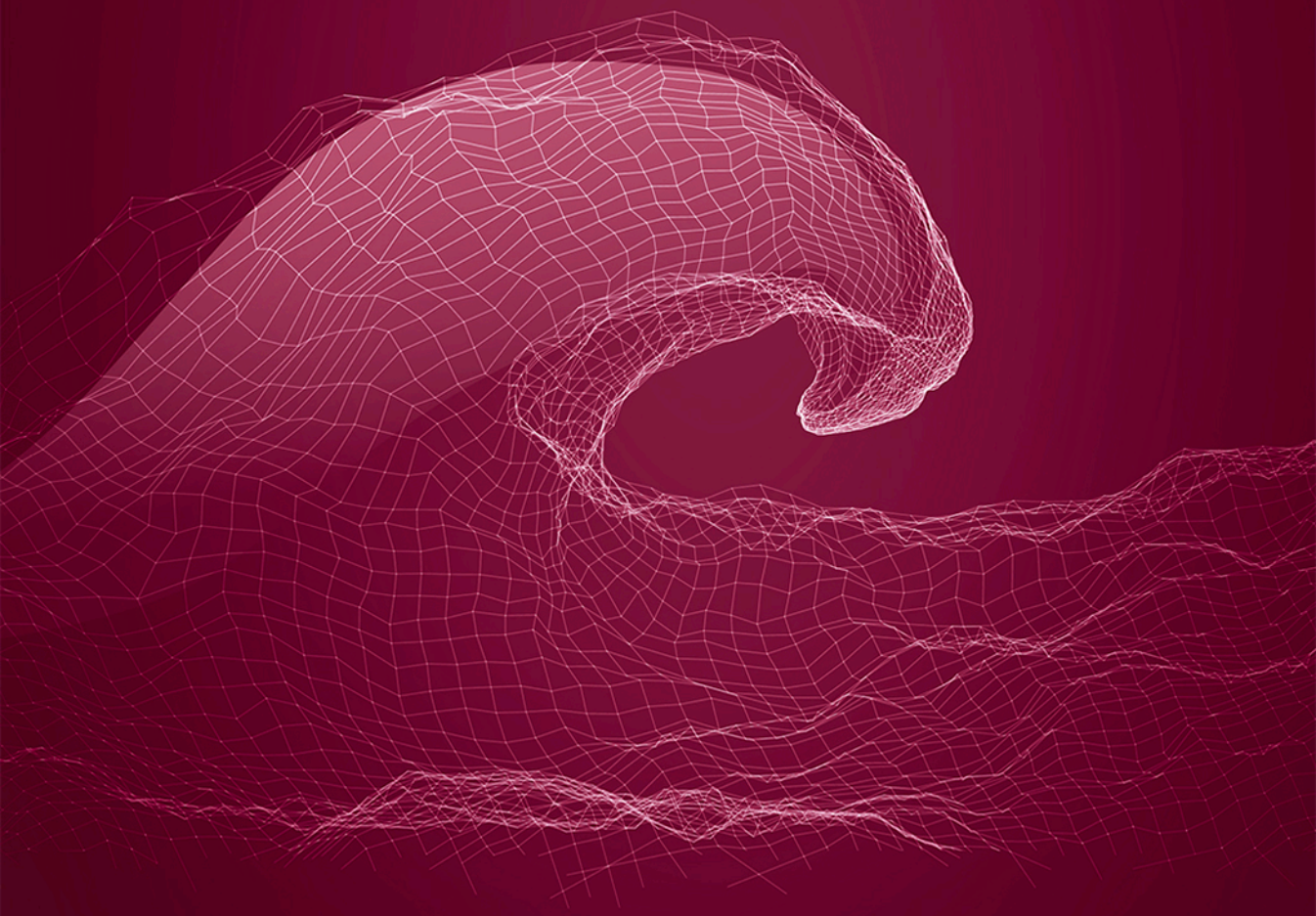
NUMERICAL METHODS IN PHYSICS WITH PYTHON

SECOND EDITION



ALEX GEZERLIS

Numerical Methods in Physics with Python

Bringing together idiomatic Python programming, foundational numerical methods, and physics applications, this is an ideal standalone textbook for courses on computational physics. All the frequently used numerical methods in physics are explained, including foundational techniques and hidden gems on topics such as linear algebra, differential equations, root-finding, interpolation, and integration. The second edition of this introductory book features several new codes and 140 new problems (many on physics applications), as well as new sections on the singular-value decomposition, derivative-free optimization, Bayesian linear regression, neural networks, and partial differential equations. The last section in each chapter is an in-depth project, tackling physics problems that cannot be solved without the use of a computer. Written primarily for students studying computational physics, this textbook brings the non-specialist quickly up to speed with Python before looking in detail at the numerical methods often used in the subject.

Alex Gezerlis is Professor of Physics at the University of Guelph. Before moving to Canada, he worked in Germany, the United States, and Greece. He has received several research awards, grants, and allocations on supercomputing facilities. He has taught undergraduate and graduate courses on computational methods, as well as courses on quantum field theory, subatomic physics, and science communication.

Praise for the Second Edition

"Gezerlis' book *Numerical Methods in Physics* with Python is a beautiful example of how an established subject can be brought to the next level by making it very accessible and by introducing several insightful and interdisciplinary applications. This second edition considerably extends the set of exercises, resulting in an extremely useful resource for both students and teachers. Strongly recommended!"

Sonia Bacca, *Johannes Gutenberg-Universität Mainz*

"This new edition of *Numerical Methods*… is another great example of Gezerlis' passion for teaching and for doing so carefully and precisely. Especially welcome, in my view, are the addition of problems at the end of each chapter and the discussion of singular value decomposition (SVD) and Bayesian methods. The SVD is one of the crown jewels of linear algebra which modern students interested in machine learning will surely find beneficial. To physics, computer science, or engineering students mesmerized by the fast Fourier transform, Gezerlis' excellent explanation of it in Chapter 6 is likely to shed some light on the underlying divide-and-conquer algorithm, which is an essential classic."

Joaquin Drut, *University of North Carolina at Chapel Hill*

"A fantastic addition as an introductory textbook for computational physics. The book is timely, and the author made thoughtful and in my view many wise choices. The book is comprehensive and yet accessible to undergraduate students."

Shiwei Zhang, *Flatiron Institute and College of William & Mary*

Praise for the First Edition

"I enthusiastically recommend *Numerical Methods in Physics with Python* by Professor Gezerlis to any advanced undergraduate or graduate student who would like to acquire a solid understanding of the basic numerical methods used in physics. The methods are demonstrated with Python, a relatively compact, accessible computer language, allowing the reader to focus on understanding how the methods work rather than on how to program them. Each chapter offers a self-contained, clear, and engaging presentation of the relevant numerical methods, and captivates the reader with well-motivated physics examples and interesting physics projects. Written by a leading expert in computational physics, this outstanding textbook is unique in that it focuses on teaching basic numerical methods while also including a number of modern numerical techniques that are usually not covered in computational physics textbooks."

Yoram Alhassid, *Yale University*

"In *Numerical Methods in Physics with Python* by Gezerlis, one finds a resource that has been sorely missing! As the usage of Python has become widespread, it is too often the case that students take libraries, functions, and codes and apply them without a solid understanding of what is truly being done 'under the hood' and why. Gezerlis' book fills this gap with clarity and rigor by covering a broad number of topics relevant for physics, describing the underlying techniques and implementing them in detail. It should be an important resource for anyone applying numerical techniques to study physics."

Luis Lehner, *Perimeter Institute*

"Gezerlis' text takes a venerable subject – numerical techniques in physics – and brings it up to date and makes it accessible to modern undergraduate curricula through a popular, open-source programming language. Although the focus remains squarely on numerical techniques, each new lesson is motivated by topics commonly encountered in physics and concludes with a practical hands-on project to help cement the students' understanding. The net result is a textbook which fills an important and unique niche in pedagogy and scope, as well as a valuable reference for advanced students and practicing scientists."

Brian Metzger, *Columbia University*

Numerical Methods in Physics with Python

Second Edition

ALEX GEZERLIS
University of Guelph

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment, a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781009303859

DOI: 10.1017/9781009303897

First published 2020
Second edition published 2023

A catalogue record for this publication is available from the British Library.

ISBN 978-1-009-30385-9 Hardback
ISBN 978-1-009-30386-6 Paperback

Additional resources for this publication at www.cambridge.org/gezerlis2 and www.numphyspy.org

To Marcos, ψυχή βαθιά

My soul, rather than yearn for life immortal,
press into service every shift at your disposal.

Pindar

Contents

Preface

> The health of the eye seems to demand a horizon.
> We are never tired, so long as we can see far enough.
>
> Ralph Waldo Emerson

This is a textbook for advanced undergraduate (or beginning graduate) courses on Computational Physics. To explain what this means, I first go over what this book is *not*.

First, this is not a text that focuses mainly on physics applications and basic programming, only bringing up numerical methods as the need arises. It's true that such an approach would have the benefit of giving rise to beautiful visualizations and helping students gain confidence in using computers to study science. The disadvantage of this approach is that it tends to rely on external libraries, i.e., "black boxes". To make an analogy with non-computational physics, we teach students calculus before seeing how it helps us do physics. In other words, an instructor would not claim that derivatives are important but already well-studied, so we'll just employ a package that takes care of them. That being said, a physics-applications-first approach may be appropriate for a more introductory course (the type with a textbook that has the answers in the back) or perhaps as a computational addendum to an existing text on mechanics, electromagnetism, and so on.

Second, this is not a text addressing a small subset of modern computational methods. Depending on the instructor's interests and expertise, computational courses sometimes specialize on a single theme, such as: simulations (e.g., molecular dynamics or Monte Carlo), data analysis (e.g., uncertainty quantification), or partial differential equations (e.g., continuum dynamics). Such a targeted approach has the advantage of being intimately connected to research, at the cost of assuming students have picked up the necessary foundational material from elsewhere. To return to the analogy with non-computational physics, a first course on electromagnetism would never skip over things like basic electrostatics to get directly to, say, the Yang–Mills Lagrangian just because non-abelian gauge theory is more "current". Even so, an approach that focuses on modern computational technology is relevant to a more advanced course: once students have mastered the foundations, they can turn to state-of-the-art methods that tackle research problems.

The present text attempts to strike a happy medium: a broad spectrum of numerical methods is studied in detail and then applied to questions from undergraduate physics, via idiomatic implementations in the Python programming language. When selecting and discussing topics, I have prioritized pedagogy over novelty; this is reflected in the chapter titles, which are pretty standard. Of course, my views on what is pedagogically superior are mine alone, so the end result also happens to be original in some respects. Below, I touch upon some of the main features of this book, with a view to orienting the reader.

- **Idiomatic Python:** the book employs Python 3, which is a popular, open-source programming language. A pedagogical choice I have made is to start out with standard Python, use it for a few chapters, and only then turn to the NumPy library; I have found that this helps students who are new to programming in Python effectively distinguish between lists and NumPy arrays. The first chapter includes a discussion of modern programming idioms, which allow me to write shorter codes in the following chapters, thereby emphasizing the numerical method over programming details. This is somewhat counterintuitive: teaching more "advanced" programming than is usual in computational-physics books allows the programming to recede into the background. In other words, not having to fight with the programming language every step of the way makes it *easier* to focus on the physics (or the math).
- **Modern numerical-analysis techniques:** I devote an entire chapter to questions of numerical precision and roundoff error; I hope that the lessons learned there will pay off when studying the following chapters, which typically focus more on approximation-error themes. While this is not a volume on numerical analysis, it does contain a bit more on applied math than is typical: in addition to standard topics, this also includes modern techniques that haven't made it to computational-physics books before (e.g., automatic differentiation or interpolation at Chebyshev points). Similarly, the section on errors in linear algebra glances toward monographs on matrix perturbation theory. To paraphrase Forman Acton [2], the idea here is to ensure that the next generation does not think that an obligatory decimal point is slightly demeaning.
- **Methods "from scratch":** chapters typically start with a pedagogical discussion of a crude algorithm and then advance to more complex methods, in several cases also covering state-of-the-art techniques (when they do not require elaborate bookkeeping). Considerable effort is expended toward motivating and explaining each technique as it is being introduced. Similarly, the chapters are ordered in such a way that the presentation is cumulative. Thus, the book attempts to discuss things "from scratch", i.e., without referring to specialized background or more advanced references; physicists do not expect lemmas and theorems, but do expect to be convinced.[1] Throughout the text, the phrases "it can be shown"[2] and "stated without proof" are actively avoided, so this book may also be used in a flipped classroom, perhaps even for self-study. As part of this approach, I frequently cover things like convergence properties, operation counts, and the error scaling of different numerical methods. When space constraints made it impossible to reach for *simplex munditiis* in explaining a given method, I quietly omitted that method. This is intended as a "first book" on the subject, which should enable students to confidently move on to more advanced expositions.
- **Methods implemented:** while the equations and figures help explain why a method should work, the insight that can be gleaned from an existing implementation of a given algorithm is crucial. I have worked hard to ensure that these code listings are embedded in the main discussion, not tossed aside at the end of the chapter or in an online supplement. Even so, each implementation is typically given its own subsection, in order to

[1] *Nullius in verba*, the motto of the Royal Society, comes to mind. The idea, though not the wording, can clearly be traced to Heraclitus' fragment 50: "Listen, not to me, but to reason".

[2] An instance of *proof by omission*, but still better than "it can be easily shown" (*proof by intimidation*).

help instructors who are pressed for time in their selection of material. Since I wanted to keep the example programs easy to talk about, they are quite short, never longer than a page. In an attempt to avoid the use of black boxes, I list and discuss implementations of methods that are sometimes considered advanced (e.g., the QR eigenvalue method or the fast Fourier transform). While high-quality libraries like NumPy and SciPy contain implementations of such methods, the point of a book like this one is precisely to teach students how and why a given method works. The programs provided (whose filenames also appear in the book's index) can function as templates for further code development on the student's part, e.g., when solving the end-of-chapter problems.

- **Clear separation between numerical method and physics problem:** each chapter focuses on a given numerical theme. The first section always discusses physics scenarios that touch upon the relevant tools; these "motivational" topics are part of the standard undergrad physics curriculum, ranging from classical mechanics, through electromagnetism and statistical mechanics, to quantum mechanics. The bulk of the chapter then focuses on several numerical methods and their implementation, typically without bringing up physics examples. The last numbered section in each chapter is a Project: in addition to involving topics that were introduced in earlier sections (or chapters), these physics projects allow students to carry out calculations they wouldn't attempt without the help of a computer. These projects also provide a first taste of "programming-in-the-large". As a result of this design choice, the book may also be useful to beginning physics students or even students in other areas of science and engineering (with a more limited physics background). Even the primary audience may benefit from the structure of the text in the future, when tackling different physics questions. In the same spirit, the physics-oriented problems in each chapter's problem set are labelled with [$\mathcal{P}$]; these are placed near the end, presupposing the maturity developed while working on the earlier problems. (Since most problems involve some coding, the ones that are purely analytical are labelled with [$\mathcal{A}$], into the bargain.)
- **Second edition includes six new sections on:**
  - the singular-value decomposition (section 4.5),
  - derivative-free optimization (sections 5.5.2 and 5.6.5),
  - maximum-likelihood and Bayesian approaches to linear regression (section 6.6),
  - non-linear fitting via the Gauss–Newton method and neural networks (section 6.7),
  - finite-difference approaches to the diffusion equation (section 8.5.2).

  Six original codes are associated with these sections. Section 6.6 may be of special benefit to readers interested in experimental physics. I found that brief yet meaty introductions to these ideas are useful to physics students, at both the undergraduate and graduate levels. As always, the point was to avoid the dreaded phrase "it turns out that", i.e., the use of (analytical or programming) black boxes. In addition to the totally new material, using the book in a classroom setting has inspired a very large number of other modifications throughout the volume, ranging from minor tweaks (e.g., now explicitly citing problem numbers in the main text) to complete rewrites of selected first-edition sections. From start to finish, I have tried to navigate between Scylla (familiar notation obscuring conceptual subtleties) and Charybdis (too many strange-looking symbols).

- **Second edition includes 140 new problems on:** (a) extensions of techniques introduced in the main text, (b) topics that would otherwise take too many pages to discuss (e.g., problems 5.38, 5.39, and 5.40 on constrained minimization), and (c) a large number of physical applications: I have now included problems on standard themes (e.g., problem 7.65 on the Ising model in two dimensions, problem 8.58 on molecular dynamics for the Lennard–Jones potential, or problem 8.59 on the scattering of a wave packet from a barrier) as well as on topics that I have not encountered in other computational-physics textbooks (e.g., problem 6.67 on credible intervals for a relativistic particle's mass or problem 7.63 on the dimensional regularization of loop integrals). Sometimes a given physical theme carries over across chapters, for example: the Roche potential is visualized in problem 1.17, it is then extremized in problem 5.50 to find the Lagrange points, the volume of the Roche lobe is computed via quadrature in problem 7.56, and the Arenstorf orbit is arrived at by solving differential equations in problem 8.46.
  A word on solutions: standard practice is that computational-physics textbook authors either produce no solutions to the problems or provide solutions only to instructors teaching for-credit courses out of the textbook. I have followed the latter route, but I'm also providing (online) a subset of the solutions to all readers, as a self-study resource.
- **Topic sequence for different courses:** like many textbooks, this one contains more material than can be covered in a single semester. Here are two sample courses:
  - *Advanced undergraduate course:* sections 1.1–1.5, 2.1–2.4.3, 2.5.2, 3.1–3.3, 1.6, 4.1, 4.2.1–4.2.3, 4.3, 4.4.1, 5.1–5.2, 5.4, 5.5.2, 6.1–6.2.2, 6.5, 7.1–7.3, 7.5, 7.7.1–7.7.4, 8.1–8.3.1, 8.4.1, 8.5. Labs focus on Python programming; lectures mainly address numerical methods; physics content limited to motivation and homework assignments.
  - *Beginning graduate course:* appendix B, sections 2.1–2.5, 3.4, 4.1–4.6, 5.1, 5.3–5.6, 6.1, 6.2.2–6.2.3, 6.4–6.8, 7.1, 7.4–7.8, 8.1–8.4, 8.6. Python and an undergrad numerical course are prerequisites. Increased focus on analytical manipulations; programming limited to homework; lectures' physics content determined by a student poll.

Alas, adding 150 pages of new material for the second edition ran the risk of making this volume too expensive. With that in mind, I abridged sections 4.2 and 4.6, placing the original versions in the online supplement at `www.numphyspy.org`. This book continues to be dear to my heart; I hope the reader gets to share some of my excitement for the subject.

On the Epigraphs

I have translated 14 of the quotes appearing as epigraphs myself; in the remaining instances the original was in English. All 17 quotes are not protected by copyright. The sources are: Dedication: Pindar, *Pythian Odes*, 3.61–62 (~474 BCE), Preface: Ralph Waldo Emerson, *Nature*, Chapter III (1836 CE), Chapter 1: Immanuel Kant, *Lectures VI*, Philosophical Encyclopedia (~1780 CE), Chapter 2: Georg Wilhelm Friedrich Hegel, *The Phenomenology of Spirit*, Paragraph 74 (1807 CE), Chapter 3: Emily Dickinson, *Poem F372/J341* (1862 CE), Chapter 4: Vergil, *Georgics*, Book II, Line 412 (~29 BCE), Chapter 5: Karl

Kraus, *The Last Days of Mankind*, Act I, Scene 22 (~1918 CE), Chapter 6: Gabriel Lippmann, quoted in Henri Poincaré, *Calcul des probabilités*, Second edn, Section 108 (1912 CE), Chapter 7: Thucydides, *History of the Peloponnesian War*, Book IV, Paragraph 40 (~420 BCE), Chapter 8: Sophocles, *Oedipus Tyrannus*, Line 486 (~429 BCE), Postscript: Socrates, quoted in Diogenes Laërtius, *Lives and Opinions of Eminent Philosophers*, Book 2 (~220 CE), Appendix A: Aristotle, *Metaphysics* Book III (B), 1001a1 (~330 BCE), Appendix B: Thomas Aquinas, *Commentary on Aristotle's Metaphysics*, Book IV (Γ), Lesson 1, Chapter 2, Commentary (1270 CE), Appendix C: Parmenides, *Fragment 5* (~475 BCE), (Online) Appendix D: Callimachus, *Fragment 465* (~250 BCE), Bibliography: Michel Eyquem de Montaigne, *Essay III.13*, On Experience (1588 CE), Index: James Joyce, *Ulysses*, Episode 16, Eumaeus (1922 CE).

Acknowledgments

My understanding of numerical methods has benefited tremendously from reading many books and papers. In the bibliography I mention only the works I consulted while writing.

I have learned a lot from my graduate students, my collaborators, as well as members of the wider nuclear physics, cold-atom, and astrophysics communities. This textbook is a pedagogical endeavor but it has unavoidably been influenced by my research, which is supported by the Natural Sciences and Engineering Research Council of Canada and the Canada Foundation for Innovation.

I would like to thank my Editor at Cambridge University Press, Vince Higgs, for his sagacious advice. Margaret Patterson did an excellent job copyediting both editions of the book, while Suresh Kumar helped with advanced LaTeX tricks. I am grateful to the anonymous reviewers for the positive feedback and suggestions for additions.

I wish to acknowledge the students taking the classes I taught; their occasional vacant looks resulted in my adding more explanatory material, whereas their (infrequent, even at 8:30 am) yawns made me shorten some sections. Eric Poisson made wide-ranging comments on the lecture notes that turned into the first edition. Aman Agarwal, Eli Bendersky, Liliana Caballero, Ryan Curry, Victoria Leaker, Benjamin Morling, Tristan Pitre and Sangeet-Pal Pannu spotted issues with individual sections or problems. Buried in this book are conceptual distinctions that attempt to preempt misconceptions which are both natural and widespread; I have enjoyed working with Martin Williams on spin-off journal articles.

"What makes us who we are is our choice of good or bad, not our opinion about it" (Aristotle, *Nicomachean Ethics*, 1112a3); Marcos Gezerlis made sure I approached technical subtleties with a similar outlook. Marcos was instrumental in expanding the new material into its current form, immediately picking up on the lacunae in early drafts; I knew I could stop writing when he reached the desired state of staring at nothing in particular while muttering "I understand now". I am indebted to my family (especially Myrsine and Ariadne), *inter multa alia*, for orienting me toward the subjective universal. While several people have helped me refine this book, any remaining poor choices are my responsibility.

Idiomatic Python 1

> It's not always about speculating; at some point one must think about practice.
>
> Immanuel Kant

This chapter is *not* intended as an introduction to programming in general or to programming with Python. A tutorial on the Python programming language can be found in the online supplement to this book; if you're still learning what variables, loops, and functions are, we recommend you go to our tutorial (see appendix A) before proceeding with the rest of this chapter. You might also want to have a look at the (official) Python Tutorial at `www.python.org`. Reference [41] is a readable book-length introduction to Python (also available online); Ref. [71] is another introduction, with nice material on visualization. Programming, like most other activities, is something you learn by doing. Thus, you should always try out programming-related material as you read it: *there is no royal road to programming*. Even if you have solid programming skills but no familiarity with Python, we recommend you work your way through one of the above resources, to familiarize yourself with the basic syntax. In what follows, we will take it for granted that you have worked through our tutorial and have modified the different examples to carry out further tasks. This includes solving many of the programming problems we pose there.

What this chapter *does* provide is a quick summary of Python features, with an emphasis on those which the reader is more likely not to have encountered in the past. In other words, even if you are already familiar with the Python programming language, you will most likely still benefit from reading this short chapter. Observe that the title at the top of this page is *Idiomatic* Python: this refers to coding in a *Pythonic* manner. The motive is not to proselytize but, rather, to let the reader work with the language (i.e., not against it); we aim to show how to write Python code that feels "natural". If this book was using, say, Julia or Rust instead of Python, we would still be making the same point: one should try to do the best job possible with the tools at one's disposal. As noted in the Preface, the use of idioms allows us to write shorter codes in the rest of the book, thereby emphasizing the numerical method over programming details; this is not merely an aesthetic concern but a question of cognitive consonance.

At a more mundane level, this chapter contains all the Python-related reference material we will need in this volume: reserved words, library functions, tables, and figures. Keeping the present chapter short is intended to help you when you're working through the following chapters and need to quickly look something up. Before summarizing Python features, we make some big-picture comments on the choice of language, as well as on programming in general.

1.1 Why Python?

Since computational physics is a fun subject, it is only appropriate that the programming involved should also be as pleasant as possible. In this book, we use Python 3, a popular, open-source programming language that has been described as "pseudocode that executes". Python is especially nice in that it doesn't require lots of boilerplate code; that, combined with the fact that one can use Python interactively, make it easy to write new programs. This is great from a pedagogical perspective, since it allows a beginner to start using the language without having to first study lengthy volumes. Importantly, Python's syntax is reasonably simple and leads to very readable code. Even so, Python is very expressive, allowing you to do more in a single line than is possible in many other languages. Furthermore, Python is cross-platform, providing a similar experience on Windows and Unix-like systems. Finally, Python comes with "batteries included": its standard library allows you to do a lot of useful work, without having to implement basic/unrelated things (e.g., sorting a list of numbers) yourself.

In addition to the functionality contained in core Python and in the standard library, Python is associated with a wider ecosystem, which includes libraries like Matplotlib, used to visualize data. Another member of the Python ecosystem, especially relevant to us, is the NumPy library (NumPy stands for "Numerical Python"); containing numerical arrays and several related functions, NumPy is one of the main reasons Python is so attractive for computational work. Another fundamental library is SciPy ("Scientific Python"), which provides many routines that carry out tasks like numerical integration and optimization in an efficient manner. A pedagogical choice we have made in this book is to start out with standard Python, use it for a few chapters, and only then turn to the `numpy` library; this is done in order to help students who are new to Python (or to programming in general) effectively distinguish between Python lists and `numpy` arrays. The latter are then used in the context of linear algebra (chapter 4), where they are indispensable, both in terms of expressiveness and in terms of efficiency.

Speaking of which, it's worth noting at the outset that, since our programs are intended to be easy to read, in some cases we have to sacrifice efficiency.[1] Our implementations are intended to be pedagogical, i.e., they are meant to teach you how and why a given numerical method works; thus, we almost never employ NumPy or SciPy functionality (other than `numpy` arrays), but produce our own functions, instead. We make some comments on alternative implementations here and there, but the general assumption is that you will be able to write your own codes using different approaches (or programming languages) once you've understood the underlying numerical method. If all you are interested in is a quick calculation, then Python along with its ecosystem is likely going to be your one-stop shop. As your work becomes more computationally challenging, you may need to switch to a compiled language; most work on supercomputers is carried out using languages like Fortran or C++ (or sometimes even C). Of course, even if you need to produce a hyperefficient code for your research, the insight you may gain from building a prototype in Python could

[1] Thus, we do not talk about things like Python's Global Interpreter Lock, cache misses, page faults, and so on.

be invaluable; similarly, you could write most of your code in Python and re-express a few performance-critical components using a compiled language. We hope that the lessons you pick up here (both on the numerical methods and on programming in general) will serve you well if you need to employ another environment in the future.

The decision to focus on Python (and NumPy) idioms is coupled to the aforementioned points on Python's expressiveness and readability: idiomatic code makes it easier to conquer the complexity that arises when developing software. (Of course, it does require you to first become comfortable with the idioms.) That being said, our presentation will be *selective*; Python has many other features that we will not go into. Most notably, we don't discuss how to define classes of your own or how to handle exceptions; the list of omitted features is actually very long.[2] While many features we leave out are very important, discussing them would interfere with the learning process for students who are still mastering the basics of programming. Even so, we do introduce topics that haven't often made it into computational-science texts (e.g., list comprehensions, dictionaries, for-else, array manipulation via slicing and `@`) and use them repeatedly in the rest of the book.

We sometimes point to further functionality in Python. For more, have a look at the bibliography and at The Python Language Reference (as well as The Python Standard Library Reference). Once you've mastered the basics of core Python, you may find books like Ref. [120] and Ref. [132] a worthwhile investment. On the wider theme of developing good programming skills, volumes like Ref. [103] can be enriching, as is also true of any book written by Brian Kernighan. Here we provide only the briefest of summaries.

1.2 Code Quality

We will not be too strict in this book about coding guidelines. Issues like code layout can be important, but most of the programs we will write are so short that this won't matter too much. If you'd like to learn more about this topic, your first point of reference should be PEP 8 – Style Guide for Python Code. Often more important than issues of code layout[3] are questions about how you write and check your programs. Here is some general advice:

- **Code readability matters** Make sure to target your program to humans, not the computer. This means that you should avoid using "clever" tricks. Thus, you should use good variable names and write comments that add value (instead of repeating the code). The human code reader that will benefit from this is first and foremost yourself, when you come back to your programs some months later.
- **Be careful, not swift, when coding** Debugging is typically more difficult than coding itself. Instead of spending two minutes writing a program that doesn't work and then requires you to spend two hours fixing it up, try to spend 10 minutes on designing the code and then carefully converting your ideas into program lines. It doesn't hurt to also use Python interactively (while building the program file) to test out components of the code one-by-one or to fuse different parts together.

[2] For example: decorators, coroutines, or type hints.

[3] Discussion of which, more often than not, sheds light on the narcissism of minor differences ("bikeshedding").

- **Untested code is wrong code** Make sure your program is working correctly. If you have an example where you already know the answer, make sure your code gives that answer. Manually step through a number of cases (i.e., mentally, or on paper, do the calculations the program is supposed to carry out). This, combined with judiciously placed print-outs of intermediate variables, can go a long way toward ensuring that everything is as it should be. When modifying your program, ensure it still gives the original answer when you specialize the problem to the one you started with.
- **Write functions that do one thing well** Instead of carrying out a bunch of unrelated operations in sequence, you should structure your code so that it makes use of well-named (and well-thought-out) functions that do one thing and do it well. You should break down the tasks to be carried out and logically separate those into distinct functions. If you design these well, in the future you will be able to modify your programs to carry out much more challenging tasks, by only adding a few lines of new code (instead of having to change dozens of lines in an existing "spaghetti" code).
- **Use trusted libraries** In most of this book we are "reinventing the wheel", because we want to understand how things work (or don't work). Later in life, you should not have to always use "hand-made" versions of standard algorithms. As mentioned, there exist good (widely employed and tested) libraries like `numpy` that you should learn to make use of. The same thing holds, obviously, for the standard Python library: you should generally employ its features instead of "rolling your own".

One could add (much) more advice along these lines. Since our scope here is much more limited, we conclude by pointing out that in the Python ecosystem (or around it) there's extensive infrastructure [128] to carry out version control (e.g., `git`), testing (e.g., `doctest` and `unittest`), as well as debugging (e.g., `pdb`), program profiling and optimization, among other things. You should also have a look at the `pylint` tool.

1.3 Summary of Python Features

1.3.1 Basics

Python can be used interactively: this is when you see the Python prompt `>>>`, also known as a chevron. You don't need to use Python interactively: like other programming languages, the most common way of writing and running programs is to store the code in a file. Linear combinations of these two ways of using Python are also available, fusing interactive sessions and program files. In any case, your program is always executed by the Python interpreter. Appendix A points you in the direction of tools you could employ.

Like other languages (e.g., C or Fortran), Python employs variables, which can be integers, complex numbers, etc. Unlike those languages, Python is a dynamically typed language, so variables get their type from their value, e.g., `x = 0.5` creates a floating-point variable (a "float"). It may help you to think of Python values as being produced first and labels being attached to them after that. Numbers like `0.5` or strings like `"Hello"`, are

known as *literals*. If you wish to print the value of a variable, you use the `print()` built-in function, i.e., `print(x)`. Further functionality is available in the form of standard-library modules, e.g., you can `import` the `sqrt` function that is to be found in the `math` module. Users can define their own modules: we will do so repeatedly. You can carry out arithmetic with variables, e.g., `x**y` raises `x` to the `y`-th power or `x//y` does "floor division". It's usually a good idea to group related operations using parentheses. Python also supports augmented assignment, e.g., `x += 1` or even multiple assignment, e.g., `x, y = 0.5, "Hello"`. This gives rise to a nifty way to swap two variables: `x, y = y, x`.

Comments are an important feature of programming languages: they are text that is ignored by the computer but can be very helpful to humans reading the code. That human may be yourself in a few months, at which point you may have forgotten the purpose or details of the code you're inspecting. Python allows you to write both single-line comments, via `#`, or docstrings (short for "documentation strings"), via the use of triple quotation marks. Crucially, we don't include explanatory comments in our code examples, since this is a book which explicitly discusses programming features in the main text. That being said, in your own codes (which are not embedded in a book discussing them) you should always include comments.

1.3.2 Control Flow

Control flow refers to programming constructs where not every line of code gets executed in order. A classic example is conditional execution via the `if` statement:

```
>>> if x!=0:
...       print("x is non-zero")
```

Indentation is important in Python: the line after `if` is indented, reflecting the fact that it belongs to the corresponding scenario. Similarly, the colon, `:`, at the end of the line containing the `if` is also syntactically important. If you wanted to take care of other possibilities, you could use another indented block starting with `else:` or `elif x==0:`. In the case of boolean variables, a common idiom is to write: `if flag:` instead of `if flag==True:`.

Another concept in control flow is the loop, i.e., the repetition of a code block. You can do this via `while`, which is typically used when you don't know ahead of time how many iterations you are going to need, e.g., `while x>0:`. Like conditional expressions, a `while` loop tests a condition; it then keeps repeating the body of the loop until the condition is no longer true, in which case the body of the block is jumped over and execution resumes from the following (non-indented) line. We sometimes like to be able to break out of a loop: if a condition in the middle of the loop body is met, then: (a) if we use `break` we will proceed to the first statement *after* the loop, or (b) if we use `continue` we skip not the entire loop, but the rest of the loop body *for the present iteration*.

A third control-flow construct is a `for` loop: this arises when you need to repeat a certain action a fixed number of times. For example, by saying `for i in range(3):` you will repeat whatever follows (and is indented) three times. Like C, *Python uses 0-based indexing* (which we will shorten to "0-indexing"), meaning that the indices go as 0, 1, 2 in this

case. In general, `range(n)` gives the integers from `0` to `n-1` and, similarly, `range(m, n, i)` gives the integers from `m` to `n-1` in steps of `i`. Above, we mentioned how to use `print()` to produce output; this can be placed inside a loop to print out many numbers, each on a separate line; if you want to place all the output on the same line you do:

```
>>> for i in range(1,15,2):
...       print(0.01*i, end=" ")
```

that is, we've said `end=" "` after passing in the argument we wish to print. As we'll discuss in the following subsection, Python's `for` loop is incredibly versatile.

1.3.3 Data Structures

Python supports container entities, called data structures; we will mainly be using lists.

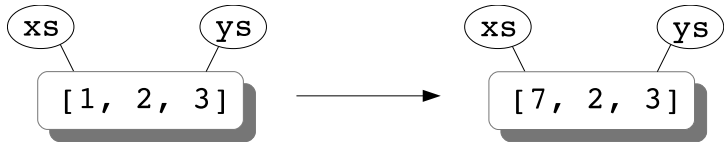
Lists A list is a container of elements; it can can grow when you need it to. Elements can have different types. You use square brackets and comma-separated elements when creating a list, e.g., `zs = [5, 1+2j, -2.0]`. You also use square brackets when indexing into a list, e.g., `zs[0]` is the first element and `zs[-1]` the last one. Lists are mutable sequences, meaning we can change their elements, e.g., `zs[1] = 9`, or introduce new elements, via `append()`. The combination of `for` loops and `append()` provides us with a powerful way to populate a list. For example:

```
>>> xs = []
>>> for i in range(20):
...       xs.append(0.1*i)
```

where we started with an empty list. In the following section, we'll see a more idiomatic way of accomplishing the same task. You can concatenate two lists via the addition operator, e.g., `zs = xs + ys`; the logical consequence of this is the idiom whereby a list can be populated with several (identical) elements using a one-liner, `xs = 10*[0]`. There are several built-in functions (applicable to lists) that often come in handy, most notably `sum()` and `len()`.

Python supports a feature called slicing, which allows us to take a slice out of an existing list. Slicing, like indexing, uses square brackets: the difference is that slicing uses two integers, with a colon in between, e.g., `ws[2:5]` gives you the elements `ws[2]` up to (but not including) the element `ws[5]`. Slicing obeys convenient defaults, in that we can omit one of the integers in `ws[m:n]` without adverse consequences. Omitting the first index is interpreted as using a first index of 0, and omitting the second index is interpreted as using a second index equal to the number of elements. You can also include a third index: in `ws[m:n:i]` we go in steps of `i`. Note that list slicing uses colons, whereas the arguments of `range()` are comma-separated. Except for that, the pattern of `start`, `end`, `stride` is the same.

We are now in a position to discuss how copying works. In Python a new list, which is



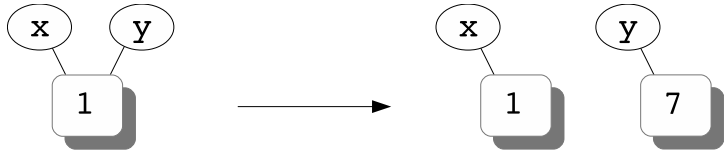
Labelling and modifying a mutable object (in this case, a list) Fig. 1.1

assigned to be equal to an old list, is simply the old list by another name. This is illustrated in Fig. 1.1, which corresponds to the three steps `xs = [1,2,3]`, followed by `ys = xs`, and then `ys[0] = 7`. In other words, in Python we're not really dealing with variables, but with *labels* attached to values, since `xs` and `ys` are just different names for the same entity. When we type `ys[0] = 7` we are not creating a new value, simply modifying the underlying entity that both the `xs` and `ys` labels are attached to. Incidentally, things are different for simpler variables, e.g., `x=1; y=x; y=7; print(x)` prints `1` since `7` is a new value, not a modification of the value `x` is attached to. This is illustrated in Fig. 1.2, where we see that, while initially both variable names were labelling the same value, when we type `y=7` we create a new value (since the number 7 is a new entity, not a modification of the number 1) and then attach the `y` label to it.

Crucially, *when you slice you get a new list*, meaning that if you give a new name to a slice of a list and then modify that, then the original list is unaffected. For example, `xs = [1,2,3]`, followed by `ys = xs[1:]`, and then `ys[0] = 7` does not affect `xs`. This fact (namely, that slices don't provide views on the original list but can be manipulated separately) can be combined with another nice feature (namely, that when slicing one can actually omit both indices) to create a copy of the entire list, e.g., `ys = xs[:]`. This is a shallow copy, so if you need a deep copy, you should use the function `deepcopy()` from the standard module `copy`; the difference is immaterial here.

Tuples Tuples can be (somewhat unfairly) described as immutable lists. They are sequences that can neither change nor grow. They are defined using parentheses instead of square brackets, e.g., `xs = (1,2,3)`, but you can even omit the parentheses, `xs = 1,2,3`. Tuple elements are accessed the same way that list elements are, namely with square brackets, e.g., `xs[2]`.

Strings Strings can also be viewed as sequences, e.g., if `name = "Mary"` then `name[-1]` is the character `'y'`. Note that you can use either single or double quotation marks. Like tuples, strings are immutable. As with tuples, we can use + to concatenate two strings. A



Labelling immutable objects (in this case, integers) Fig. 1.2

useful function that acts on strings is `format()`: it uses *positional* arguments, numbered starting from 0, within curly braces. For example:

```
>>> x, y = 3.1, -2.5
>>> "{0} {1}".format(x, y)
'3.1 -2.5'
```

The overall structure is string-dot-format-arguments. This can lead to powerful ways of formatting strings, e.g.,

```
>>> "{0:1.15f} {1}".format(x, y)
'3.100000000000000 -2.5'
```

Here we also introduced a colon, this time followed by `1.15f`, where `1` gives the number of digits before the decimal point, `15` gives the number of digits after the decimal point, and `f` is a type specifier (that leads to the result shown for floats).

Dictionaries Python also supports dictionaries, which are called associative arrays in computer science (they're called maps in C++). You can think of dictionaries as being similar to lists or tuples, but instead of being limited to integer indices, with a dictionary you can use strings or floats as *keys*. In other words, dictionaries contain key and value pairs. The syntax for creating them involves curly braces (compare with square brackets for lists and parentheses for tuples), with the key-value pair being separated by a colon. For example, `htow = {1.41: 31.3, 1.45: 36.7, 1.48: 42.4}` is a dictionary associating heights to weights. In this case both the keys and the values are floats. We access a dictionary value (for a specific key) by using the name of the dictionary, square brackets, and the key we're interested in: this returns the value associated with that key, e.g., `htow[1.45]`. In other words, indexing uses square brackets for lists, tuples, strings, and dictionaries. If the specific key is not present, then we get an error. Note, however, that accessing a key that is not present *and then assigning* actually works: this is a standard way key:value pairs are introduced into a dictionary, e.g., `htow[1.43] = 32.9`.

1.3.4 User-Defined Functions

If our programs simply carried out a bunch of operations in sequence, inside several loops, their logic would soon become unwieldy. Instead, we are able to group together logically related operations and create what are called user-defined functions: just as in our earlier section on control flow, this refers to lines of code that are not necessarily executed in the order in which they appear inside the program file. For example, while the `math` module contains a function called `exp()`, we could create our own function called, say, `compexp()` as in section 2.4.4, which, e.g., uses a different algorithm to get to the answer. The way we introduce our own functions is via the `def` keyword, along with a function name and a colon at the end of the line, as well as the (by now expected) indentation of the code block that follows. Here's a function that computes the sum from 1 up to some integer:

```
>>> def sumofints(nmax):
...       val = sum(range(1,nmax+1))
...       return val
```

We are taking in the integer up to which we're summing as a parameter. We then ensure that `range()` goes up to (but not including) `nmax+1` (i.e., it includes `nmax`). We split the body of the function into two lines: first we define a new variable and then we *return* it, though we could have simply used a single line, `return sum(range(1,nmax+1))`. This function can be called (in the rest of the program) by saying `x = sumofints(42)`.

The function we just defined took in one parameter and returned one value. It could have, instead, taken in no parameters, e.g., summing the integers up to some constant; we would then call it by `x = sumofints()`. Similarly, it could have printed out the result, inside the function body, instead of returning it to the external world; in that case, where no `return` statement was used, the `x` in `x = sumofints(42)` would have the value `None`. Analogously, we could be dealing with several input parameters, or several return values, expressed by `def sumofints(nmin,nmax):`, or `return val1, val2`, respectively. The latter case is implicitly making use of a tuple.

We say that a variable that's either a parameter of a function or is defined inside the function is *local* to that function. If you're familiar with the terminology other languages use (pass-by-value or pass-by-reference), then note that Python employs *pass-by-assignment*, which for immutable objects behaves like pass-by-value (you *can't* change what's outside) and for mutable objects behaves like pass-by-reference (you *can* change what's outside), if you're not re-assigning. It's often a bad idea to change the external world from inside a function: it's best simply to return a value that contains what you need to communicate to the external world. This can become wasteful, but here we opt for conceptual clarity, always returning values without changing the external world. This is a style inspired by *functional programming*, which aims at avoiding *side effects*, i.e., changes that are not visible in the return value. (Unless you're a purist, input/output is fine.) Python also supports *nested functions* and *closures*. On a related note, Python contains the keywords `global` and `nonlocal` as well as function one-liners via `lambda`; some of these features are briefly touched upon in problem 1.4.

A related feature of Python is the ability to provide default parameter values:

```
>>> def cosder(x, h=0.01):
...       return (cos(x+h) - cos(x))/h
```

You can call this function with either `cosder(0.)` or `cosder(0., 0.001)`; in the former case, `h` has the value 0.01. Basically, the second argument here is *optional*. As a matter of good practice, you should make sure to always use immutable default parameter values. Finally, note that in Python one has the ability to define a function that deals with an indefinite number of positional or keyword arguments. The syntax for this is `*args` and `**kwargs`, but a detailed discussion would take us too far afield.

A pleasant feature of Python is that *functions are first-class objects*. As a result, we

can pass them in as arguments to other functions; for example, instead of hard-coding the cosine as in our previous function, we could say:

```
>>> def der(f, x, h=0.01):
...       return (f(x+h) - f(x))/h
```

which is called by passing in as the first argument the function of your choice, e.g., `der(sin, 0., 0.05)`. Note how `f` is a regular parameter, but is used inside the function the same way we use functions (by passing arguments to them inside parentheses). We passed in the name of the function, `sin`, as the first argument and the `x` as the second argument.[4] As a rule of thumb, you should pass a function in as an argument if you foresee that you might be passing in another function in its place in the future. If you basically expect to always keep carrying out the same task, there's no need to add yet another parameter to your function definition. Incidentally, we really meant it when we said that in Python functions are first-class objects. You could even have a list whose elements are functions, e.g., `funcs = [sumofints, cos]`. Similarly, problem 1.2 explores a dictionary that contains functions as values (or keys).

1.4 Core-Python Idioms

We are now in a position to discuss Pythonic idioms: these are syntactic features that allow us to perform tasks more straightforwardly than would have been possible with the syntax introduced above. Using such alternative syntax to make the code more concise and expressive helps us write new programs, but also makes the lives of future readers easier. Of course, you do have to exercise your judgement.[5]

1.4.1 List Comprehensions

At the start of section 1.3.3, we saw how to populate a list: start with an empty one and use `append()` inside a `for` loop to add the elements you need. List comprehensions (often shortened to *listcomps*) provide us with another way of setting up lists. The earlier example can be replaced by `xs = [0.1*i for i in range(20)]`. This is much more compact (one line vs three). Note that when using a list comprehension the loop that steps through the elements of some other sequence (in this case, the result of stepping through `range()`) is placed *inside* the list we are creating! This particular syntax is at first sight a bit unusual, but very convenient and strongly recommended.

It's a worthwhile exercise to replace hand-rolled versions of code using listcomps. For example, if you need a new list whose elements are two times the value of each element in `xs`, you should *not* say `ys = 2*xs`: as mentioned earlier, this concatenates the two lists, which is not what we are after. Instead, what *does* work is `ys = [2*x for x in xs]`. More

[4] This means that we did *not* pass in `sin()` or `sin(x)`, as those wouldn't work.

[5] "A foolish consistency is the hobgoblin of little minds" (Ralph Waldo Emerson, *Self-Reliance*).

generally, if you need to apply a function to every element of a list, you could simply do so on the fly: `ws = [f(x) for x in xs]`. Another powerful feature lets you "prune" a list as you're creating it, e.g., `zs = [2*x for x in xs if x>0.3]`; this doubles an element only if that element is greater than 0.3 (otherwise it doesn't even introduce it).

1.4.2 Iterating Idiomatically

Our earlier example, `ys = [2*x for x in xs]`, is an instance of a significant syntactic feature: a `for` loop is not limited to iterating through a collection of integers in fixed steps (as in our earlier `for i in range(20)`) but can iterate through the list elements themselves *directly*.[6] This is a general aspect of iterating in Python, a topic we now turn to; the following advice applies to all loops (i.e., not only to listcomps).

One list Assuming the list `xs` already exists, you may be tempted to iterate through its elements via something like `for i in range(len(xs)):` then in the loop body you would get a specific element by indexing, i.e., `xs[i]`. The Pythonic alternative is to step through the elements themselves, i.e., `for x in xs:` and then simply use `x` in the loop body. Instead of iterating through an index, which is what the error-prone syntax `range(len(xs))` is doing, this uses Python's `in` to iterate directly through the list elements.

Sometimes, you need to iterate through the elements of a list `xs` in reverse: the old-school (C-influenced) way to do this is `for i in range(len(xs)-1, -1, -1):`, followed by indexing, i.e., `xs[i]`. This works, but all those `-1`'s don't make for light reading; instead, use Python's built-in `reversed()` function, saying `for x in reversed(xs):` and then using `x` directly. A final use case: you often need access to both the index showing an element's place in the list, and the element itself. The "traditional" solution would involve `for i in range(len(xs)):` followed by using `i` and `xs[i]` in the loop body. The Pythonic alternative is to use the built-in `enumerate()` function, `for i, x in enumerate(xs):` and then use `i` and `x` directly; this is more readable and less error-prone.

Two lists We sometimes want to iterate through two lists, `xs` and `ys`, in parallel. You should be getting the hang of things by now; the unPythonic way to do this would be `for i in range(len(xs)):`, followed by using `xs[i]` and `ys[i]` in the loop body. The Pythonic solution is to say `for x, y in zip(xs,ys):` and then use `x` and `y` directly. The `zip()` built-in function creates an iterable entity consisting of tuples, fusing the 0th element in `xs` with the 0th element in `ys`, the 1st element in `xs` with the 1st element[7] in `ys`, and so on.

Dictionaries We can use `for` to iterate Pythonically through more than just lists; in the tutorial you have learned that it also works for lines in a file. Similarly, we can iterate through the *keys* of a dictionary; for our earlier height-to-weight dictionary, you could say `for h in htow:` and then use `h` and `htow[h]` in the loop body. An even more Pythonic way of doing the same thing uses the `items()` method of dictionaries to produce all the key-value pairs: `for h,w in htow.items():` lets you use `h` and `w` inside the loop body.

[6] In other words, Python's `for` is similar to the `foreach` that some other languages have.
[7] In English we say "first", "second", etc. We'll use numbers, e.g., 0th, when using the 0-indexing convention.

Code 1.1 forelse.py

```
def look(target,names):
      for name in names:
            if name==target:
                  val = name
                  break
      else:
            val = None
      return val

names = ["Alice", "Bob", "Eve"]
print(look("Eve", names))
print(look("Jack", names))
```

For-else In this section we've spent some time discussing the line where `for` appears. We now turn to a way of using `for` loops that is different altogether: similarly to `if` statements, you can follow a `for` loop by an `else` (!). This is somewhat counterintuitive, but can be very helpful. The way this works is that the `for` loop is run as usual: if no `break` is encountered during execution of the `for` block, then control proceeds to the `else` block. If a `break` is encountered during execution of the `for` block, then the `else` block is not run. (Try re-reading the last two sentences after you study code 1.1.)

The `else` in a `for` loop is nice when we are looking for an item within a sequence and we need to do one thing if we find it and a different thing if we don't. An example is given in code 1.1, the first full program in the book. We list such codes in boxes with the filename at the top; we strongly recommend you download these Python codes and use them in parallel to reading the main text;[8] you would run this by typing `python forelse.py` or something along those lines; note that, unlike other languages you may have used in the past, there is no *compilation* stage. This specific code uses several of the Python features mentioned in this chapter: it starts with defining a function, `look()`, which we will discuss in more detail below. The main program uses a listcomp to produce a list of strings and then calls the `look()` function twice, passing in a different first argument each time; we don't even need a variable to hold the first argument(s), using string literal(s) directly. Since we are no longer using Python interactively, we employ `print()` to ensure that the output is printed on the screen (instead of being evaluated and then thrown away).

Turning to the `look()` function itself, it takes in two parameters: one is (supposed to be) a target string and the other one is a list of strings. The latter could be very long, e.g., the first names listed in a phone book. Note that there are three levels of indentation involved here: the function body gets indented, the `for` loop body gets indented, and then the conditional

[8] "One cannot so well grasp a thing and make it one's own, when it has been learned from someone else, as when one has discovered it oneself" (René Descartes, *Discourse on the Method*, part VI).

expression body also gets indented. Incidentally, our `for` loop is iterating through the list elements directly, as per our earlier admonition, instead of using error-prone indices. You should spend some time ensuring that you understand what's going on. Crucially, the `else` is indented at the level of the `for`, not at the level of the `if`. We also took the opportunity to employ another idiom mentioned above: `None` is used to denote the absence of a value. The two possibilities that are at play (target in sequence or target not in sequence) are probed by the two function calls in the main program. When the target is found, a `break` is executed, so the `else` is not run. When the target is not found, the `else` is run. It might help you to think of this `else` as being equivalent to `nobreak`: the code in that block is only executed when no `break` is encountered in the main part of the `for` loop. (Of course, this is only a mnemonic, since `nobreak` is not a reserved word in Python.) We will use the for-else idiom repeatedly in this volume (especially in chapter 5), whenever we are faced with an iterative task which may plausibly fail, in which case we wish to communicate that fact to the rest of the program. Since even some expert users are uncomfortable with the for-else idiom, problem 1.5 gives you a tour of the alternatives.

1.5 Basic Plotting with `matplotlib`

We will now visualize relationships between numbers via `matplotlib`, a plotting library (i.e., not part of core Python) which can produce quality figures: all the plots in this book were created using `matplotlib`. Inside the `matplotlib` package is the `matplotlib.pyplot` module, which is used to produce figures in a MATLAB-like environment.

A simple example is given in code 1.2. This starts by importing `matplotlib.pyplot` in the (standard) way which allows us to use it below without repeated typing of unnecessary characters. We then define a function, `plotex()`, that takes care of the plotting, whereas the main program simply introduces four list comprehensions and then calls our function. The listcomps also employ idiomatic iteration, in the spirit of applying what you learned in the previous section. If you're still a beginner, you may be wondering why we defined a Python function in this code. An important design principle in computer science goes by the name of *separation of concerns* (or sometimes *information hiding* or *encapsulation*): each aspect of the program should be handled separately. In our case, this means that each component of our task should be handled in a separate function.

Let's discuss this function in more detail. Its parameters are (meant to be) four lists, namely two pairs of x_i and y_i values. The function body starts by using `xlabel()` and `ylabel()` to provide labels for the x and y axes. It then creates individual curves/sets of points by using `matplotlib`'s function `plot()`, passing in the x-axis values as the first argument and the y-axis values as the second argument. The third positional argument to `plot()` is the *format string*: this corresponds to the color and point/line type. In the first case, we used `r` for red and `-` for a solid line. Of course, figures in this book are in black and white, but you can produce the color version using the corresponding Python code. In order to help you interpret this and other format strings, we list allowed colors and some

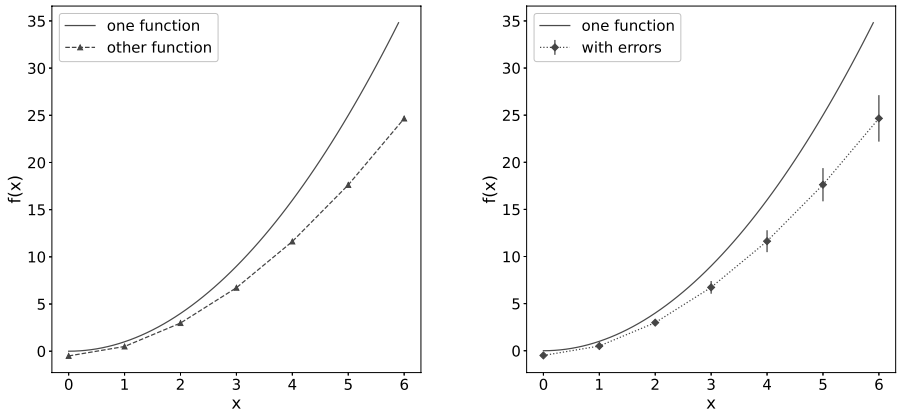
Code 1.2 plotex.py

```
import matplotlib.pyplot as plt

def plotex(cxs,cys,dxs,dys):
      plt.xlabel('x', fontsize=20)
      plt.ylabel('f(x)', fontsize=20)
      plt.plot(cxs, cys, 'r-', label='one function')
      plt.plot(dxs, dys, 'b--^', label='other function')
      plt.legend()
      plt.show()

cxs = [0.1*i for i in range(60)]
cys = [x**2 for x in cxs]
dxs = [i for i in range(7)]
dys = [x**1.8 - 0.5 for x in dxs]

plotex(cxs, cys, dxs, dys)
```

of the most important line styles/markers in table 1.1. The fourth argument to `plot()` is a keyword argument containing the label corresponding to the curve. In the second call to `plot()` we pass in a different format string and label (and, obviously, different lists); observe that we used two style options in the format string: `--` to denote a dashed line and `^` to denote the points with a triangle marker. The function concludes by calling `legend()`, which is responsible for making the legend appear, and `show()`, which makes the plot actually appear on our screen.

Table 1.1 Color, line styles, and markers in `matplotlib`

Character	Color	Character	Description
`'b'`	blue	`'-'`	solid line style
`'g'`	green	`'--'`	dashed line style
`'r'`	red	`'-.'`	dash-dot line style
`'c'`	cyan	`':'`	dotted line style
`'m'`	magenta	`'o'`	circle marker
`'y'`	yellow	`'s'`	square marker
`'k'`	black	`'D'`	diamond marker
`'w'`	white	`'^'`	triangle-up marker



Examples of figures produced using `matplotlib` **Fig. 1.3**

The result of running this program is in the left panel of Fig. 1.3. A scenario that pops up very often in practice involves plotting points with error bars:

```
dyerrs = [0.1*y for y in dys]
plt.errorbar(dxs, dys, dyerrs, fmt='b:D', label='with errors')
```

where we have called `errorbar()` to plot the points with error bars: the three positional arguments here are the x values, the y values, and the errors in the y values. After that, we pass in the format string using the keyword argument `fmt` and the label as usual. We thereby produce the right panel of Fig. 1.3.

We could fine-tune almost all aspects of our plots, including basic things like line width, font size, and so on. For example, we could get TeX-like equations by putting dollar signs inside our string, e.g., 'x_i' appears as x_i. We could control which values are displayed via `xlim()` and `ylim()`, we could employ a log-scale for one or both of the axes (using `xscale()` or `yscale()`), and much more. The online documentation can help you go beyond these basic features. Finally, we note that instead of providing `matplotlib` with Python lists as input, you could be using NumPy arrays; this is the topic we now turn to.

1.6 NumPy Idioms

NumPy arrays are not used in chapters 2 and 3 so, if you are still new to Python, you should focus on mastering Python lists: how to grow them, modify them, etc. Thus, you should *skip this section and the corresponding NumPy tutorial for now* and come back when you are about to start reading chapter 4. If you're feeling brave you can keep reading, but know that it is important to distinguish between Python lists and NumPy arrays.

In our list-based codes, we typically carry out the same operation over and over again, a fixed number of times, e.g., `contribs = [w*f(x) for w,x in zip(ws,xs)]`; this list com-

Table 1.2 Commonly used numpy data types

Type	Variants
Integer	`int8`, `int16`, `int32`, `int64`
Float	`float16`, `float32`, `float64`, `longdouble`
Complex	`complex64`, `complex128`, `complex256`

prehension uses the `zip()` built-in to step through two lists in parallel. Similarly, our Python lists are almost always "homogeneous", in the sense that they contain elements of only one type. This raises the natural question: wouldn't it make more sense to carry out such tasks using a homogeneous, fixed-length container? This is precisely what the Numerical Python (NumPy) array object does: as a result, it is fast and space-efficient. It also allows us to avoid, for the most part, having to write loops: even in our listcomps, which are more concise than standard loops, there is a need to explicitly step through each element one by one. Numerical Python arrays often obviate such syntax, letting us carry out mathematical operations on entire blocks of data in one step.[9] The standard convention is to import `numpy` with a shortened name: `import numpy as np`.

One-dimensional arrays One-dimensional (1d) arrays are direct replacements for lists. The easiest way to make an array is via the `array()` function, which takes in a sequence and returns an array containing the data that was passed in, e.g., `ys = np.array(contribs)`; the `array()` function is part of `numpy`, so we had to say `np.array()` to access it. There is both an `array()` function and an array object involved here: the former created the latter. Printing out an array, the commas are stripped, so you can focus on the numbers. If you want to see how many elements are in the array `ys`, use the `size` attribute, via `ys.size`. Remember: NumPy arrays are fixed-length, so the total number of elements cannot change. Another very useful attribute arrays have is `dtype`, namely the data type of the elements; table 1.2 collects several important data types. When creating an array, the data type can also be explicitly provided, e.g., `zs = np.array([5, 8], dtype=np.float32)`.

NumPy contains several handy functions that help you produce pre-populated arrays, e.g., `np.zeros(5)`, `np.ones(4)`, or `np.arange(6)`. The latter also works with float arguments, however, as we will learn in the following chapter, this invites trouble. For example, `np.arange(1.5,1.75,0.05)` and `np.arange(1.5,1.8,0.05)` behave quite differently. Instead, use the `linspace()` function to get a specified number of evenly spaced elements over a specified interval, e.g., `np.linspace(1.5,1.75,6)` or `np.linspace(1.5,1.8,7)`. There also exists a function called `logspace()`, which produces a logarithmically spaced grid of points.

Indexing for arrays works as for lists, namely with square brackets, e.g., `zs[2]`. Slicing appears, at first sight, to also be identical to how slicing of lists works. However, there is a crucial difference, namely that *array slices are views on the original array*; let's revisit our

[9] From here onward, we will not keep referring to these new containers as `numpy` arrays: they'll simply be called arrays, the same way the core-Python lists are simply called lists.

earlier example. For arrays, `xs = np.array([1,2,3])`, followed by `ys = xs[1:]`, and then `ys[0] = 7` *does* affect `xs`. NumPy arrays are efficient, eliminating the need to copy data: of course, one should always be mindful that different slices all refer to the same underlying array. For the few cases where you do need a true copy, you can use `np.copy()`, e.g., `ys = np.copy(xs[1:])` followed by `ys[0] = 7` does not affect `xs`. We could, just as well, copy the entire array over, without impacting the original. In what follows, we frequently make copies of arrays inside functions, in the spirit of impacting the external world only through the return value of a function.

Another difference between lists and arrays has to with *broadcasting*. Qualitatively, this means that NumPy often knows how to handle entities whose dimensionalities don't quite match. For example, `xs = np.zeros(5)` followed by `xs[:] = 7`, leads to an array of five sevens. Remember, since array slices are views on the original array, `xs[:]` cannot be used to create a copy of the array, but it *can* be used to broadcast one number onto many slots; this syntax is known as an "everything slice". Without it, `xs = 7` leads to `xs` becoming an integer (number) variable, not an array, which isn't what we want.

The real strength of arrays is that they let you carry out operations such as `xs + ys`: if these were lists, you would be concatenating them, but for arrays addition is interpreted as an elementwise operation (i.e., each pair of elements is added together).[10] If you wanted to do the same thing with lists you would need a listcomp and `zip()`. This ability to carry out such batch operations on all the elements of an array (or two) at once is often called *vectorization*. You could also use other operations, e.g., to sum the results of pairwise multiplication you simply say `np.sum(xs*ys)`. This is simply evaluating the scalar product of two vectors, in one short expression. Equally interesting is the ability NumPy has to also combine an array with a scalar: this follows from the aforementioned *broadcasting*, whereby NumPy knows how to interpret entities with different (but compatible) shapes. For example, `2*xs` doubles each array element; this is very different from what we saw in the case of lists. You can think of what's happening here as the value `2` being "stretched" into an array with the same size as `xs` and then an elementwise multiplication taking place. Needless to say, you could carry out several other operations with scalars, e.g., `1/xs`. Such combinations of broadcasting (whereby you can carry out operations between scalars and arrays) and vectorization (whereby you write one expression but the calculation is carried out for all elements) can be hard to grasp at first, but are very powerful (both expressive and efficient) once you get used to them.

Another very useful function, `np.where()`, helps you find specific indices where a condition is met, e.g., `np.where(2 == xs)` returns a tuple of arrays, so we would be interested in its 0th element. NumPy also contains several functions that look similar to corresponding `math` functions, e.g., `np.sqrt()`; these are are designed to take in entire arrays so they are almost always faster. NumPy also has functions that take in an array and return a scalar, like the `np.sum()` we encountered above. Another very helpful function is `np.argmax()`, which returns the index of the maximum value. You can also use NumPy's functionality to create your own functions that can handle arrays. For example, our earlier `contribs = [w*f(x) for w,x in zip(ws,xs)]` can be condensed into the much cleaner `contribs =`

[10] Conversely, if you wish to concatenate two arrays you cannot use addition; use `np.concatenate()`.

Table 1.3 Important attributes of numpy arrays

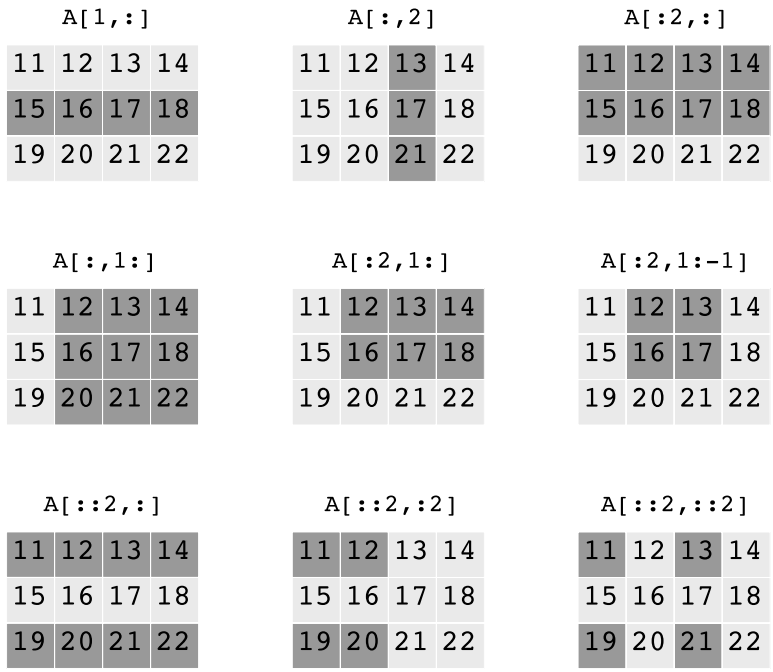
Attribute	Description
`dtype`	Data type of array elements
`ndim`	Number of dimensions of array
`shape`	Tuple with number of elements for each dimension
`size`	Total number of elements in array

`ws*fa(xs)`, where `fa()` is designed to take in arrays (so, e.g., it uses `np.sqrt()` instead of `math.sqrt()`).

Two-dimensional arrays In core Python, matrices can be represented by lists of lists which are, however, quite clunky (as you'll further experience in the following section). In Python a list-of-lists is introduced by, e.g., `LL = [[11,12], [13,14], [15,16]]`. Just like for one-dimensional arrays, we can say `A = np.array(LL)` to produce a two-dimensional (2d) array that contains the elements in `LL`. If you type `print(A)` the Python interpreter knows how to strip the commas and split the output over three rows, making it easy to see that it's a two-dimensional entity, similar to a mathematical matrix.

Much of what we saw on creating one-dimensional arrays carries over to the two-dimensional case. For example, `A.size` is 6: this is the total number of elements, including both rows and columns. Another attribute is `ndim`, which tells us the dimensionality of the array: `A.ndim` is 2 for our example. The number of dimensions can be thought of as the number of distinct "axes" according to which we are listing elements. Incidentally, our terminology may be confusing to those coming from a linear-algebra background. In linear algebra, we say that a matrix $\mathbf{A}$ with m rows and n columns has "dimensions" m and n, or sometimes that it has dimensions $m \times n$. In contradistinction to this, the NumPy convention is to refer to an array like `A` as having "dimension 2", since it's made up of rows and columns (i.e., how many elements are in each row and in each column doesn't matter). There exists yet another attribute, `shape`, which returns a tuple containing the number of elements in each dimension, e.g., `A.shape` is `(3, 2)`. Table 1.3 collects the attributes we'll need. You can also create two-dimensional arrays by passing a tuple with the desired shape to the appropriate function, e.g., `np.zeros((3,2))` or `np.ones((4,6))`. Another function helps you make a square identity matrix via, e.g., `np.identity(4)`. All of these functions produce floats, by default.

If you want to produce an array starting from a hand-rolled list, but wish to avoid the Python list-of-lists syntax (with double square brackets, as well as several commas), you can start from a one-dimensional list, which is then converted into a one-dimensional array, which in its turn is re-shaped into a two-dimensional array via the `reshape()` function, e.g., `A = np.array([11,12,13,14,15,16]).reshape(3,2)`. Here's another example: `A = np.arange(11,23).reshape(3,4)`. Observe how conveniently we've obviated the multitude of square brackets and commas of `LL`.



Examples of slicing a 3×4 two-dimensional array

Fig. 1.4

It is now time to see one of the nicer features of two-dimensional arrays in NumPy: the intuitive way to access elements. To access a specific element of `LL`, you have to say, e.g., `LL[2][1]`. Recall that we are using Python 0-indexing, so the rows are numbered 0th, 1st, 2nd, and analogously for the columns. This double pair of brackets, separating the numbers from each other, is quite cumbersome and also different from how matrix notation is usually done, i.e., A_{ij} or $A_{i,j}$. Thus, it comes as a relief to see that NumPy array elements can be indexed simply by using only one pair of square brackets and a comma-separated pair of numbers, e.g., `A[2, 1]`.

Slicing is equally intuitive, e.g., `A[1,:]` picks a specific row and uses an everything-slice for the columns, and therefore leads to that entire row. Figure 1.4 shows a few other examples: the highlighting shows the elements that are chosen by the slice shown at the top of each matrix. The most interesting of these is probably `A[:2,1:-1]`, which employs the Python convention of using `-1` to denote the last element, in order to slice the "middle" columns: `1:-1` avoids the first/0th column and the last column. We also show a few examples that employ a stride when slicing. To summarize, when we use two numbers to index (such as `A[2,1]`), we go from a two-dimensional array to a number. When we use one number to index/slice (such as `A[:,2]`), we go from a two-dimensional array to a one-dimensional array. When we use slices (such as `A[::2,:]`), we get collections of rows, columns, individual elements, etc. We can combine what we just learned about slicing two-dimensional arrays with what we already know about NumPy broadcasting. For

example, `A[:,:] = 7` overwrites the entire matrix and you could do something analogous for selected rows, columns, etc.

Similarly to the one-dimensional case, *vectorized* operations for two-dimensional arrays allow us to, say, add together (elementwise) two square matrices, `A + B`. On the other hand, a simple multiplication is *also* carried out elementwise, i.e., each element in `A` is multiplied by the corresponding element in `B` when saying `A*B`. This may be unexpected behavior, if you were looking for a matrix multiplication. To repeat, *array operations are always carried out elementwise*, so when you multiply two two-dimensional arrays you get the *Hadamard product*, $(\mathbf{A} \odot \mathbf{B})_{ij} = A_{ij}B_{ij}$. The matrix multiplication that you know from linear algebra follows from a different formula, namely $(\mathbf{AB})_{ij} = \sum_k A_{ik}B_{kj}$, see Eq. (C.10). From Python 3.5 and onward[11] you can multiply two matrices using the `@` infix operator, i.e., `A@B`. In older versions you had to say `np.dot(A,B)`, which was not as intuitive as one would have liked. There are many other operations we could carry out, e.g., we can multiply together a two-dimensional and a one-dimensional matrix, `A@xs` or `xs@A`. It's easy to come up with more convoluted examples; here's another one: `xs@A@ys` is much easier to write (and read) than `np.dot(xs,np.dot(A,ys))`. Problem 1.8 studies the dot product of two one-dimensional arrays in detail: the main takeaway is that we compute it via `xs@ys`.[12] In a math course, to take the dot product (also known as the *inner product*) of two vectors you had to first take the transpose of the first vector (to have the dimensions match); conveniently, NumPy takes care of all of that for us, allowing us to simply say `xs@ys`.[13] Finally, *broadcasting* with two-dimensional arrays works just as you'd expect, e.g., `A/2` halves all the matrix elements.

NumPy contains several other important functions, with self-explanatory names. Since we just mentioned the (potential) need to take the transpose: NumPy has a function called `transpose()`; we prefer to access the transpose as an array attribute, i.e., `A.T`. You may recall our singing the praises of the `for` loop in Python; well, another handy idiom involves iterating over rows via `for row in A:` or over columns via `for column in A.T:`; marvel at how expressive both these options are. Iterating over columns will come in very handy in section 4.4.4 when we will be handling eigenvectors of a matrix.

In linear algebra the product of an $n \times 1$ column vector and a $1 \times n$ row vector produces an $n \times n$ matrix. If you try to do this "naively" in NumPy using 1d arrays, you will be disappointed: both `xs@ys` and `xs@(ys.T)` give a number (the same one). Inspect the `shape` attribute of `ys` and of `ys.T` to see what's going on. What you really need is the function `np.outer()`, which computes the outer product, e.g., `np.outer(xs, ys)`.

All the NumPy functions we mentioned earlier, such as `np.sqrt()`, also work on two-dimensional arrays, and the same holds for subtraction, powers, etc. Intriguingly, functions like `np.sum()`, `np.amin()`, and `np.amax()` also take in an (optional) argument `axis`: if you set `axis = 0` you operate across rows (column-wise) and `axis = 1` operates across columns (row-wise). Similarly, we can create our own user-defined functions that know how to handle an entire two-dimensional array/matrix at once. There are also several handy functions that are intuitively easier to grasp for the case of two-dimensional ar-

[11] Python 3.5 was already 8 years old when the present textbook came out.

[12] Make sure not to get confused by the fact that `A@B` produces a matrix but `xs@ys` produces a number.

[13] Similarly, when multiplying a vector by a matrix, `xs@A` is just as simple to write as `A@xs`.

rays, e.g., `np.diag()` to get the diagonal elements, or `np.tril()` to get the lower triangle of an array and `np.triu()` to get the upper triangle. In chapter 8 we will also need to use `np.meshgrid()`, which takes in two coordinate vectors and returns coordinate matrices. Another helpful function is `np.fill_diagonal()` which allows you to efficiently update the diagonal of a matrix you've already created. Finally, `np.trace()` often comes in handy.

The default storage-format for a NumPy array in memory is *row major*: as a beginner, you shouldn't worry about this too much, but it means that rows are stored in order, one after the other. This is the same format used by the C programming language. If you wish to use, instead, Fortran's column-major format, presumably in order to interoperate with code in that language, you simply pass in an appropriate keyword argument when creating the array. If you structure your code in the "natural" way, i.e., first looping through rows and then through columns, all should be well, so you can ignore this paragraph.

1.7 Project: Visualizing Electric Fields

Each chapter concludes with a physical application of techniques introduced up to that point. Since this is only the first chapter, we haven't covered any numerical methods yet. Even so, we can already start to look at some physics that is not so accessible without a computer by using `matplotlib` for more than just basic plotting. We will visualize a vector field, i.e., draw field lines for the electric field produced by several point charges.

1.7.1 Electric Field of a Distribution of Point Charges

Very briefly, let us recall *Coulomb's law*: the force on a test charge Q located at point P (at the position $\mathbf{r}$), coming from a single point charge q_0 located at $\mathbf{r}_0$ is given by:

$$\mathbf{F}_0 = k\frac{q_0 Q}{(\mathbf{r}-\mathbf{r}_0)^2}\frac{\mathbf{r}-\mathbf{r}_0}{|\mathbf{r}-\mathbf{r}_0|} \tag{1.1}$$

where Coulomb's constant is $k = 1/(4\pi\epsilon_0)$ in SI units (and ϵ_0 is the permittivity of free space). The force is proportional to the product of the two charges, inversely proportional to the square of the distance between the two charges, and points along the line from charge q_0 to charge Q. The electric field is then the ratio of the force $\mathbf{F}_0$ with the test charge Q in the limit where the magnitude of the test charge goes to zero. In practice, this gives us:

$$\mathbf{E}_0(\mathbf{r}) = kq_0\frac{\mathbf{r}-\mathbf{r}_0}{|\mathbf{r}-\mathbf{r}_0|^3} \tag{1.2}$$

where we cancelled out the Q and also took the opportunity to combine the two denominators. This is the electric field at the location $\mathbf{r}$ due to the point charge q_0 at $\mathbf{r}_0$.

If we were faced with more than one point charge, we could apply the *principle of superposition*: the total force on Q is made up of the vector sum of the individual forces acting on Q. As a result, if we were dealing with the n point charges $q_0, q_1, \ldots, q_{n-1}$ located at $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n-1}$ (respectively) then the situation would be that shown in Fig. 1.5. Our figure is in two dimensions for ease of viewing, but the formalism applies equally well to three dimensions. The total electric field at the location $\mathbf{r}$ is:

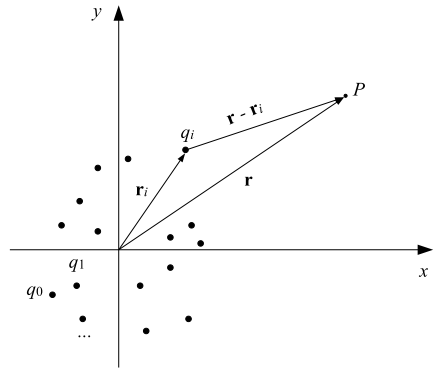


Fig. 1.5 Physical configuration made up of n point charges

$$\mathbf{E}(\mathbf{r}) = \sum_{i=0}^{n-1} \mathbf{E}_i(\mathbf{r}) = \sum_{i=0}^{n-1} kq_i \frac{\mathbf{r} - \mathbf{r}_i}{|\mathbf{r} - \mathbf{r}_i|^3} \tag{1.3}$$

namely, a sum of the individual electric field contributions, $\mathbf{E}_i(\mathbf{r})$. Note that you can consider this total electric field at any point in space, $\mathbf{r}$. Note, also, that the electric field is a vector quantity: at any point in space this $\mathbf{E}$ has a magnitude and a direction. One way of visualizing vector fields consists of drawing *field lines*, namely imaginary curves that help us keep track of the direction of the field. More specifically, the tangent of a field line at a given point gives us the direction of the electric field at that point. Field lines do not cross; they start at positive charges ("sources") and end at negative charges ("sinks").

1.7.2 Plotting Field Lines

We will plot the electric field lines in Python; while more sophisticated ways of visualizing a vector field exist (e.g., line integral convolution), what we describe below should be enough to give you a qualitative feel for things. While plotting functions (or even libraries) tend to change much faster than other aspects of the programming infrastructure, the principles discussed apply no matter what the specific implementation looks like.

We are faced with two tasks: first, we need to produce the electric field (vector) at several points near the charges as per Eq. (1.3) and, second, we need to plot the field lines in such a way that we can physically interpret what is happening. As in the previous code, we make a Python function for each task. For simplicity, we start from a problem with only two point charges (of equal magnitude and opposite sign). Also, we restrict ourselves to two dimensions (the Cartesian x and y).

Code 1.3 is a Python implementation, where Coulomb's constant is divided out for simplicity. We start by importing `numpy` and `matplotlib`, since the heavy lifting will be done

vectorfield.py **Code 1.3**

```
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt
from copy import deepcopy

def makefield(xs, ys):
      qtopos = {1: (-1,0), -1: (1,0)}
      n = len(xs)
      Exs = [[0. for k in range(n)] for j in range(n)]
      Eys = deepcopy(Exs)
      for j,x in enumerate(xs):
            for k,y in enumerate(ys):
                  for q,pos in qtopos.items():
                        posx, posy = pos
                        R = sqrt((x - posx)**2 + (y - posy)**2)
                        Exs[k][j] += q*(x - posx)/R**3
                        Eys[k][j] += q*(y - posy)/R**3
      return Exs, Eys

def plotfield(boxl,n):
      xs = [-boxl + i*2*boxl/(n-1) for i in range(n)]
      ys = xs[:]
      Exs, Eys = makefield(xs, ys)
      xs=np.array(xs); ys=np.array(ys)
      Exs=np.array(Exs); Eys=np.array(Eys)
      plt.streamplot(xs, ys, Exs, Eys, density=1.5, color='m')
      plt.xlabel('$x$')
      plt.ylabel('$y$')
      plt.show()

plotfield(2.,20)
```

by the function `streamplot()`, which expects NumPy arrays as input. We also import the square root and the `deepcopy()` function, which can create a distinct list-of-lists.

The function `makefield()` takes in two lists, `xs` and `ys`, corresponding to the coordinates at which we wish to evaluate the electric field (x and y together make up $\mathbf{r}$). We also need some way of storing the $\mathbf{r}_i$ at which the point charges are located. We have opted to store these in a dictionary, which maps from charge q_i to position $\mathbf{r}_i$—take some time to consider

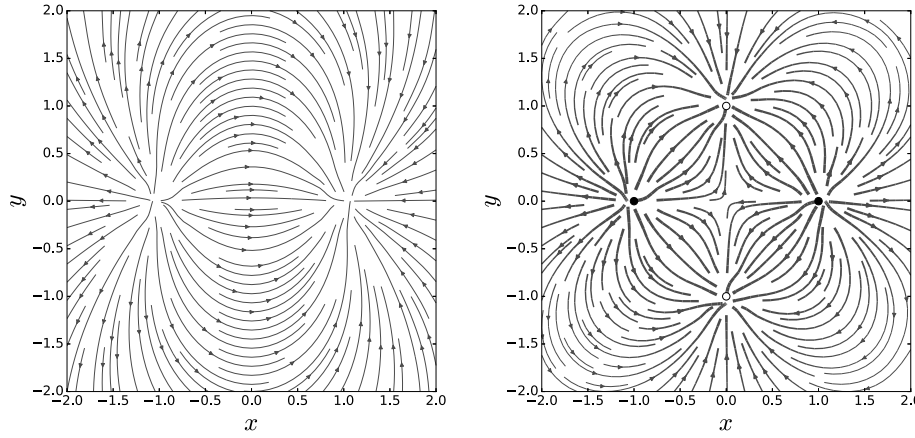
alternative ("manual") implementations. For each position $\mathbf{r}$ we need to evaluate $\mathbf{E}(\mathbf{r})$: in two dimensions, this is made up of $E_x(\mathbf{r})$ and $E_y(\mathbf{r})$, namely the two Cartesian components of the total electric field. Focusing on only one of these for the moment, say $E_x(\mathbf{r})$, we realize that we need to store its value for any possible $\mathbf{r}$, i.e., for any possible x and y values. We decide to use a list-of-lists, produced by a nested list comprehension. We then create another list-of-lists, for $E_y(\mathbf{r})$. We need to map out (i.e., store) the value of the x and y components of the total electric field, at all the desired values of the vector $\mathbf{r}$, namely, on a two-dimensional grid made up of `xs` and `ys`. This entails computing the electric field (contribution from a given point charge q_i) at all possible y's for a given x, and then iterating over all possible x's. We also need to iterate over our point charges q_i and their locations $\mathbf{r}_i$ (i.e., the different terms in the sum of Eq. (1.3)); we do this by saying `for q, pos in qtopos.items():` at which point we unpack `pos` into `posx` and `posy`.

We thus end up with three nested loops: one over possible x values, one over possible y values, and one over i. All three of these are written idiomatically, employing `items()` and `enumerate()`. The latter was used to ensure that we won't only have access to the x and the y values, which are needed for the right-hand side of Eq. (1.3), but also to two indices (j and k) that will help us store the electric-field components in the appropriate list-of-lists entry, e.g., `Exs[k][j]`.[14] This storing is carried out after defining a helper variable to keep track of the vector magnitude that appears in the denominator in Eq. (1.3). You should think about the += a little bit: since the left-hand side is for given j and k, the summation is carried out only when we iterate over the q_i (and $\mathbf{r}_i$). Incidentally, our idiomatic iteration over the point charges means that we don't even need an explicit i index.

Our second function, `plotfield()`, is where we build our two-dimensional grid for the `xs` and `ys`.[15] We take in as parameters the length L and the number of points n we wish to use in each dimension and create our `xs` using a list comprehension; all we're doing is picking x's from $-L$ to L. We then create a copy of `xs` and name it `ys`. After this, we call our very own `makefield()` to produce the two lists-of-lists containing $E_x(\mathbf{r})$ and $E_y(\mathbf{r})$ for many different choices of $\mathbf{r}$. The core of the present function consists of a call to `matplotlib`'s function `streamplot()`; this expects NumPy arrays instead of Python lists, so we convert everything over. If you skipped section 1.6, as you were instructed to do, you should relax: this call to `np.array()` is all you need to know for now (and until chapter 4). We also pass in to `streamplot()` two (optional) arguments, to ensure that we have a larger density of field lines and to choose the color. Most importantly, `streamplot()` knows how to take in `Exs` and `Eys` and output a plot containing curves with arrows, exactly like what we are trying to do. We also introduce x and y labels, using dollar signs to make the symbols look nicer.

The result of running this code is shown in the left panel of Fig. 1.6. Despite the fact that the charges are not represented by a symbol in this plot, you can easily tell that you are dealing with a positive charge on the left and a negative charge on the right. At this point, we realize that a proper graphic representation of field lines also has another feature: the density of field lines should correspond to the strength of the field (i.e., its magnitude). Our

[14] The confusing index order follows `streamplot()`'s documentation (see also page 630).
[15] This would all be much easier with two-dimensional arrays, but you skipped that section.



Visualizing the electric fields resulting from two and four point charges

Fig. 1.6

figure has discarded that information: the `density` argument we passed in had a constant value. This is a limitation of the `streamplot()` function.

There is a way to represent both the direction (as we already did) and the strength of the field using `streamplot()`, using the optional `linewidth` parameter. The argument passed in to `linewidth` can be a two-dimensional NumPy array, which keeps track of the strength at each point on the grid; it's probably better to pass in, instead, the logarithm of the magnitude at each point (possibly also using an offset). We show the result of extending our code to also include line width in the right panel of Fig. 1.6, where a stronger field is shown using a thicker line. This clearly shows that the field strength is larger near the charges (and in between them) than it is as you go far away from them. To make things interesting, this shows a different situation than the previous plot did: we are now dealing with four charges (two positive and two negative, all of equal magnitude). We also took the opportunity to employ symbols representing the position of the point charges themselves.

Problems

1.1 Study the following program, meant to evaluate the factorial of a positive integer:

```
def fact(n):
      return 1 if n==0 else n*fact(n-1)
print(fact(10))
```

This uses Python's version of the *ternary operator*. Crucially, it also uses *recursion*: we are writing the solution to our problem in terms of the solution of a smaller version of the same problem. The function then calls itself repeatedly. At some point we reach the *base case*, where the answer can be directly given. Recursion is helpful when the problem you are solving is amenable to a "divide-and-conquer" approach,

as we will see in section 6.4.3.3. For the example above, recursion is quite unnecessary: write a Python function that evaluates the factorial *iteratively*.

1.2 Produce a function, `descr()`, which describes properties of the function and argument value that are passed in to it. Define it using `def descr(f, x):`.

(a) Write the function body; this should separately print out `f`, `x`, and `f(x)`. Now call it repeatedly, for a number of user-defined and Python standard library functions. Notice that when you print out `f` the output is not human friendly.
(b) Pass in as the first argument not the function but a string containing the function name. You now need a mechanism that converts the string you passed in to a function (since you still need to print out `f(x)`). Knowing the list of possibilities, create a dictionary that uses strings as keys and functions as values.
(c) Modify the previous program so that the first argument that is passed in to `descr()` is a function, not a string. To produce the same output, you must also modify the body of the function since your dictionary will now be different.

1.3 Iterate through a list `xs` in reverse, printing out both the (decreasing) index and each element itself. Come up with both Pythonic and unPythonic solutions.

1.4 Imagine you have access to two Python functions, `fa()` and `fb()`, with one parameter each.[16] You now wish to apply the function `der()` that was defined in section 1.3.4 to a linear combination of `fa()` and `fb()`, namely `a*fa(x) + b*fb(x)`.

(a) Your initial reaction might be to define a third function, `fc()`; but what if you don't actually know the values of the `a` and `b` coefficients until you're already inside some other Python function, say, `caller()`? Based on what we've introduced in the main text, one idea is to define `fc()` to take three parameters (`x`, `a`, and `b`). This would then not interoperate with `der()`, which expects as its first argument a function taking in a single parameter. Try this.
(b) Your next instinct may be to modify the definition of `der()`: sometimes that's OK, but often it's just asking for trouble. What you're really after is a throwaway function that knows about the values of variables defined in the same scope. Do so via an anonymous function (via Python's `lambda` keyword) right where you need it. Look up the documentation and implement this solution.
(c) You may next be tempted to actually *name* your lambda function and then use it in `der()`. Try this. Unfortunately, this is explicitly discouraged in PEP 8.[17]
(d) Instead, define a *nested function* `func()` inside `caller()`, the latter calling `der()`.
(e) Use a *closure*: `func()` is nested inside `caller()`, the latter returning `func()`; recall that in Python you use the function name without the parentheses.

1.5 We will now elaborate on the for-else idiom introduced in `forelse.py`.

(a) First, we realize that our `look()` function, while helpful from a pedagogical perspective, is somewhat pointless: you didn't really need to write a function to

[16] The same idea applies to the two functions `sumofints()` and `cos()` that we encountered in the main text.
[17] It's easy to see why: if you feel the need to name your... anonymous function, then perhaps you shouldn't be using an anonymous function in the first place.

check for membership in a list. Directly use Python's `in` keyword to test whether or not a given string is to be found in a given list of strings.

(b) Now make `look()` slightly more useful by having it return not only the value (i.e., the string) but also the index where that value was found.

(c) Experiment with making `look()` shorter, by avoiding the for-else idiom altogether. First, do this the simplest way, i.e., by providing the default option `val = None` *before* you enter the loop. Second, avoid using `break` altogether by opting for two exit points in your function, i.e., two separate `return` statements.[18]

(d) The for-else idiom is actually better than the alternatives when exceptions (which we don't discuss elsewhere) are involved. Check the documentation to see how you can `raise` a `ValueError` in the `else` block of your `for` loop.

1.6 The following is implicitly defining a recurrence relation:

```
f0,f1 = 0,1
for i in range(n-1):
      f0,f1 = f1,f0+2*f1
```

We will now produce increasingly fancier versions of this code snippet.

(a) Define a function that takes in the cardinal number n and returns the corresponding latest value following the above recurrence relation. In other words, for $n = 0$ you should get 0, for $n = 1$ you should get 1, for $n = 2$ you should get 2, for $n = 3$ you should get 5, and so on.

(b) Define a *recursive* function taking in the cardinal number n and returning the corresponding latest value. The interface of the function will be identical to that of the previous part (the implementation will be different).

(c) Define a similar function that is more efficient. Outside the function, define a dictionary `ntoval = {0:0, 1:1}`. Inside the function, you should check to see if the n that was passed in exists as a key in `ntoval`: if it does, then simply return the corresponding value; if it doesn't, then carry out the necessary computation and augment the dictionary with a new key-value pair.

(d) If you take separation of concerns seriously, you may be feeling uncomfortable about accessing and modifying `ntoval` inside your function (since it is not being passed in as a parameter). Write a new function that looks like the one in the previous part, but takes in two parameters: `n` and `ntoval`.

(e) While part (d) respects separation of concerns, unfortunately it is not actually efficient. Write a similar function which uses a *mutable default parameter value*, i.e., it is defined by saying `def f5(n, ntoval = {0:0, 1:1}):`.

Test all five functions with $n = 8$: each of them should return 408. The functions in parts (c) and (e) should be efficient in the sense that if you now call them with, say, $n = 6$ they won't need to recompute the answer since they have already done so.

[18] Elsewhere in this book we always employ a single exit point, to make our codes easier to reason about.

1.7 This problem studies the quantity $(1 + 1/n)^n$ where $n = 10^1, 10^2, \ldots, 10^7$. Print out a nicely formatted table where the three columns are: (a) the value of n, (b) the quantity of interest computed with single-precision floating-point numbers, (c) the same quantity but now using doubles. You will need to use NumPy to get the singles to work. Keep in mind that the numbers shown in the second and third columns should be different (if they aren't, you're doing it wrong).

1.8 Investigate the relative efficiency of multiplying two one-dimensional NumPy arrays, `as` and `bs`; these should be large and with non-constant content. Do this in four distinct ways: (a) `sum(as*bs)`, (b) `np.sum(as*bs)`, (c) `np.dot(as,bs)`, and (d) `as@bs`. You may wish to use the `default_timer()` function from the `timeit` module. To produce meaningful timing results, repeat such calculations thousands of times (at least).

1.9 Take two matrices, $\mathbf{A}$ and $\mathbf{B}$, which are $n \times m'$ and $m' \times m$, respectively. Implement matrix multiplication, without relying on `numpy`'s `@` or `dot()` as applied to matrices.

(a) Write the most obvious implementation you can think of, which takes in $\mathbf{A}$ and $\mathbf{B}$ and returns a new $\mathbf{C}$. Use three loops.
(b) Write a function without the third loop by applying `@` to vectors.
(c) Write a third function that takes in $\mathbf{A}$ and $\mathbf{B}$ and returns $\mathbf{C}$, but this time these are lists-of-lists, instead of arrays.
(d) Test the above three functions by employing specific examples for $\mathbf{A}$ and $\mathbf{B}$, say 3×4 and 4×2, respectively.

1.10 Write your own functions that implement functionality similar to: (a) `np.argmax()`, (b) `np.where()`, and (c) `np.all()`, where the input will be a one-dimensional NumPy array. Note that `np.where()` is equivalent to `np.nonzero()` for this case.

1.11 Rewrite code 5.9, i.e., `action.py`, such that:

(a) The two lines involving `arr[1:-1]` now use an explicit loop and index.
(b) The calls to `np.fill_diagonal()` are replaced by explicit loop(s) and indices.

Compare the expressiveness (and total line-count) in your code vs that in `action.py`.

1.12 [$\mathcal{P}$] We now help you produce the right panel of Fig. 1.6:

(a) First try to produce the curves themselves. You'll need to appropriately place two positive and two negative charges and plot the resulting field lines. (Remember that dictionaries don't let you use duplicate keys, for good reason.)
(b) Introduce line width, by producing a list of lists containing the logarithm of the square root of the sum of the squares of the components of the electric field at that point, i.e., $\log\left[\sqrt{(E_x(\mathbf{r}))^2 + (E_y(\mathbf{r}))^2}\right]$ at each $\mathbf{r}$.
(c) Figure out how to add circles/dots (of color corresponding to the charge sign).
(d) Re-do this plot for the case of four positive (and equal) charges.

1.13 [$\mathcal{P}$] Problem 1.12 was made easier by the fact that you knew what the answer had to look like. You now need to produce a plot for the field lines (including line width) for the case of four alternating charges on a line. Place all four charges on the x axis, and

give their x_i positions the values -1, -0.5, 0.5, and 1. The leftmost charge should be positive and the next one negative, and so on (they are all of equal magnitude).

1.14 [$\mathcal{P}$] Study methane *isotherms* with the van der Waals equation of state, Eq. (5.2).

(a) Plot 40 isotherms (i.e., constant-T curves, showing P vs v), where T goes from 162 to 210 K, v goes from $1.5b$ to $9b$ and the curves look smooth.

(b) If you solved the previous part correctly, you should barely be able to tell the different curves apart. Beautify your plot by employing an automated color map.

1.15 [$\mathcal{P}$] We address the problem of *wave interference*, by repurposing `vectorfield.py`. Work in two dimensions (just like in section 1.7) and assume that the combined effect of two objects dropped in a water basin can be modelled by:

$$W(\mathbf{r}) = A\sin(k|\mathbf{r}-\mathbf{r}_0|) + A\sin(k|\mathbf{r}-\mathbf{r}_1|) \tag{1.4}$$

You can think of this as an updated version of Eq. (1.3), where we are dealing with the linear addition of sinusoidal waves, instead of electric-field contributions. For simplicity, take $A = 1$, $k = 2\pi/0.3$, $\mathbf{r}_0 = (-1\ \ 0)^T$, and $\mathbf{r}_1 = (1\ \ 0)^T$. Since we are now faced with a density plot, you should employ (not `streamplot()` but) `imshow()` from Matplotlib; in each of x and y your plot should extend from -4 to $+4$.

1.16 [$\mathcal{P}$] We will now examine the simple (yet non-linear) pendulum on the *phase plane*. The total energy of a pendulum of mass m and length l is:

$$E = \frac{1}{2}ml^2\dot{\theta}^2 + mgl(1-\cos\theta) \tag{1.5}$$

where θ is the angle from the vertical. Your task is to create a plot of $\dot{\theta}$ vs θ, where you will show the contours of constant E using Matplotlib's `contour()`; feel free to repurpose `vectorfield.py`. It's best to vary θ from -2π to $+2\pi$; you may also have to play around with the `levels` parameter; take $m = 1$ kg, $l = 1$ m, and $g = 9.8$ m/s^2. Physically interpret the different regions you encounter; roughly speaking, you should be seeing open curves[19] (above and below) and eye-shaped areas (near the middle). The boundary between the two regions is known as a *separatrix*.

1.17 [$\mathcal{P}$] We will now discuss how to visualize *binary stars*. While stars in well-detached binaries will have spherical shapes, stars in close binaries will experience tidal distortions and will have nearly ellipsoidal shapes. The *Roche model* was introduced to account for either possibility: it studies the total gravitational potential for two masses that are in circular orbit (about their barycenter). Switching to a coordinate frame that eliminates the circular orbit of the two masses (a co-rotating/"synodic" frame), the (dimensionless) Roche potential at any point (x, y, z) can be written as:

$$\Phi = \frac{m_0}{\sqrt{(x-m_1)^2+y^2+z^2}} + \frac{m_1}{\sqrt{(x+m_0)^2+y^2+z^2}} + \frac{x^2+y^2}{2} \tag{1.6}$$

The (dimensionless) masses can be written in terms of the mass ratio $q = m_0/m_1 \leq 1$ ($m_0 = q/(1+q)$ and $m_1 = 1/(1+q)$). We focus on the orbital plane ($z = 0$).

[19] The open curves have constant amplitude as θ is increased. We will see a different scenario in problem 8.43.

(a) Repurpose `vectorfield.py`, using Matplotlib's `contour()`, to visualize the curves of constant gravitational potential for the case of $q = 0.4$. Pass in an appropriate `levels` *list* to make your figure look good. (Note that a star's surface *is* an equipotential surface.) The characteristic ∞ shape you find in the middle is made up of two *Roche lobes*; a Roche lobe tells you the maximum volume that a star in a binary can occupy without losing gravitational control of its constituents.

(b) Another way you can visualize the Roche potential of Eq. (1.6) is via a surface plot. You can create such a figure via Matplotlib's `plot_surface()`. Go over its documentation, which will explain that you will first need to pass in `projection='3d'` when creating your axes. Make sure you can easily identify the different ridges and wells corresponding to the two stars as well as the area around them (e.g., via the use of `set_zlim()`).

1.18 [$\mathcal{P}$] The problem of a single particle in a one-dimensional lattice is a staple of introductory courses on solid-state physics; we will see how to tackle it for the case of a general potential in problem 4.56 (after learning about linear-algebra techniques). Here, we address a specific case, that of the *Kronig–Penney model*, involving an infinite periodic array of potential barriers (or, viewed from another perspective, of wells) of rectangular shape. Writing down the wave function within the barrier (of width $L - M$) and separately that within the well (of width M), and then imposing continuity of the wave function and of its derivative, one arrives at the condition:

$$\begin{aligned} \cos(KL) &= \cos(k_0 M)\cosh[k_1(L-M)] + \frac{k_1^2 - k_0^2}{2k_0 k_1}\sin(k_0 M)\sinh[k_1(L-M)] \\ k_0 &= \sqrt{2mE/\hbar^2}, \qquad k_1 = \sqrt{2m(V_0 - E)/\hbar^2} \end{aligned} \tag{1.7}$$

V_0 is the barrier height, E the energy, and K the wave number in *Bloch's theorem*:

$$\psi(x + L) = e^{iKL}\psi(x) \tag{1.8}$$

This wave number obeys $-\pi < KL < \pi$. Equation (1.7) implicitly provides us with the energy-dispersion relation, i.e., $E(K)$. However, we don't know how to solve complicated equations yet (we tackle this challenge in problem 5.45, after learning about root-finding). Our approach here will be to: hand-pick a value of E, compute the right-hand side of the equation, and check to see if its absolute value is larger than unity; if so, then there is no solution for K. If, however, the absolute value of the right-hand side is smaller than unity, then we can find the value of K by taking the inverse cosine and dividing by L. (Since the cosine is an even function, you need to consider both K and $-K$ as equally acceptable solutions.) Take $L = 1, M = 0.4, \hbar^2/m = 1$ and study five thousand E values from (roughly) 0 to 110. Plot E vs K for the two cases of: (a) $V_0 = 0$ and (b) $V_0 = 35$. Do yourself a favor and employ the special functions contained in the complex-number-aware module `cmath` (i.e., not in `math`). Your plot for $V_0 = 0$ should be a parabola, which corresponds to a plane-wave dispersion; of course, your results will lie in $-\pi < KL < \pi$, so you will find a "folded" version of the parabola. Your plot for $V_0 = 35$ should exhibit *band gaps*, resulting from the fact that there are energy regions for which no solution exists.

Numbers 2

Should we not be concerned that this fear of erring is already the error itself?
Georg Wilhelm Friedrich Hegel

2.1 Motivation

We don't really have to provide a motivation regarding the importance of numbers in physics: both experimental measurements and theoretical calculations produce specific numbers. Preferably, one also estimates the uncertainty associated with these values.

We have, semi-arbitrarily, chosen to discuss a staple of undergrad physics education, the photoelectric effect. This arises when a metal surface is illuminated with electromagnetic radiation of a given frequency ν and electrons come out. In 1905 Einstein posited that quantization is a feature of the radiation itself. Thus, a light quantum (a *photon*) gives up its energy ($h\nu$, where h is now known as *Planck's constant*) to an electron, which has to break free of the material, coming out with a kinetic energy of:

$$T = h\nu - W \tag{2.1}$$

where W is the work function that relates to the specific material. The maximum kinetic energy T of the photoelectrons could be extracted from the potential energy of the electric field needed to stop them, via $T = eV_s$, where V_s is the stopping potential and e is the charge of the electron. Together, these two relations give us:

$$eV_s = h\nu - W \tag{2.2}$$

Thus, if one produces data relating V_s with ν, the slope would give us h/e.

In 1916, R. A. Millikan published a paper [107] titled "A direct photoelectric determination of Planck's h", where he did precisely that. Millikan's device included a remotely controlled knife that would shave a thin surface off the metal; this led to considerably enhanced photocurrents. It's easy to see why Millikan described his entire experimental setup as a "machine shop in vacuo". Results from this paper are shown in Fig. 2.1. Having extracted the slope h/e, the author then proceeded to compute h by "inserting my value of e".[1] The value Millikan extracted for Planck's constant was:

$$h = 6.56 \times 10^{-27} \text{erg s} \tag{2.3}$$

In his discussion of Fig. 2.1, Millikan stated that "it is a conservative estimate to place

[1] Recall that Millikan had measured e very accurately with his oil-drop experiment in 1913.

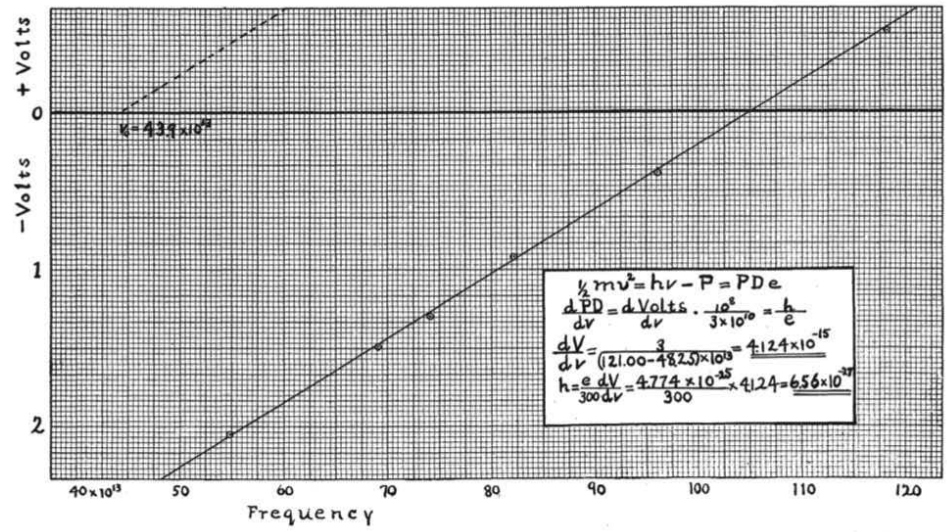


Fig. 2.1 Millikan's data on the photoelectric effect. Reprinted figure with permission from R. A. Millikan, *Phys. Rev.* **7**, 355 (1916), Copyright 1916 by the American Physical Society.

the maximum uncertainty in the slope at about 1 in 200 or 0.5 per cent".Translating this to the above units, we find an uncertainty estimate of 0.03×10^{-27}erg s. Richardson and Compton's earlier experimental results [122] had a slope uncertainty that was larger than 60 percent. The above extraction is to be compared with the modern determination of $h = 6.626070040(81) \times 10^{-27}$erg s. Overall, Millikan's result was a huge improvement on earlier works and was important in the acceptance of Einstein's work and of light quanta.[2]

For the sake of completeness, we note that the aforementioned error estimate follows from *assuming* a linear relationship. It would have probably been best to start from Millikan's earlier comment that "the maximum possible error in locating any of the intercepts is say two hundredths of a volt" and then do a least-squares fit, as explained in problem 6.65. This isolated example already serves to highlight that even individual experimental measurements have associated uncertainties; nowadays, experimental data points are always given along with a corresponding error bar. Instead of multiplying the examples where experiment and theory interact fruitfully, we now turn to the main theme of this chapter, which is the presence of errors when storing and computing numbers.

2.2 Errors

In this text we use the word *accuracy* to describe the match of a value with the (possibly unknown) true value. On the other hand, we use the word *precision* to denote how many

[2] Intriguingly, Millikan himself was far from being convinced that quantum theory was relevant here, speaking of "the bold, not to say the reckless, hypothesis of an electro-magnetic light corpuscle".

digits we can use in a mathematical operation, whether these digits are correct or not. An inaccurate result arises when we have an error. This can happen for a variety of reasons, only one of which is limited precision.[3] Excluding "human error" and measurement uncertainty in the input data, there are typically two types of errors we have to deal with in numerical computing: approximation error and rounding error. In more detail:

- **Approximation errors** These are sometimes known as *truncation errors*. Here's an example. You are trying to approximate the exponential, e^x, using its Taylor series:

$$y = \sum_{n=0}^{n_{max}} \frac{x^n}{n!} \tag{2.4}$$

  Obviously, we are limiting the sum to the terms up to n_{max} (i.e., we are including the terms labelled 0, 1, ..., n_{max} and dropping the terms from $n_{max} + 1$ to ∞). As a result, it's fairly obvious that the value of y for a given x may depend on n_{max}. In principle, at the mere cost of running one's calculation longer, one can get a better answer.[4]
- **Roundoff errors** These are also known as *rounding errors*. This type of error appears every time a calculation is carried out using floating-point numbers: since these don't have infinite precision, some information is lost. Here's an example: using real numbers, it is easy to see that $(\sqrt{2})^2 - 2 = 0$. However, when carrying out the same operation in Python we get a non-zero answer:

```
>>> (sqrt(2))**2 - 2
4.440892098500626e-16
```

  This is because $\sqrt{2}$ cannot be evaluated with infinitely many digits on the computer. Thus, the (slightly inaccurate) result for `sqrt(2)` is then used to carry out a second calculation, namely the squaring. Finally, the subtraction is yet another mathematical operation that can lead to rounding error.[5] Often, roundoff errors do not go away even if you run the calculation longer.

In the present chapter we will talk quite a bit about roundoff errors. In the next chapter we talk about the combined effect of approximation and roundoff errors. The chapters after that typically focus only on approximation errors, i.e., on estimating how well a specific method performs in principle. Before we get that far, however, let us first try to introduce some basic concepts, without limiting ourselves to any one kind of error.

2.2.1 Absolute and Relative Error

Assume we are studying a quantity whose exact value is x. If $\tilde{x}$ is an approximate value for it, then we can define the *absolute error* as follows:[6]

[3] The value 1.23456789123456 is precisely determined yet, viewed as an approximation of π, quite inaccurate.
[4] But keep reading, since roundoff error becomes important here, too.
[5] Again, this is discussed much more thoroughly in the rest of the chapter.
[6] Other authors employ another definition of the absolute error, which differs by a minus sign (see also page 459).

$$\Delta x = \tilde{x} - x \tag{2.5}$$

We don't specify at this point the source of this absolute error: it could be uncertainties in the input data, an inaccuracy introduced by our imperfect earlier calculation, or the result of roundoff error (possibly accumulated over several computations). For example:

$$x_0 = 1.000, \qquad \tilde{x}_0 = 0.999 \tag{2.6}$$

corresponds to an absolute error of $\Delta x_0 = -10^{-3}$. This also allows us to see that the absolute error, as defined, can be either positive or negative. If you need it to be positive (say, in order to take its logarithm), simply take the absolute value.

We are usually interested in defining an *error bound* of the form:

$$|\Delta x| \leq \epsilon \tag{2.7}$$

or, equivalently:

$$|\tilde{x} - x| \leq \epsilon \tag{2.8}$$

where we hope that ϵ is "small". Having access to such an error bound means that we can state something very specific regarding the (unknown) exact value x:

$$\tilde{x} - \epsilon \leq x \leq \tilde{x} + \epsilon \tag{2.9}$$

This means that, even though we don't know the exact value x, we do know that it could be at most $\tilde{x} + \epsilon$ and at the least $\tilde{x} - \epsilon$. Keep in mind that if you know the actual absolute error, as in our $\Delta x_0 = -10^{-3}$ example above, then, from Eq. (2.5), you know that:

$$x = \tilde{x} - \Delta x \tag{2.10}$$

and there's no need for inequalities. The inequalities come into the picture when you don't know the actual value of the absolute error and only know a bound for the magnitude of the error. The error bound notation $|\Delta x| \leq \epsilon$ is sometimes rewritten in the form $x = \tilde{x} \pm \epsilon$, though you should be careful: this employs our definition of *maximal error* (i.e., the worst-case scenario) as above, not the usual *standard error* (i.e., the statistical concept you may have encountered in a lab course).

Of course, even at this early stage, one should think about exactly what we mean by "small". Our earlier case of $\Delta x_0 = -10^{-3}$ probably fits the bill. But what about:

$$x_1 = 1\,000\,000\,000.0, \qquad \tilde{x}_1 = 999\,999\,999.0 \tag{2.11}$$

which corresponds to an absolute error of $\Delta x_1 = -1$? Obviously, this absolute error is larger (in magnitude) than $\Delta x_0 = -10^{-3}$. On the other hand, it's not too far-fetched to say that there's something wrong with this comparison: x_1 is much larger (in magnitude) than x_0, so even though our approximate value $\tilde{x}_1$ is off by a unit, it "feels" closer to the corresponding exact value than $\tilde{x}_0$ was.

This is resolved by introducing a new definition. As before, we are interested in a quantity whose exact value is x and an approximate value for it is $\tilde{x}$. Assuming $x \neq 0$, we can define the *relative error* as follows:

$$\delta x = \frac{\Delta x}{x} = \frac{\tilde{x} - x}{x} \tag{2.12}$$

Obviously, this is simply the absolute error Δx divided by the exact value x. As before, we are not specifying the source of this relative error (input-data uncertainties, roundoff, etc.). Another way to express the relative error is:

$$\tilde{x} = x(1 + \delta x) \tag{2.13}$$

You should convince yourself that this directly follows from Eq. (2.12). We will use this formulation repeatedly in what follows.[7]

Let's apply our definition of the relative error to the earlier examples:

$$\delta x_0 = \frac{0.999 - 1.000}{1.000} = -10^{-3}, \qquad \delta x_1 = \frac{999\,999\,999.0 - 1\,000\,000\,000.0}{1\,000\,000\,000.0} = -10^{-9} \tag{2.14}$$

The definition of the relative error is consistent with our intuition: $\tilde{x}_1$ is, indeed, a much better estimate of x_1 than $\tilde{x}_0$ is of x_0. Quite frequently, the relative error is given as a percentage: δx_0 is a relative error of -0.1% whereas δx_1 is a relative error of $-10^{-7}\%$.

In physics the values of an observable can vary by several orders of magnitude (according to density, temperature, and so on), so it is wise to employ the scale-independent concept of the relative error, when possible.[8] Just like for the case of the absolute error, we can also introduce a *bound for the relative error*:

$$|\delta x| = \left|\frac{\Delta x}{x}\right| \leq \epsilon \tag{2.15}$$

where now the phrase "ϵ is small" is unambiguous (since it doesn't depend on whether or not x is large). Finally, we note that the definiton of the relative error in Eq. (2.12) involves x in the denominator. If we have access to the exact value (as in our examples with x_0 and x_1 above), all is well. However, if we don't actually know the exact value, it is sometimes more convenient to use, instead, the approximate value $\tilde{x}$ in the denominator. Problem 2.2 discusses this alternative definition and its connection with what we discussed above.

2.2.2 Error Propagation

So far, we have examined the concepts of the absolute error and of the relative error (as well as the corresponding error bounds); no details were provided regarding the mathematical

[7] You can also expand the parentheses and identify $x\delta x = \Delta x$, to see that this leads to $\tilde{x} = x + \Delta x$.

[8] Once again, if you need the relative error to be positive, simply take the absolute value.

operations carried out using these values. We will now discuss the elementary operations (addition, subtraction, multiplication, division), to give you some insights into combining approximate values together. One of our goals is to see what happens when we put together the error bounds for two numbers a and b to produce an error bound for a third number x, i.e., we will study error propagation. We will also study more general scenarios, in which one is faced with more complicated mathematical operations (and possibly more than two numbers being operated on). In what follows, it's important to keep in mind that these are *maximal errors*, so our results will be different (and likely more pessimistic) than what you may have encountered in a standard course on experimental measurements.[9]

2.2.2.1 Addition or Subtraction

We are faced with two real numbers a and b, and wish to take their difference:

$$x = a - b \tag{2.16}$$

As usual, we don't know the exact values, but only the approximate values $\tilde{a}$ and $\tilde{b}$, so what we form instead is the difference of these:

$$\tilde{x} = \tilde{a} - \tilde{b} \tag{2.17}$$

Let us now apply Eq. (2.10) twice:

$$\tilde{a} = a + \Delta a, \qquad \tilde{b} = b + \Delta b \tag{2.18}$$

Plugging the last four equations into the definition of the absolute error, Eq. (2.5), we have:

$$\Delta x = \tilde{x} - x = (a + \Delta a) - (b + \Delta b) - (a - b) = \Delta a - \Delta b \tag{2.19}$$

In the third equality we cancelled what we could.

We now recall that we are interested in finding relations between error bounds. Thus, we take the absolute value and then use the triangle inequality to find:

$$|\Delta x| \leq |\Delta a| + |\Delta b| \tag{2.20}$$

You should convince yourself that a fully analogous derivation leads to exactly the same result for the case of addition of the two numbers a and b. Thus, our main conclusion so far is that *in addition and subtraction adding together the bounds for the absolute errors in the two numbers gives us the bound for the absolute error in the result*.

Let's look at an example. Assume that we have:

$$|4.56 - a| \leq 0.14, \qquad |1.23 - b| \leq 0.03 \tag{2.21}$$

(If you are in any way confused by this notation, look up Eq. (2.7) or Eq. (2.8).) Our finding in Eq. (2.20) implies that the following relation will hold, when $x = a - b$:

$$|3.33 - x| \leq 0.17 \tag{2.22}$$

[9] "It is probable that many quite improbable things should happen" (Aristotle, *Poetics*, 1456a25).

It's easy to see that this error bound, simply the sum of the two error bounds we started with, is larger than either of them. If we didn't have access to Eq. (2.20), we could have arrived at the same result the long way: (a) when a has the greatest possible value (4.70) and b has the smallest possible value (1.20), we get the greatest possible value for $a - b$, namely 3.50, and (b) when a has the smallest possible value (4.42) and b has the greatest possible value (1.26), we get the smallest possible value for $a - b$, namely 3.16.

As our main result in Eq. (2.20), applied just now in a specific example, shows, we simply add up the absolute error bounds. Again, this is different from what you do when you are faced with a "standard error" (i.e., the standard deviation of the sampling distribution of a statistic): in that case, the absolute errors add "in quadrature"; we will address this scenario in due course, see Eq. (6.129). We repeat that our result is more pessimistic (i.e., tries to account for the worst-case scenario). We won't keep repeating this warning below.

2.2.2.2 Catastrophic Cancellation

Let us examine the most interesting special case: $a \approx b$ (for which case $x = a - b$ is small). Dividing our result in Eq. (2.20) by x gives us the relative error (bound) in x:

$$|\delta x| = \left|\frac{\Delta x}{x}\right| \leq \frac{|\Delta a| + |\Delta b|}{|a - b|} \tag{2.23}$$

Now, express Δa and Δb in terms of the corresponding relative error: $\Delta a = a\delta a$ and $\Delta b = b\delta b$. Since $a \approx b$, you can factor $|a|$ out:

$$|\delta x| \leq (|\delta a| + |\delta b|)\frac{|a|}{|a - b|} \tag{2.24}$$

It's easy to see that if $a \approx b$ then $|a - b|$ will be much smaller than $|a|$ so, since the fraction will be large, the relative errors δa and δb will be magnified.[10]

Let's look at an example. Assume that we have:

$$|1.25 - a| \leq 0.03, \qquad |1.20 - b| \leq 0.03 \tag{2.25}$$

that is, a relative error (bound) $\delta a \approx 0.03/1.25 = 0.024$ or roughly 2.4% (this is approximate, because we divided by $\tilde{a}$, not by the, unknown, a). Similarly, the other relative error (bound) is $\delta b \approx 0.03/1.20 = 0.025$ or roughly 2.5%. From Eq. (2.24) we see that the relative error for the difference will obey:

$$|\delta x| \leq (0.024 + 0.025)\frac{1.25}{0.05} = 1.225 \tag{2.26}$$

where the right-hand side is an approximation (using $\tilde{a}$ and $\tilde{x}$). This shows us that two numbers with roughly 2.5% relative errors were subtracted and the result has a relative error which is more than one hundred percent! This is sometimes known as *subtractive or catastrophic cancellation*. For the purists, we note that *catastrophic cancellation* refers to the case where the two numbers we are subtracting are themselves subject to errors, as above.

[10] This specific issue doesn't arise in the case of addition, since there the denominator doesn't have to be tiny.

There also exists the scenario of *benign cancellation*, which shows up when you subtract quantities that are exactly known (though that is rarely the case in practice) or when the result of the subtraction does not need to be too accurate for what follows. The distinction between catastrophic and benign cancellation is further explained in problems 2.6 and 2.7 (including the classic example of a simple quadratic equation) and in section 2.4 below.

2.2.2.3 Multiplication or Division

We are faced with two real numbers a and b, and wish to take their product:

$$x = ab \tag{2.27}$$

As usual, we don't know the exact values, but only the approximate values $\tilde{a}$ and $\tilde{b}$, so what we form instead is the product of these:

$$\tilde{x} = \tilde{a}\tilde{b} \tag{2.28}$$

With some foresight, we will now apply Eq. (2.13) twice:

$$\tilde{a} = a(1 + \delta a), \qquad \tilde{b} = b(1 + \delta b) \tag{2.29}$$

Plugging the last four equations into the definition of the relative error, Eq. (2.12), we have:

$$\delta x = \frac{\tilde{x} - x}{x} = \frac{\tilde{a}\tilde{b} - ab}{ab} = \frac{a(1 + \delta a)b(1 + \delta b) - ab}{ab} = 1 + \delta a + \delta b + \delta a \delta b - 1 = \delta a + \delta b \tag{2.30}$$

In the fourth equality we cancelled the ab. In the fifth equality we cancelled the unit and we dropped $\delta a \delta b$ since this is a higher-order term (it is the product of two small terms).

We now recall that we are interested in finding relations between error bounds. Thus, we take the absolute value and then use the triangle inequality to find:

$$|\delta x| \leq |\delta a| + |\delta b| \tag{2.31}$$

In problem 2.1 you will carry out the analogous derivation for the case of division of the two numbers a and b, finding exactly the same result. Thus, our new conclusion is that *in multiplication and division adding together the bounds for the relative errors in the two numbers gives us the bound for the relative error in the result*. Observe that for addition or subtraction we were summing *absolute* error bounds, whereas here we are summing *relative* error bounds. Thus, typically multiplication and division don't cause too much trouble, whereas addition and (especially) subtraction can cause headaches.

Let's look at an example. Assume that we have the same numbers as in Eq. (2.25):

$$|1.25 - a| \leq 0.03, \qquad |1.20 - b| \leq 0.03 \tag{2.32}$$

that is, a relative error $\delta a \approx 0.03/1.25 = 0.024$ of roughly 2.4% and a relative error $\delta b \approx 0.03/1.20 = 0.025$ of roughly 2.5%. Our finding in Eq. (2.31) implies that:

$$\frac{|1.5 - x|}{|x|} \leq 0.049 \tag{2.33}$$

namely a relative error bound of roughly 4.9%. It's easy to see that this error bound, while larger than either of the relative error bounds we started with, is nowhere near as dramatic as what we found when we subtracted the same two numbers.

2.2.2.4 General Error Propagation: One Variable

We have studied (maximal) error propagation for a couple of simple cases of combining two numbers (subtraction and multiplication). But what about the error when you do something more complicated to a single number, e.g., take its square root or its logarithm?

Let us go over some notation. As per Eq. (2.5), the absolute error in a variable x is:

$$\Delta x = \tilde{x} - x \tag{2.34}$$

We now turn to a more involved quantity, namely $y = f(x)$. We wish to calculate:

$$\Delta y = \tilde{y} - y = f(\tilde{x}) - f(x) \tag{2.35}$$

What we'll do is to Taylor expand $f(\tilde{x})$ around x. This gives us:

$$\Delta y = f(x + \Delta x) - f(x) = f(x) + \frac{df(x)}{dx}\Delta x + \frac{1}{2}\frac{d^2 f(x)}{dx^2}(\Delta x)^2 + \cdots - f(x) \approx \frac{df(x)}{dx}\Delta x \tag{2.36}$$

In the third step we cancelled the $f(x)$ and disregarded the $(\Delta x)^2$ term and higher-order contributions: assuming Δx is small, this is legitimate. We have thereby shown that:

$$\Delta y \approx \frac{df(x)}{dx}\Delta x \tag{2.37}$$

In other words, the absolute *condition number* is $df(x)/dx$: this determines how strongly the absolute error in x will affect the absolute error in y. If you were faced with, say, $y = x^3$, you could estimate the absolute error in y by taking a derivative: $\Delta y \approx 3x^2\Delta x$. Obviously, when x is large the absolute error Δx gets amplified due to the presence of $3x^2$ in front.

A formal aside: let's define the *forward error* to be $\tilde{f}(x) - f(x)$; here x is the exact problem, $f(x)$ is the exact solution to the exact problem, and $\tilde{f}(x)$ is an approximate solution to the exact problem. Then, to take $f(\tilde{x}) = \tilde{f}(x)$ $(= \tilde{y})$ is to search for the approximate problem $\tilde{x}$ for which the exact solution $(f(\tilde{x}))$ is equal to the approximate solution to the exact problem $(\tilde{f}(x))$. In that case, $\tilde{x} - x$ is known as the *backward error* and bounding it is known as *backward error analysis*. As an example, let's tackle $f(x) = \sqrt{x}$ at $x = 3$, implying $y = \sqrt{3}$. Take $\tilde{y} = \mathbf{\sqrt{3}} = 1.8$ to be the result of our own (poor) approximation to the square-root function. The forward error is $\tilde{f}(x) - f(x) = \mathbf{\sqrt{3}} - \sqrt{3} \approx 0.068$; we notice that $f(\tilde{x}) = \sqrt{3.24} = 1.8 = \mathbf{\sqrt{3}} = \tilde{f}(x)$, so the backward error is $\tilde{x} - x = 0.24$.

It is straightforward to use Eq. (2.37) to get an estimate for the relative error in y:

$$\delta y = \frac{\Delta y}{y} \approx \frac{1}{f(x)}\frac{df(x)}{dx}\Delta x \tag{2.38}$$

If we now multiply and divide by x (assuming, of course, that $x \neq 0$) we find:

$$\delta y \approx \frac{x}{f(x)} \frac{df(x)}{dx} \delta x \tag{2.39}$$

which is our desired relation connecting δy with δx. Analogously to what we saw for the case of the absolute error, our finding here shows us that the coefficient in front of δx determines how strongly the relative error in x will affect the relative error in y. For example, if you were faced with $y = x^4$, then you could estimate the relative error in y as follows:

$$\delta y \approx \frac{x}{x^4} 4x^3 \delta x = 4\delta x \tag{2.40}$$

that is, the relative error in y is worse than the relative error in x, but not dramatically so.[11]

2.2.2.5 General Error Propagation: Many Variables

For future reference, we oberve that it is reasonably straightforward to generalize our approach above to the case of a function of many variables, i.e., $y = f(x_0, x_1, \ldots, x_{n-1})$: the total error Δy would then have contributions from each Δx_i and each partial derivative:

$$\Delta y \approx \sum_{i=0}^{n-1} \frac{\partial f}{\partial x_i} \Delta x_i \tag{2.41}$$

This is a general formula, which can be applied to functions of varying complexity. As a trivial check, we consider the case:

$$y = x_0 - x_1 \tag{2.42}$$

which a moment's consideration will convince you is nothing other than the subtraction of two numbers, as per Eq. (2.16). Applying our new general result to this simple case:

$$\Delta y \approx \Delta x_0 - \Delta x_1 \tag{2.43}$$

which is precisely the result we found in Eq. (2.19).

Equation (2.41) can now be used to produce a relationship for the relative error:

$$\delta y \approx \sum_{i=0}^{n-1} \frac{x_i}{f(x_0, x_1, \ldots, x_{n-1})} \frac{\partial f}{\partial x_i} \delta x_i \tag{2.44}$$

You should be able to see that this formula can be applied to almost all possible scenarios.[12] An elementary test would be to take:

$$y = x_0 x_1 \tag{2.45}$$

[11] You will apply both Eq. (2.37) and Eq. (2.39) to other functions when you solve problem 2.3.

[12] Of course, in deriving our last result we assumed that $y \neq 0$ and that $x_i \neq 0$.

which is simply the multiplication of two numbers, as per Eq. (2.27). Applying our new general result for the relative error to this simple case, we find:

$$\delta y \approx \frac{x_0}{x_0 x_1} x_1 \delta x_0 + \frac{x_1}{x_0 x_1} x_0 \delta x_1 = \delta x_0 + \delta x_1 \tag{2.46}$$

which is precisely the result in Eq. (2.30).

You will benefit from trying out more complicated cases, e.g., $y = \sqrt{x_0} + x_1^3 \log x_2$. In your resulting expression for δy, the coefficient in front of each δx_i tells you by how much (or if) the corresponding relative error is amplified.

2.3 Representing Real Numbers

Our discussion so far has been general: the source of the error, leading to an absolute or relative error or error bound, has not been specified. At this point, we will turn our attention to roundoff errors. In order to do that, we first go over the representation of real numbers on the computer and then discuss simple examples of mathematical operations. For the rest of the chapter, we will work on roundoff error alone: this will result from the representation of a number (i.e., storing that number) or the representation of an operation (e.g., carrying out a subtraction).

2.3.1 Basics

Computers use electrical circuits, which communicate using signals. The simplest such signals are *on* and *off*. These two possibilities are encoded in what is known as a *binary digit or bit*: bits can take on only two possible values, by convention 0 or 1.[13] All types of numbers are stored in binary form, i.e., as collections of 0s and 1s.

Python integers actually have unlimited precision, so we won't have to worry about them too much. In this book, we mainly deal with real numbers, so let's briefly see how those are represented on the computer. Most commonly, real numbers are stored using *floating-point representation*. This has the general form:

$$\pm \text{ mantissa} \times 10^{\text{exponent}} \tag{2.47}$$

For example, the speed of light in scientific notation is $+2.997\,924\,58 \times 10^8$ m/s.

Computers only store a finite number of bits, so cannot store exactly all possible real numbers. In other words, there are "only" finitely many exact representations/*machine numbers*.[14] These come in two varieties: normal and subnormal numbers. There are three ways of losing precision, as shown qualitatively in Fig. 2.2: *underflow* for very small numbers, *overflow* for very large numbers, and *rounding* for decimal numbers whose value falls

[13] You can contrast this to decimal digits, which can take on 10 different values, from 0 to 9.

[14] That is, finitely many decimal numbers that can be stored exactly using a floating-point representation.

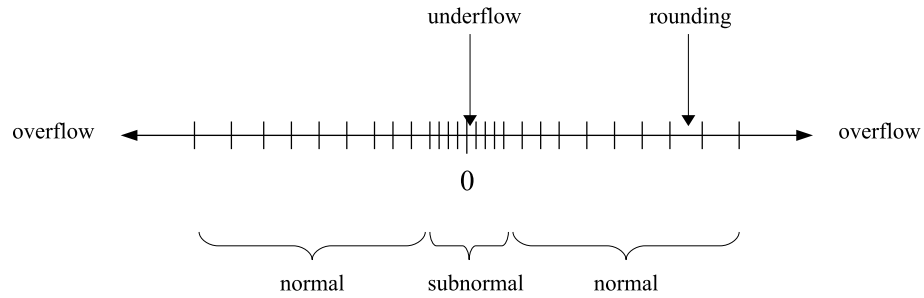


Fig. 2.2 Illustration of exactly representable floating-point numbers

between two exactly representable numbers. For more on these topics, you should look at appendix B; here we limit ourselves to simply quoting some results.

Python employs what are known as *double-precision floating point numbers*, also called *doubles*; their storage uses 64 bits in total. Doubles can represent:

$$\pm 4.9 \times 10^{-324} \leftrightarrow \pm 1.8 \times 10^{308} \tag{2.48}$$

This refers to the ability to store very large or very small numbers. Most of this ability is found in the term corresponding to the exponent. For doubles, if we try to represent a number that's larger than 1.8×10^{308} we get *overflow*. Similarly, if we try to represent a number that's smaller than 4.9×10^{-324} we get *underflow*. Keep in mind that being able to represent 4.9×10^{-324} does *not* mean that we are able to store 324 significant figures in a double. The number of significant figures (and the related concept of *precision*) is found in the coefficient in front (e.g., `1.8` or `1.234567`). For doubles, the precision is 1 part in $2^{52} \approx 2.2 \times 10^{-16}$, which amounts to 15 or 16 decimal digits.

2.3.2 Overflow

We can explore the above results programmatically. We will start from an appropriately large value, in order to shorten the number of output lines:

```
>>> large = 2.**1021
for i in range(3):
...       large *= 2
...       print(i, large)
0 4.49423283715579e+307
1 8.98846567431158e+307
2 inf
```

This is what we expected: from $2^{1024} \approx 1.7976 \times 10^{308}$ and onward we are no longer able to store the result in a double. You can check this by saying:

```
>>> 8.98846567431158e+307*1.999999999999999
1.797693134862315e+308
```

Multiplying by 2 would have led to `inf`, as above. Problem 2.4 investigates when underflow occurs.

2.3.3 Machine Precision

We already mentioned that the precision for doubles is limited to $\approx 2.2 \times 10^{-16}$. The precision is related to the distance between two vertical lines in the figure above for a given region of interest: as we noted, anything between the two lines gets rounded, either to the left or to the right. We now turn to the question of carrying out arithmetic operations using such numbers: this gives rise to the all-important question of *rounding*. This question arises every time we are trying to combine two floating-point numbers but the answer is not an exactly representable floating-point number. For example, 1 and 10 can be exactly represented as doubles, but $1/10$ cannot.

We first address an even simpler problem: five-digit decimal arithmetic. Let's assume we want to add together the two numbers 0.12345 and 1.2345. One could notice that $1.2345 = 0.12345 \times 10^1$ while $0.12345 = 0.12345 \times 10^0$, i.e., in an (imagined) system that used five-digit decimal arithmetic these two numbers would have the same mantissa and different exponents. However, that doesn't help us when adding: to add two mantissas we have to align them to use the same exponent (since that's how addition works). Adding them as real numbers (i.e., not five-digit decimal numbers) we get:

$$0.12345 + 1.2345 = 1.35795 \tag{2.49}$$

But our answer now contains six decimal digits, 1.35795. Since we're limited to five-digit decimal numbers, this leaves us with the option of *chopping* the result down to 1.3579 or *rounding* it up to 1.3580. Problems like this one also appear in other arithmetic operations and for other representational systems (like binary).

Turning back to the question of the machine representation of doubles, we try to make the concept of "precision" more specific. We define the *machine precision* ϵ_m as follows: it is the gap between the number 1.0, on the one hand, and the smallest possible number $\tilde{x}$ that is larger than 1.0 ($\tilde{x} > 1.0$), on the other hand. If you have read appendix B, you should know that (given the form of the mantissa for normal doubles) the smallest number we can represent obeying $\tilde{x} > 1.0$ is $1 + 2^{-52}$. The gap between this number and 1 is $2^{-52} \approx 2.2 \times 10^{-16}$. In other words:

$$\epsilon_m \approx 2.2 \times 10^{-16} \tag{2.50}$$

We will make use of this repeatedly in what follows.

Instead of delving into a study of binary arithmetic (analogous to what we did above for five-digit decimal arithmetic), let us investigate the definition of machine precision using

Python. We start with a small number and keep halving it, after which operation we add it on to `1.0`: at some point, this is going to give us back `1.0`: we then call the gap between `1.0` and the last number > 1 the machine precision. Explicitly:

```
>>> small = 1/2**50
>>> for i in range(3):
...       small /= 2
...       print(i, 1 + small, small)
0 1.0000000000000004 4.440892098500626e-16
1 1.0000000000000002 2.220446049250313e-16
2 1.0 1.1102230246251565e-16
```

As you can see, we started `small` at an appropriately small value, in order to shorten the number of output lines. We can further explore this topic interactively. At first sight, the results below might be confusing:

```
>>> 1. + 2.3e-16
1.0000000000000002
>>> 1. + 1.6e-16
1.0000000000000002
>>> 1. + 1.12e-16
1.0000000000000002
>>> 1. + 1.1e-16
1.0
```

We found that there exist numbers smaller than `2.2e-16` that when added to 1 lead to a result that is larger than 1. If you pay close attention, you will notice that `1. + 1.6e-16` or `1. + 1.12e-16` are rounded to the same number that `1. + 2.22e-16` corresponds to (namely, `1.0000000000000002`). However, below a certain point, we start rounding down to 1. In other words, for some values of `small` below the machine precision, we have a computed value of `1.+small` that is not 1, but corresponds to $1 + \epsilon_m$.[15]

Take a moment to appreciate that you can use a double to store a tiny number like 10^{-300}, but this doesn't mean that you can store $1 + 10^{-300}$: to do so, you would need 301 digits of precision (and all you have is 16).

2.3.4 Revisiting Subtraction

We discussed in an earlier section how bad catastrophic cancellation can be. At the time, we were investigating general errors, which could have come from several sources. Let us try to specifically investigate what happens in the case of subtraction when the *only* errors involved are those due to roundoff, i.e., let us assume that the relative error δx is a consequence of the fact that, generally speaking, x cannot be represented exactly on the

[15] Some authors call the smallest value of `small` for which `1.+small` doesn't round down to 1 the *unit roundoff* u: it is easy to see that this is related to the machine precision by $\epsilon_m = 2u$.

computer (i.e., it is typically not a machine number). Thus, when replacing x by its nearest double-precision floating-point number, we make a roundoff error: without getting into more detail, we will estimate the relative error's magnitude using the machine precision, which as we saw earlier is $\epsilon_m \approx 2.2 \times 10^{-16}$.[16] Obviously, this is only an estimate: for example, the error made when rounding to a subnormal number may be smaller (since subnormals are more closely spaced). Another example: if x is a machine number, then it can be exactly represented by $\tilde{x}$, so the relative error is actually 0. Our point in using ϵ_m is simply that one should typically not trust floating-point numbers for more than 15–16 (relative) decimal digits of precision.[17] If you think of ϵ_m as an error bound, then the fact that sometimes the error is smaller is OK.

Since we will estimate the relative error δx using ϵ_m, we see that the absolute error can be considerably larger: from Eq. (2.12) we know that $\Delta x = x\delta x$, so if x is very large then since we are taking $\epsilon_m \approx 2.2 \times 10^{-16}$ as being fixed at that value, it's obvious that Δx can become quite large. Let's take a specific example: given a relative error $\approx 10^{-16}$, a specific double of magnitude $\approx 10^{22}$ will have an absolute error in its last digit, of order 10^6.

Given that we use ϵ_m to find a general relative error in representing a real number via the use of a floating-point number, we can re-express our result for catastrophic cancellation from Eq. (2.24) as:

$$|\delta x| \leq \frac{|a|}{|a-b|}\, 2\epsilon_m \tag{2.51}$$

Due to $|a|/|a-b|$, even the *relative* error can be much larger than ϵ_m.

It might be worth making a distinction at this point between: (a) the loss of significant figures when subtracting two nearly equal numbers, and (b) the loss of even more digits when carrying out the subtraction using floating-point numbers (which have finite precision). Let's start from the first case. Subtract two nearly equal numbers, each of which has 20 significant figures:

$$1.2345678912345678912 - 1.2345678900000000000 = 0.0000000012345678912 \tag{2.52}$$

Note that here we subtracted real numbers (not floating-point representations) and wrote out the answer explicitly. It's easy to see that we started with 20 significant figures and ended up with 11 significant figures, even though we're dealing with real numbers/infinite precision. We now turn to the second case, which is the carrying out of this subtraction using floating-point numbers. We can use Python (doubles) to be explicit. We have:

```
>>> 1.2345678912345678912 - 1.2345678900000000000
1.234568003383174e-09
```

Comparing to the answer we had above (for real numbers), we see that we only match the first 6 (out of 11) significant figures. This is partly a result of the fact that each of our initial numbers is not represented on the computer using the full 20 significant figures, but only at most 16 digits. Explicitly:

[16] Note, however, that ϵ_m was defined in a related but slighty different context (for values around 1.0).

[17] Incidentally, though we loosely use the term significant figures here and elsewhere, it's better to keep in mind the relative error, instead, which is a more precise and base-independent measure.

```
>>> 1.2345678912345678912
1.234567891234568
>>> 1.2345678900000000000
1.23456789
>>> 1.234567891234568 - 1.23456789
1.234568003383174e-09
```

This shows that we lose precision in the first number, which then has to lead to loss of precision in the result for the subtraction.

It's easy to see that things are even worse than that, though: using real numbers, the subtraction $1.234567891234568 - 1.23456789$ would give us 0.000000001234568. Instead of that, we get `1.234568003383174e-09`. This is not hard to understand: a number like `1.234567891234568` typically has an absolute error in the last digit, i.e., of the order of 10^{-15}, so the result of the subtraction generally cannot be trusted beyond that absolute order (`1.234568003383174e-09` can be rewritten as `1234568.003383174e-15`). This is a result of the fact that in addition or subtraction the *absolute* error in the result comes from adding the absolute errors in the two numbers.

Just in case the conclusion is still not "clicking", let us try to discuss what's going on using relative errors. Here the exact number is $x = 0.0000000012345678912$ while the approximate value is $\tilde{x} = 0.000000001234568003383174$. Since this is one of those situations where we can actually evaluate the error, let us do so. The definition in Eq. (2.12) gives us $\delta x \approx 9.08684 \times 10^{-8}$. Again, observe that we started with two numbers with relative errors of order 10^{-16}, but subtracting them led to a relative error of order roughly 10^{-7}. Another way to look at this result is to say that we have explicitly evaluated the left-hand side of Eq. (2.51), δx. Now, let us evaluate the right-hand side of that equation. Here $a = 1.2345678912345678912$ and $a - b = 0.0000000012345678912$. The right-hand side comes out to be $|a|\, 2\epsilon_m/|a-b| \approx 2.2 \times 10^{-7}$. Thus, we find that the inequality is obeyed (of course), but the actual relative error is a factor of a few smaller than what the error bound would lead us to believe. This isn't too hard to explain: most obviously, the right-hand side of Eq. (2.51) contains a 2, stemming from the assumption that both $\tilde{a}$ and $\tilde{b}$ have a relative error of ϵ_m, but in our case b didn't change when we typed it into the Python interpreter.

You should repeat the above exercise (or something like it) for the cases of addition, multiplication, or division. It's easy to come up with examples where you add two numbers, one of which is poorly constrained, and then you get a large absolute error in the result (but still not as dramatic as in catastrophic cancellation). On the other hand, since in multiplication and division only the relative errors add up, you can convince yourself that since your starting numbers each have a relative error bound of roughly 10^{-16}, the error bound for the result is at worst twice that, which is typically not that bad.

2.3.5 Comparing Floats

Since only machine numbers can be represented exactly (other numbers are rounded to the nearest machine number), we have to be careful when comparing floating-point numbers.

We won't go into the question of how operations with floating-point numbers are actually implemented, but some examples may help explain the core issues.

Specifically, you should (almost) never compare two floating point variables $\tilde{x}$ and $\tilde{y}$ for equality: you might have an analytical expectation that the corresponding two real numbers x and y should be the same, but if the values of $\tilde{x}$ and $\tilde{y}$ are arrived at via different routes, their floating-point representations may well be different. A famous example:

```
>>> xt = 0.1 + 0.2
>>> yt = 0.3
>>> xt == yt
False
>>> xt
0.30000000000000004
>>> yt
0.3
```

The solution is to (almost) never compare two floating-point variables for equality: instead, take the absolute value of their difference, and check if that is smaller than some acceptable threshold, e.g., 10^{-10} or 10^{-12}. To apply this to our example above:

```
>>> abs(xt-yt)
5.551115123125783e-17
>>> abs(xt-yt) < 1.e-12
True
```

which behaves as one would expect.[18]

The above recipe (called an *absolute epsilon*) is fine when comparing natural-sized numbers. However, there are situations where it can lead us astray. For example:

```
>>> xt = 12345678912.345
>>> yt = 12345678912.346
>>> abs(xt-yt)
0.0010013580322265625
>>> abs(xt-yt) < 1.e-12
False
```

This makes it look like these two numbers are really different from each other, though it's plain to see that they aren't. The solution is to employ a *relative epsilon*: instead of comparing the two numbers to check whether they match up to a given small number, take into account the magnitude of the numbers themselves, thereby making the comparison relative. To do so, we first introduce a helper function:

[18] Well, almost: if you were carrying out the subtraction between 0.30000000000000004 and 0.3 using real numbers, you would expect their difference to be 0.00000000000000004. Instead, since they are floats, the answer turns out to be `5.551115123125783e-17`.

```
>>> def findmax(x,y):
...       return max(abs(x),abs(y))
```

This picks the largest magnitude, which we then use to carry out our comparison:

```
>>> xt = 12345678912.345
>>> yt = 12345678912.346
>>> abs(xt-yt)/findmax(xt,yt) < 1.e-12
True
```

Finally, note that there are situations where it's perfectly fine to compare two floats for equality, e.g., `while 1.+small != 1.:`. This specific comparison works: 1.0 is exactly representable in double-precision, so the only scenario where `1.+small` is equal to `1.0` is when the result gets rounded to `1.0`. Of course, this codicil to our rule (you can't compare two floats for equality, except when you can) in practice appears most often when comparing a variable to a literal: there's nothing wrong with saying `if xt == 10.0:` since 10.0 is a machine number and `xt` can plausibly round up or down to that value.

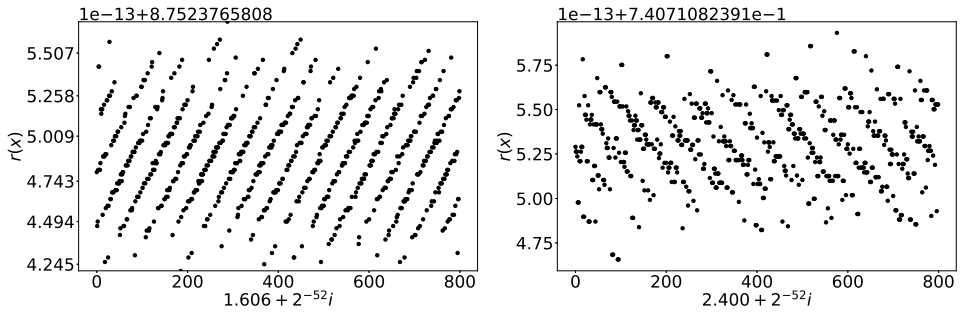
2.4 Rounding Errors in the Wild

Most of what we've had to say about roundoff error up to this point has focused on a single elementary mathematical operation (e.g., one subtraction or one addition). Of course, in actual applications one is faced with many more calculations (e.g., taking the square root, exponentiating), often carried out in sequence. It is often said that rounding error for the case where many iterations are involved leads to roundoff error buildup. This is not incorrect, but more often than not we are faced with one or two iterations that cause a problem, which can typically not be undone after that point. Thus, in the present section we turn to a study of more involved cases of rounding error.

2.4.1 Are Roundoff Errors Random?

At this point, many texts on computational science discuss a standard problem, that of roundoff error propagation when trying to evaluate the sum of n numbers x_0, x_1, ..., x_{n-1}. One way to go about this is by applying our discussion from section 2.2.2.1. If each number has the same error bound ϵ, it's easy to convince yourself that since the absolute errors will add up, you will be left with a total error bound that is $n\epsilon$. This is most likely too pessimistic: it's hard to believe that the errors for n numbers would never cancel, i.e., they would all have the same sign and maximal magnitude.

What's frequently done, instead, is to assume that the errors in the different terms are independent, use the theory of random variables, and derive a result for the scaling with n of the absolute or relative error for the sum $\sum_{i=0}^{n-1} x_i$. We will address essentially the same



Value of rational function discussed in the main text, at two different regions **Fig. 2.3**

problem when we introduce the central limit theorem in problem 6.47. There we will find that, if ϵ is the standard deviation for one number, then the standard error for the sum turns out to be $\sqrt{n}\epsilon$. For large n, it's clear that $\sqrt{n}$ grows much more slowly than n. Of course, this is comparing apples (maximal errors) with oranges (standard errors).

One has to pause, however, to think about the assumption that the errors of these x_i's are stochastically independent: many of the examples we will discuss in the rest of this chapter would have been impossible if the errors were independent. To take one case, the next contribution in a Taylor expansion is clearly correlated to the previous contribution. Rounding errors are not random: you get the same (predictable) answer every time you repeat the same computation. If you wish to study them probabilistically, you will need correlated, discrete (i.e., not continuous) random variables. These are points that, while known to experts (see, e.g., N. Higham's book [68]), are too often obscured in introductory textbooks. The confusion may arise from the fact that the roundoff error in the finite-precision *representation* of a real number may be modelled using a simple distribution; this is wholly different from the roundoff error in a *computation* involving floating-point numbers.

Perhaps it's best to consider a specific example.[19] Let's look at the rational function:

$$r(x) = \frac{4x^4 - 59x^3 + 324x^2 - 751x + 622}{x^4 - 14x^3 + 72x^2 - 151x + 112} \tag{2.53}$$

In problem 2.11 you will learn to code up polynomials using an efficient and accurate approach known as Horner's rule. In problem 2.12, you will apply what you've learned to this specific rational function. You will find what's shown in Fig. 2.3: this is clear evidence that our method of evaluating the function is sensitive to roundoff error: the typical error is $\approx 10^{-13}$. What's more, the roundoff error follows striking patterns: the left panel shows that the error is not uniformly random.[20] To highlight the fact that the pattern on the left panel is not simply a fluke, the right panel picks a different region and finds a different pattern (again, a mostly non-random one). In the aforementioned problem, you will not only reproduce these results, but also see how one could do better.

Where does this leave us? Since assuming totally independent standard errors is not warranted and assuming maximal errors is too pessimistic, how do we proceed? The answer is

[19] "When we perceive the whole at once, as in numerical computations, all agree in one judgment" (Samuel Johnson, *The History of Rasselas, Prince of Abyssinia*, chapter XXVIII).

[20] In which case we would be seeing "noise" instead of straight lines.

that one has to approach each problem separately, so there is no general result for the scaling with n (beyond the formal relation in Eq. (2.44)). As noted, often only a few rounding errors are the dominant contributions to the final error, so the question of finding a scaling with n is moot. We hope that by seeing several cases of things going wrong (and how to fix them), the reader will learn to identify the main classes of potential problems.

2.4.2 Compensated Summation

We now turn to a crucial issue regarding operations with floats; in short, due to roundoff errors, when you're dealing with floating-point numbers the associative law of algebra does not necessarily hold. You know that `0.1` added to `0.2` does not give `0.3`, but things are even worse than that: *the result of operations involving floating-point numbers may depend on the order in which these operations are carried out*. Here's a simple example:

```
>>> (0.7 + 0.1) + 0.3
1.0999999999999999
>>> 0.7 + (0.1 + 0.3)
1.1
```

Once again, operations that "should" give the same answers (i.e., that *do* give the same answer when dealing with real numbers) may not. This behavior is more than just a curiosity: it can have real-world consequences. In fact, here's an even more dramatic example:

```
>>> xt = 1.e20; yt = -1.e20; zt = 1.
>>> (xt + yt) + zt
1.0
>>> xt + (yt + zt)
0.0
```

In the first case, the two large numbers, `xt` and `yt`, cancel each other and we are left with the unit as the answer. In the second case, we face a situation similar to that in subsection 2.3.3: adding 1 to the (negative) large number `yt` simply rounds to `yt`; this is analogous to the `1. + small` we encountered on page 48, only this time we're faced with `large + 1.` and it is the unit that is dropped. Then, `xt` and `yt` cancel each other out (as before). If you're finding these examples a bit disconcerting, you are in good company.

Once you think about the problem more carefully, you might reach the conclusion that the issue that arose here is not too problematic: you were summing up numbers of wildly varying magnitudes, so you cannot trust the final answer too much. Unfortunately, as we'll see in section 2.4.4, sometimes you may not even be aware of the fact that the intermediate values in a calculation are large and of opposite signs (leading to cancellations), in which case you might not even know how much you should trust the final answer. A lesson that keeps recurring in this chapter is that you should get used to reasoning about your calculation, in contradistinction to blindly trusting whatever the computer produces.

We don't want to sound too pessimistic, so we will now see how to sum up numbers

kahansum.py **Code 2.1**

```
def kahansum(xs):
      s = 0.; e = 0.
      for x in xs:
            temp = s
            y = x + e
            s = temp + y
            e = (temp - s) + y
      return s

if __name__ == '__main__':
      xs = [0.7, 0.1, 0.3]
      print(sum(xs), kahansum(xs))
```

very accurately. Our task is simply to sum up the elements of a list. In problem 2.15, we will see that often one can simply sort the numbers and then add them up starting with the smallest one. There are, however, scenarios where sorting the numbers to be summed is not only costly but goes against the task you need to carry out. Most notably, when solving initial-value problems in the study of ordinary differential equations (see chapter 8), the terms *have* to be added in the same order as that in which they are produced.

Here we will employ a nice trick, called *compensated summation* or *Kahan summation*. Qualitatively, what this does is to estimate the rounding error in each addition and then compensate for it with a correction term. More specifically, if you are adding together two numbers (a and b) and $\tilde{s}$ is your best floating-point representation for the sum, then if:

$$e = (a - \tilde{s}) + b \tag{2.54}$$

we can compute $\tilde{e}$ to get an estimate of the error $(a+b)-\tilde{s}$, namely the information that was lost when we evaluated $\tilde{s}$. While this doesn't help us when all we're doing is summing two numbers[21] it can really help when we are summing *many* numbers: add in this correction to the next term in your series, before adding that term to the partial sum.

Typically, compensated summation is more accurate when you are adding a large number of terms, but it can also be applied to the case we encountered at the start of the present section. Code 2.1 provides a Python implementation. This does what we described around Eq. (2.54): it estimates the error in the previous addition and then compensates for it. Note how our new function does not need any "length" arguments, since it simply steps through the elements of the list. We then encounter a major new syntactic feature of Python: the line `if __name__ == '__main__':` checks to see if we're running the present file as the main program (which we are). In this case, including this extra check is actually unnecessary: we could have just defined our function and then called it. We will see the importance of

[21] $\tilde{s} + \tilde{e}$ doesn't get you anywhere, since $\tilde{s}$ was already the best we could do!

this further check later, when we wish to call `kahansum()` without running the rest of the code. The output is `1.0999999999999999 1.1`. Thus, this simple function turns out to cure the problem we encountered earlier on. We don't want to spend too much time on the topic, but you should play around with compensated summation, trying to find cases where the direct sum does a poor job (here's another example: `xs = [123456789 + 0.01*i for i in range(10)]`).

Even our progress has its limitations: if you take `xs = [1.e20, 1., -1.e20]`, which is (a modified version of) the second example from the start of this section, you will see that compensated summation doesn't lead to improved accuracy. In general, if:

$$\sum_i |x_i| \gg \left| \sum_i x_i \right| \tag{2.55}$$

then compensated summation is not guaranteed to give a small relative error. On a different note, Kahan summation requires more computations than regular summation: this performance penalty won't matter to us in this book, but it may matter in real-world applications.

2.4.3 Naive vs Manipulated Expressions

We now go over a simple example showing how easy it is to lose accuracy if one is not careful; at the same time, we will see how straightforward it is to carry out an analytical manipulation that avoids the problem. The task at hand is to evaluate the function:

$$f(x) = \frac{1}{\sqrt{x^2+1} - x} \tag{2.56}$$

for large values of x. A Python implementation using list comprehensions is given in code 2.2. The output of running this code is:

```
10000 19999.99977764674
100000 200000.22333140278
1000000 1999984.77112922
10000000 19884107.85185185
```

The answer appears to be getting increasingly worse as the x is increased. Well, maybe: this all depends on what we expect the correct answer to be. On the other hand, the code/expression we are using is certainly not robust, as you can easily see by running the test case of $x = 10^8$. In Python this leads to a `ZeroDivisionError` since the terms in the denominator are evaluated as being equal. This is happening because for large values of x, we know that $x^2 + 1 \approx x^2$. We need to evaluate the square root very accurately if we want to be able to subtract a nearly equal number from it.

An easy way to avoid this problem consists of rewriting the starting expression:

$$f(x) = \frac{1}{\sqrt{x^2+1} - x} = \frac{\sqrt{x^2+1} + x}{(\sqrt{x^2+1} - x)(\sqrt{x^2+1} + x)} = \frac{\sqrt{x^2+1} + x}{x^2 + 1 - x^2} = \sqrt{x^2+1} + x \tag{2.57}$$

naiveval.py **Code 2.2**

```
from math import sqrt

def naiveval(x):
      return 1/(sqrt(x**2 + 1) - x)

xs = [10**i for i in range(4,8)]
ys = [naiveval(x) for x in xs]
for x, y in zip(xs, ys):
      print(x, y)
```

In the second equality we multiplied numerator and denominator by the same expression. In the third equality we used a well-known identity in the denominator. In the fourth equality we cancelled terms in the denominator. Notice that this expression no longer requires a subtraction. If you implement the new expression, you will get the output:

```
10000 20000.000050000002
100000 200000.00000499998
1000000 2000000.0000005001
10000000 20000000.000000052
```

The errors now behave much better: for $x \gg 1$ we have $x^2 + 1 \approx x^2$, so we are essentially printing out $2x$. There are several other cases where a simple rewriting of the initial expression can avoid bad numerical accuracy issues (often by avoiding a subtraction).

2.4.4 Computing the Exponential Function

We now turn to an example where several calculations are carried out in sequence. Thus, if something goes wrong we must carefully sift through intermediate results to see what went wrong (and when). We focus on the task of computing the exponential function (assuming we have no access to a math library), by using the Taylor/Maclaurin series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{2.58}$$

We're clearly not going to sum infinitely many terms, so we approximate this expansion by keeping only the terms up to n_{max}:

$$e^x \approx \sum_{n=0}^{n_{max}} \frac{x^n}{n!} \tag{2.59}$$

A naive implementation of this algorithm would divide x raised to increasingly large powers by increasingly large factorials, summing the result of such divisions up to a specified