Texts in Computational Science and Engineering

15

Editors

More information about this series at http://www.springer.com/series/5151

Svein Linge • Hans Petter Langtangen

Programming for Computations - Python

A Gentle Introduction to Numerical Simulations with Python 3.6

Second Edition

Springer Open

Svein Linge
Fac of Tech, Natural Sci & Maritime Sci
University of South-Eastern Norway
Porsgrunn, Norway

Hans Petter Langtangen
Simula Research Laboratory BioComp
Lysaker, Norway

ISSN 1611-0994 ISSN 2197-179X (electronic)
Texts in Computational Science and Engineering
ISBN 978-3-030-16876-6 ISBN 978-3-030-16877-3 (eBook)
https://doi.org/10.1007/978-3-030-16877-3

Mathematics Subject Classification (2010): 26-01, 34A05, 34A30, 34A34, 39-01, 40-01, 65D15, 65D25, 65D30, 68-01, 68N01, 68N19, 68N30, 70-01, 92D25, 97-04, 97U50

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To My Family
Thanks to my dear wife, Katrin, and our lovely children, Stian, Mia, and Magnus, for their love, support, and patience. I am a very lucky man.

To Hans Petter
Dear friend and coauthor, it is so sad you are no longer among us.[1] *Thank you for everything. I dedicate this second edition of our book to you.*

Porsgrunn, Norway Svein Linge
June 2018

[1] Professor Hans Petter Langtangen (https://en.wikipedia.org/wiki/Hans_Petter_Langtangen) passed away with cancer on the 10th of October, 2016.

Preface

Computing, in the sense of doing mathematical calculations, is a skill that mankind has developed over thousands of years. Programming, on the other hand, is in its infancy, with a history that spans a few decades only. Both topics are vastly comprehensive and usually taught as separate subjects in educational institutions around the world, especially at the undergraduate level. This book is about the *combination* of the two, because computing today becomes so much more powerful when combined with programming.

Most universities and colleges implicitly require students to specialize in computer science if they want to learn the craft of programming, since other student programs usually do not offer programming to an extent demanded for really mastering this craft. Common arguments claim that it is sufficient with a brief introduction, that there is not enough room for learning programming in addition to all other must-have subjects, and that there is so much software available that few really need to program themselves. A consequence is that engineering students often graduate with shallow knowledge about programming, unless they happened to choose the computer science direction.

We think this is an unfortunate situation. There is no doubt that practicing engineers and scientists need to know their pen-and-paper mathematics. They must also be able to run off-the-shelf software for important standard tasks and will certainly do that a lot. Nevertheless, the benefits of mastering programming are many.

Why Learn Programming?

1. Ready-made software is limited to handling certain standard problems. What do you do when the problem at hand is not covered by the software you bought? Fortunately, a lot of modern software systems are extensible via programming. In fact, many systems demand parts of the problem specification (e.g., material models) to be specified by computer code.
2. With programming skills, you may extend the flexibility of existing software packages by combining them. For example, you may integrate packages that do not speak to each other from the outset. This makes the work flow simpler, more efficient, and more reliable, and it puts you in a position to attack new problems.

3. It is easy to use excellent ready-made software the wrong way. The insight in programming and the mathematics behind is fundamental for understanding complex software, avoiding pitfalls, and becoming a safe user.
4. Bugs (errors in computer code) are present in most larger computer programs (also in the ones from the shop!). What do you do when your ready-made software gives unexpected results? Is it a bug, is it the wrong use, or is it the mathematically correct result? Experience with programming of mathematics gives you a good background for answering these questions. The one who can program can also make tailored code for a simplified problem setting and use that to verify the computations done with off-the-shelf software.
5. Lots of skilled people around the world solve computational problems by writing their own code and offering those for free on the Internet. To take advantage of this truly great source of software in a reliable way, one must normally be able to understand and possibly modify computer code offered by others.
6. It is recognized worldwide that students struggle with mathematics and physics. Too many find such subjects difficult and boring. With programming, we can execute the good old subjects in a brand new way! According to the authors' own experience, students find it much more motivating and enlightening when programming is made an integrated part of mathematics and physical science courses. In particular, the problem being solved can be much more realistic than when the mathematics is restricted to what you can do with pen and paper.
7. Finally, we launch our most important argument for learning computer programming: the *algorithmic thinking* that comes with the process of writing a program for a computational problem enforces a thorough understanding of both the problem and the solution method. We can simply quote the famous Norwegian computer scientist Kristen Nygaard: "Programming is understanding."

In the authors' experience, programming is an excellent pedagogical tool for understanding mathematics: "You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program" (Alan Perlis, computer scientist, 1922–1990). Consider, for example, integration. A numerical method for integration has a much stronger focus on what the integral actually is and means compared to analytical methods, where much time and effort must be devoted to integration by parts, integration by substitution, etc. Moreover, when programming the numerical integration formula, it becomes evident that it works for "all" mathematical functions and that the implementation should be in terms of a *general* function applicable to "all" integrals. In this way, students learn to recognize a special problem as belonging to a class of problems (e.g., integration, differential equations, root finding), for which we have general numerical methods implemented in a widely applicable software. When they write this software, as we do in this book, they learn how to generalize and increase the abstraction level of the mathematical problem. When they use this software, they learn how a special case should be attacked by general methods and software for the class of problems that comprises the special case at hand. This is the power of mathematics in a nutshell, and it is paramount that students understand this way of thinking.

Target Audience and Background Knowledge This book was written for students, teachers, engineers, and scientists who know *nothing* about programming and numerical methods from before but who seek a *minimum* of the fundamental skills required to get started with programming as a tool for solving scientific and engineering problems. Some knowledge of one- and multivariable calculus is assumed. The basic programming concepts are presented in Chaps. 1–5 (about 150 pages), before practical applications of these concepts are demonstrated in important mathematical subjects addressed in the remaining parts of the book (Chaps. 6–9). Each chapter is followed by a set of exercises that covers a wide range of application areas, e.g., biology, geology, statistics, physics, and mathematics. The exercises were particularly designed to bring across important points from the text.

Learning the very basics of programming should not take long, but as with any other craft, mastering the skill requires continued and extensive practice. Some beginning practice is gained through Chaps. 6–9, but the authors strongly emphasize that this is only a start. Students should continue to practice programming in subsequent courses, while those who exercise self-study should keep up the learning process through continued application of the craft. The book is a good starting point when teaching computer programming as an integrated part of standard university courses in mathematics and natural science. In our experience, such an integration is doable and indeed rewarding.

Numerical Methods An overall goal with this book is to motivate computer programming as a very powerful tool for doing mathematics. All examples are related to mathematics and its use in engineering and science. However, to solve mathematical problems through computer programming, we need numerical methods. Explaining basic numerical methods is therefore an integral part of the book. Our choice of topics is governed by what is most needed in science and engineering, as well as in the teaching of applied natural science courses. Mathematical models are then central, with differential equations constituting the most frequent type of models. Consequently, the numerical focus in this book is on differential equations. As soft pedagogical starters for the programming of mathematics, we have chosen the topics of numerical integration and root finding. We remark that the book is deliberately brief on numerical methods. This is because our focus is on *implementing* numerical algorithms, and to develop reliable, working programs, the programmer must be confident about the basic ideas of the numerical approximations involved.

The Computer Language: Python We have chosen to use the programming language Python, because this language gives a very compact and readable code that closely resembles the mathematical recipe for solving the problem at hand. Python also has a gentle learning curve.

Other computer languages, like Fortran, C, and C++, have a strong position in science and engineering. During the last two decades, however, there has been a significant shift in popularity from these compiled languages to more high-level and easier-to-read languages, for instance, MATLAB, Python, R, Maple, Mathematica, and IDL. This latter class of languages is computationally less efficient but superior with respect to overall human problem-solving efficiency. This book emphasizes

how to think like a programmer, rather than focusing on technical language details. Thus, the book should put the reader in a good position for learning other programming languages later, including the classic ones: Fortran, C, and C++.

How This Book Is Different There are numerous texts on computer programming and numerical methods, so how does the present one differ from the existing literature? Compared to standard books on numerical methods, our book has a much stronger emphasis on the craft of programming and on verification. We want to give students a thorough understanding of how to think about programming as a problem-solving method and how to provide convincing evidence for program correctness.

Even though there are lots of books on numerical methods where many algorithms have a corresponding computer implementation (see, e.g., [1, 3–6, 10, 15–17, 20, 23, 25, 27–31]—the latter two apply Python), it is often assumed that the reader "can program" beforehand. The present book teaches the craft of structured programming along with the fundamental ideas of numerical methods. In this book, unit testing and corresponding test functions are introduced early on. We also put much emphasis on coding algorithms as *functions*, as opposed to "flat programs," which often dominate in the literature and among practitioners. Functions are reusable because they utilize the general formulation of a mathematical algorithm such that it becomes applicable to a large class of problems.

There are also numerous books on computer programming, but not many that really emphasize how to *think* about programming in the context of numerical methods and scientific applications. One such book is [11], which gives a comprehensive introduction to Python programming and the thinking about programming as a computer scientist.

Sometimes, however, one needs a text like the present one. It does not go so deep into language-specific details, but rather targets the shortest path to reliable mathematical problem-solving through programming. With this attitude in mind, a lot of topics were left out of the present book, simply because they were not *strictly* needed in the mathematical problem-solving process. Examples of such topics are object-oriented programming and Python dictionaries (of which the latter omission is possibly subject to more debate). If you find the present book too shallow, [11] might be the right choice for you. That source should also work nicely as a more in-depth successor of the present text.

Whenever the need for a *structured introduction to programming* arises in science and engineering courses, the present book may be your option, either for self-study or for use in organized teaching. The thinking, habits, and practice covered herein will put readers in a firm position for utilizing and understanding the power of computers for problem-solving in science and engineering.

Changes to the First Edition

1. All code is now in Python version 3.6 (the previous edition was based on Python version 2.7).
2. In the first edition, the introduction to programming was basically covered in 50 pages by Chap. 1 (*The First Few Steps*) and Chap. 2 (*Basic Constructions*). This *is* enough to get going, but many readers soon want more details. In this second edition, these two chapters have therefore been extended and split up into five

chapters. Explanations are now more complete, previous examples have been modified, new examples have been added, and more. In particular, the importing of code is now elaborated on in a greater detail, so is the making of modules. Also, Sect. 4.2 is new, illustrating the important stepwise strategy of code writing through a dedicated example. The five first chapters now cover about 150 pages that explain, in a brief and simple manner, all the code basics required to follow the remaining parts of the book.

3. The new Chap. 6 (*Computing Integrals and Testing Code*) and Chap. 7 (*Solving Nonlinear Algebraic Equations*) are seen as gentle first applications of programming to problem-solving in mathematics. Both these chapters now precede the mathematically more challenging Chaps. 8 and 9, which treat basic numerical solving of ODEs and PDEs, respectively (the chapter *Solving Nonlinear Algebraic Equations* was, in the first edition, the final chapter of the book, but it seems more appropriate to let it act as a "warm-up" chapter, together with the new Chap. 6, for the two final chapters on differential equation solving).
4. Section 8.1 (*Filling a Water Tank: Two Cases*) is new, particularly written for readers who lack experience with differential equations.
5. Section 8.5 (*Rate of Convergence*) is new, explaining convergence rate related to differential equations.
6. New exercises have been added, e.g., on Fibonacci numbers, the Leapfrog method, Adams-Bashforth methods, and more.
7. Errors and typos have been corrected, and many explanations have been reformulated throughout.

Supplementary Materials All program and data files referred to herein are available from the book's (2nd edition) primary web site:
https://github.com/slgit/prog4comp_2.

Acknowledgments We want to thank all students who attended the courses *FM1006 Modelling and simulation of dynamic systems*, *FM1115 Scientific Computing*, *FB1012 Mathematics*, and *FB2112 Physics* at the University of South-Eastern Norway over the last 5–10 years. They worked their way through early versions of this text and gave us constructive and positive feedback that helped us correct errors and improve the book in so many ways. Special acknowledgment goes to Guandong Kou, Edirisinghe V. P. J. Manjula, and Yapi Donatien Achou for their careful reading of the manuscript (first edition) and constructive suggestions for improvement. The constructive feedback and good suggestions received from Om Prakash Chapagain (second edition) is also highly appreciated. We thank all our good colleagues at the University of South-Eastern Norway, the University of Oslo, and Simula Research Laboratory for their continued support and interest, for the enlightening discussions, and for providing such an inspiring environment for teaching and science.

Special thanks go to Prof. Kent-Andre Mardal, the University of Oslo, for his insightful comments and suggestions.

The authors would also like to thank the Springer team with Dr. Martin Peters, Thanh-Ha Le Thi, and Leonie Kunz for the effective editorial and production process.

This text was written in the DocOnce[1] [12] markup language.

Lysaker, Norway — Hans Petter Langtangen
December 2015

Porsgrunn, Norway — Svein Linge
June 2018

[1] https://github.com/hplgit/doconce.

Abstract

This second edition of the book presents computer programming as a key method for solving mathematical problems and represents a major revision: all code is now written in Python version 3.6 (the first edition was based on Python version 2.7). The first two chapters of the previous edition have been extended and split up into five new chapters, thus expanding the introduction to programming from 50 to 150 pages. Throughout, explanations are now more complete, previous examples have been modified, and new sections, examples, and exercises have been added. Also, errors and typos have been corrected. The book was inspired by the Springer book TCSE 6, *A Primer on Scientific Programming with Python* (by Langtangen), but the style is more accessible and concise in keeping with the needs of engineering students. The book outlines the shortest possible path from no previous experience with programming to a set of skills that allows the students to write simple programs for solving common mathematical problems with numerical methods in engineering and science courses. The emphasis is on generic algorithms, clean design of programs, use of functions, and automatic tests for verification.

Contents

List of Exercises