$  for  $t \rightarrow \infty$ , rational can do this.
- For stiff problems, we need implicit methods.

# Chapter 10

## Partial Differential Equations

### 10.1 Preliminary Remarks

In partial differential equations (PDEs), there are more than one independent variables. Typically, these are different spatial coordinates  $x, y, z$  (stationary or steady-state problems) or space and time (unsteady problems). Hence, the occurring derivatives are *partial* ones. The PDE describes the underlying physics, whereas a concrete problem (with a unique solution) is defined only with the help of additional boundary conditions or mixed boundary (with respect to space) and initial (with respect to time) conditions. Altogether, we speak of *boundary value problems* or of *initial-boundary value problems*. Compared with ODEs, an important difference (and origin of problems, see Chapter 11) is the fact that we, now, may have to deal with very complicated domains instead of simple time intervals.

A both simple and important class of PDEs is the so-called *general linear PDE of second order in  $d$  dimensions*:

$$\sum_{i,j=1}^d a_{ij}(\mathbf{x})u_{x_i x_j}(\mathbf{x}) + \sum_{i=1}^d a_i(\mathbf{x})u_{x_i}(\mathbf{x}) + a(\mathbf{x})u(\mathbf{x}) = f(\mathbf{x}). \quad (10.1)$$

Here,  $\mathbf{x} := (x_1, \dots, x_d)$  denotes the vector of variables  $x_j$  (space or time). There are three different subclasses:

- elliptic* in  $\mathbf{x}$ : the (w.l.o.g. symmetric) matrix  $A(\mathbf{x}) = (a_{ij}(\mathbf{x}))_{i,j}$  is positive definite or negative definite;
- hyperbolic* in  $\mathbf{x}$ :  $A$  has one positive and  $n - 1$  negative eigenvalues or vice versa;
- parabolic* in  $\mathbf{x}$ : one eigenvalue of  $A$  is zero,  $n - 1$  eigenvalues have the same sign, and  $\text{rank}(A(\mathbf{x}), c(\mathbf{x})) = n$  with  $c(\mathbf{x}) = (a_i(\mathbf{x}))_i$ .

If the above relations hold for all  $\mathbf{x} \in \Omega$ , the PDE is called elliptic or hyperbolic or parabolic, resp. Some of the simplest examples:

elliptic	potential or Laplace equation	$\Delta u = u_{xx} + u_{yy} = 0$
hyperbolic	wave equation	$u_{xx} - u_{yy} = 0$
parabolic	heat equation	$u_{xx} - u_t = 0$

All three types have completely different properties – with respect to both analysis and numerics. In the following, we restrict ourselves to the stationary elliptic case and have a short glance at the most important discretization schemes.

## 10.2 Finite Differences (FD)

Here, the domain  $\Omega$  is given a generally regular grid with (possibly different) mesh widths  $h_x$ ,  $h_y$ , and  $h_z$ . In the standard case, we have equidistant grids in each direction. However, adaptive refinement is possible, too (block-wise, for example). Note that one has to take care of the so-called *hanging nodes* (grid points without a neighbour on one or some sides). The single derivatives are approximated like in the previous chapter:

$$\begin{aligned} \frac{\partial u}{\partial x}(\xi) &\doteq \frac{u(\xi+h) - u(\xi)}{h}, \frac{u(\xi+h) - u(\xi-h)}{2h}, \frac{u(\xi) - u(\xi-h)}{h} \\ &\text{(forward difference, central difference, backward difference)} \\ \frac{\partial^2 u}{\partial x^2}(\xi) &\doteq \frac{u(\xi+h) - 2u(\xi) + u(\xi-h)}{h^2} \\ &\dots \end{aligned}$$

In each grid point – which corresponds to one degree of freedom or one value to be determined in the case of a scalar solution function – the corresponding difference equation is formulated, i. e. the discrete counterpart of the PDE. If we have Dirichlet boundary conditions (function values given at the boundary), we get exactly one discrete equation for each inner point (the given boundary values are put on the right-hand side of the respective discrete equation). For Neumann boundary conditions (normal derivative given at the boundary), these lead to slightly modified difference stars near and on the boundary. However, in both cases, we end up with a system of  $M$  linear equations in  $M$  unknowns.

A simple example: Suppose we have to solve the 2 D Laplace equation  $-\Delta u = 0$  in the interior of the unit square  $\Omega = ]0, 1[{}^2$ . We use the minus sign in order to obtain positive diagonal entries in the matrix afterwards. We use a quadratic and equidistant  $(N+1) \times (N+1)$  grid; hence,  $h_x = h_y = 1/N$ , and the number of unknowns or degrees of freedom is  $M := (N-1)^2$ . For the matrix  $A$ ,  $A \in \mathbb{R}^{M \times M}$  holds; for the right-hand side  $b$ , we have  $b \in \mathbb{R}^M$ , and for the solution vector  $x$  which contains the approximations to  $u$  in the respective grid points, we have  $x \in \mathbb{R}^M$  as well. If the above discrete approximation is used for the second derivatives,  $A$  is a *sparse* matrix: at most five entries per matrix row are non-zero, not depending on  $M$ . If we arrange the discrete unknowns in a suitable way,  $A$  gets some band structure. Of course, the properties of  $A$  have an important impact on the selection of the iterative solver for the system of linear equations.

For Dirichlet boundary conditions, all diagonal entries are 4, and there are two, three, or four positions per row with the entry  $-1$ . In rows corresponding to grid points next to the boundary, there are less than four  $-1$  entries, because the given boundary values are not treated as unknowns, but added to the right-hand side  $b$  of the system of linear equations, see Fig. 10.1.

For Neumann boundary conditions, the diagonal elements are 2, 3, or 4 – depending on the respective grid point's position (next to the boundary or not). Accordingly, the number of  $-1$  entries is two, three, or four. The modified diagonal entries are due to the fact that the difference of a boundary value and the value in the neighbouring grid point (corresponding to the diagonal element) is considered as the given normal derivative at the boundary, replaced by the prescribed value, and added to the equation's right-hand side, see Fig. 10.2.

The quality of the chosen discretization is reflected by its order of convergence, that is the order of the error of the solution  $u_h$  computed numerically compared with the exact solution  $u$  of the given problem. Typically, this error (measured in some given norm) is of the order  $\|u - u_h\| = O(h^k)$  ( $k = 2$  for the above approximation of the second derivatives and for the maximum norm), if  $h$  denotes the mesh width (now supposed to be uniform for all directions). There are at least two strategies to improve this result: by reducing the mesh width (expensive, since the mesh width  $h$

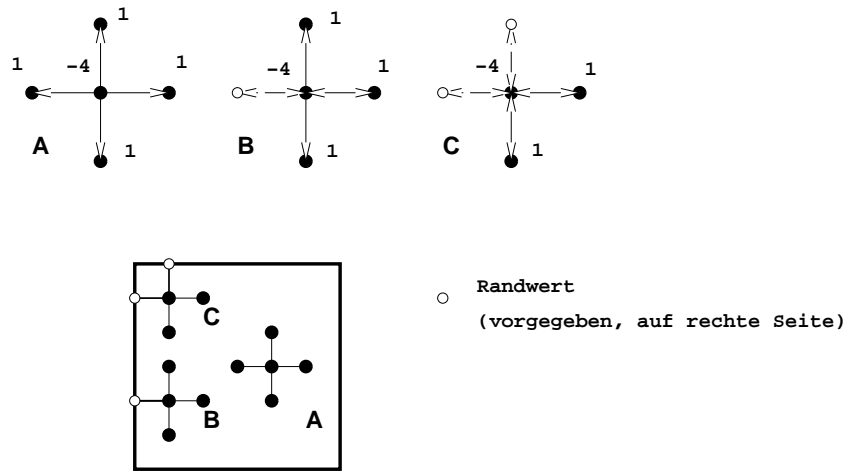


Figure 10.1: Different “difference stars” in 2 D (Dirichlet)

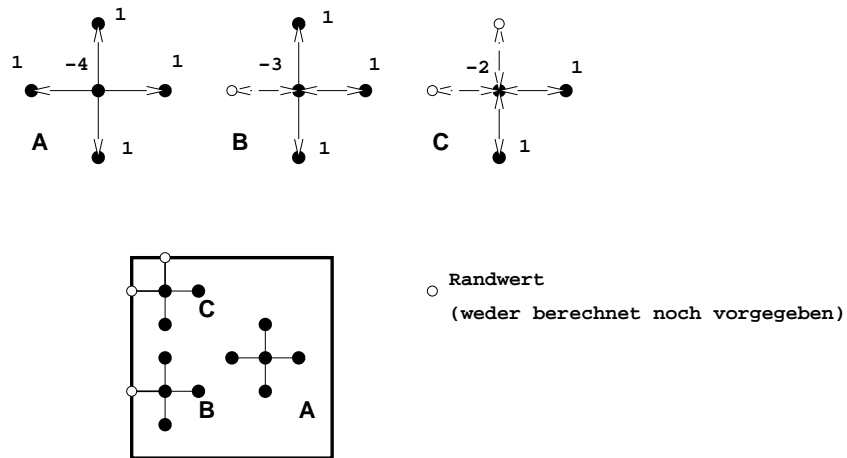


Figure 10.2: Different “difference stars” in 2 D (Neumann)

stands for  $O(h^{-d})$  grid points in the  $d$ -dimensional case: the *curse of dimension* strikes!) or by increasing the order of approximation  $k$  with the help of more sophisticated and more accurate discrete approximations of the derivatives. For the latter strategy, more neighbouring grid points have to be involved – which leads, of course, to more non-zero entries in the matrix  $A$ .

### 10.3 Finite Elements (FE)

The approach underlying the finite element method is a completely different one. Instead of approximating the PDE  $Lu = f$  with its differential operator  $L$  (the Laplacian  $\Delta$ , for example) and the right-hand side  $f$  via the direct discretization of all occurring derivatives, we now require that  $Lu = f$  must be fulfilled in some *weak* sense only, i. e. with respect to some scalar product with a finite number of test functions  $\psi_l$ :

$$Lu = f \quad \longrightarrow \quad \langle Lu, \psi_l \rangle = \langle f, \psi_l \rangle \quad 1 \leq l \leq N.$$

One can show that this corresponds to a *variational approach*: The solution  $u$  also solves a minimization problem of some suitable energy functional (again the detour via minimization!). The

test functions  $\psi_l$  are simple functions with compact support, they are non-zero only on small sub-domains of  $\Omega$ . This weakening corresponds to a partitioning of  $\Omega$  into *finite elements*. In contrast to finite differences, the spectrum concerning the position of the grid points or the shape of the elements (element types) is much bigger than with finite differences. At this point, the mathematician's point of view meets the engineer's one (the origins of the application-oriented finite element method are in civil engineering or computational mechanics), for whom this technique primarily helps to substructure his (generally complicated) object of desire in simpler and standardized elements. Then, local element solutions are generated, which are afterwards assembled or compiled to a global solution of the problem.

The single steps:

**1. substructuring or grid generation:**

Partitioning of the domain into single "elements"; the vertices where different elements meet (nodes) form a grid  $\Omega_n$ .

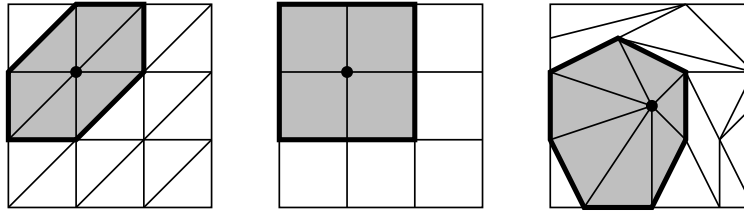


Figure 10.3: Regular triangular (left) and quadrilateral (centre) as well as irregular grid (right) with the support of potential basis or ansatz functions

In each grid point  $x_k$ , an *ansatz function*  $\varphi_k$  is defined, for example a hat function in 1D or a pagoda function in 2D (see Fig. 10.4).

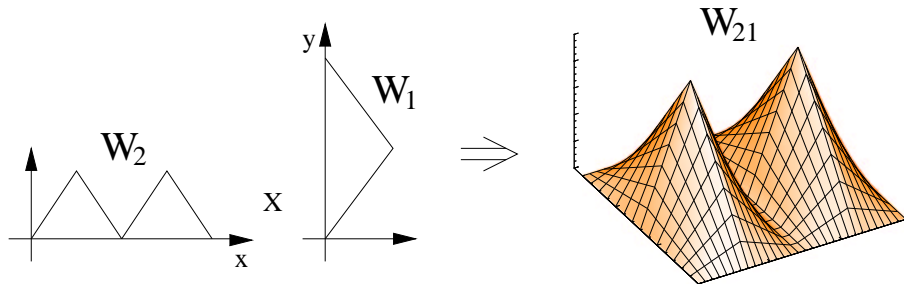


Figure 10.4: Hat function (1D) and pagoda function (2D) as typical basis functions on intervals or rectangles as elements

This  $\varphi_k$  vanishes on all but the neighbouring finite elements (finite support; in some cases, the support may extend over more elements – but it is still local). Together, all ansatz functions span a linear space  $V_n$ , the so-called *ansatz space*:

$$V_n := \text{span}\{\varphi_k : x_k \in \Omega_n\}.$$

Actually, the  $\varphi_k$  form a basis of  $V_n$ . Now, we look for an approximation to the solution  $u$  of our PDE or boundary value problem in this space  $V_n$ .

**2. Weak form of the PDE:**

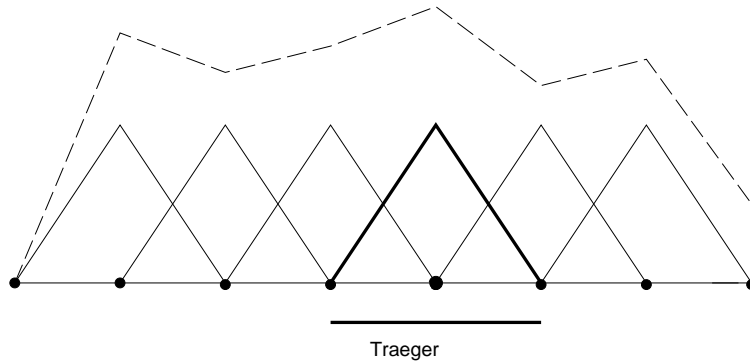


Figure 10.5: Piecewise linear nodal point basis

In a second step, we weaken the PDE  $Lu = f$  on  $\Omega$  to

$$\int_{\Omega} Lu \cdot \psi_l \, d\mathbf{x} = \int_{\Omega} f \cdot \psi_l \, d\mathbf{x}$$

for certain *test functions*  $\psi_l$  (*method of weighted residuals, Galerkin approach*). Thus, besides the ansatz space  $V_n$ , there is a second finite-dimensional vector space now, the *test space* spanned by the  $\psi_l$ . Test functions and ansatz functions can be chosen in an independent way. If they are the same, we call the approach *Ritz-Galerkin method*. Otherwise, we speak of *Petrov-Galerkin methods*. In the first case, the weak form means

$$\int_{\Omega} Lu \cdot \varphi \, d\mathbf{x} = \int_{\Omega} f \cdot \varphi \, d\mathbf{x} \quad \forall \varphi \in V_n .$$

Due to the linearity of  $V_n$ , it is sufficient to require this for all *basis functions*  $\varphi_k$  of  $V_n$  instead of for all functions  $\varphi \in V_n$ . Furthermore, in our context, we restrict ourselves to *linear* differential operators  $L$  like the Laplacian  $\Delta$ . With the *bilinear form* or scalar product

$$a(u, v) := \int_{\Omega} Lu \cdot v \, d\mathbf{x}$$

and the *linear form*

$$b(v) := \int_{\Omega} f \cdot v \, d\mathbf{x}$$

for  $u, v \in V_n$ , we finally get the following system of linear equations:

$$a(u, \varphi_k) = b(\varphi_k) \quad \forall \varphi_k \in V_n .$$

3. **Discrete Approximation:** Next, we have to take into account that we are looking for an approximation  $u_n$  to  $u$  in  $V_n$ , i. e.  $u_n$  is a linear combination of the basis functions  $\varphi_l$ :

$$u_n = \sum_l \alpha_l \cdot \varphi_l .$$

We insert this approach for  $u_n$  into the above system of linear equations and obtain

$$a(u, \varphi_k) = a\left(\sum_l \alpha_l \cdot \varphi_l, \varphi_k\right) = \sum_l \alpha_l \cdot a(\varphi_l, \varphi_k) = b(\varphi_k) \quad \forall \varphi_k \in V_n .$$

All  $a(\varphi_l, \varphi_k)$  and  $b(\varphi_k)$  depend on the given problem only but not on the current approximation. Hence, they have to be computed only once at the beginning of the solution process.

Consequently, we have a system of linear equations  $Ax = b$  in the unknowns  $\alpha_i$ :

$$\begin{aligned} A &= (a(\varphi_i, \varphi_j))_{i,j}, \\ b &= (b(\varphi_i))_i + \langle \text{influence of boundary conditions} \rangle, \\ x &= (\alpha_i)_i. \end{aligned}$$

The matrix  $A$  is called *stiffness matrix*.

4. **Solution of the system of linear equations:** The solution  $x$  of the system of linear equations  $Ax = b$  represents the finite element approximation in  $V_n$  to our continuous problem's solution. Hence, the stiffness matrix and its properties are crucial for the (iterative) solution process. The optimum would be a diagonal matrix (i. e., the  $\{\varphi_k\}$  are  $a$ -orthogonal), because then, solving  $Ax = b$  would be trivial. Some concluding remarks concerning  $A$ :

- Often,  $A$  is symmetric positive definite.
- If we use so-called *nodal point bases* (all basis functions have the same shape and size of support), then  $A$  is sparse and has a band structure. This is due to the fact that, for all basis functions  $\varphi_i$  and  $\varphi_j$  with non-overlapping supports, we have

$$a_{i,j} = a(\varphi_i, \varphi_j) = \int_{\Omega} L\varphi_i \cdot \varphi_j \, dx = 0.$$

Here, the term “finite elements” gets very obvious.

- If we use so-called *hierarchical bases* (different size of the supports depending on the respective point's position and hierarchical level, see Fig. 10.6), more supports do overlap. For example, the support of the basis function of the highest hierarchical level overlaps with all other supports. This results in a fill-in of the stiffness matrix, but the matrix keeps its sparse character (within each level, there is no overlap at all). On the average, we get  $O(n)$  non-zero entries per matrix row, if  $n$  denotes the finest resolution ( $h = 2^{-n}$ ) or the number of levels, respectively.

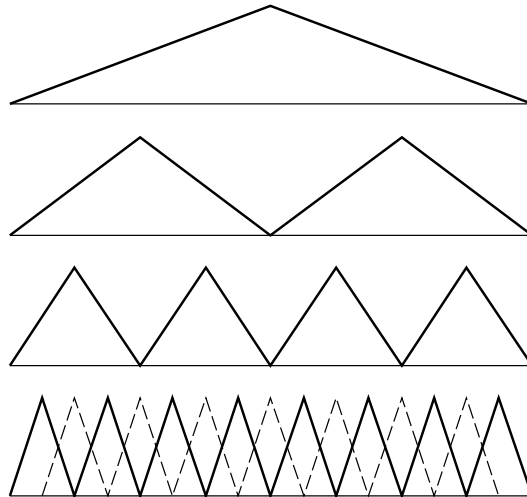


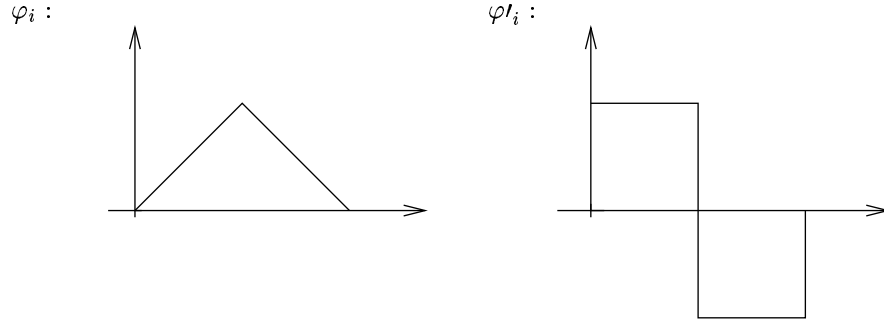
Figure 10.6: Nodal point basis (bottom) und hierarchical basis (all rows, but only the bold functions) in 1D

- It must be our objective to construct a basis which leads to a stiffness matrix that allows for a fast solution of the system of equations. Ideally,  $A$  is diagonal; a more realistic scenario is to require that  $A$ 's properties (spectrum etc.) should result in a fast convergence of the standard iterative methods discussed in Chapter 8, for example.

As an example, we consider the one-dimensional Poisson equation  $-\Delta u = f$  or  $-u'' = f$ , resp. The first formula of Green (remember your analysis course or just believe it!) provides

$$a_{i,j} = \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j \, dx = \int_{\Omega} \varphi_i' \cdot \varphi_j' \, dx.$$

If we use the piecewise linear hierarchical basis, we have the following situation:



Obviously, we have  $a_{i,j} \neq 0 \Leftrightarrow i = j$ . Hence, this is one of the really rare situations where a diagonal stiffness matrix can be easily obtained!

Concerning statements of convergence, the following property of *best approximation* (lemma of C ea) is of crucial importance:

Let  $u_n^{FE} \in V_n$  be the finite element approximation to the solution  $u$  of the boundary value problem with symmetric and positive definite  $a$  or  $A$ , respectively. Then, with respect to the norm  $\|u\|_a := \sqrt{a(u, u)}$ , the so-called *energy norm*, the following estimate holds:

$$\|u - u_n^{FE}\|_a \leq \inf_{u_n \in V_n} \|u - u_n\|_a.$$

This means that the finite element solution  $u_n^{FE}$  is the best possible approximation to  $u$  in  $V_n$ .

The interesting aspect of this lemma is that the interpolation error in  $V_n$  and its energy norm can be determined without bigger problems, usually. Since the interpolant  $u_n^I \in V_n$  of  $u$  in  $V_n$ , of course, is one candidate for the infimum process living in  $V_n$ , we immediately have an upper bound for the error of the finite element solution  $u_n^{FE}$ . As a result, convergence statements with respect to the (problem-dependent)  $\|\cdot\|_a$ -norm are, in general, simple in a finite element context, whereas corresponding statements with respect to the (not problem-dependent)  $L_\infty$ - or  $L_2$ -norm are often much more complicated to obtain.

Thus, the central task in the (mathematical) finite element world is:

Find finite-dimensional approximation spaces  $V_n$  with bases  $\{\varphi_k\}$  such that, firstly, the resulting approximate solutions  $u_n$  are high-quality approximations to the exact solution  $u$  (better and better for increasing  $n$ ), and that, secondly, the  $u_n$  are simple (i. e. cheap) to construct.

## 10.4 Finite Volumes

Finite volume methods are widespread in computational fluid dynamics, in particular. There, the main idea is to directly implement the classical conservation laws of continuum mechanics on small volume elements. Since flows are not a core topic in the COMMAS program, we won't go into detail, here.

## 10.5 Unsteady Problems

So far, we have only discussed stationary problems. The situation becomes more difficult, if we encounter time-dependence. The simplest way to take into account time at least to some extent is the *quasi stationary* approach: Successively, time is increased by one time step, and the new resulting stationary problem is solved. Obviously, such a proceeding does not neglect changes in time completely. However, on the other hand, it is clear that this can be sufficiently accurate for problems with small changes in time only – solving the stationary heat equation means using vanishing time derivatives, which is not correct in general, of course.

Hence, we will have to tackle the simultaneous discretization of both space and time. Here, stability problems (as we already know them from ODEs) are quite frequent: time step  $\delta t$  and spatial mesh width  $h$  are often linked in the sense of some law like “a small  $h$  requires a small  $\delta t$ ”. This is annoying, but plausible: it doesn’t make sense to take into account each tiny spatial detail, if we look at the scenario only every 30 minutes!

This short outlook must be enough here. You will get in touch with this topic in several other COMMAS courses!

# Chapter 11

## Grid Generation

The numerical treatment of partial differential equations requires the approximate description and discretization of the underlying computational domain. There exists a real zoo of strategies for generating and refining suitable *grids* or *meshes*. Of course, the selection of an appropriate strategy depends on the chosen discretization technique for the PDE (*point* discretizations, for example finite differences, or *cell* discretizations, for example finite volumes or finite elements) as well as on the complexity of the given geometry. Especially for real-life technical objects, it can take orders of magnitude more man-power to construct a grid than it does to compute and analyze the approximate solution on the grid. Therefore, grid generation is a very important issue for scientific computing. Hence, in this section, we want to give a short overview of some of the most relevant approaches for grid generation and mesh refinement.

During grid generation, a number of aspects have to be taken into account:

- **Accuracy:** The generated mesh must be (locally) dense enough to represent the essential physical phenomena.
- **Boundary approximation:** The generated mesh should allow a sufficiently accurate representation of the domain's boundaries and, thus, of the respective boundary conditions.
- **Computational efficiency:** The overhead for the organization of the grid structure (storage, CPU time) should not be too big, and the performance of the whole solution process on supercomputers (distributing the mesh!) should not suffer from the chosen mesh type.
- **Numerical adequacy:** Features with a negative impact on numerical efficiency (too small or big angles of the elements, too big distortions of elements or grids, etc.) should be avoided.

In general, we distinguish *structured* and *unstructured* grids. We speak of structured grids, if the points or elements are constructed following some regular process from which the geometric and topological information (coordinates, neighbour relationships) can be directly derived. In contrast to that, unstructured grids allow more flexibility, with grid points whose position might even result from a random process, but require the explicit storage of the grid's relevant geometric and topological information. However, in both situations, the generation of some starting grid is only part of the job, since, in many cases, the optimal position of grid points in the sense of a cost-benefit ratio can not be predicted *before* the solution process, but depends on the computed solution and, thus, has to take into account properties of it. Hence, dynamically adaptive meshes that are coupled with the physical solution are of crucial importance, too.

Finally, standard parallelization strategies for PDEs are based on some subdivision or decomposition of the underlying domain and, thus, of the respective grid. Therefore, the questions how easy an equal distribution of the grid points among the processors can be obtained, and how com-

munication between the different subsets of grid points can be reduced (small interfaces, graph partitioning strategies) are important, too, but will not be discussed in detail.

## 11.1 Structured Grids

### 11.1.1 Composite Grids

The notion of *composite grids* was the key to the treatment of general 3D geometries with the help of structured grids. The idea is just to subdivide the region of concern into a number of subdomains which are of a simpler form that allows the generation of a structured mesh. While the gain won't be that much for domains arising with porous media, e. g., it definitely is a breakthrough for many technical configurations, if one thinks of objects that have been created with the help of some CAD system and that are represented via the CSG (constructive solid geometry<sup>1</sup>) scheme, for example. There are several variants (see Fig. 11.1):

- **Block** or **patched** grids: Here, the different structured grids on the subregions are fitted together along interfaces (with or without continuity).
- **Overlaid** or **chimera** grids: Here, the grids are composed of completely independent subdomain grids that overlap an overall background grid or other component grids. A lot of attention has to be paid to the organization of data transfer between grids via appropriate interpolation techniques.

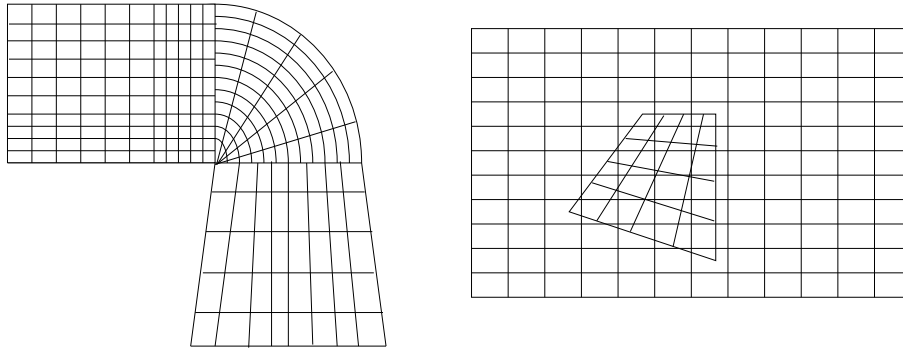


Figure 11.1: Composite grids: patched (left) and chimera (right)

### 11.1.2 Block-Structured Grids

Block-structured grids are very popular in computational fluid dynamics. Actually, their arrival opened the door to real-life CFD applications. In a block-structured approach, the domain is subdivided into several logically rectangular subdomains which fit together in an unstructured way. Afterwards, each of the subdomains gets a logically rectangular grid with possibly curvilinear coordinates. At the interfaces, continuity is ensured – the grid points coincide (see Fig. 11.2). The block-structured approach enables us to combine the advantages of both boundary-fitted grids and strictly structured grids.

<sup>1</sup> CSG is a standard CAD representation scheme for rigid bodies. Here, simple parametrized primitives like bricks or cylinders are used to form complicated objects with the help of the Boolean operations union, intersection, and set difference.

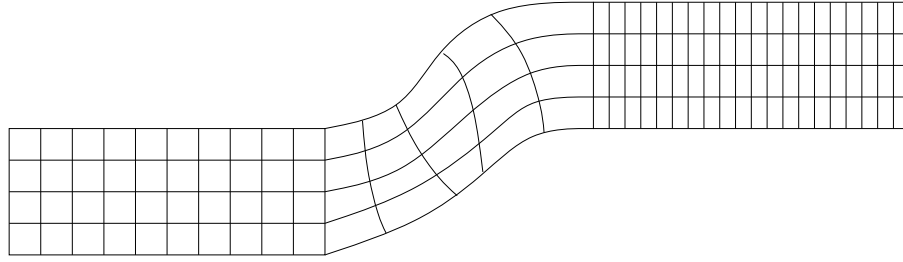


Figure 11.2: A block-structured grid

### 11.1.3 Elliptic Generators

A lot of grid generation techniques are based on the idea of a mapping or transformation  $\Psi : \Omega' \rightarrow \Omega$  from the unit square or unit cube  $\Omega'$  to the computational domain  $\Omega$  on which the given PDE is to be solved. One defines the grid on the simple geometry  $\Omega'$  and obtains a grid on the target geometry  $\Omega$  by applying the mapping to the grid lines and points. Or one applies a suitable mapping to the PDE, too, and solves the transformed PDE on  $\Omega'$  with its simple grid. A famous representative of this class are *elliptic generators*, where the coordinates on the respective domain are obtained as solutions of systems of elliptic PDEs. This ensures that we get very smooth grid lines in the interior – even if the boundaries are not smooth (see the left part of Fig. 11.3). It is mainly this advantage that makes elliptic grid generators that popular, in spite of the relatively high cost. However, grid control, i. e. the explicit control of the grid points' and lines' position, is often quite difficult.

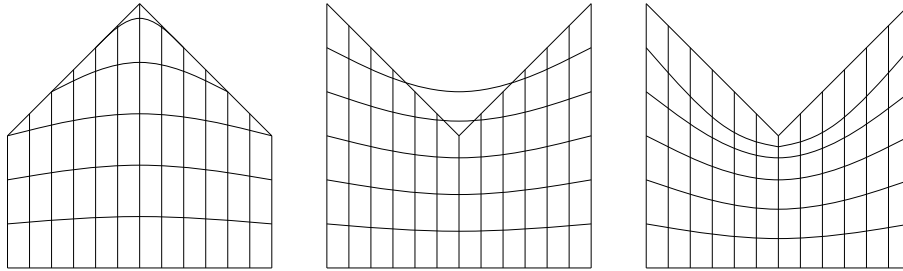


Figure 11.3: Grids by elliptic methods (convex (left) and nonconvex case (centre and right))

We regard a simple 2D example. A grid shall be generated on a (logically rectangular) domain  $\Omega$  bounded by four curves  $c_{0,y}$ ,  $c_{1,y}$ ,  $c_{x,0}$ , and  $c_{x,1}$  (see Fig. 11.4). Then, one possibility would be to define a system of non-coupled Laplacians for the components  $\xi(x, y)$  and  $\eta(x, y)$  of the mapping  $\Psi(x, y)$ ,

$$\begin{aligned} \Delta \xi(x, y) &= 0 & \text{on } \Omega' = ]0, 1[^2, \\ \Delta \eta(x, y) &= 0 & \text{on } \Omega', \end{aligned} \tag{11.1}$$

with Dirichlet boundary conditions (two sets for the two PDE!)

$$\begin{aligned} \begin{pmatrix} \xi(x, 0) \\ \eta(x, 0) \end{pmatrix} &= c_{x,0}(x), & \begin{pmatrix} \xi(x, 1) \\ \eta(x, 1) \end{pmatrix} &= c_{x,1}(x), \\ \begin{pmatrix} \xi(0, y) \\ \eta(0, y) \end{pmatrix} &= c_{0,y}(y), & \begin{pmatrix} \xi(1, y) \\ \eta(1, y) \end{pmatrix} &= c_{1,y}(y). \end{aligned} \tag{11.2}$$

A drawback of this simple approach is the fact that, in the nonconvex case, the resulting grid lines may leave the domain (see the centre part of Fig. 11.3). Here, *inverse elliptic methods*, where the

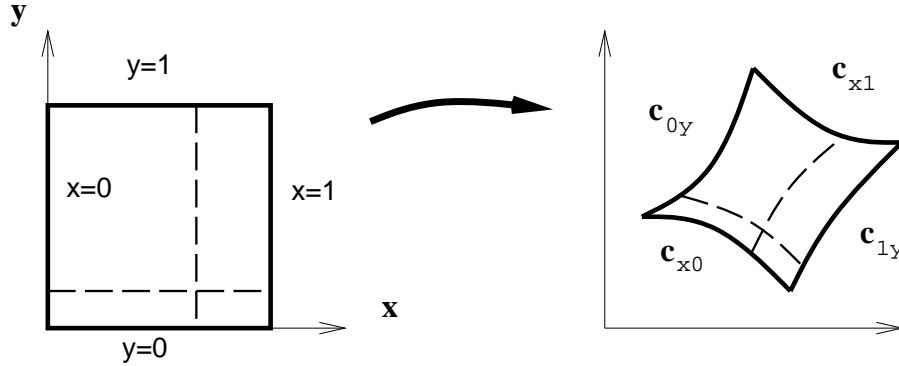


Figure 11.4: Mapping and elliptic grid generation

Laplacians are defined on the curvilinear domain  $\Omega$  and then transformed to the unit square, can serve as a remedy (see the right part of Fig. 11.3). The price for this is a more complicated and coupled system of PDEs to be solved.

#### 11.1.4 Hyperbolic Generators

Instead of solving an elliptic system, a computational grid can be constructed by solving a hyperbolic system, too. Due to the character of hyperbolic PDEs, however, only one boundary with respect to each coordinate can be specified. Hence, hyperbolic generators are appropriate for calculations on physically unbounded regions. Here, the advantage of the hyperbolic approach is a gain in speed compared with elliptic generators.

#### 11.1.5 Algebraic Generators

*Algebraic* or *interpolation-based* generators do not require to solve a whole system of PDEs in order to get a grid for another PDE's solution, but they are based on simple interpolation techniques. The most famous one is the so-called *transfinite interpolation* or, as known in computer-aided geometric design (CAGD), the *Coons patch*. Again, we restrict ourselves to the 2D case and define a grid for the domain  $\Omega$  given in the right part of Fig. 11.4 whose boundaries are specified by the four curves  $c_{0,y}, \dots, c_{x,1}$ . We can summarize all four curves in one single definition  $c(x, y)$ , defined for  $x \in \{0, 1\}$  or  $y \in \{0, 1\}$ , and introduce three interpolaton operators  $F_1$ ,  $F_2$ , and  $F_1 F_2$  that map the function  $c(x, y)$  defined on the boundary of  $\Omega'$ , only, to a function given on the whole  $\Omega'$ :

$$\begin{aligned}
 F_1(x, y) &:= (1 - x) \cdot c(0, y) + x \cdot c(1, y), \\
 F_2(x, y) &:= (1 - y) \cdot c(x, 0) + y \cdot c(x, 1), \\
 F_1 F_2(x, y) &:= (1 - x)(1 - y) \cdot c(0, 0) + x(1 - y) \cdot c(1, 0) \\
 &\quad + (1 - x)y \cdot c(0, 1) + xy \cdot c(1, 1).
 \end{aligned} \tag{11.3}$$

Then, the transfinite interpolation operator  $TF$  is defined as

$$TF(x, y) := (F_1 + F_2 - F_1 F_2)(x, y). \tag{11.4}$$

As it can be easily shown,  $TF$  performs an interpolation of all four boundary curves into the interior of  $\Omega$  (the four corners and the boundary curves are reproduced).

The 3D extension is straightforward. The transfinite interpolation provides a very cheap and easy way to control grid generation (explicit lines which shall become coordinate lines can be taken

into account as additional inner or virtual boundaries). However, boundaries that are not smooth are inherited into the interior of the domain, which is a serious drawback. Furthermore, once again, the resulting grid lines may leave the domain. In Fig. 11.5, two examples shall illustrate the principle of transfinite interpolation.

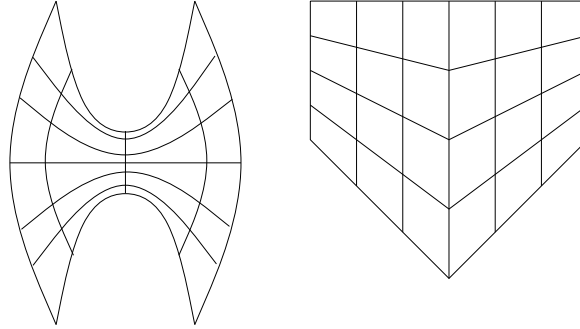


Figure 11.5: Transfinite interpolation

### 11.1.6 Adaptive Grids

Many numerical simulations require the dynamical and adaptive refinement of meshes during the computation, if one reaches the following point: Investing more everywhere is no longer feasible for economic reasons, stopping already now means that you haven't obtained a sufficiently accurate solution. Thus, more has to be invested locally at some points or in some subregions, whether others are already treated sufficiently well. There are several strategies for going in that direction. First, one might decide to *refine* the grid in the critical regions (the classical mesh refinement or *h-adaptation*). Second, one might increase the order of approximation (by using higher order finite elements or higher order stencils, which does the *p-adaptation*<sup>2</sup>). Finally, if no more work can be invested, the idea might be to *rearrange* the grid points in order to get an error that is balanced better. In the context of grid generation, we are primarily interested in the first case. For structured grids, the most common way to refine a given mesh is block adaptation: Identify the critical regions (with the help of some *error indicator* estimating the local error in that point or region, and start a simple block refinement (double mesh width in the respective block, for example). Special attention has to be paid to *hanging nodes*, since discontinuities have to be avoided. See Fig. 11.6 for a simple example.

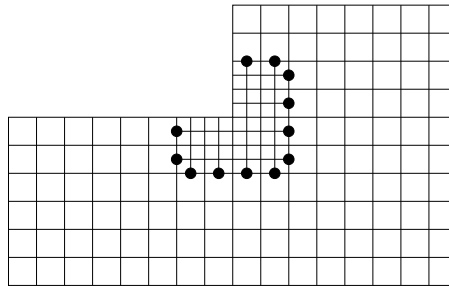


Figure 11.6: Block adaptation and hanging nodes: If there is a non-zero in a point marked with a black circle, then the point has this value from the fine grid's point of view, but it is just the difference of its two neighbours' values from the adjacent coarse cell's view.

<sup>2</sup>The parameter  $p$  stands for the order or polynomial degree of the used approach.

## 11.2 Unstructured Grids

Unstructured grids are closely related to finite element methods, and one usually thinks of (nearly) arbitrary triangulations or tetrahedral grids.

### 11.2.1 Delaunay Triangulations, Voronoi Diagrams

Suppose you have to find a grid for some domain  $\Omega$ . Moreover, suppose you have already determined discrete points for your grid. How can you derive elements in the finite element sense? One solution is very old and goes back to Dirichlet and Voronoi. For a given set of points  $P_i, i = 1, \dots, N$ , the *Voronoi regions*  $V_i$  can be defined as

$$V_i := \{P : \|P - P_i\| < \|P - P_j\| \forall j \neq i\}. \quad (11.5)$$

This means that  $V_i$  contains all points that are closer to  $P_i$  than to any other of our grid points  $P_j$ . Overall, the result is the so-called *Voronoi diagram*, a subdivision of the 2D or 3D domain into polygons or polyhedra, respectively. If one draws a line between all pairs of points that are neighbouring in the sense that they are located in adjacent Voronoi regions, the result is a set of disjoint triangles (a *Delaunay triangulation*, see Fig. 11.7 for a simple example) or tetrahedra covering the convex hull of the  $P_i$ . Delaunay triangulations are used very often, since they have some properties both favourable for efficient algorithms for their construction and for the numerical solution process afterwards.

In the following, for reasons of simplicity, we will speak of triangles and triangulations for general dimensionality  $d$ .

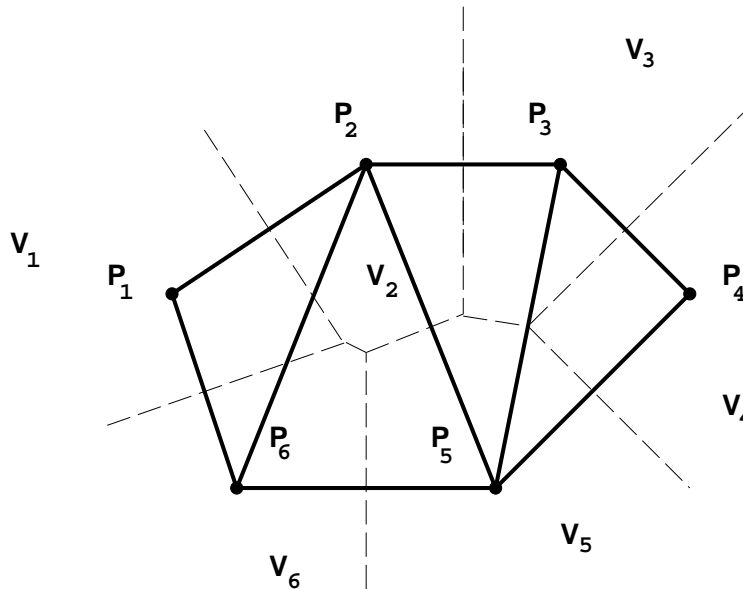


Figure 11.7: Voronoi diagram (dashed lines) and Delaunay triangulation (solid lines) in 2D

### 11.2.2 Point Creation

Now, we know what to do when we have grid points – but how to get these? There are several possible ways:

- **Independent generation:** Use the grid points of a structured grid (which has nothing to do with the later triangulation), for example a polar grid in the case of a ring domain, as starting point.
- **Superposition and successive subdivision:** Superimpose a regular grid over the domain with the help of a recursive subdivision approach (quad- or octrees, for example).
- **Successive subdivision of a boundary-based triangulation:** Start with a given boundary point distribution, generate the Delaunay triangulation, and continue with subdividing the triangles by introducing new points at the triangles' centroids according to certain rules (reduce big triangles etc.), until resolution is high enough.
- **Point and line sources:** Apart from points on the boundary, point or line sources in the interior of the domain can be introduced. They shall help to ensure that the successive subdivision of the initial Delaunay triangulation gets especially refined near the sources. This is helpful, for example, if one knows the position of singularities or boundary layers.

### 11.2.3 Other Techniques

Of course, there are more strategies to generate unstructured grids than the Delaunay approach:

- **Advancing front methods:** Here, the idea is to start from the boundary (the starting *front* – a line in 2D, a surface in 3D) and advance step by step into the interior of the domain. In 2D, with points  $P_i$ ,  $i = 1, \dots, n$ , and edges  $P_i P_j$ , the basic algorithmic scheme looks like

choose an edge on the current front, say  $P_1 P_2$ ;  
 create a new point  $P$  at equal distance  $d$  from  $P_1$  and  $P_2$ ;  
 determine all points  $Q \in \{P, P_1, \dots, P_n\}$  lying within a circle of radius  $r$  around  $P$ ;  
 order these points  $Q$  in distance from  $P$ ;  
 for all those points  $Q$ , form triangles  $(P_1 P_2 Q)$  and accept the first suitable one;  
 if the accepted  $Q$  is a new point, add it to the list of points;  
 add new edges to the front and remove the old edge  $P_1 P_2$ ;

(11.6)

A new triangle is suitable if it does not produce edge intersections and if it satisfies certain quality conditions (concerning angles etc.). This loop is repeated until the front is empty (see Fig. 11.8).

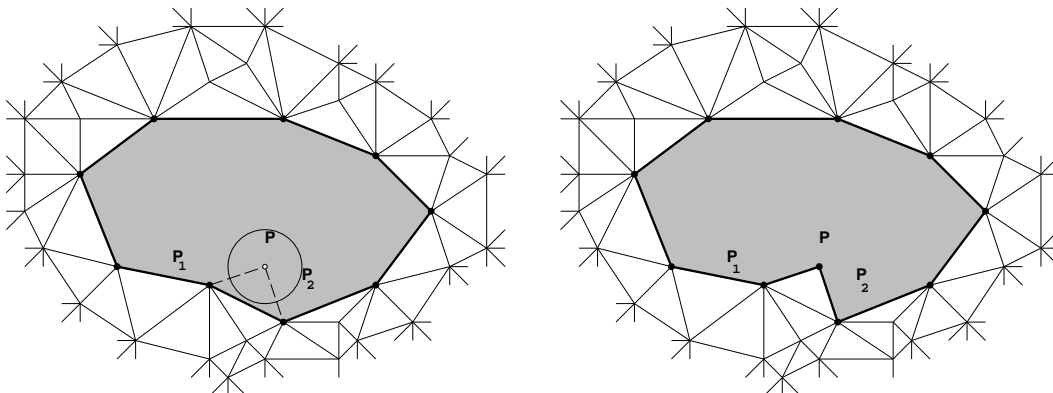


Figure 11.8: Advancing front algorithm: one step

Note that the resolution is controlled via the parameter  $d$ , and that an alignment to specific directions is possible, if additional directional parameters are introduced.

- **Spacetrees:** Spacetrees (quadtrees in 2 D, octrees in 3 D) are based upon a recursive regular subdivision of a  $d$ -dimensional cube containing the domain. Until some given tolerance is reached, a square or a cube is subdivided into four squares or eight cubes, respectively, if the domain's boundary intersects it. Otherwise, it can be regarded as 'completely inside' or 'completely outside' the domain. In order to get a boundary conforming grid, the pure spacetree is modified by appropriate cutting and reconnecting to triangles and tetrahedra.
- **Hybrid grids:** In order to achieve an optimum compromise between regularity and flexibility, it is possible to combine the two grid types in the form of *hybrid* or *structured-unstructured* grids.

### 11.2.4 Adaptive Grids

Adaptive grids are somewhat natural in an unstructured context, since we have dynamic data structures, anyway. Thus, mesh adaptation is much more popular for unstructured grids than it is in the structured case. Nevertheless, it is important to note that it is not at all restricted to unstructured grids.

Once we have produced an unstructured mesh of our domain, the need for adaptive refinement usually occurs during the computational part, i. e. during the solution of the PDE(s). Hence, the solution process should provide information whether and, if at all, where to refine. Overall, we need

- a *global error estimator* telling us at the end whether our computed result is sufficiently accurate or whether we should start another computation with a finer mesh;
- a *refinement criterion* that determines the aim of refinement (balancing the error over all grid points, keeping it below a certain bound everywhere, etc.);
- a *local error estimator* or *indicator* telling us during the computation whether it would be helpful to refine the mesh locally at that point;
- a *refinement procedure* that determines the process how the grid shall be (technically) refined. Figure 11.9 shows several possible strategies for that. Note that, as in the structured case, the problem of hanging nodes may occur.

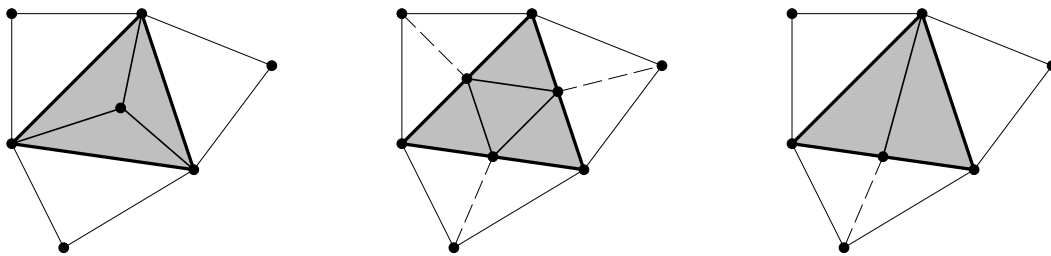


Figure 11.9: Some ways of refinement (with treatment of hanging nodes): adding the centroid (left), *red* subdivision (centre), and *green* one (right)

## 11.3 Methods for Varying Geometries

Recently, the focus of research interest has turned to varying geometries or moving grids instead of stationary configurations. The spectrum of applications is huge, and we want to mention only

some interesting fields:

- free surfaces (injection moulding, melting or freezing processes, e. g.);
- multiphase flows (steam bubbles transporting heat in water in coolers, e. g.);
- fluid-structure interactions (mechanical valves opened or closed by a flowing liquid, tent-roof constructions subject to wind).

Of course, a permanent remeshing after each modification of the configuration is far beyond present and future computational possibilities. Thus, other strategies than the iteration of the techniques presented above have to be applied.

### 11.3.1 Front Tracking Methods

The so-called *front tracking methods* describe the boundary or *interface* between the two phases (two liquid phases or liquid and solid phase, etc.) directly, i.e. they update a geometric representation (free-from curves or surfaces) due to the current movements. Obviously, this allows to take into account any modification in a quite accurate way. However, if we have to deal with complicated 3D geometry changes, or if even changes of topology occur (two bubbles unite), we run into methodical and computational problems. Therefore, in these cases, we can not afford the direct representation of the interface.

### 11.3.2 Front Capturing Methods

At that point, *front capturing methods* enter the game. They indicate the position of the interface in an indirect way, with the help of some global quantity. Thus, we are less precise, but front capturing methods allow a much more straightforward handling of varying geometries. Two representatives of this class are *marker-and-cell (MAC)* methods and the *volume-of-fluid (VOF)* method. Both were developed for CFD applications, and we, hence, base the following short discussion on flow problems. The MAC approach is based on structured rectangular grids. In each cell, we have a number of (virtual) particles moving with the flow field. Cells without particles do not belong to the liquid phase, cells with particles adjacent to empty cells form the surface or interface, and the other cells are located in the interior of the liquid's domain. The VOF method uses a global variable, the relative volume of the liquid phase in each cell, in order to describe the position of the interface. According to the flow field, the values of this VOF variable are updated after each time step.

### 11.3.3 Clicking and Sliding Mesh Techniques

If the movements are regular or periodic (oscillations, slow rotations of a stirrer), one might think of keeping most of the fluid grid fixed and moving only the respective structures and the adjacent flow cells. This part of the grid *slides*, and a *click* is necessary, when neighbour relations jump from one grid point to the next.

One final remark shall close the discussion of grid generation. Global grid generation is a problem that primarily occurs when we look at things from the so-called *Eulerian point of view*. Here, we assume a fixed reference system or grid, into which moving structures have to be integrated in a somewhat artificial way. Another way of looking at things provides the *Lagrangian point of view*, where no global reference system is imposed, but all 'players' have their local reference systems. This approach, which seems to be less natural at first glance, is sometimes advantageous for problems with varying geometry. The so-called *Arbitrary Lagrangian-Eulerian (ALE)* method combines both viewpoints for an efficient realization of moving structures. For example, moving

bubbles have their own local description when they move in the (regular and fixed) mesh of the fluid domain. At the end, we just want to do some name dropping and mention some methods with reduced importance of a certain grid and, thus, a big potential with respect to movements: *particle methods*, *Lattice Boltzmann Automata (LBA)*, and *gridless discretizations*.

## Chapter 12

# Parallel Computing