Chapter 0

Introduction

0.1 Course Fundamentals

0.1.1 Focus of this Course

Emphasis is put

▷ on **algorithms** (principles, computational cost, scope, and limitations),

▷ on (efficient and stable) **implementation** in **C++** based on the numerical linear algebra template library **EIGEN**, a **Domain Specific Language** (DSL) embedded into **C++**.

▷ on **numerical experiments** (design and interpretation).

§0.1.1.1 (**Aspects outside the scope of this course**) No emphasis will be put on

- theory and proofs (unless essential for derivation and understanding of algorithms).

  ☞ 401-3651-00L Numerical Methods for Elliptic and Parabolic Partial Differential Equations
  401-3652-00L Numerical Methods for Hyperbolic Partial Differential Equations
  (both courses offered in BSc Mathematics)

- hardware aware implementation (cache hierarchies, CPU pipelining, vectorization, etc.)

  ☞ 263-0007-00L Advanced System Lab (How To Write Fast Numerical Code, Prof. M. Püschel, D-INFK)

- issues of high-performance computing (HPC, shard and distributed memory parallelisation, vectorization)

  ☞ 151-0107-20L High Performance Computing for Science and Engineering (HPCSE, Prof. P. Koumoutsakos, D-MAVT)
  263-2800-00L Design of Parallel and High-Performance Computing (Prof. T. Höfler, D-INFK)

However, note that these other courses partly rely on knowledge of elementary numerical methods, which is covered in this course.
Contents

§0.1.1.2 (Prequisites)  This course will take for granted basic knowledge of linear algebra, calculus, and programming, that you should have acquired during your first year at ETH.

Numerical Methods

- Linear systems of equations
- Eigenvalue problems
- Least squares problems
- Interpolation and approximation
- Quadrature
- Filtering & FFT

Analysis | Linear algebra | Programming (in C++)

§0.1.1.3 (Numerical methods: A motley toolbox)

This course discusses elementary numerical methods and techniques

They are vastly different in terms of ideas, design, analysis, and scope of application. They are the items in a toolbox, some only loosely related by the common purpose of being building blocks for codes for numerical simulation.

Do not expect much coherence between the chapters of this course!

A purpose-oriented notion of “Numerical methods for CSE”:

A: “Stop putting a hammer, a level, and duct tape in one box! They have nothing to do with each other!”

B: “I might need any of these tools when fixing something about the house”

§0.1.1.4 (Dependencies of topics)  Despite the diverse nature of the individual topics covered in this course, some depend on others for providing essential building blocks. The following directed graph tries to capture these relationships. The arrows have to be read as “uses results or algorithms of”.

0. Introduction, 0.1. Course Fundamentals
Any one-semester course “Numerical methods for CSE” will cover only selected chapters and sections of this document. Only topics addressed in class or in homework problems will be relevant for exams!

§0.1.5 (Relevance of this course) I am a student of computer science. After the exam, may I safely forget everything I have learned in this mandatory “numerical methods” course? No, because it is highly likely that other courses or projects will rely on the contents of this course:

- Singular value decomposition
- Least squares
- Function approximation
- Numerical quadrature
- Numerical integration
- Interpolation
- Least squares
- Eigensolvers
- Sparse linear systems
- Numerical integration

\[ \Rightarrow \text{Computational statistics, machine learning} \]
\[ \Rightarrow \text{machine learning, Numerical methods for PDEs} \]
\[ \Rightarrow \text{Computer graphics} \]
\[ \Rightarrow \text{Graph theoretic algorithms} \]
\[ \Rightarrow \text{Computer animation} \]
Hardly anyone will need everything covered in this course, but most of you will need something.

0.1.2 Goals

This course is meant to impart

✦ knowledge of some fundamental algorithms forming the basis of numerical simulations,
✦ familiarity with essential terms in numerical mathematics and the techniques used for the analysis of numerical algorithms
✦ the skill to choose the appropriate numerical methods for concrete problems,
✦ the ability to interpret numerical results,
✦ proficiency in implementing numerical algorithms efficiently in C++, using numerical libraries.

Indispensable: Learning by doing (➔ exercises)

0.1.3 Literature

Parts of the following textbooks may be used as supplementary reading for this course. References to relevant sections will be provided in the course material.

Studying extra literature is not important for following this course!


Comprehensive introduction to numerical methods with an algorithmic focus based on MATLAB.
(Target audience: students of engineering subjects)


Good reference for large parts of this course; provides a lot of simple examples and lucid explanations, but also rigorous mathematical treatment.
(Target audience: undergraduate students in science and engineering)
Available for download as PDF


Gives detailed description and mathematical analysis of algorithms and relies on MATLAB. Profound treatment of theory way beyond the scope of this course. (Target audience: undergraduates in mathematics)

Classical introductory numerical analysis text with many examples and detailed discussion of algorithms. (Target audience: undergraduates in mathematics and engineering) Can be obtained from website.


Modern discussion of numerical methods with profound treatment of theoretical aspects (Target audience: undergraduate students in mathematics).


Comprehensive treatment of elementary numerical methods with an algorithmic focus.

D-INFK maintains a webpage with links to some of these books.

Essential prerequisite for this course is a solid knowledge in linear algebra and calculus. Familiarity with the topics covered in the first semester courses is taken for granted, see


0.2 Teaching Style and Model

0.2.1 Flipped Classroom

This course will depart from the usual academic teaching arrangement centering around classes taught by a lecturer addressing an audience in a lecture hall.

<table>
<thead>
<tr>
<th>A flipped-classroom course</th>
</tr>
</thead>
<tbody>
<tr>
<td>This course will follow the flipped-classroom paradigm:</td>
</tr>
<tr>
<td>Learning by self-study guided by</td>
</tr>
<tr>
<td>instruction videos</td>
</tr>
<tr>
<td>tablet notes</td>
</tr>
<tr>
<td>lecture notes</td>
</tr>
<tr>
<td>interactive Q&amp;A sessions</td>
</tr>
<tr>
<td>homeworks</td>
</tr>
<tr>
<td>tutorial classes</td>
</tr>
</tbody>
</table>

All the course material will be published online through the course Moodle Page. All notes jotted down by the lecturer during the creation of videos or during the Q&A sessions will be made available as PDF.

0.2.1.1 Course Videos

In the flipped-classroom teaching model regular lectures will be replaced with pre-recorded videos. These videos are not commercial-grade clips, but resemble video recordings from a standard classroom setting; they convey the development of the material on a tablet accompanied by the lecturer’s voice.
The Videos will be published through
1. the course Moodle Page (see beside),
2. and as .mp4-files on PolyBox
   (password required).

Every video comes with a PDF containing the tablet notes taken during the creation of the video. However, the PDF may have been corrected, updated, or supplemented later.

§0.2.1.2 ("Pause" and "fast forward") Videos have two big advantages:

You can stop a video at any time, whenever
- you need more time to think,
- you want to look up related information,
- you want to work for yourself.

Make use of this possibility!

The video portal also allows you to play the videos at 1.5× speed. This can be useful, if the current topic is very clear to you. You can also skip entire parts using the scroll bar. The same functionality (fast playing and skipping) is offered by most video players, for instance the VLC media player.

§0.2.1.3 (Review questions) Most lecture units (corresponding to a video) are accompanied with a list of review questions. You should try to answer them off the top of your head without consulting any written material shortly after you have finished studying the unit.

In case you are utterly clueless about how to approach a review question, you probably need to refresh some of the unit’s topics.

§0.2.1.4 (List of available tutorial videos) This is the list of available video tutorials as of October 13, 2022:

1.  Video tutorial for Chapter 0 “Introduction”: (16 minutes) Download link, tablet notes

2.  Video tutorial for Section 1.1.1 “Notations and Classes of Matrices”: (7 minutes) Download link, tablet notes
   → review questions 1.1.2.9

3.  Video tutorial for Section 1.2.1 "EIGEN": (11 minutes) Download link, tablet notes
4. Video tutorial for Section 1.2.3 "(Dense) Matrix Storage Formats": (10 minutes) Download link, tablet notes
→ review questions 1.2.3.11

5. Video tutorial for Section 1.4 "Computational Effort": (29 minutes) Download link, tablet notes
→ review questions 1.4.3.11

6. Video tutorial for Section 1.5 "Machine Arithmetic and Consequences": (16 minutes) Download link, tablet notes
→ review questions 1.5.3.18

7. Video tutorial for Section 1.5.4 "Cancellation": (22 minutes) Download link, tablet notes
→ review questions 1.5.4.33

8. Video tutorial for Section 1.5.5 "Numerical Stability": (17 minutes) Download link, tablet notes
→ review questions 1.5.5.23

9. Video tutorial for Section 2.1 & Section 2.2.1 "Introduction and Theory: Linear Systems of Equations (LSEs)": (6 minutes) Download link, tablet notes
→ review questions 2.2.1.7

10. Video tutorial for Ex. 2.1.0.3 "Nodal Analysis of Linear Electric Circuits": (8 minutes) Download link, tablet notes
→ review questions 2.1.0.8

11. Video tutorial for Section 2.2.2 "Sensitivity of Linear Systems": (15 minutes) Download link, tablet notes
→ review questions 2.2.2.12

12. Video tutorial for Section 2.3 & Section 2.5 "Gaussian Elimination": (17 minutes) Download link, tablet notes
→ review questions 2.3.2.21

13. Video tutorial for Section 2.6 "Exploiting Structure when Solving Linear Systems": (17 minutes) Download link, tablet notes
→ review questions 2.6.0.25

14. Video tutorial for Section 2.7.1 "Sparse Matrix Storage Formats": (10 minutes) Download link, tablet notes
→ review questions 2.7.1.5
15. Video tutorial for Section 2.7.2 "Sparse Matrices in E\textsc{igen}": (6 minutes) Download link, tablet notes
   → review questions 2.7.2.17

16. Video tutorial for Section 2.7.3 "Direct Solution of Sparse Linear Systems of Equations": (10 minutes) Download link, tablet notes
   → review questions 2.7.3.7

17. Video tutorial for Section 3.0.1 "Overdetermined Linear Systems of Equations: Examples": (12 minutes) Download link, tablet notes
   → review questions 3.0.1.11

18. Video tutorial for Section 3.1.1 "Least Squares Solutions": (9 minutes) Download link, tablet notes
   → review questions 3.1.1.14

19. Video tutorial for Section 3.1.2 "Normal Equations": (16 minutes) Download link, tablet notes
   → review questions 3.1.2.23

20. Video tutorial for Section 3.1.3 "Moore-Penrose Pseudoinverse": (8 minutes) Download link, tablet notes
   → review questions 3.1.3.8

21. Video tutorial for Section 3.2 "Normal Equation Methods": (12 minutes) Download link, tablet notes
   → review questions 3.2.0.11

22. Video tutorial for Section 3.3 "Orthogonal Transformation Methods": (10 minutes) Download link, tablet notes
   → review questions 3.3.2.3

23. Video tutorial for Section 3.3.3.1 "QR-Decomposition: Theory": (11 minutes) Download link, tablet notes
   → review questions 3.3.3.8

24. Video tutorial for Section 3.3.3.2 & Section 3.3.3.4 "Computation of QR-Decomposition, QR-Decomposition in E\textsc{igen}": (32 minutes) Download link, tablet notes
   → review questions 3.3.3.28

25. Video tutorial for Section 3.3.4 "QR-Based Solver for Linear Least Squares Problems": (9 minutes) Download link, tablet notes
   → review questions 3.3.4.8

26. Video tutorial for Section 3.3.5 "Modification Techniques for QR-Decomposition": (25 minutes) Download link, tablet notes
27. Video tutorial for Section 3.4.1 "Singular Value Decomposition: Definition and Theory": (13 minutes) Download link, tablet notes

→ review questions 3.4.1.15

28. Video tutorial for Section 3.4.2 "SVD in EIGEN": (9 minutes) Download link, tablet notes

→ review questions 3.4.2.10

29. Video tutorial for Section 3.4.3 "Solving General Least-Squares Problems by SVD": (14 minutes) Download link, tablet notes

→ review questions 3.4.3.17

30. Video tutorial for Section 3.4.4.1 "Norm-Constrained Extrema of Quadratic Forms": (11 minutes) Download link, tablet notes

→ review questions 3.4.4.13

31. Video tutorial for Section 3.4.4.2 "Best Low-Rank Approximation": (13 minutes) Download link, tablet notes

→ review questions 3.4.4.25

32. Video tutorial for Section 3.4.4.3 "Principal Component Data Analysis (PCA)"": (28 minutes) Download link, tablet notes

→ review questions 3.4.4.51

33. Video tutorial for Section 3.6 "Constrained Least Squares": (23 minutes) Download link, tablet notes

→ review questions 3.6.2.1

34. Video tutorial for Section 4.1.1 "Discrete Finite Linear Time-Invariant Causal Channels/Filters": (11 minutes) Download link, tablet notes

→ review questions 4.1.1.13

35. Video tutorial for Section 4.1.2 "LT-FIR Linear Mappings": (12 minutes) Download link, tablet notes

→ review questions 4.1.2.10

36. Video tutorial for Section 4.1.3 "Discrete Convolutions": (9 minutes) Download link, tablet notes

→ review questions 4.1.3.12

37. Video tutorial for Section 4.1.4 "Periodic Convolutions": (12 minutes) Download link, tablet notes

→ review questions 4.1.4.19
38. Video tutorial for Section 4.2.1 "Diagonalizing Circulant Matrices": (17 minutes) Download link, tablet notes
→ review questions 4.2.1.23

39. Video tutorial for Section 4.2.2 "Discrete Convolution via DFT": (7 minutes) Download link, tablet notes
→ review questions 4.2.2.6

40. Video tutorial for Section 4.2.3 "Frequency filtering via DFT": (20 minutes) Download link, tablet notes
→ review questions 4.2.3.11

41. Video tutorial for Section 4.2.5 "Two-Dimensional DFT": (20 minutes) Download link, tablet notes
→ review questions 4.2.5.23

42. Video tutorial for Section 4.3 "Fast Fourier Transform (FFT)": (16 minutes) Download link, tablet notes
→ review questions 4.3.0.13

43. Video tutorial for Section 4.5 "Toeplitz Matrix Techniques": (20 minutes) Download link, tablet notes
→ review questions 4.5.3.8

44. Video tutorial for Section 5.1 "Abstract Interpolation": (16 minutes) Download link, tablet notes
→ review questions 5.1.0.27

45. Video tutorial for Section 5.2.1 "Uni-Variate Polynomials": (7 minutes) Download link, tablet notes
→ review questions 5.2.1.8

46. Video tutorial for Section 5.2.2 "Polynomial Interpolation: Theory": (6 minutes) Download link, tablet notes
→ review questions 5.2.2.19

47. Video tutorial for Section 5.2.3 "Polynomial Interpolation: Algorithms": (18 minutes) Download link, tablet notes
→ review questions 5.2.3.14

48. Video tutorial for Section 5.2.3.3 "Extrapolation to Zero": (12 minutes) Download link, tablet notes
→ review questions 5.2.3.20

49. Video tutorial for Section 5.2.3.4 "Newton Basis and Divided Differences": (17 minutes) Download link, tablet notes
50. Video tutorial for Section 5.2.4 "Polynomial Interpolation: Sensitivity": (13 minutes) Download link, tablet notes

51. Video tutorial for Section 5.3 "Shape-Preserving Interpolation": (23 minutes) Download link, tablet notes

52. Video tutorial for Section 5.4.1 "Spline Function Spaces": (9 minutes) Download link, tablet notes

53. Video tutorial for Section 5.4.2 "Cubic Spline Interpolation": (14 minutes) Download link, tablet notes

54. Video tutorial for Section 5.4.3 "Structural Properties of Cubic Spline Interpolants": (12 minutes) Download link, tablet notes

55. Video tutorial for Section 5.6 "Trigonometric Interpolation": (14 minutes) Download link, tablet notes

56. Video tutorial for Section 5.7 "Least Squares Data Fitting": (13 minutes) Download link, tablet notes

57. Video tutorial for Section 6.1 "Approximation of Functions in 1D: Introduction": (7 minutes) Download link, tablet notes

58. Video tutorial for Section 6.2 "Polynomial Approximation: Theory": (13 minutes) Download link, tablet notes

59. Video tutorial for Section 6.2.2 "Error Estimates for Polynomial Interpolation": (12 minutes) Download link, tablet notes

60. Video tutorial for Section 6.2.2.2 "Error Estimates for Polynomial Interpolation: Interpolands of Finite Smoothness": (17 minutes) Download link, tablet notes
61. Video tutorial for Section 6.2.2.3 "Error Estimates for Polynomial Interpolation: Analytic Interpolands": (27 minutes) Download link, tablet notes
   → review questions 6.2.2.69

62. Video tutorial for Section 6.2.3.1 "Chebychev Interpolation: Motivation and Definition": (21 minutes) Download link, tablet notes
   → review questions 6.2.3.13

63. Video tutorial for Section 6.2.3.2 "Chebychev Interpolation Error Estimates": (14 minutes) Download link, tablet notes
   → review questions 6.2.3.30

64. Video tutorial for Section 6.2.3.3 "Chebychev Interpolation: Computational Aspects": (11 minutes) Download link, tablet notes
   → review questions 6.2.3.44

65. Video tutorial for Section 6.5.1 "Approximation by Trigonometric Interpolation": (5 minutes) Download link, tablet notes
   → review questions 6.5.1.6

66. Video tutorial for Section 6.5.2 "Trigonometric Interpolation Error Estimates": (14 minutes) Download link, tablet notes
   → review questions 6.5.2.26

67. Video tutorial for Section 6.5.3 "Trigonometric Interpolation of Analytic Periodic Functions": (16 minutes) Download link, tablet notes
   → review questions 6.5.3.18

68. Video tutorial for Section 6.6.1 "Piecewise Polynomial Lagrange Interpolation": (17 minutes) Download link, tablet notes

69. Video tutorial for Section 6.6.2 "Cubic Hermite and Spline Interpolation: Error Estimates": (10 minutes) Download link, tablet notes

70. Video tutorial for Section 7.1 "Numerical Quadrature: Introduction": (4 minutes) Download link, tablet notes
   → review questions 7.1.0.5

71. Video tutorial for Section 7.2 "Quadrature Formulas/Rules": (13 minutes) Download link, tablet notes
   → review questions 7.2.0.15

72. Video tutorial for Section 7.3 "Polynomial Quadrature Formulas": (9 minutes) Download link, tablet notes
   → review questions 7.3.0.12
73. Video tutorial for Section 7.4.1 "Order of a Quadrature Rule": (9 minutes) Download link, tablet notes
→ review questions 7.4.1.12

74. Video tutorial for Section 7.4.2 "Maximal-Order Quadrature Rules": (16 minutes) Download link, tablet notes
→ review questions 7.4.2.27

75. Video tutorial for Section 7.4.3 "(Gauss-Legendre) Quadrature Error Estimates": (18 minutes) Download link, tablet notes
→ review questions 7.4.3.16

76. Video tutorial for Section 7.5 "Composite Quadrature": (18 minutes) Download link, tablet notes
→ review questions 7.5.0.26

77. Video tutorial for Section 7.6 "Adaptive Quadrature": (13 minutes) Download link, tablet notes
→ review questions 7.6.0.20

78. Video tutorial for Section 8.1 "Iterative Methods for Non-Linear Systems of Equations: Introduction": (6 minutes) Download link, tablet notes
→ review questions 8.1.0.6

79. Video tutorial for Section 8.2.1 "Iterative Methods: Fundamental Concepts": (6 minutes) Download link, tablet notes
→ review questions 8.2.1.11

80. Video tutorial for Section 8.2.2 "Iterative Methods: Speed of Convergence": (15 minutes) Download link, tablet notes
→ review questions 8.2.2.16

81. Video tutorial for Section 8.2.3 "Iterative Methods: Termination Criteria/Stopping Rules": (14 minutes) Download link, tablet notes
→ review questions 8.2.3.10

82. Video tutorial for Section 8.3 "Fixed-Point Iterations": (12 minutes) Download link, tablet notes
→ review questions 8.3.2.21

83. Video tutorial for Section 8.4.1 "Finding Zeros of Scalar Functions: Bisection": (7 minutes) Download link, tablet notes
→ review questions 8.4.1.4

84. Video tutorial for Section 8.4.2.1 "Newton Method in the Scalar Case": (20 minutes) Download link, tablet notes
→ review questions 8.4.2.16

85. Video tutorial for Section 8.4.2.3 "Multi-Point Methods": (12 minutes) Download link, tablet notes

→ review questions 8.4.2.41

86. Video tutorial for Section 8.4.3 "Asymptotic Efficiency of Iterative Methods for Zero Finding": (10 minutes) Download link, tablet notes

→ review questions 8.4.3.15

87. Video tutorial for Section 8.5.1 "The Newton Iteration in $\mathbb{R}^n$ (I)": (10 minutes) Download link, tablet notes

→ review questions 8.5.1.46

88. Video tutorial for § 8.5.1.15 "Multi-dimensional Differentiation": (20 minutes) Download link, tablet notes

89. Video tutorial for Section 8.5.1 "The Newton Iteration in $\mathbb{R}^n$ (II)": (15 minutes) Download link, tablet notes

90. Video tutorial for Section 8.5.2 "Convergence of Newton's Method": (9 minutes) Download link, tablet notes

→ review questions 8.5.2.8

91. Video tutorial for Section 8.5.3 "Termination of Newton Iteration": (7 minutes) Download link, tablet notes

→ review questions 8.5.3.9

92. Video tutorial for Section 8.5.4 "Damped Newton Method": (11 minutes) Download link, tablet notes

→ review questions 8.5.4.8

93. Video tutorial for Section 8.6 "Quasi-Newton Method": (15 minutes) Download link, tablet notes

→ review questions 8.6.0.22

94. Video tutorial for Section 8.7 "Non-linear Least Squares": (7 minutes) Download link, tablet notes

→ review questions 8.7.0.10

95. Video tutorial for Section 8.7.1 "Non-linear Least Squares: (Damped) Newton Method": (13 minutes) Download link, tablet notes

→ review questions 8.7.1.9

96. Video tutorial for Section 8.7.2 "(Trust-region) Gauss-Newton Method": (13 minutes) Download link, tablet notes

→ review questions 8.7.3.3
Necessary corrections and updates of the lecture document will sometimes lead to changes in the numbering of paragraphs and formulas, which, of course, cannot be applied to the recorded videos. However, these changes will be taken into account into the tablet notes supplied for every video.

0.2.1.2 Following the Course

<table>
<thead>
<tr>
<th>Weekly study assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>• For every week there is a list of course units and associated videos published on the course Moodle Page.</td>
</tr>
<tr>
<td>• The corresponding contents must be studied in that same week.</td>
</tr>
</tbody>
</table>

§0.2.1.6 (How to organize your learning)

☛ Develop a routine: Plan fixed slots, with a total duration of four hours, for studying for the course material in your weekly calendar. This does not include homework.

☛ Choose a stable setting, in which you can really concentrate (quiet area, headphones, coffee, etc.)

☛ Take breaks, when concentration is declining, usually after 20 to 45 minutes, but avoid online distractions during breaks.

You must not procrastinate!

Do not put off studying for this course. Dependencies between the topics will make it very hard to catch up.

§0.2.1.7 (“Personalized learning”) The flipped classroom model allows students to pursue their preferred ways of studying. The following approaches can be tried.

• Traditional: You watch the assigned videos similar to attending a conventional classroom lecture. Afterwards digest the material based on the tablet notes and/or the lecture document. Finally, answer the review questions and look up more information in the lecture document.

• Reading-centered: You work through the unit reading the tablet notes, and, sometimes, related sections of the lecture document. You occasionally watch parts of the videos, in case some considerations and arguments have not become clear to you already.

Collaborative studying is encouraged:

• You may watch course videos together with classmates.
• You may meet to discuss course units.
• You may solve homework problems in a group assigning different parts to different members.

☛ Explaining to others is a great way to deepen understanding.

☛ It is easy to sustain motivation and avoid distraction in a peer study group.
0.2.2 Clarifications and Frank Words

§0.2.2.1 (“Lecture notes”)

The PDF you are reading is referred to as lecture document and is an important source of information, but this course document is neither a textbook nor comprehensive lecture notes. They are meant to supplement and be supplemented by explanations given in the videos.

Some pieces of advice:

✦ The lecture document is only partly designed to be self-contained and can/should be studied in parts in addition to attending to watching the course videos and/or reading the tablet notes.
✦ This text is not meant for mere reading, but for working with,
✦ Turn pages all the time and follow the numerous cross-references,
✦ study the relevant section of the course material when doing homework problems,
✦ You may study referenced literature to refresh prerequisite knowledge and for alternative presentation of the material (from a different angle, maybe), but be careful about not getting confused or distracted by information overload.

§0.2.2.2 (Comprehension is a process . . .)

✦ This course will require

  hard work  —  perseverance  —  patience

✦ Do not expect to understand everything at once. Most students will
  • understand about one third of the material when watching videos and studying the course material
  • understand another third when making a serious effort to solve the homework problems,
  • hopefully understand the remaining third when studying for the main examination after the end of the course.

  Perseverance will be rewarded!

§0.2.2.3 (Expected workload)

(I) You are a student in the BSc/MSc programme of Computational Science and Engineering (CSE) or others. Then you are taking the full version of the course (401-2673-*) , which is endowed with 9 ECTS credits, which roughly corresponds to a total workload of 270 hours:

\[
270 \text{ hours} = 180 \text{ hours} + 90 \text{ hours},
\]

which amounts to a massive

\[
\text{average workload} \approx 10 - 14 \text{ hours per week}.
\]
If you are a student in the BSc Computer Science you are offered a trimmed version of the course, which is worth 7 ECTS credits. Though a very loose relationship, this roughly indicates a total workload of 180 hours:

\[
180 \text{ hours} = 110 \text{ hours during term} + 70 \text{ hours exam preparation}.
\]

This indicates that you should brace for an average workload \( \approx 7 - 9 \) hours per week.

For both versions of the course your efforts have to be split between

- watching videos and/or studying the course material,
- and solving homework problems,
- attending Q&A sessions and tutorials.

where homework may keep you busy for 5 – 7 hours every week for the full version.

Of course, all these are averages and the workload may vary between different weeks.

### 0.2.3 Requests

The lecturers very much welcome and, putting it even more strongly, rather depend on feedback and suggestions of the students taking the course for continuous improvement of the course contents and presentation. Therefore all participants are strongly encouraged to get involved actively and contribute in the following ways:

#### §0.2.3.1 (Reporting errors)

As the documents for this course will always be in a state of flux, they will inevitably and invariably teem with small errors, mainly typos and omissions.

For error reporting we use the DISCUNA online collaboration platform that runs in the browser. DISCUNA allows to attach various types of annotations to shared PDF documents, see instruction video.

Please report errors in the lecture material through the DISCUNA NumCSE Community to which various course-related documents have already been uploaded.

In the beginning of the teaching period you receive a join link of the form https://app.discuna.com/<JOIN CODE>. Open the link in a web browser and it will take you to the DISCUNA community page.

To report an error,

1. select the corresponding PDF document (chapter of the lecture document of homework problem) in the left sidebar,
2. press the prominent white-on-blue +-button in the right sidebar,
3. click on the displayed PDF where the error is located,
4. then in the pop-up window choose the “Error” category,
5. and add a title and,
6. if the title does not tell everything, a short description.

In case you cannot or do not want to link an error to a particular point in the PDF, you may just click on the title page of the respective chapter. Then, please precisely specify the concerned section and the number of the paragraph, remark, equation etc. Do not give page numbers as they may change with updates to the documents.

Note that chapter PDFs and homework problem files will gradually be added to the DISCUNA NumCSE community. Hence, the final chapters will not be accessible in the beginning of the course.

§0.2.3.2 (Pointing out technical problems) The DISCUNA NumCSE Community is equipped with a chat channel “Technical Problems”. In case you encounter a problem affecting the videos, the course web pages, or the PDF documents supplied online, say, severely distorted or missing audio tracks or a faulty link, instantly post a comment to this channel with a short description of the problem. You can do this after clicking on the channel name in the left sidebar in the community

§0.2.3.3 (Providing comments and suggestions) The chat channel “General Comments” of the DISCUNA NumCSE Community is meant for letting the lecturer know about weaknesses of the contents, structure, and presentation of the course and how they can be remedied. Your statements should be constructive and address specific parts or aspects of the course.

Regularly, students attending the course remark that they have found online resources like instruction videos that they think present some of the course material in a much clearer and better structured way. It is important that you tell the lecturer about those online resources so that he can include pointers to them and get inspiration. Use the “General Comments” channel also for this purpose. State clearly, which part of the course you are referring to, and briefly explain why the online resource is superior or a valuable supplement.

0.2.4 Assignments

A steady and persistent effort spent on homework problems is essential for success in this course.

You should expect to spend 3-5 hours per week on trying to solve the homework problems. Since many involve small coding projects, the time it will take an individual student to arrive at a solution is hard to predict.

For the sake of efficiency:

Avoid coding errors (bugs) in your homework coding projects!

The problems are published online together with plenty of hints. A master solution will also be made available, but it is foolish to read the master solution parallel to working on a problem sheet, because trying to find the solution on one’s own is essential for developing problem solving skills, though it may occasionally be frustrating.

§0.2.4.1 (Homeworks and tutors’ corrections)

✦ The weekly assignments will be a few problems from the NCSE Problem Collection available online as PDF, see course Moodle page for the link. The particular problems to be solved will be communicated through that Moodle page every week.

Please note that this problem collection is being extended throughout the semester. Thus, make sure that you obtain the most current version every week. A polybox link will also be distributed;
if you install the Polybox Client the most current version of all course documents will always be uploaded to your machine.

✦ Some or all of the problems of an assignment sheet will be discussed in the tutorial classes at least one week after the problems have been assigned.

✦ Your tutors are happy to examine your solutions and give you feedback: You may either hand them your solution papers during the tutorial session (put your name on every sheet and clearly mark the problems you want to be inspected) or upload a scan/photo through the CODEXPERT upload interface, see § 0.2.4.2 below. You are encouraged to hand in incomplete and wrong solutions, so that you can receive valuable feedback even on incomplete or failed attempts.

✦ Your tutors will automatically have access to all your homework codes, see § 0.2.4.2 below.

§0.2.4.2 (CODEXPERT C++ online IDE and testing environment)

CODEXPERT has been developed at ETH as an online IDE for small programming assignment and coding homeworks. It will be used in this course for all C++ homework problems.

Please study the documentation!

CODEXPERT also offers the possibility of uploading any files to a private area (connected with a homework problem) that, beside you, only the tutor in charge of your exercise group can access. This is the preferred option for sharing (scans/photos of) your solutions of homework problem with your tutor.

Note that CODEXPERT will also be using for the coding problems of the main examination.

0.2.5 Information on Examinations

§0.2.5.1 (Examinations during the teaching period) From the ETH course directory:

An optional 30-minutes mid-term exam and an optional 30-minutes end-term exam will be held during the teaching period. The grades of these interim examinations will be taken into account through a BONUS of up to 30% for the final grade.

The term exams will be conducted as closed book examinations on paper. The dates of the exams will be communicated in the beginning of the term and published on the course webpage. The term exams can neither be repeated nor be taken remotely.

The final grade is computed according to the formula

\[ G := 0.25 \cdot \left\lceil 4 \cdot \max\{G_s, 0.85G_s + 0.15g_m, 0.85G_s + 0.15g_e, 0.7G_s + 0.15g_m + 0.15g_e\} \right\rceil, \]  

where \( G_s \) denotes the smallest integer \( \geq x \).

§0.2.5.3 (Main examination during exam session)

✦ Three-hour written examination involving coding problems to be done at the computer. The date of the exam will be set and communicated by the ETH exam office, and will also be published on the course webpage.
The coding part of the exam has to be done using **CODEXPERT**.

Subjects of examination:

- All topics that have been addressed in a video listed on the course Moodle page or in any assigned homework problem

The lecture document contains much more material than covered in class. All these extra topics are not relevant for the exam.

Lecture document (as PDF), the **EIGEN** documentation, and the online **C++ REFERENCE PAGES** will be available PDF during the examination. The corresponding final version of the lecture document will be made available at least two weeks before the exam.

No other materials may be used during the exam.

The homework problem collection cannot be accessed during the exam.

In case you come to the conclusion that you have too little time to prepare for the main exam a few weeks before the exam, contemplate withdrawing in order not to squander an attempt.

§0.2.5.4 (Repeating the main exam)

- Bonus points earned in term exams in last year's course can be taken into account for this course's main exam.

- If you want to take this option, please declare this intention by email to the course organizers before the mid-term exam. Otherwise, your bonus will be based on the results of this year's term exams.

0.3 Programming in C++

**C++20** is the *current* ANSI/ISO standard for the programming language C++. On the one hand, it offers a wealth of features and possibilities. On the other hand, this can be confusing and even be prone to inconsistencies. A major cause of inconsistent design is the requirement with backward compatibility with the C programming language and the earlier standard C++ 98.

However, C++ has become the main language in computational science and engineering and high performance computing. Therefore this course relies on C++ to discuss the implementation of numerical methods.

In fact C++ is a blend of different programming paradigms:

- an object oriented core providing classes, inheritance, and runtime polymorphism,

- a powerful template mechanism for parametric types and partial specialization, enabling *template meta-programming* and compile-time polymorphism,

- a collection of abstract data containers and basic algorithms provided by the **Standard Template Library** (STL).
Supplementary literature. A popular book for learning C++ that has been upgraded to include the C++11 standard is [LLM12].

The book [Jos12] gives a comprehensive presentation of the new features of C++11 compared to earlier versions of C++.


The following sections highlight a few particular aspects of C++ that may be important for code development in this course.

The version of the course for BSc students of Computer Science includes a two-week introduction to C++ in the beginning of the course.

0.3.1 Function Arguments and Overloading

§0.3.1.1 (Function overloading, [LLM12, Sect. 6.4]) Argument types are an integral part of a function declaration in C++. Hence the following functions are different

- int* f(int); // use this in the case of a single numeric argument
- double f(int*); // use only, if pointer to a integer is given
- void f(const MyClass&); // use when called for a MyClass object

and the compiler selects the function to be used depending on the type of the arguments following rather sophisticated rules, refer to overload resolution rules. Complications arise, because implicit type conversions have to be taken into account. In case of ambiguity a compile-time error will be triggered. Functions cannot be distinguished by return type!

For member functions (methods) of classes an additional distinction can be introduced by the const specifier:

```cpp
struct MyClass {
  double f(double); // use for a mutable object of type MyClass
  double f(double) const; // use this version for a constant object
  ...
};
```

The second version of the method f is invoked for constant objects of type MyClass.

§0.3.1.2 (Operator overloading [LLM12, Chapter 14]) In C++ unary and binary operators like =, ==, +, -, *, /, +=, -=, *=, /=, %, &&, ||, «, », etc. are regarded as functions with a fixed number of arguments (one or two). For built-in numeric and logic types they are defined already. They can be extended to any other type, for instance

```cpp
MyClass operator + (const MyClass& , const MyClass&);
MyClass operator + (const MyClass& , double);
MyClass operator + (const MyClass& ); // unary + !
```
The same selection rules as for function overloading apply. Of course, operators can also be introduced as class member functions.

C++ gives complete freedom to overload operators. However, the semantics of the new operators should be close to the customary use of the operator.

§0.3.1.3 (Passing arguments by value and by reference [LLM12, Sect. 6.2]) Consider a generic function declared as follows:

```cpp
void f(MyClass x); // Argument x passed by value.
```

When f is invoked, a temporary copy of the argument is created through the copy constructor or the move constructor of MyClass. The new temporary object is a local variable inside the function body.

When a function is declared as follows

```cpp
void f(MyClass &x); // Argument x passed by reference.
```

then the argument is passed to the scope of the function and can be changed inside the function. No copies are created. If one wants to avoid the creation of temporary objects, which may be costly, but also wants to indicate that the argument will not be modified inside f, then the declaration should read

```cpp
void f(const MyClass &x); // Argument x passed by constant reference.
```

New in C++11 is move semantics, enabled in the following definition

```cpp
void f(const MyClass &&x); // Optional shallow copy
```

In this case, if the scope of the object passed as the argument is merely the function or std::move () tags it as disposable, the move constructor of MyClass is invoked, which will usually do a shallow copy only. Refer to Code 0.3.5.10 for an example.

0.3.2 Templates

§0.3.2.1 (Function templates) The template mechanism supports parameterization of definitions of classes and functions by type. An example of a function templates is

```cpp
template<typename ScalarType, typename VectorType>
VectorType saxpy(ScalarType alpha, const VectorType &x, const VectorType &y)
{
    return (alpha*x+y);
}
```

Depending on the concrete type of the arguments the compiler will instantiate particular versions of this function, for instance saxpy<float, double>, when alpha is of type float and both x and y are of type double. In this case the return type will be double.

For the above example the compiler will be able to deduce the types ScalarType and VectorType from the arguments. The programmer can also specify the types directly through the < >-syntax as in

```cpp
saxpy<double, double>(a,x,y);
```

If an instantiation for all arguments of type double is desired. In case, the arguments do not supply enough information about the type parameters, specifying (some of) them through < > is mandatory.

§0.3.2.2 (Class templates) A class template defines a class depending on one or more type parameters, for instance
template <typename T>
class MyClsTempl {
public:
    using parm_t = T; // T-dependent type
    MyClsTempl(); // Default constructor
    MyClsTempl(const T&); // Constructor with an argument
    template <typename U>
    T memfn(const T&, const U&) const; // Templated member function
private:
    T *ptr; // Data member, T-pointer
};

Types MyClsTempl<T> for a concrete choice of T are instantiated when a corresponding object is declared, for instance via

double x = 3.14;
MyClass myobj; // Default construction of an object
MyClsTempl<double> tinstd; // Instantiation for T = double
MyClsTempl<MyClass> mytinst(myobj); // Instantiation for T = MyClass
MyClass ret = mytinst.memfn(myobj,x); // Instantiation of member
    function for U = double, automatic type deduction

The types spawned by a template for different parameter types have nothing to do with each other.

0.3.3 Function Objects and Lambda Functions

A function object is an object of a type that provides an overloaded “function call” operator (). Function objects can be implemented in two different ways:

(I) through special classes like the following that realizes a function \( \mathbb{R} \to \mathbb{R} \)

    class MyFun {
    public:
        ...
        double operator (double x) const; // Evaluation operator
        ...
    };

    The evaluation operator can take more than one argument and need not be declared const.

(II) through lambda functions, an “anonymous function” defined as

\[
[<\text{capture list}>] (<\text{arguments}>) \to <\text{return type}> \{ \text{body}; \}
\]

where <capture list> is a list of variables from the local scope to be passed to the lambda function; an & indicates passing by reference,

<arguments> is a comma separated list of function arguments complete with types,

<return type> is an optional return type; often the compiler will be able to deduce the return type from the definition of the function.
Function classes should be used, when the function is needed in different places, whereas lambda functions for short functions intended for single use.

C++ code 0.3.3.1: Demonstration of use of lambda function ➔ GITLAB

```cpp
int main() {
    // initialize a vector from an initializer list
    std::vector<double> v({1.2, 2.3, 3.4, 4.5, 6.6, 7.8});
    // A vector of the same length
    std::vector<double> w(v.size());
    // Do cumulative summation of v and store result in w
    double sum = 0;
    std::transform(v.begin(), v.end(), w.begin(),
                   [&sum](double x) { sum += x; return sum; });
    cout << "sum = " << sum << " , w = [ ";
    for(auto x : w) cout << x << ' ';
    cout << ' ] ' << endl;
    return(0);
}
```

In this code the lambda function captures the local variable `sum` by reference, which enables the lambda function to change its value in the surrounding scope.

§0.3.3.2 (Function type wrappers) The special class `std::function` provides types for general polymorphic function wrappers.

```cpp
C++ code 0.3.3.3: Use of std::function ➔ GITLAB

double binop(double arg1, double arg2) { return (arg1/arg2); }

void stdfunctiontest(void) {
    // Vector of objects of a particular signature
    std::vector<std::function<double(double, double)>> fnvec;
    // Store reference to a regular function
    fnvec.push_back(binop);
    // Store a lambda function
    fnvec.push_back([]() (double x, double y) -> double { return y/x; });
    for (auto fn : fnvec) { std::cout << fn(3,2) << std::endl; }
}
```

In this example an object of type `std::function<double (double, double)>` can hold a regular function taking two `double` arguments and returning another `double` or a lambda function with the same signature. Guess the output of `stdfunctiontest`!

§0.3.3.4 (Recorder objects) In the case of routines that perform some numerical computations we are often interested in the final result only. Occasionally we may also want to screen intermediate results. The following example demonstrates the use of an optional object for collecting information while the function is being executed. If no such object is supplied, an idle lambda function is passed, which incurs absolutely no runtime overhead.

C++ code 0.3.3.5: An example of a function taking a recorder object. ➔ GITLAB

```cpp
template <typename RECORDER = std::function<void(int, int)>>
unsigned int myloopfunction(
    unsigned int n, unsigned int val = 1,
    RECORDER &rec = [](int, int) -> void {})
```
for (unsigned int i = 0; i < n; ++i) {
    rec(i, val); // Removed by the compiler for the default argument
    if (val % 2 == 0) {
        val /= 2;
    } else {
        val *= 3;
        val++;
    }
}
rec(n, val);
return val;

C++ code 0.3.3.6: Calling myloopfunction() ➤ GITLAB

std::cout << "myloopfunction(10, 1) = " << myloopfunction(10, 1) << std::endl;
// Run with recorder
std::vector<std::pair<int, int>> store();
std::cout << "myloopfunction(10, 1) = "
    << myloopfunction(10, 1,
    &store)(int n, int val) -> void {
        store.emplace_back(n, val);
    }
    << std::endl;
std::cout << "History:" << std::endl;
for (auto &i : store) {
    std::cout << i.first << " \rightarrow " << i.second << std::endl;
}

0.3.4 Multiple Return Values

In Python it is customary to return several variables from a function call, which, in fact, amounts to returning a tuple of mixed-type objects:

def f(a, b):
    return min(a, b), max(a, b), (a+b)/2
x, y, z = f(1, 2)

In C++ this is also possible by using the tuple utility. For instance, the following function computes the minimal and maximal element of a vector and also returns its cumulative sum. It returns all these values.

C++ code 0.3.4.1: Function with multiple return values ➤ GITLAB

template<typename T>
std::tuple<T, T, std::vector<T>> extcumsum(const std::vector<T> &v) {
    // Local summation variable captured by reference by lambda function
    T sum{};
    // temporary vector for returning cumulative sum
    std::vector<T> w{};
    // cumulative summation
    std::transform(v.cbegin(), v.cend(), back_inserter(w),
    [&sum](T x) { sum += x; return(sum); });
    return (std::make_tuple(*std::min_element(v.cbegin(), v.cend()),
        *std::max_element(v.cbegin(), v.cend()),
        std::move(w)));

0. Introduction, 0.3. Programming in C++
This code snippet shows how to extract the individual components of the tuple returned by the previous function.

**C++ code 0.3.4.2: Calling a function with multiple return values**

```cpp
int main () {
    // initialize a vector from an initializer list
    std::vector<double> v({1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8});
    // Variables for return values
    double minv, maxv; // Extremal elements
    std::vector<double> cs; // Cumulative sums
    std::tie(minv, maxv, cs) = extcumsum(v);
    cout << "min = " << minv << " , max = " << maxv << endl;
    cout << "cs = [ " ; for(double x : cs) cout << x << " ; " ; cout << "]" << endl;
    return(0);
}
```

Be careful: many temporary objects might be created! A demonstration of this hidden cost is given in Exp. 0.3.5.27. From C++17 a more compact syntax is available:

**C++ code 0.3.4.3: Calling a function with multiple return values**

```cpp
int main () {
    // initialize a vector from an initializer list
    std::vector<double> v({1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8});
    // Definition of variables and assignment of return values all at once
    auto [minv, maxv, cs] = extcumsum(v);
    cout << "min = " << minv << " , max = " << maxv << endl;
    cout << "cs = [ " ; for(double x : cs) cout << x << " ; " ; cout << "]" << endl;
    return(0);
}
```

**Remark 0.3.4.4 ("auto" considered harmful)** C++ is a strongly typed programming language and every variable must have a precise type. However, the developer of templated classes and functions may not know the type of some variables in advance, because it can be deduced only after instantiation through the compiler. The `auto` keyword has been introduced to handle this situation.

There is a temptation to use `auto` profligately, because it is convenient, in particular when using templated data types. However, this denies a major benefit of types, consistency checking at compile time and, as a developer, one may eventually lose track of the types completely, which can lead to errors that are hard to detect.

Thus, the use of `auto` should be avoided, unless in the following situations:

- for variables inside templated functions or classes, whose precise type will only become clear during instantiation,
- for lambda functions, see Section 0.3.3,
- for return values of templated library (member) functions, whose type is “impossible to deduce” by the user. An example is expression templates in Eigen, refer to Rem. 1.2.1.11 below.
0.3.5 A Vector Class

Since C++ is an object oriented programming language, datatypes defined by classes play a pivotal role in every C++ program. Here, we demonstrate the main ingredients of a class definition and other important facilities of C++ for the class MyVector meant for objects representing vectors from $\mathbb{R}^n$. The codes can be found in ➤ GITLAB. A similar vector class is presented in [Fri19, Ch. 6].

```cpp
namespace myvec {
    class MyVector {
        public:
            using value_t = double;

            // Constructor creating constant vector, also default constructor
            explicit MyVector(std::size_t n = 0, double val = 0.0);

            // Constructor: initialization from an STL container
            template<typename Container> MyVector(const Container &v);

            // Constructor: initialization from an STL iterator range
            template<typename Iterator> MyVector(Iterator first, Iterator last);

            // Copy constructor, computational cost $O(n)$
            MyVector(const MyVector &mv);

            // Move constructor, computational cost $O(1)$
            MyVector(MyVector &&mv);

            // Copy assignment operator, computational cost $O(n)$
            MyVector &operator = (const MyVector &mv);

            // Move assignment operator, computational cost $O(1)$
            MyVector &operator = (MyVector &&mv);

            // Destructor
            virtual ~MyVector();

            // Type conversion to STL vector
            operator std::vector<double> () const;

            // Returns length of vector
            std::size_t size(void) const { return n; }

            // Access operators: rvalue & lvalue, with range check
            double operator [](std::size_t i) const;

            double &operator [](std::size_t i);

            // Comparison operators
            bool operator == (const MyVector &mv) const;

            bool operator != (const MyVector &mv) const;

            // Transformation of a vector by a function $\mathbb{R} \rightarrow \mathbb{R}$
            template<typename Functor>
            MyVector &transform(Functor &f);

            // Overloaded arithmetic operators
            // In place vector addition: $x += y$;
            MyVector &operator +=(const MyVector &mv);

            // In place vector subtraction: $x -= y$;
            MyVector &operator -= (const MyVector &mv);

            // In place scalar multiplication: $x *= a$;
            MyVector &operator *= (double alpha);

            // In place scalar division: $x /= a$;
            MyVector &operator /= (double alpha);

            // Vector addition
            MyVector operator + (MyVector mv) const;

            // Vector subtraction
            MyVector operator - (const MyVector &mv) const;

            // Scalar multiplication from right and left: $x = a \cdot y$; $x = y \cdot a$
            MyVector operator * (double alpha) const;

            friend MyVector operator * (double alpha, const MyVector &);
        }
    }
}
```
Note the use of a public static data member dbg in Line 63 that can be used to control debugging output by setting `MyVector::dbg = true` or `MyVector::dbg = false`.

Remark 0.3.5.2 (Contiguous arrays in C++) The class `MyVector` uses a C-style array and dynamic memory management with `new` and `delete` to store the vector components. This is for demonstration purposes only and not recommended.

**Arrays in C++**

In C++ use the STL container `std::vector<T>` for storing data in contiguous memory locations. Exception: use `std::array<T>`, if the number of elements is known at compile time.

§0.3.5.4 (Member and friend functions of `MyVector` ➔ GITLAB)

**C++ code 0.3.5.5: Constructor for constant vector, also default constructor, see Line 6 in Code 0.3.5.1 ➔ GITLAB**

```
MyVector MyVector(std::size_t _n, double _a):n(_n), data(nullptr) {
    if (dbg) cout << "{Constructor MyVector(" << _n << ") called" << ")" << endl;
    if (_n > 0) data = new double [_n];
    for (std::size_t l=0;l<_n;++l) data[l] = _a;
}
```

This constructor can also serve as default constructor (a constructor that can be invoked without any argument), because defaults are supplied for all its arguments.

The following two constructors initialize a vector from sequential containers according to the conventions of the STL.

**C++ code 0.3.5.6: Templated constructors copying vector entries from an STL container ➔ GITLAB**

```
template<typename Container>
```
2 MyVector::MyVector(const Container &v): n(v.size()), data(nullptr) {
3     if (dbg) cout << "\{MyVector\(\text{length } n\) constructed from container\}" << '}' << endl;
4     if (n > 0) {
5         double *tmp = (data = new double [n]);
6         for (auto i : v) *tmp++ = i; // foreach loop
7     }
8 }
9
Note the use of the new C++ 11 facility of a "foreach loop" iterating through a container in Line 7.

C++ code 0.3.5.7: Constructor initializing vector from STL iterator range ➔ GITLAB
1 template<typename Iterator>
2 MyVector::MyVector(Iterator first, Iterator last): n(0), data(nullptr) {
3     n = std::distance(first, last);
4     if (dbg) cout << "\{MyVector\(\text{length } n\) constructed from range\}" << '}' << endl;
5     if (n > 0) {
6         data = new double [n];
7         std::copy(first, last, data);
8     }
9 }
10
The use of these constructors is demonstrated in the following code

C++ code 0.3.5.8: Initialization of a MyVector object from an STL vector ➔ GITLAB
1 int main() {
2     myvec::MyVector::dbg = true;
3     std::vector<int> ivec = { 1,2,3,5,7,11,13 }; // initializer list
4     myvec::MyVector v1(ivec.cbegin(), ivec.cend());
5     myvec::MyVector v2(ivec);
6     myvec::MyVector vr(ivec.cbegin(), ivec.cend());
7     cout << "v1 = " << v1 << endl;
8     cout << "v2 = " << v2 << endl;
9     cout << "vr = " << vr << endl;
10    return(0);
11 }

The following output is produced:

\{MyVector\(\text{length 7}\) constructed from range\}
\{MyVector\(\text{length 7}\) constructed from container\}
\{MyVector\(\text{length 7}\) constructed from range\}
v1 = [ 1,2,3,5,7,11,13 ]
v2 = [ 1,2,3,5,7,11,13 ]
vr = [ 13,11,7,5,3,2,1 ]
\{Destructor for MyVector\(\text{length = 7}\)\}
\{Destructor for MyVector\(\text{length = 7}\)\}
\{Destructor for MyVector\(\text{length = 7}\)\}

The copy constructor listed next relies on the \textit{STL algorithm} \verb|std::copy| to copy the elements of an existing object into a newly created object. This takes \(n\) operations.
An important new feature of C++11 is move semantics which helps avoid expensive copy operations. The following implementation just performs a shallow copy of pointers and, thus, for large \( n \) is much cheaper than a call to the copy constructor from Code 0.3.5.9. The source vector is left in an empty vector state.

The following code demonstrates the use of `std::move()` to mark a vector object as disposable and allow the compiler to use the move constructor. The code also uses left multiplication with a scalar, see Code 0.3.5.23.

This code produces the following output. We observe that \( v1 \) is empty after its data have been “stolen” by \( v2 \).
We observe that the object \( v_1 \) is reset after having been moved to \( v_3 \).

Use `std::move` only for special purposes like above and only if an object has a move constructor. Otherwise a 'move' will trigger a plain copy operation. In particular, do not use `std::move` on objects at the end of their scope, e.g., within `return` statements.

The next operator effects copy assignment of an rvalue `MyVector` object to an lvalue `MyVector`. This involves \( O(n) \) operations.

```cpp
MyVector &MyVector::operator = (const MyVector &mv) {
  if (dbg) cout << "{ Copy assignment of MyVector (length " << n << " <= " << mv.n << ") " << '}' << endl;
  if (this == &mv) return (*this);
  if (n != mv.n) {
    n = mv.n;
    if (data != nullptr) delete [] data;
    if (n > 0) data = new double [n]; else data = nullptr;
  }
  if (n > 0) std::copy_n(mv.data, n, data);
  return (*this);
}
```

The move semantics is realized by an assignment operator relying on shallow copying.

```cpp
MyVector &MyVector::operator = (MyVector &&mv) {
  if (dbg) cout << "{ Move assignment of MyVector (length " << n << " <= " << mv.n << ") " << '}' << endl;
  if (data != nullptr) delete [] data;
  n = mv.n; data = mv.data;
  mv.n = 0; mv.data = nullptr;
  return (*this);
}
```

The destructor releases memory allocated by `new` during construction or assignment.

```cpp
MyVector::~MyVector(void) {
  if (dbg) cout << "{ Destructor for MyVector (length = " << n << ") " << '}' << endl;
  if (data != nullptr) delete [] data;
}
```

The `operator` keyword is also used to define implicit type conversions.

```cpp
MyVector::operator std::vector<double> () const {
```

0. Introduction, 0.3. Programming in C++
The bracket operator \[\] can be used to fetch and set vector components. Note that index range checking is performed; an exception is thrown for invalid indices. The following code also gives an example of operator overloading as discussed in § 0.3.1.2.

C++ code 0.3.5.16: rvalue and lvalue access operators ➤ GITLAB

```cpp
double MyVector::operator [] (std::size_t i) const {
    if (i >= n) throw(std::logic_error("\[\] out of range"));
    return data[i];
}
double &MyVector::operator [] (std::size_t i) {
    if (i >= n) throw(std::logic_error("\[\] out of range"));
    return data[i];
}
```

Componentwise direct comparison of vectors. Can be dangerous in numerical codes, cf. Rem. 1.5.3.15.

C++ code 0.3.5.17: Comparison operators ➤ GITLAB

```cpp
bool MyVector::operator == (const MyVector &mv) const {
    if (dbg) cout << "{Comparison ==: " << n << " <-> " << mv.n << "}" << endl;
    if (n != mv.n) return(false);
    else {
        for (std::size_t l=0; l<n; ++l)
            if (data[l] != mv.data[l]) return(false);
        return(true);
    }
}
bool MyVector::operator != (const MyVector &mv) const {
    return (!(*this == mv));
}
```

The transform method applies a function to every vector component and overwrites it with the value returned by the function. The function is passed as an object of a type providing a \( () \)-operator that accepts a single argument convertible to \texttt{double} and returns a value convertible to \texttt{double}.

C++ code 0.3.5.18: Transformation of a vector through a functor double \(\rightarrow\) double ➤ GITLAB

```cpp
template <typename Functor>
MyVector &MyVector::transform(Functor &f) {
    for (std::size_t l=0; l<n; ++l) data[l] = f(data[l]);
    return(*this);
}
```

The following code demonstrates the use of the transform method in combination with

1. a function object of the following type
2. A **lambda function** defined directly inside the call to `transform`.

```cpp
class SimpleFunction {
    double _a = 1.0;
public:
    SimpleFunction(double _a = 1.0):cnt(0), a(_a) {} // internal counter
    int cnt;
    const double a; // increment value
    double operator()(double x) { cnt++; return(x+a); }
};
```

```cpp
text
int main() {
    myvec::MyVector::dbg = false;
    double a = 2.0;  // increment
    int cnt = 0;  // external counter used by lambda function
    myvec::MyVector mv(std::vector<double>(
        {1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9}));
    mv.transform([a,&cnt] (double x) { cnt++; return(x+a); });
    cout << cnt << " operations, mv transformed = " << mv << endl;
    SimpleFunction trf(a); mv.transform(trf);
    cout << trf.cnt << " operations, mv transformed = " << mv << endl;
    mv.transform(SimpleFunction(-4.0));
    cout << "Final vector = " << mv << endl;
    return(0);
}
```

The output is

8 operations, mv transformed = [3.2, 4.3, 5.4, 6.5, 7.6, 8.7, 9.8, 10.9]  
8 operations, mv transformed = [5.2, 6.3, 7.4, 8.5, 9.6, 10.7, 11.8, 12.9]  
Final vector = [1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9]

Operator overloading provides the "natural" vector operations in \( \mathbb{R}^n \) both in place and with a new vector created for the result.

```cpp
class MyVector {
public:
    MyVector operator+=(const MyVector &mv) {
        if ( dbg ) cout << "{ operator +=, MyVector of length " << n << " }" << endl;
        if ( n != mv.n ) throw(std::logic_error("+=: vector size mismatch"));
        for(std::size_t l=0;l<n;++l) data[l] += mv.data[l];
        return(*this);
    }

class MyVector {
public:
    MyVector operator-=(const MyVector &mv) {
        if ( dbg ) cout << "{ operator -=, MyVector of length " << n << " }" << endl;
        if ( n != mv.n ) throw(std::logic_error("-=: vector size mismatch"));
        for(std::size_t l=0;l<n;++l) data[l] -= mv.data[l];
        return(*this);
    }

class MyVector {
public:
    MyVector operator*=(double alpha) {
        if ( dbg ) cout << "{ operator *=, MyVector of length " << n << " }" << endl;
```
```cpp
for (std::size_t l = 0; l < n; ++l) data[l] *= alpha;
return(*this);
}

MyVector &MyVector::operator /=(double alpha) {
    if (dbg) cout << "{operator /=, MyVector of length "
                 << n << "}' << endl;
    for (std::size_t l = 0; l < n; ++l) data[l] /= alpha;
    return(*this);
}
```

C++ code 0.3.5.22: Binary arithmetic operators (two arguments) ➤ GITLAB

```cpp
MyVector MyVector::operator + (MyVector mv) const {
    if (dbg) cout << "{operator +, MyVector of length "
                 << n << "}' << endl;
    if (n != mv.n) throw(std::logic_error("+: vector size mismatch"));
    MyVector tmp(*this); tmp += mv;
    return(mv);
}
```

```cpp
MyVector MyVector::operator - (const MyVector &mv) const {
    if (dbg) cout << "{operator -, MyVector of length "
                 << n << "}' << endl;
    if (n != mv.n) throw(std::logic_error("+: vector size mismatch"));
    MyVector tmp(*this); tmp -= mv;
    return(tmp);
}
```

```cpp
MyVector MyVector::operator * (double alpha) const {
    if (dbg) cout << "{operator *, MyVector of length "
                 << n << "}' << endl;
    MyVector tmp(*this); tmp *= alpha;
    return(tmp);
}
```

```cpp
MyVector MyVector::operator / (double alpha) const {
    if (dbg) cout << "{operator /, MyVector of length " << n << "}' << endl;
    MyVector tmp(*this); tmp /= alpha;
    return(tmp);
}
```

C++ code 0.3.5.23: Non-member function for left multiplication with a scalar ➤ GITLAB

```cpp
MyVector operator * (double alpha, const MyVector &mv) {
    if (MyVector::dbg) cout << "{operator a*, MyVector of length "
                             << mv.n << "}' << endl;
    MyVector tmp(mv); tmp *= alpha;
    return(tmp);
}
```

C++ code 0.3.5.24: Euclidean norm ➤ GITLAB

```cpp
double MyVector::norm(void) const {
    if (dbg) cout << "{norm: MyVector of length " << n << "}' << endl;
    double s = 0;
    for (std::size_t l = 0; l < n; ++l) data[l] -= alpha;
    return(*this);
}
```
Adopting the notation in some linear algebra texts, the operator \( * \) has been chosen to designate the Euclidean inner product:

```cpp
double MyVector::operator * (const MyVector &mv) const {
    if (dbg) cout << "{ dot *, MyVector of length " << n << " }" << endl;
    if (n != mv.n) throw (std::logic_error("dot: vector size mismatch"));
    double s = 0;
    for (std::size_t l = 0; l < n; ++l) s += (data[l] * mv.data[l]);
    return (s);
}
```

At least for debugging purposes every reasonably complex class should be equipped with output functionality.

**C++ code 0.3.5.26: Non-member function output operator ➔ GITLAB**

```cpp```
```cpp
std::ostream &operator << (std::ostream &o, const MyVector &mv) {
    o << "[ ";
    for (std::size_t l = 0; l < mv.n; ++l)
        o << mv.data[l] << (l == mv.n-1? ' ' : ', ');
    return (o << "]");
}
```
Several temporary objects are created and destroyed and quite a few copy operations take place. The situation would be worse unless move semantics was available; if we had not supplied a move constructor, a few more copy operations would have been triggered. Even worse, the frequent copying of data runs a high risk of cache misses. This is certainly not an efficient way to do elementary vector operations though it looks elegant at first glance.

EXAMPLE 0.3.5.29 (Gram-Schmidt orthonormalization based on MyVector implementation) Gram-Schmidt orthonormalization has been taught in linear algebra and its theory will be revisited in § 1.5.1.1. Here we use this simple algorithm from linear algebra to demonstrate the use of the vector class MyVector defined in Code 0.3.5.1.

The templated function gramschmidt takes a sequence of vectors stored in a std::vector object. The actual vector type is passed as a template parameter. It has to supply length and norm member functions as well as in place arithmetic operations -=, /= and +=. Note the use of the highlighted methods of the std::vector class.

```cpp
template <typename Vec>
std::vector<Vec> gramschmidt(const std::vector<Vec> &A, double eps = 1E-14) {
    const int k = A.size(); // no. of vectors to orthogonalize
    const int n = A[0].size(); // length of vectors
    cout << " gramschmidt orthonormalization for " << k << ' ' << n << "-vectors " << endl;

    std::vector<Vec> Q((A[0]/A[0].norm())); // output vectors
    for (int j=1; (j<k) && (j<n); ++j) {
        Q.push_back(A[j]);
        for (int l=0; l<j; ++l) Q.back() -= (A[j]*Q[l]).Q[l];
        if (Q.back().norm() < eps*A[j].norm()) { // premature termination ?
            Q.pop_back(); break;
        }
        Q.back() /= Q.back().norm(); // normalization
    }
}
```

C++ code 0.3.5.30: templated function for Gram-Schmidt orthonormalization ➔ GITLAB
This driver program calls a function that initializes a sequence of vectors and then orthonormalizes them by means of the Gram-Schmidt algorithm. Eventually orthonormality of the computed vectors is tested. Please pay attention to

- the use of auto to avoid cumbersome type declarations,
- the for loops following the “foreach” syntax.
- automatic indirect template type deduction for the templated function \texttt{gramschmidt} from its argument. In Line 6 the function \texttt{gramschmidt<MyVector>} is instantiated.

\textbf{C++ code 0.3.5.31: Driver code for Gram-Schmidt orthonormalization}

```cpp
int main() {
    myvec::MyVector::dbg = false;
    const int n = 7; const int k = 7;
    auto A = initvectors(n,k,[](int i,int j) {
        return std::min(i+1,j+1); });
    auto Q = gramschmidt(A); // instantiate template for MyVector
    cout << '"Set of vectors to be orthonormalized:" << endl;
    for (const auto &a : A) { cout << a << endl; }
    cout << '"Output of Gram-Schmidt orthonormalization: " << endl;
    for (const auto &q : Q) { cout << q << endl; }
    cout << '"Testing orthogonality:" << endl;
    for (const auto &qi : Q) {
        for (const auto &qj : Q)
            cout << std::setprecision(3) << std::setw(9) << qi * qj << ' ';
        cout << endl; }
    return(0);
}
```

This initialization function takes a functor argument as discussed in Section 0.3.3.

\textbf{C++ code 0.3.5.32: Initialization of a set of vectors through a functor with two arguments}

```cpp
template<typename Functor>
std::vector<myvec::MyVector> initvectors(std::size_t n, std::size_t k, Functor &f) {
    std::vector<MyVector> A{};
    for (int j=0;j<k;++j) {
        A.push_back(MyVector(n));
        for (int i=0;i<n;++i)
            (A.back())[i] = f(i, j);
    }
    return(A);
}
```
Bibliography


