Clemens Heitzinger

Algorithms with JULIA

Optimization, Machine Learning, and Differential Equations Using the JULIA Language

Springer

Algorithms with JULIA

Clemens Heitzinger

Algorithms with JULIA

Optimization, Machine Learning, and Differential Equations Using the JULIA Language

Springer

Clemens Heitzinger
Center for Artificial Intelligence
and Machine Learning (CAIML)
and
Department of Mathematics
and Geoinformation
Technische Universität Wien
Vienna, Austria

ISBN 978-3-031-16559-7 ISBN 978-3-031-16560-3 (eBook)
https://doi.org/10.1007/978-3-031-16560-3

Mathematics Subject Classification: 65-XX, 34K28, 65Mxx, 65M06, 65M08, 65M60, 65Kxx, 65Yxx, 62M45, 68T05

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

To M.S.

Foreword

Students of applied mathematics are often confronted with textbooks that either cover the mathematical principles and concepts of mathematical models or with textbooks which introduce the basic language structures of a programming language. Many authors fail to cover the underlying mathematical theory of models, which is crucial in understanding the applicability of models – and especially their limitations – to real world problems. On the other hand, many textbooks and monographs fail to address the crucial step from the algorithmic formulation of an applied problem to the actual implementation and solution in the form of an executable program. This book brilliantly combines these two aspects using a high level open source computer language and covers many areas of continuum model based areas of natural and social sciences, applied mathematics and engineering. JULIA is a high-level, high-performance, dynamic programming language which can be used to write any application in numerical analysis and computational science. Clemens Heitzinger has gone through great lengths to organize this book into sequences that make sense for the beginner as well as for the expert in one particular field.

The applied topics are carefully chosen, from the most relevant standard areas like ordinary and partial differential equations and optimization to more recent fields of interest like machine learning and neural networks. The chapters on ordinary and partial differential equations include examples of how to use existing packages included in the JULIA software. In the chapter about optimization the methods for standard local optimization are nicely explained. However, this book also contains a very relevant chapter about global optimization, including methods such as simulated annealing and agent based optimization algorithms. All this is not something usually found in the same book. Again, the global optimization theory, as far as the general theory exists, is well presented and the application examples (and, most importantly, the benchmark problems) are well chosen. One chapter – concerned with the currently maybe most relevant area – introduces practical problem solving in the field of machine learning. The author covers the basic approach of learning via artificial neural networks as well as probabilistic methods based on Bayesian theory. Again, the topics and examples are well chosen, the underlying theory is well explained, and the solutions of the chosen application problems are immediately implementable in JULIA.

Clemens Heitzinger has been involved in some of the most impressive applications in engineering and the applied physical sciences, covering microelectronics, sensors and biomedical applications. This book covers both the theoretical and practical aspects of this part of modern science. The approach taken in this book is novel in the sense that it goes into quite some detail in the theoretical background, while at the same time being based on a modern computing platform. In a sense, this work serves the role of two books. It is much more than a cookbook for "how to solve problems with JULIA," but also a good introduction to the most relevant problems in continuum model based science and engineering. At the same time, it gives the novice in JULIA programming a good introduction on how to use this higher level programming language. It can therefore be used as a text for students in an advanced graduate level course as well as a monograph by the researcher planning to solve actual problems by programming in JULIA.

Tempe, April 2022 *Christian Ringhofer*

Preface

Why computation? The middle of the last century marks the beginning of a new era in the history of mathematics. Although calculators and computers had been envisioned centuries before and mechanical calculators or calculating machines were in widespread use already in the nineteenth century, only the invention of purely electronic computers made it possible to perform calculations on increasingly large scales. The reason is quite simple: mechanical and electromechanical calculators (see Fig. 0.1) are severely limited by friction.

The advent of electronic computers and later the rise of the integrated circuit have resulted in portable devices of astounding computational power at tiny power consumption (see Fig. 0.2). Computations that were unthinkable a few decades ago can now be performed at low cost and at great speed. These developments in the physical realm have resulted in the birth of new mathematical disciplines. Computer algebra, scientific computing, machine learning, artificial intelligence, and related areas are concerned with solving abstract mathematical problems as well as scientific and data-science problems correctly, precisely, and efficiently.

Although lots of computational power are available today, fundamental questions will always have to be answered. How should the computations be structured? What are the advantages and disadvantages of various algorithms? How accurate will the results be? How can we best take advantage of the computational resources available to us? These are fundamental questions that lead to new and fascinating mathematical problems. In this sense, the invention of electronic computers has had and will have a twofold influence on mathematics: computers are both an enabling technology and a source of new mathematical problems.

There is no doubt that computers and mathematical algorithms have impacted our lives in many ways. In many engineering disciplines, it has become common to perform simulations for the rational design and the optimization of all kinds of devices and processes. Simulations can be much cheaper than performing many experiments and they provide theoretical and quantitative insights. Examples are airplanes, combustion engines, antennas, and the construc-

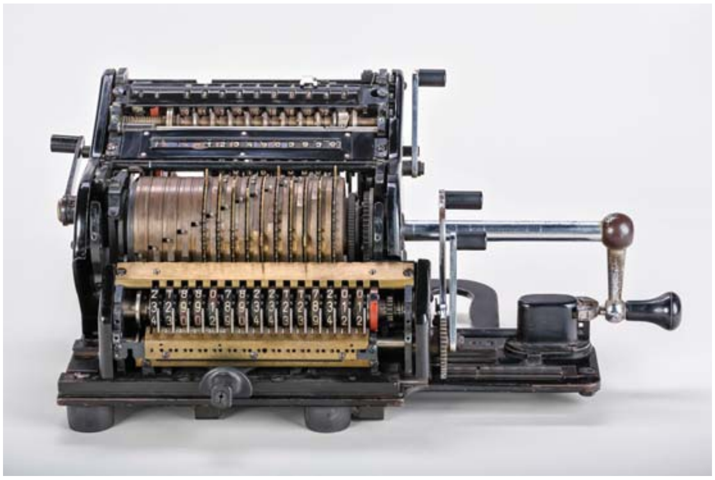

Fig. 0.1 A Brunsviga 15 mechanical calculator, produced by Brunsviga-Maschinenwerke Grimme, Natalis & Co. AG, Braunschweig, between 1934 and 1947. The shrouds are removed to reveal the internal mechanic. (Photo by CEphoto, Uwe Aranas, no changes, license CC-BY-SA-3.0, https://creativecommons.org/licenses/by-sa/3.0/.)

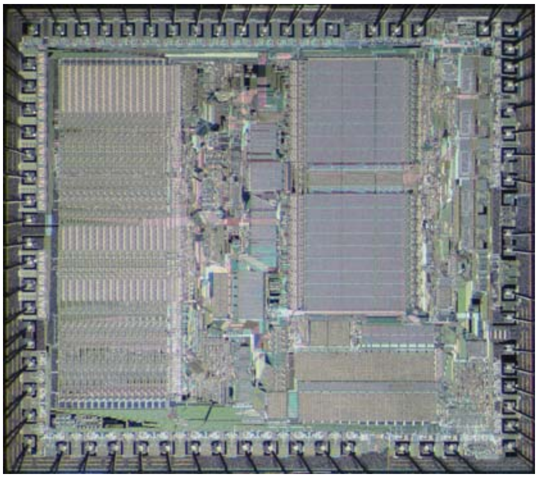

Fig. 0.2 Motorola 68000 CPU. (Photo by Pauli Rautakorpi, no changes, license CC-BY-3.0, https://creativecommons.org/licenses/by/3.0/.)

tion of bridges and other buildings. Large-scale computations are also behind search engines, financial services, and other data-intensive industries. There is probably not a single hour in our daily lives when we do not use a service or a device that has only become possible by computers and mathematical algorithms or that has been much improved by them.

Using this book, you will learn a modern, general-purpose, and efficient programming language, namely JULIA, as well as some of the most important methods in optimization, machine learning, and differential equations and how they work. These three fields, optimization, machine learning, and differential equations, have been chosen because they cover a wide range of computational tasks in science, engineering, and industry.

Methods and algorithms in these areas will be discussed in sufficient detail to arrive at a complete understanding. You will understand how the computational approaches work starting from the basic mathematical theory. Important results and proofs will be given in each chapter (and can be skipped on first reading). Based on these foundations, you will be provided with the knowledge to implement the algorithms and your own variants. To this end, sample programs and hints for implementation in JULIA are provided.

The ultimate purpose of this book is to provide the reader both with a working knowledge of the JULIA programming languages as well as with more than a superficial understanding of modern topics in three important computational fields. Using the algorithms and the sample codes for leading problems, you will be able to translate the theory into working knowledge in order to solve your scientific, engineering, mathematical, or industrial problems.

How is this book unique? This book strives to provide a modern, practical, and well founded perspective on algorithms in optimization, machine-learning, and differential equations. Hence there are two points how the present book differs from other books in this area.

First, the topics were selected with a modern view of computation in mind. As mathematics and computation evolve, we are able to solve more and more difficult problems. These advances are reflected in the material in this book. For example, topics such as artificial neural networks, computational Bayesian estimation, and partial differential equations are discussed, but numerically solving systems of linear equations is not, since you will most likely not write your own program to do so due the availability of well tested libraries (also immediately available in JULIA).

Optimization is of great value in almost all disciplines. Differential equations are of great utility in many cases where fundamental relationships between the known and unknown variables exist, for example in physics, chemistry, and many engineering disciplines. Furthermore, machine learning in particular is an area that has benefited from increases in computational power and available memory and that is of utmost importance when large amounts of data are available, but fundamental relationships are unknown.

Second, the JULIA language, a rather young language designed with scientific and technical computing in mind, is used to implement the algorithms. Its implementation includes a compiler and a type system that leads to fast compiled code. It builds on modern and general programming concepts so that it is usable for many different purposes. It comes with linear-algebra algorithms, sparse matrices, and a package system. It is open source and its syntax is easy to learn. Because of these reasons, it has quickly gained popularity in scientific computing and machine learning.

In a larger historic perspective, there has been a divide between the FORTRAN and the LISP families of languages since the early days of computing. JULIA can be seen as a successor of FORTRAN in the path of MATLAB, while it incorporates several important concepts from languages in the LISP family. From this point of view, it can be seen as the intersection of two very different paths in the history of programming languages.

For whom is this book useful? The book has been written with various audiences in mind, namely

- mathematicians,
- computer scientists, and
- computational scientists and engineers of any kind.

I have tried to combine the mathematical, algorithmic, and computational aspects into one exposition, because this intersection is challenging, exciting, and useful. These three aspects must be included in holistic views of the various subjects in this book, and only a holistic view can provide deeper understanding and appreciation of the results and algorithms and the paths that lead there.

The prerequisites are the usual ones for a book in these areas: a certain mathematical proficiency is required in the form of the basics of linear algebra and calculus. Proficiency in programming is helpful, but not required, when you take the next step and begin to implement algorithms. A prior knowledge of JULIA is not required, as its main as well as several of its advanced features are presented in this book.

The book is perfectly readable and usable by skipping some details and sections marked as advanced material on first reading. In any case, stories as complete as possible are told in each chapter in order to meet the intellectual demand of a complete and self-contained treatment.

Therefore, in order to provide a complete and self-contained exposition, proofs of the main results are included. They can be skipped, however, altogether by readers who are not interested in the proofs or in classes where they are not required. The level of exposition is mathematically rigorous, yet care was taken that it remains accessible to a large audience with a background in linear algebra and calculus.

Most readers will eventually be interested in quantitative solutions of their mathematical problems. This means that the theoretic mathematical results should be translated into correct and hopefully also efficient computer program. To this end, the theoretical background is accompanied not only by algorithms, but in some cases also by sample programs that illustrate the best practice in implementation. Various exercises have been added in order to assist exploring the variants of the algorithms and related problems.

In summary, this book has been written with a few goals in mind. First, the choice of mathematical subjects and computational problems is modern and relevant in the practice of applied mathematics, computer science, and engineering.

Second, the book teaches a modern programming language that is especially useful in technical and scientific applications, while also providing high-level and advanced programming concepts. Third, in addition to self-study, the book can be used as a textbook for courses in these areas. By choosing the chapters of interest, the course can be tailored to various needs. The exercises deepen the theory and help practice translating the theory into useful programs.

Acknowledgments. Finally, it is my pleasure to acknowledge the interest and support by Klaus Stricker and his department. I would also like to acknowledge the students in Vienna who helped improve the manuscript.

Wien, March 2022 *Clemens Heitzinger*

Contents

Part I
The JULIA Language

Chapter 1
An Introduction to the JULIA Language

Dicebat Bernardus Carnotensis
nos esse quasi nanos gigantium humeris insidentes,
ut possimus plura eis et remotiora videre,
non utique proprii visus acumine aut eminentia corporis,
sed quia in altum subvehimur et extollimur magnitudine gigantea.

—John of Salisbury, *Metalogicon* (1159)

Abstract This chapter provides a first introduction to the JULIA language. A brief historic overview of programming languages is given, which is important to understand the dichotomy between static programming languages such as FORTRAN and dynamic languages such as LISP as well as their uses. Then an overview of JULIA discusses some important design choices and major attributes of the language as well as why and how it is useful and meets today's requirements. Finally, we learn how to start JULIA, how to interact with it, how to run JULIA programs, and how to install packages. Finally, various development environments are presented, and commands for accessing help and documentation from within the JULIA system itself are explained.

1.1 Brief Historic Overview

The calculations an electronic computers performs are eventually performed by the movement of electrons. Therefore the execution of any computer program that implements an algorithm depends on several layers of hard- and software. Transistors, which gate electron flow, and memory are arranged into integrated circuits whose task is to store and to execute machine code and to store data. In the early days of electronic computers, programs were written directly in machine code, but soon more abstract ways to specify programs than machine code and assembly language were sought.

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_1

FORTRAN (the Formula Translating System), is generally considered the oldest high-level programming language and dates back to the 1950s. It has been standardized for most of its existence. In many applications, it has been largely superseded by the commercial MATLAB software due to its convenient plotting routines, built-in availability of standard numerical linear-algebra algorithms, interactive use, large collection of additional software packages, and hence increased productivity.

LISP (the List Processor), is the second-oldest high-level programming language and was originally specified in 1958 [8, 9]. It gave rise to a large and diverse family of languages and it has influenced many other programming languages. COMMON LISP [1] is its most prominent member. Although LISP has never been as popular by far as FORTRAN for numerical computations, it is ideally suited for symbolic computations and computer algebra. The computer-algebra system MACSYMA [6] was written in LISP and continues to be developed in the form of MAXIMA [7]. COMMON LISP has also been playing a major role in artificial intelligence.

While FORTRAN emphasizes vectors and matrices as data structures, early LISP dialects emphasized data structures such as lists and symbols, which are especially useful for symbolic computations. It may just be an unfortunate historic coincidence that no single language in the early days of computing accommodated these two different needs in an efficient manner, and this fact may have resulted in a historic divide. In any case, a programming language that unifies efficient symbolic and numerical computations is certainly highly useful in mathematics.

Large-scale numerical computations require – at least on today's computing architectures – various layers of hard- and software: transistors, integrated circuits, machine code, assembly code, operating system, compiler, programming language, and algorithm. Of course, this book is not concerned with the layers (and technological marvels) up to and including the operating system. However, the top three layers, namely the choices of compiler, programming language, and algorithm, may have a profound impact on the final result or may be instrumental in being able to obtain a result at all.

Why is the choice of programming language important? There are three times in the life of a program:

- First, the program is written, i.e., an algorithm is implemented.
- Second, the program is compiled (or interpreted).
- Finally, the program is executed and produces output.

These three times result in requirements on programming languages suitable for the task at hand. High-level programming languages are preferable, since the time we have at our disposal to implement an algorithm or to find the most suitable numerical method is always limited. High-level programming languages by definition offer features that reduce implementation time. At the same time, the final programs should be efficient. Therefore the availability of state-of-the art compiler technology is another important requirement. The efficiency of the generated code is often linked to the type system of the programming languages. Therefore the type system and the programming language should have been designed such that they support the task of the compiler to generate fast code with minimal burden on the programmer.

In the past, programming languages were usually standardized and had several implementations. Nowadays, the situation is different; many popular programming languages are not standardized, and their single implementations serve as their specifications. Therefore the choice of programming language often severely limits the choice of compiler. This means that the choices of programming language and compiler are not independent, and one often has to decide on a combination of both.

A final requirement or consideration is the availability of well-tested libraries so that algorithmic and numerical wheels do not have to be reinvented.

1.2 An Overview of JULIA

MATLAB is probably the most well-known and widely used programming language in scientific computing and engineering. It has gained this position mainly by being a more convenient and more productive alternative to FORTRAN. MATLAB can be used interactively, many numerical algorithms are either built-in or available as packages, and plotting is easy. This is a large productivity gain compared to writing a FORTRAN program from scratch or downloading and installing libraries.

The programming language used in this book is JULIA [2]. JULIA is a high-level, high-performance, and dynamic programming language that has been developed with scientific and technical computing in mind. It offers features that make it very well suited for computing in science, engineering, and machine learning in view of the requirements posed in Sect. 1.2, while some of the features are unique for a programming language in this field. An overview of the key features of the JULIA language is given in the following.

1.2.1 The Reproducibility of Science and Open Source

JULIA is open source and distributed under the so called MIT license. Apart from concerns about licensing costs, the access to source code is essential for the reproducibility of science. Reproducibility is one of the main principles of the scientific method [10, 4] and hence also of artificial intelligence, machine learning, scientific computing, and computational science [12].

Reproducibility is important whenever calculations are performed. By using an open-source operating system and an open-source implementation of the programming language, it is – at least in principle – possible to know precisely which software was executed and which instructions were performed. Furthermore, it is also possible – again at least in principle – to check the correctness of all the software involved.

Unfortunately, it is likely that any large piece of software contains errors; an example is discussed in [3]. Therefore access to the source code is a reasonable requirement whenever errors in the implementation of a programming language are encountered or its behavior is to be verified.

1.2.2 Compiler

The implementation of the JULIA language includes a native-code compiler. More precisely, it includes a just-in-time compiler based on the LLVM compiler infrastructure. Employing a high-performance and well supported compiler in combination with JULIA's design yields performance that is – as microbenchmarks show – within a small factor of C and FORTRAN and faster in some benchmarks. By leveraging state-of-the-art compiler technology, JULIA achieves performance that is comparable to other popular programming languages not only for calls of library functions, but also for user defined functions.

1.2.3 Libraries and Numerical Linear Algebra

JULIA uses standard packages for numerical linear algebra such as BLAS (Basic Linear Algebra Subprograms), LAPACK (Linear Algebra Package), and SUITESPARSE (a collection of sparse-matrix software). These libraries are standard among other programming languages and software systems so that the performance of many linear-algebra algorithms in JULIA should be comparable if not identical to the performance in these other languages.

1.2.4 Interactivity

The dichotomy between the FORTRAN and LISP families of languages manifests itself clearly in the question whether functions can be called interactively or not. While FORTRAN programs follow a strict write-compile-execute cycle, LISP systems allow the interactive execution of expressions as well as the interactive definition and compilation of functions [11]. At the same, they usually support file compilation, saving of memory-image files, and generation of binaries.

Interactivity has a large effect on productivity, especially in explorative problem solving and programming. It makes it possible to implement complicated algorithms step by step and to immediately test them piece by piece. An interactive environment allows to set up complicated test cases and keep them in memory while defining new functions. Only the redefined functions need to be compiled so that potentially long compilation times are avoided when developing programs interactively.

1.2.5 High-Level Programming Concepts

JULIA provides several high-level programming concepts and is heavily influenced by the LISP family of languages, although its syntax is much closer to mathematical notation.

JULIA supports object oriented programming, provides a dynamic type system, and uses types for documentation, optimization, and dispatch. Users can of course define new types, and the user defined types are as fast and as compactly represented in memory as the built-in ones.

A dispatch mechanism decides which version of a function to run depending on the types of its arguments. JULIA uses the most general form of dispatch, namely multiple dispatch, similar to CLOS (COMMON LISP Object System) [5]. Multiple dispatch makes it possible to define function behavior depending on many combinations of argument types and not only the type of the first argument. JULIA also automatically generates efficient, specialized code for the different combinations of argument types. In this way, it provides both a very general approach to types, objects, and function dispatch, while sophisticated compilation techniques ensure that efficient code is generated.

JULIA is a homoiconic programming language, meaning that programs are represented in a data structure in a built-in type of the language itself. In other words, code is naturally represented in the language itself, and therefore code is easily treated as data. This makes it possible to write programs that write programs in a straightforward manner. Such program writing programs are called macros in the LISP family of languages, and macros can hence easily be defined in JULIA. As a homoiconic programming language, JULIA also provides other metaprogramming facilities.

In summary, these state-of-the-art programming concepts render JULIA not only useful in certain application areas, but also render it suitable for general programming tasks. As modern algorithms use a larger variety of data structures than scalars, vectors, and arrays, the support of user defined types and multiple dispatch are welcome for implementing sophisticated algorithms.

1.2.6 Interoperability

As mentioned in Sect. 1.2.3, JULIA uses external libraries for numerical linear algebra. In addition to this built-in use of external libraries, external libraries can generally be called easily from JULIA programs. External functions in C and FORTRAN shared libraries can be called without writing any wrapper code, and they can even be called directly from JULIA's interactive prompt. Furthermore, the `PyCall` package makes it possible to call PYTHON code. Other operating-system processes can also be invoked and managed from within JULIA by using its shell-like capabilities.

Vice versa, JULIA itself can be built as a shared library so that users can call their JULIA functions from within their C or FORTRAN programs.

1.2.7 Package System

JULIA comes with a package system that is both easy to use and easy to contribute to. It is straightforward to install packages and their dependencies, to update them, and to remove them. Many packages in the package system build on libraries written in other languages and provide interfaces to external functionalities in a JULIA-like style.

1.2.8 Parallel and Distributed Computing

JULIA comes with built-in functionality to run programs in parallel. The first type of parallel execution is running on a single computer and harnessing the power of the multiple cores of a CPU or of the multiple CPUs of a computer. The second type are clusters spanning multiple computers. For both types of parallel execution, introspective features make adding, removing, and querying the processes in a cluster straightforward.

Parallel function execution is achieved by just using the parallel version of a mapping function or by a parallel **`for`** loop. For parallel algorithms that require non-trivial communication, functions for moving data, for synchronization, for scheduling, and for shared arrays are available as well.

1.2.9 Availability on Common Operating Systems

JULIA is available on macOS, Linux, Windows, and FreeBSD. It can be downloaded as a binary distribution, it is available through various package managers, and the user can download and build the JULIA source code and its dependencies with a few commands.

1.3 Using JULIA and Accessing Documentation

1.3.1 Starting JULIA

After installing JULIA, you can start it by double-clicking the JULIA icon (depending on your system and installation) or by running the JULIA executable from the command line. If you just type

```
> julia
```

at the command line, JULIA starts, displays a banner, and prompts you for input with its own prompt `julia>`. You can quit the interactive session by typing `exit()` and pressing the return key or by typing control-d.

To run a JULIA program saved in a file called file `program.jl` non-interactively, type

```
> julia program.jl
```

at the command line. If your program is supposed to take arguments from the command line, you can simply pass them at the end of the command line and they will be available in JULIA in the global variable `ARGS` as an array of strings.

```
> julia program.jl arg1 arg2 arg3
```

You can also pass code to be executed directly from the command line to JULIA by using the `-e` command-line option. Then the traditional example looks like this.

```
> julia -e 'println("Hello, world!")'
Hello, world!
```

Note that the quotes around the JULIA code depend on your command shell and may differ from `'`.

When JULIA code is executed via the `-e` command-line option, the arguments are stored in `ARGS` as well. We can try to retrieve the command-line arguments passed to JULIA code like this.

```
> julia -e 'ARGS' arg1 arg2 arg3
```

However, nothing is printed. Using `-e`, the expression is only evaluated, but the result is not printed. To print a value (followed by a newline), we can use the command-line option `-E`, which evaluates an expression and shows the result.

```
> julia -E 'ARGS' arg1 arg2 arg3
["arg1", "arg2", "arg3"]
```

Another possibility is to use the `println` function.

```
> julia -e 'println(ARGS)' arg1 arg2 arg3
["arg1", "arg2", "arg3"]
```

The output is the printed representation of an array containing the three strings shown. To print each element of the `ARGS` array on a separate line, you can `map` the `println` function over the array stored in the variable `ARGS`.

```
> julia -e 'map(println, ARGS)' arg1 arg2 arg3
arg1
arg2
arg3
```

Running `julia --help` from your command line gives an overview of the many options of the JULIA executable. Command-line arguments for the JULIA executable must be passed before the name of the file to be executed as in this example.

```
> julia --optimize program.jl arg1 arg2 arg3
```

Any JULIA code you put into the file `$HOME/.julia/config/startup.jl` in your home directory will be executed every time JULIA is started.

So far we have seen how to run your JULIA programs from the command line. Although this is the usual way how JULIA programs are run in production environments, it is only one way to run a JULIA program. While developing programs, interactive sessions connected to an editor are much preferred. The features of interactive sessions are explained next.

1.3.2 The Read-Eval-Print Loop

JULIA is often used interactively via its REPL . The abbreviation REPL is short for read-eval-print loop and has its roots in LISP implementations. The three parts of the REPL are the following.

Read: An expression typed by the user is read. An error is raised if it is not syntactically correct.
Eval: The expression is evaluated. The result is a value, unless an error was raised.
Print: The value is printed or – if an error occurred – the error message is displayed. Finally, a new prompt is displayed and the loop is repeated.

This implies that each expression in JULIA returns a value, just as in LISP. (Therefore it has been said about LISP programmers that they know the value of everything, but the (computational) cost of nothing.) There is no expression that does not return a value. Displaying a value can, however, be suppressed by appending a semicolon `;` at the end of the input.

```
julia> "Hello, world!"
"Hello, world!"
julia> "Hello, everyone!";
julia> ans
```

```
"Hello, everyone!"
```

This example shows that the shortest hello-world program in JULIA is just the string `"Hello, world!"`. It also shows that the variable `ans`, short for answer, is bound to the value of the previous expression evaluated by the REPL irrespective whether it was printed or not. The variable `ans` is only bound in REPLs.

In addition to strings, numbers such as integers and floating-point numbers also evaluate to themselves and can be entered as usual. Furthermore, it is possible to type additional underscores _ in order to divide long numbers and make them easier to read. The groups do not have to contain three digits.

```
julia> 1_000_000_000 * 0.000_000_001
1.0
julia> 1_2_3
123
```

You can load a JULIA source file called `"file.jl"` and evaluate the expressions it contains using `include("file.jl")`. Since the function `include` works recursively, this also makes it possible to split programs into various files. During development, however, smaller pieces of code are usually evaluated (see Sect. 1.3.5). Larger JULIA programs, on the other hand, such as packages whose source code is distributed online, are usually installed and loaded as packages much more easily than by working with single files (see Sect. 1.3.4).

When you working in the JULIA REPL, you can execute shell commands conveniently by switching to shell mode, which is entered by just typing a semicolon `;`. Then the JULIA prompt changes and shell commands such `ls` or `ps` can be executed. Typing backspace switches the prompt back to the usual JULIA prompt.

You can save lots of typing at the REPL using autocompletion. If you press the tab key, the symbol you started typing is completed, or – if the completion is not unique – completions are suggested after pressing tab a second time. This feature also works in shell mode, where it is convenient to complete names of directories and files.

The REPL remembers the expressions it has evaluated previously. Expressions from previous interactive sessions are also stored in the file `$HOME/.julia/logs/repl_history.jl`. The straightforward way to access previous expressions is using the up and down arrow keys. But you can also search the history forwards and backwards with control-s and control-r, respectively. Other keyboard commands analogous to the EMACS text editor are available as well.

1.3.3 Help and Documentation

Similarly to the shell mode, you can enter the help mode of the REPL by typing a question mark `?` at the JULIA prompt. The prompt changes and you can enter a string to search for. You can again use the tab key to complete the string you typed as a symbol or to view possible completions. After pressing enter, the documentation is searched and you will be presented with the documentation for the symbol you entered or further matches of the string you typed.

To search all documentation for a string, you can use the `apropos` function.

```
julia> apropos("Euler")
Base.MathConstants.gamma
Base.MathConstants.eulergamma
```

The `@doc` macro allows you to access the documentation string of any symbol and also to change it. Documentation is accessed by `@doc(`*symbol*`)` or simpler by `@doc` *symbol* (see Chap. 7 for more information about macros and how to use them).

```
julia> @doc @doc
```

Table 1.1 contains a list of functions and macros that are useful to inspect the state of the JULIA executable or to interact with the operating system. The contexts in which several of these functions and macros are useful will become clearer later, but they are collected here for reference.

The value of the constant `VERSION` is of type `VersionNumber` and can easily be compared to other values of the same type. This makes it possible to write programs that work with different versions of JULIA.

```
julia> VERSION >= v"1.0"
true
julia> VERSION >= v"2.0"
false
```

1.3.4 Handling Packages

The plain way to handle packages is to use the functions provided by the `Pkg` package, which we must load first.

```
julia> import Pkg
```

The most important functions in this package are `Pkg.add`, `Pkg.rm`, and `Pkg.update`. For example, to install a package called `CSV` (for reading and writing comma separated values), we can use the `Pkg.add` function.

```
julia> Pkg.add("CSV")
```

To remove it, we can use the `Pkg.rm` function.

```
julia> Pkg.rm("CSV")
```

Table 1.1 Generally useful functions and macros.

Function or macro	Description
`ans`	variable with the last evaluated expression
`apropos`	search documentation for a string
`atexit`	register a function to be called at exit
`atreplinit`	register a function to be called before starting a REPL
`clipboard`	send a string and receive a string from the clipboard
`@doc`	access and modify documentation strings
`dump`	show user visible structure of a value
`edit`[a,b]	edit a file or function definition
`@edit`[b]	edit a function definition
`ENV`	operating-system environment variables
`exit`	exit the JULIA executable, possibly supplying an exit code
`fieldnames`	return an array of the fields of a type
`include`	evaluate a source file, files are fetched from node 1
`isinteractive`	whether JULIA is running interactively
`less`	show a file or function definition
`@less`	show a function definition
`methods`	return the methods of a function
`methodswith`	return methods with an argument of the given type
`names`	return an array of the names exported by a module
`@show`	show an expression and return it
`summary`	briefly describe a value
`@time`	print timing and allocation information, return value
`@timed`	return value as well as timing and allocation information
`@timev`	verbose version of `@time`
`VERSION`	the version number, a value of type `VersionNumber`
`versioninfo`	print version information about JULIA, libraries, and packages
`which`[a]	return the applicable method
`which`[a]	return the module where a variable was defined
`@which`	return the applicable method

[a] The behavior depends on the types of the arguments.
[b] The editor called is the one given by `ENV["EDITOR"]`.

To update all installed packages, we can use the `Pkg.update` function.

```
julia> Pkg.update()
```

Another mode that can be entered from the REPL is the package mode for handling packages. It is entered by typing `]` at the JULIA prompt. The prompt changes to end in `pkg>`. Typing tab at the prompt shows a list of all commands available in package mode. For example, to install the `CSV` package, type `add CSV` at the package prompt; to remove it, type `remove CSV`; and to update all packages type `update` at the package prompt.

1.3.5 Developing JULIA Programs

Because of the REPL, developing JULIA programs resembles developing LISP, PYTHON, or MATLAB programs in the sense that it may become a very interactive process. A good practice is to compose larger programs out of functions that are small enough to serve a single, well-defined purpose, to be understood without unnecessary context, and to be tested exhaustively. In short, the programmer should be able to prove the correctness of each function. Then larger programs can be assembled from these building blocks and well-thought-out data structures.

When writing functions in an editor, the REPL is useful to explore ideas and to query the built-in documentation. Function definitions and source files can quickly be loaded into the JULIA executable. Reloading definitions is usually supported by editors suitable for programming. Whenever a function definition appears satisfactory, it can immediately be tested in the REPL. Data for testing functions can be saved in variables in the REPL so that the cycle of exploring ideas, implementing them, testing the functions on realistic data, and improving them does not have to interrupted, but can be performed on the same input repeatedly. In languages that cannot be used interactively, changing a program necessitates the building of a new executable and loading test data anew, which are potentially time consuming steps. In a dynamic language such as JULIA, redefining and compiling single functions is a simple and often performed procedure and much accelerates the loop consisting of exploration, implementation, testing, and improvement.

The method to develop larger programs by abstracting its pieces into functions is very effective. In the ideal case, it is possible to show the correctness of each function at least informally. This development style should also be supported by the choice of editor and development environment. Two such environments are discussed briefly in the following.

The first development environment is EMACS in conjunction with its JULIA mode. EMACS (short for editor macros) is a LISP programmable editor which is convenient for many different text editing and programming tasks because of its many modes that provide special key bindings and interfaces to other programs. The JULIA mode makes it straightforward to interact with a JULIA REPL, to evaluate JULIA expressions and files, and to access documentation.

The second environment is `IJulia`. It is a popular way of writing shorter JULIA programs and preparing reports. `IJulia` provides a graphical user interface that runs within a web browser. It resembles a notebook, where you can input JULIA expressions and function definitions in cells. You can evaluate an input cell by typing shift-enter, and then the output is collected below the input cell. Text such as documentation or a narrative for the calculations can be saved in a notebook file as well. Finally, the notebook can be exported to various formats such as JULIA code in a text file, PDF, HTML, and MARKDOWN.

`IJulia` is in fact a JULIA package and can be installed as described in Sect. 1.3.4. Then is loaded and started using these two commands.

```
julia> import IJulia
julia> IJulia.notebook()
```

If files in the current directory are of interest, the following function call is useful.

```
julia> IJulia.notebook(dir = pwd())
```

An alternative that has the same effect but saves some typing are the following commands.

```
julia> using IJulia
julia> notebook()
```

Problems

1.1 Install JULIA on your computer.

1.2 Install an extension for dealing with JULIA programs in your favorite text editor or install the `IJulia` package.

References

1. American National Standards Institute (ANSI), Washington, DC, USA: *Programming Language Common Lisp, ANSI INCITS 226-1994 (R2004)* (1994)
2. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: The Julia programming language. http://julialang.org
3. Durán, A., Pérez, M., Varona, J.: The misfortunes of a trio of mathematicians using computer algebra systems. Can we trust in them? *Notices of the AMS* **61**(10), 1249–1252 (2014)
4. Fisher, R.: *The Design of Experiments*. Oliver and Boyd, Edinburgh (1935)
5. Keene, S.: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley Professional (1989)
6. The Mathlab Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139: *MACSYMA Reference Manual*, Version Nine, Second Printing (1977)
7. *Maxima, a Computer Algebra System, version 5.43.0* (2019). http://maxima.sourceforge.net
8. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine (part I). *Comm. ACM* **3**, 184–195 (1960)
9. McCarthy, J.: *LISP 1.5 Programmer's Manual*. The MIT Press (1962)
10. Popper, K.: *Logik der Forschung. Zur Erkenntnistheorie der modernen Naturwissenschaft*. Verlag von Julius Springer, Wien (1935)
11. Sandewall, E.: Programming in an interactive environment: the "LISP" experience. *Computing Surveys* **10**(1), 35–71 (1978)
12. Stodden, V., Borwein, J., Bailey, D.: Setting the default to reproducible in computational science research. *SIAM News* **46**(5), 4–6 (2013)

Chapter 2
Functions

A year spent in artificial intelligence is enough to make one believe in God.

—Alan Jay Perlis

Abstract Functions are one of the most important abstractions in mathematics, and they are one of the most important concepts in programming languages as well. Functions are the main building blocks of programs. In this chapter, we learn how functions are defined in JULIA and we discuss generic functions and methods. More details are presented as well, including argument passing behavior, multiple return values, functions as first-class objects, anonymous functions, optional and keyword arguments, variable numbers of arguments, and the scopes of variables. Interactions with data types are discussed throughout the chapter to illustrate the interplay between functions, domains, and codomains.

2.1 Defining Functions

Functions are one of the most important concepts in mathematics. The definition of a mathematical function $f : X \to Y, x \mapsto y$ comprises three parts: the domain X, i.e., the set where the function is defined; the codomain Y, i.e., the set that contains all function values y; and a rule $x \mapsto y$ describing how a unique function value y is assigned to each argument $x \in X$.

The same three pieces of information are important when defining a function in any programming language. The role of the domain is played by the types of the arguments, the function values are calculated by the body of the function definition, and the role of the codomain is played by the type of the calculated value and can hopefully be inferred by the compiler. If it can be inferred, faster code can be generated for the calling function that receives the output.

The example we consider in this chapter is the definition of a function that calculates the n-th number x_n in the Fibonacci sequence defined by the recur-

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_2

rence relation

$$x_0 := 0, \qquad x_1 := 1, \qquad x_n := x_{n-1} + x_{n-2}. \tag{2.1}$$

The mathematical function corresponding to this sequence is denoted by $\mathrm{fib}\colon \mathbb{N}_0 \to \mathbb{N}_0, n \mapsto x_n$.

A straightforward translation of this recurrence relation into a JULIA function is the following.

```
function fib1(n)
    if n <= 1
        n
    else
        fib1(n-1) + fib1(n-2)
    end
end
```

The idea is to check if the argument `n` is one of the starting values or not. If it is not, then the recurrence relation is used. We note that in JULIA everything is an expression and hence returns a value, so that it is not necessary to use an explicit **`return`** statement here.

An alternative, but equivalent syntax for function definition is the following called the ternary operator.

```
fib2(n) = (n <= 1) ? n : fib2(n-1) + fib2(n-2)
```

This example shows how short functions are often defined in JULIA. Here the syntax

condition ? *consequent* : *alternative*

was used as an alternative for the **`if`** expression as well.

You can save this function definition in a file and load it into JULIA or you can type it directly into the JULIA REPL. In the REPL, JULIA will answer with the following output.

```
fib1 (generic function with 1 method)
```

We note that this function definition does not capture two pieces of information that are part of a mathematical function definition: the domain and the codomain. We have neither specified the type of `n` nor the type of the possible function values `0`, `1`, and `fib1(n-1) + fib1(n-2)`. This means that this function definition will work whenever the operations used in its definition, namely `<=`, `+`, and `-`, are defined for their arguments. For example, evaluating `fib1(5//4)` in the JULIA REPL yields `-1//2`; here `5//4` and `-1//2` are rational numbers. We will learn all about the types of numbers available in JULIA in Chap. 5. It is not obvious just by looking at the definitions of `fib1` and `fib2` which types can be used as arguments and whether JULIA can generate efficient code or not.

Therefore, we now use JULIA's introspective features to find out more about the function we just defined. JULIA told us after evaluating the function definition that we have defined a generic function with one method. The same information is obtained by typing `fib1` into the REPL, since functions evaluate to themselves in JULIA. This means that functions in JULIA are what are called generic functions in computer science. A generic function is a collection of (function) methods, where each method is responsible for a certain combination of argument types. For example, the generic function + comprises many methods.

```
julia> +
+ (generic function with 166 methods)
```

We can find the methods of a generic function using `methods`. It lists all the methods that were defined for various combinations of argument types and where they were defined.

```
julia> methods(fib1)
# 1 method for generic function "fib1":
[1] fib1(n) in Main at REPL[1]:2
```

If a new method is defined and a method already exists for this particular combination of argument types, then the old method definition is superseded.

To find out more about the codomain of the function, we can query the types of function values.

```
julia> typeof(fib1(0))
Int64
julia> typeof(0)
Int64
julia> typeof(fib1(1))
Int64
julia> typeof(1)
Int64
julia> typeof(fib1(2))
Int64
```

In the first two cases, the argument is simply returned. Therefore the type of the returned value is the same as the type of the argument. The last example implies that the type of the sum of two values of type **`Int64`** is again **`Int64`**.

If you are using a 32-bit system, then integers are by default represented by the type **`Int32`**. The default integer type is called **`Int`** and it can be either **`Int32`** or **`Int64`** depending on your system. The variable `Base.Sys.WORD_SIZE` also indicates if JULIA is running on a 32-bit or a 64-bit system.

We have just seen that literal integers such as `0` and `1` are parsed and then represented as values of type **`Int`**, which is an **`Int64`** on this particular system. **`Int64`** are (positive or negative, i.e., signed) integers that can be stored within 64 bits. This is illustrated by the following calculations.

```
julia> 2^62
4611686018427387904
julia> (-2)^62
4611686018427387904
julia> 2^63
-9223372036854775808
julia> (-2)^63
-9223372036854775808
julia> 2^64
0
julia> (-2)^64
0
```

In addition to querying the type of a value using `typeof`, we can also query a type which the minimum and maximum numbers it can represent are. This is informative when a type can only represent a finite number of values by design.

```
julia> typemin(Int64)
-9223372036854775808
julia> typemax(Int64)
9223372036854775807
julia> typemax(typeof(fib1(0)))
9223372036854775807
```

These bounds explain why 2^{63} could not be calculated above as an **`Int64`**, but $(-2)^{63}$ (barely) could.

This implies that types such as **`Int`**, **`Int32`**, and **`Int64`** can represent the mathematical structure of the ring $(\mathbb{Z}, +, \cdot)$ only if all operations during a calculation remain within the interval given by `typemin(`*type*`)` and `typemax(`*type*`)`.

If this is not assured, then it is necessary to use the **`BigInt`** type, which can represent integers provided they fit into available memory and are thus a much better representation of the ring $(\mathbb{Z}, +, \cdot)$. To illustrate the efficiency of calculations using **`BigInt`**s, we consider Mersenne prime numbers. The following function returns Mersenne prime numbers given an adequate exponent.

```
mersenne(n) = BigInt(2)^n - 1
```

The type of the return value is **`BigInt`**, since we base the calculation on 2 represented as a **`BigInt`**. In general, *type*`(x)` returns the representation of the value `x` in the type *type*; in other words, a new value converted to the type *type* is returned. It is instructive to inspect the **`BigInt`** type using `methods(`**`BigInt`**`)`, `typemin(`**`BigInt`**`)`, and `typemax(`**`BigInt`**`)`.

The largest known Mersenne prime number to date is $2^{82\,589\,933} - 1$. The following interaction shows that it can be computed within a few thousands of a second requiring less than 20 MB of memory, also showing that it has 24 862 048 digits.

```
digits(x) = floor(Integer, log10(x)+1)
```

```
julia> @time digits(mersenne(82_589_933))
  0.004653 seconds (23 allocations: 19.692 MiB)
24862048
```

The `@time` macro yields the run time and the allocated memory. Asterisk generally indicate macros; we will learn all about macros in Chap. 7.

Now we know how to define a function that can calculate arbitrarily large (only limited by the available memory) Fibonacci numbers. The next version of the Fibonacci function ensures that the codomain is the type **`BigInt`**.

```
function fib3(n)::BigInt
    if n <= 1
        n
    else
        fib3(n-1) + fib3(n-2)
    end
end
```

The syntax `::`*type* after the argument list means that the return value will be converted to the specified type. Here the base case `n <= 1` ensures that later on **`BigInt`**s are added. If the return value cannot be converted to the specified type, then an error is raised. If `::`*type* is not given, it is assumed to be `::`**`Any`**. Every value is JULIA is of type **`Any`**.

We have seen how we can specify the codomain of our function. How can we specify the domain of our function? We already know that a generic function consists of methods. The various methods that constitute a generic function are responsible for different argument types. Whenever a function is called, the types of the arguments are inspected and then the most specific matching method is called; if none exists, an error is raised. The only method we have defined for the function `fib3` is called for every argument type, since we did not specify any type for the argument `n`.

But this is not what we intend in the context of the Fibonacci sequence. It is unclear what `fib1(5//4)` or `fib1(9.5)` should be, although our implementation returns numbers in these cases. `fib1("foo")` clearly raises an error (only after trying to perform calculations), but `fib1(5//4)` and `fib1(9.5)` should raise errors as well.

How can we restrict the domain, i.e., how can we specify the types of the arguments of a method? The syntax is again *argument*`::`*type*. The domain of the next version of the Fibonacci function is the **`Integer`** type and the codomain is the **`BigInt`** type.

```
function fib4(n::Integer)::BigInt
    if n <= 1
        n
    else
        fib4(n-1) + fib4(n-2)
    end
end
```

The **Integer** type comprises **BigInt**s and the finite integer types **Int32** and **Int64** among others. It is therefore the most natural domain for our function.

We can check whether a type is a subset of another one using the subtype function *subtype*<:*supertype*. The following example shows that the **Integer** type works as intended in the method definition above. It also shows that neither **BigInt** is a subtype of **Int** nor **Int** is a subtype of **BigInt**, confirming the usefulness of the **Integer** type. The **Any** type is a supertype of every type. An argument x without any specified type is equivalent to an argument x::**Any**, just as in the case of the return value.

```
julia> Int
Int64
julia> Int32 <: Integer
true
julia> Int64 <: Integer
true
julia> BigInt <: Integer
true
julia> BigInt <: Int
false
julia> Int <: BigInt
false
julia> Integer <: Any
true
```

To check whether a value has a certain type, the isa function, which also supports infix syntax, can be used.

```
julia> isa(0, Int64)
true
julia> 0 isa Int64
true
```

Calling the generic function fib4 with arguments that are not of type **Integer** results in an error explaining that there is no matching method. Sometimes it is useful to define a method that catches all other argument types. This is achieved by the following method.

```
fib4(x::Any) = error("Only defined for integer arguments.")
```

Evaluating methods(fib4) confirms that the generic function comprises two methods.

So far we have learned how we can define the domains and codomains of JULIA functions, namely by specifying the types of the arguments and the return values. Next the question arises how efficient our implementation actually works. Calculating a few function values based on **Int**s or **BigInt**s for relatively small arguments and timing the calculations using the @time macro quickly leads us to the conclusion that we will run out of patience before we run out of integers.

We therefore examine the computational problem, namely solving the difference equation (2.1), in more detail. The difference equation

$$x_n = x_{n-1} + x_{n-2}$$

in (2.1) can be solved using the ansatz $x_n = y^n$, substituting it into (2.1), finding the two solutions $y_1 := (1 + \sqrt{5})/2$ and $y_2 := (1 - \sqrt{5})/2$ of the resulting quadratic equation, and noting that the general solution of this difference equation is the linear combination

$$x_n = c_1 y_1^n + c_2 y_2^n.$$

The two starting values x_0 and x_1 determine the two constants c_1 and c_2 as $c_1 := 1/\sqrt{5}$ and $c_2 := -1/\sqrt{5}$. Therefore the Fibonacci sequence is given by

$$\begin{aligned} x_n &= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \\ &= \operatorname{round}\left(\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \qquad \forall n \in \mathbb{N}. \end{aligned}$$

The last equality holds since $y_2 \approx -0.618$ and $|c_2| < 1/2$.

The theory of difference equations hence leads to the next function definition.

```
fib5(n::Integer)::BigInt =
    round(BigInt, ((1+sqrt(5))/2)^n / sqrt(5))
```

The argument **`BigInt`** of `round` ensures not only that a **`BigInt`** is returned instead of a floating-point value, but also that no error is raised when the number to be rounded is large.

Although we can calculate Fibonacci numbers now very quickly, it unfortunately turns out that the return value of `fib5(71)` is not equal to x_{71} (while the preceding values are correct). The smallest example to demonstrate this deficiency is the following.

```
julia> @time fib5(69) + fib5(70) == fib5(71)
  0.000005 seconds (8 allocations: 168 bytes)
false
```

Evaluating `typeof(sqrt(5))` yields a value of type **`Float64`**, which is the type of IEEE 754 double-precision floating-point numbers. It is well-known that this representation of the real numbers $\mathbb{R}$ gives 15 to 17 significant decimal digits of precision. Since x_{70} and x_{71} have 15 decimal digits, we conclude that exponentiation over the **`Float64`** type works very precisely within these limits. On the other hand, we cannot calculate more Fibonacci numbers in this manner, since the spacing between adjacent floating-point numbers becomes too large to represent integers.

We can alleviate this limitation by using the **BigFloat** type and ensuring that all calculations are performed over this type. Then the **BigFloat** value is rounded to a **BigInt** value.

```
fib6(n::Integer)::BigInt =
    round(BigInt,
          ((1+sqrt(BigFloat(5)))/2)^n / sqrt(BigFloat(5)))
```

The following function checks the calculations. The output means that all Fibonacci numbers up to and including x_{358} are calculated correctly, while `fib6(359)` is not equal to x_{359}.

```
function check_fib6(range)
    for i in range
        if fib6(i) + fib6(i+1) != fib6(i+2)
            print(i, " ")
        end
    end
end
```

```
julia> check_fib6(0:370)
357 362 366 367 368 369 370
```

We can again relate the number of bits used in this calculation to the number of decimal digits of x_{359}. The following calculation shows that we can expect at most 78 decimal digits of precision, since **BigFloat**s use 256 binary digits by default. (The precision of **BigFloat**s can be changed using `setprecision`.) At the same time, representing `fib5(359)` requires 75 decimal digits. Therefore the calculation using the exponentiation is very precise in the sense that almost all digits of the result are correct.

```
julia> digits(BigInt(2)^256)
78
julia> digits(fib6(359))
75
```

Although our detour taking advantage of an explicit formula for Fibonacci numbers turned out to be successful in the sense that we can calculate more Fibonacci numbers faster, it is not entirely satisfying for two reasons. First, the theory of difference equations provided a new angle how to attack the problem, but such a new angle is not available in general. Second, the machinery behind `BigFloats` and `round` is considerable.

Therefore we return to the recursive function `fib4` now. The inefficiency of this implementation stems from the fact that during the recursive calculation of `fib4(`n`)` the number of times that `fib4(`m`)` is called becomes larger and larger as m becomes smaller. Therefore a trade-off between computation time and memory consumption is reasonable.

The idea is to define a global variable that holds a dictionary containing the previously calculated values. (Global variables are discussed in Chap. 3, and dictionaries in Sect. 4.5.4.) The function checks if the function value has been calculated previously. If yes, it is simply returned; if not, the new value is calculated, stored, and returned. This technique is known as memoization (see Sect. 7.5).

```
global fib_cache = Dict{BigInt, BigInt}(0 => 0, 1 => 1)

function fib7(n::Integer)
    global fib_cache

    if haskey(fib_cache, n)
        fib_cache[n]
    else
        fib_cache[n] = fib7(n-1) + fib7(n-2)
    end
end
```

This implementation can calculate $x_{10\,000}$, which has 2090 digits, within a few thousands of a second using about 6 MB of memory. We will see a more general approach to memoization in Sect. 7.5.

Additional properties of the Fibonacci sequence are useful to refine this approach. It can be shown that the equalities

$$x_{2n} = x_{n+1}^2 - x_{n-1}^2 = x_n(x_{n+1} + x_{n-1}), \tag{2.2}$$

$$x_{3n} = 2x_n^3 + 3x_n x_{n+1} x_{n-1} = 5x_n^3 + 3(-1)^n x_n, \tag{2.3}$$

$$x_{4n} = 4x_n x_{n+1}(x_{n+1}^2 + 2x_n^2) - 3x_n^2(x_n^2 + 2x_{n+1}^2) \tag{2.4}$$

hold for all $n \in \mathbb{N}$.

The next version uses the last equality to reduce the number of function values that are memoized to approximately one quarter.

```
global fib_cache = Dict{BigInt, BigInt}(0 => 0, 1 => 1)

function fib8(n::Integer)
    global fib_cache

    if haskey(fib_cache, n)
        fib_cache[n]
    else
        if mod(n, 4) == 0
            m = div(n, 4)
            fib_cache[n] = (4*fib8(m)*fib8(m+1)
                            *(fib8(m+1)^2+2*fib8(m)^2)
                            -3*fib8(m)^2*(fib8(m)^2+2*fib8(m+1)^2))
        else
```

```
            fib_cache[n] = fib8(n-1) + fib8(n-2)
        end
    end
end
```

To summarize, we have approached the problem of calculating Fibonacci numbers, defined by maybe the simplest of all recursive formulas, from several very different angles. When translating the definition of a mathematical function into a JULIA function, we have seen

- that the domain of the function corresponds to the types of the arguments,
- that the codomain corresponds to the type of the return value, and
- that the development of an efficient algorithm to calculate entities defined by abstract mathematical definitions may be a highly complex task.

Mathematical functions correspond to generic functions in JULIA. Generic functions consist of (function) methods, and the method that is called depends on the types of the arguments. The function `applicable` can be used to find out if a given generic function has a method that is applicable to given arguments. The function `which` returns the method of a given generic function that matches given argument types. Given a generic function and argument types, you can use `invoke` to call the method that matches the given argument types (after converting the arguments). This allows invoking a method different from the most specific one.

Another important consideration is how to translate infinite sets and abstract concepts such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{C}$ into a necessarily finite approximation in a computer. Appropriate representations must be chosen depending on the problem to be solved, and therefore we will learn much more about built-in and user defined data types in Chapters 4 and 5. Fortunately, JULIA provides a rich set of numerical data types of various degrees of precision and makes it easy to change the types used in a program.

We have also seen in the example that it is prudent to implement tests and checks whenever and as soon as possible. A standard procedure is to compare numerical results with known exact solutions in order to check the correctness of the implementation and in order to assess the size of the error necessarily made by using approximations.

2.2 Argument Passing Behavior

In computer science, various ways to pass arguments from the caller to the called function are known. The following approaches are commonly found in programming languages.

Call by value: The arguments are evaluated and the resulting values are passed to the function and bound to local variables. The passed values are often copied into a new memory region. The function cannot make changes in the scope of the caller, since it only receives a copy.

Call by reference: The function receives a reference to a variable used as the argument. Via this reference, the function can assign a new value to the variable or modify it, and any changes are also seen by the caller of the function.

Call by sharing: If the values in a languages are objects (carrying type information in contrast to primitive types), then call by sharing is possible. In call by sharing, function arguments act as new variable bindings and assignments to function arguments are not visible to the caller. No copies of the arguments are made, however, and the values the new variable bindings refer to are identical to the passed values. Therefore changes to a mutable object are seen by the caller.

Call by value is provides additional safety, since it is impossible for the called function to effect any changes outside of its own scope. On the other hand, it is inefficient to copy all arguments, especially when many small functions are defined or the arguments occupy large memory regions such as large arrays.

Call by reference is more efficient, but a function may effect changes outside of its scope, making reasoning about program behavior much more difficult and possibly leading to subtle bugs. Call be reference is the most unsafe way of passing arguments.

JULIA uses call by sharing. Assignments to function arguments only affect the scope of the function. Still, mutable objects (such as the elements of vectors or arrays) can be changed and these changes persist and are seen by the caller. Call by sharing is a reasonable compromise between memory safety and efficiency and it is found in other dynamic languages such as LISP, SCHEME, and PYTHON.

2.3 Multiple Return Values

JULIA does not provide multiple return values per se, but uses tuples of values to the same effect. Since tuples can be created and destructured also without parentheses, the illusion of returning and receiving multiple values is created by leaving out the parentheses.

Tuples can always be created with parentheses and in many circumstances without parentheses. The same holds true when tuples are destructured in assignments. If there are fewer variables in the tuple on the left side than elements in the tuple on the right side of the assignment, then only the first elements on the right side are used. Conversely, if there are more variables in the tuple on the left side than elements on the right side, an error is raised.

```
julia> foo, bar = 0, 1
(0, 1)
julia> (foo, bar) = (2, 3)
(0, 1)
julia> (foo,) = (0, 1)
(0, 1)
julia> foo
0
```

A tuple with a single element is created using the syntax `(`*element*`,)` so that it can be distinguished from parentheses that have no effect around an expression.

```
julia> (0,)
(0,)
```

Using tuples, multiple values can be returned and received in a straightforward manner. When receiving the return values, a tuple can be assigned to a variable or the tuple can be destructured and multiple variables can be assigned.

```
function foo(a1, a2, b1, b2)
    (a1*b1 - a2*b2, a1*b2 + a2*b1)
end

julia> c = foo(1, 2, 3, 4)
(-5, 10)
julia> (c1, c2) = foo(1, 2, 3, 4)
(-5, 10)
julia> c, c1, c2
((-5, 10), -5, 10)
```

2.4 Functions as First-Class Objects

Functions are first-class objects in JULIA, and the type of each function is a subtype of the type `Function`. This means that functions can be assigned and passed as arguments just as any other data type.

```
julia> +
+ (generic function with 166 methods)
julia> +(0, 1)
1
julia> isa(+, Function)
true
julia> foo = *
* (generic function with 357 methods)
julia> foo(0, 1)
0
```

Passing functions as function arguments is not uncommon. The built-in `sort` function, for example, takes a keyword argument `lt` (short for less than) that specifies the ordering to be used. In this example, the function `>` is passed as the `lt` argument and then used to compare two values.

```
julia> sort([1, 3, 2], lt = >)
3-element Array{Int64,1}:
 3
 2
 1
```

Functions whose domains are functions are well-known in mathematics: they are functionals. The following function calculates a simple approximation (the Riemann definition) of the functional $I(f) := \int_a^b f(x)\mathrm{d}x$.

```
function riemann(f::Function, a::Number, b::Number, n::Integer)
    local sum = 0
    local h = (b-a) / n
    for i in 1:n
        sum += f((i+1/2) * h)
    end
    h * sum
end

julia> riemann(cos, 0, 2pi, 100)
1.3253900292432802e-16
```

The `QuadGK` package provides one-dimensional numerical integration using adaptive Gauss–Kronrod quadrature.

```
julia> import QuadGK
julia> QuadGK.quadgk(cos, 0, 2pi)
(3.00032335826547e-16, 5.062846180836036e-24)
```

The first element of the tuple is the estimated value of the integral, and the second an estimated upper bound for the absolute error.

The identity function is called `identity` in JULIA. Additionally, there are syntactic expressions in JULIA, listed in Table 2.1, which are translated into function calls, but the names of the functions are not obvious.

Finally, the expression *arg* `|>` *fun* is the same as *fun*`(`*arg*`)`. It allows to revert the order of function and arguments and is easier to read in certain situations.

2.5 Anonymous Functions

Anonymous functions can be created by either of two syntactic options, namely using `->` or a function definition without a function name. Especially the first syntax is useful to create simple functions and to pass them to other functions.

Table 2.1 Syntactic expressions that correspond to function calls.

Syntactic expression	Function
`[A B C...]`	`hcat`[a]
`[A, B, C, ...]`	`vcat`[b]
`[A B; C D; ...]`	`hvcat`[c]
`A'`	`adjoint`[d]
`start:stop`, `start:step:stop`	`(:)(start, [step,] stop)`
`A[i]`	`getindex`
`A[i] = x`	`setindex!`

[a] Concatenate horizontally.
[b] Concatenate vertically.
[c] Concatenate horizontally and vertically.
[d] Conjugate transpose.

```
julia> x -> 2x
#3 (generic function with 1 method)
julia> isa(x -> 2x, Function)
true
julia> (x, y) -> 2x*y
#7 (generic function with 1 method)
julia> function (x)
           2x
       end
#9 (generic function with 1 method)
```

A popular example of using anonymous functions is passing a custom ordering to the `sort` function. Instead of defining a function used only in one place, it is often more convenient to use a short anonymous function.

```
julia> sort([1.9, 1.8, 1.7], lt = (x,y) -> floor(x) < floor(y))
3-element Array{Float64,1}:
 1.9
 1.8
 1.7
```

A classical example, often found in functional programming, is the `map` function. It takes a function as its first argument and one or more collections as the remaining arguments. The number of arguments the function expects must match the number of collections passed. Then the function is applied elementwise to the collections and a collection with the results is returned.

```
julia> map(x -> 10x, [1, 2, 3])
3-element Array{Int64,1}:
 10
 20
 30
julia> map((x, y) -> 10x + y, [1, 2, 3], [4, 5, 6])
```

```
3-element Array{Int64,1}:
 14
 25
 36
```

There are variants of the `map` function, namely `map!`, `mapfoldl`, `mapfoldr`, `mapreduce`, and `mapslices`. The function `map!` stores the result in its second argument, i.e., the first sequence argument (see Sect. 2.2). It follows the convention that functions with names that end in `!` modify their arguments. This convention stems from the programming languages SCHEME.

The function `reduce` is another mainstay of functional programming. It takes an associative function as its first argument, a collection as its second, and an initial value as the (optional) keyword argument `init`. The initial value should be the neutral element for applying the function to an empty collection. `reduce` applies the function to two values from the collection (except for the initial value) repeatedly until the collection has been reduced to a single value. The functions `foldl` and `foldr` are similar, but guarantee left or right associativity.

```
julia> reduce(*, [1 2 3])
6
julia> reduce(*, [1 2 3], init = -1)
-6
```

The function `mapreduce` maps and reduces. It maps its first argument (a function) over the sequence given as the third argument and then reduces the result using the second argument (again a function) with the initial value (optionally) given as the keyword argument `init`. `mapreduce` is more efficient than using `map` and `reduce`, since the intermediary sequence is not stored. The example calculates the ℓ^p norm of a vector, where an anonymous function is used for $x \mapsto |x|^p$.

```
norm(v::Vector, p::Number) = mapreduce(x -> abs(x)^p, +, v)^(1/p)
```

2.6 Optional Arguments

Arguments often have sensible default values. For example, the ℓ^2 norm is the most popular among the ℓ^p norms. In these cases, it is convenient to declare these arguments as optional arguments. Optional arguments do not have to be specified in the argument list when the function is called.

```
norm(v::Vector, p::Number = 2) =
    mapreduce(x -> abs(x)^p, +, v)^(1/p)

julia> norm([-1, -2, -3])
3.7416573867739413
julia> norm([-1, -2, -3], 1)
6.0
```

Optional arguments are implemented as methods of the generic function. For example, a function with one optional argument is implemented as two methods (one method for no arguments and one method for one argument) and a function with two optional arguments is implemented as three methods (one method each for the three cases of zero, one, or two arguments).

2.7 Keyword Arguments

When functions have many arguments or many optional arguments in particular, it is often clearer in the caller to use keyword arguments. Keyword arguments allow the arguments to be identified not by position, but by name. Keyword arguments follow a semicolon in the function signature, and default values must be provided. The example is an implementation of the Newton method for finding roots of nonlinear functions.

```
function newton(f::Function, df::Function, x0; typ = Float64,
                tol = 1e-6, max_iterations::Integer = 100)
    @assert tol > 0
    @assert max_iterations >= 1

    local x = typ(x0)
    local i = 1
    while abs(f(x)) >= tol && i <= max_iterations
        x += -f(x)/df(x)
        @show i, x, f(x)
        i += 1
    end
    x
end
```

If you are not interested in printing the progress of the calculation, you can add a comment character # in front of the call of the `@show` macro. In this line, a tuple containing the three values `i`, `x`, and `f(x)` is created and passed to the `@show` macro. Also note that **`type`** is a reserved word in JULIA, so that we use the name `typ` for the first keyword argument instead.

The first argument is the function whose zero is sought starting from the point specified as the third argument. In this implementation we require the derivative, a function, to be passed as the second argument. The `typ` keyword argument specifies the type of the result by converting the starting value to this type. The `tol` keyword argument specifies how large the absolute value of the function value at the final point may be at most. The last keyword argument specifies the maximum number of iterations calculated and ensures that the function always returns.

We note that it is possible to specify the type of keyword and optional arguments, as we have done here for the last keyword argument.

The `@assert` macro takes an expression as its argument and evaluates it. If the value is **`true`**, the function continues; if it is **`false`**, an error is raised. Such assertions are commonly used to check that the input is valid.

Two syntactic options are valid when calling the function. The semicolon that separates the positional arguments and the keywords arguments in the function definition is often not required when calling the function and can then be replaced by a comma.

The first example illustrates that the type of the result is given by the `typ` keyword argument. Here the requested tolerance exceeds the precision of the floating-point type **`Float32`** so that the function returns after the maximum number of iterations.

```
julia> newton(sin, cos, 3.0, typ = Float32, tol = 1e-15,
              max_iterations = 5)
(i, x, f(x)) = (1, 3.1425467f0, -0.000954f0)
(i, x, f(x)) = (2, 3.1415927f0, -8.742278f-8)
(i, x, f(x)) = (3, 3.1415927f0, -8.742278f-8)
(i, x, f(x)) = (4, 3.1415927f0, -8.742278f-8)
(i, x, f(x)) = (5, 3.1415927f0, -8.742278f-8)
3.1415927f0
```

In the next example, the type is not specified so that the default type **`Float64`** is used.

```
julia> newton(sin, cos, 3.0, tol = 5e-16)
(i, x, f(x)) = (1, 3.142546543074278, -0.0009538893398264409)
(i, x, f(x)) = (2, 3.141592653300477, 2.8931624907621843e-10)
(i, x, f(x)) = (3, 3.141592653589793, 1.2246467991473532e-16)
3.141592653589793
```

We will learn more about the Newton method and its convergence behavior in Chap. 12.

Keyword arguments are ignored in method dispatch, i.e., when searching for a matching method of the generic function. Keyword arguments are only processed after the matching method has been found.

Functions can also receive a variable number of keyword arguments at the end of the argument list using the syntax `...` after the name of variable that will receive all remaining keyword arguments as a collection. The function in this example just returns the collection containing all keyword arguments it has received.

```
foo(a; b = 0, c...) = c
```

When calling this function, a semicolon must be used after the keyword argument `b` so that the keyword arguments collected in `c` can be distinguished from the preceding keyword argument `b`.

```
julia> foo(1, b = 2)
Iterators.Pairs(::NamedTuple{(),Tuple{}}, ::Tuple{}) with 0 entries
julia> foo(1, b = 2; bar = 3, baz = 4)
pairs(::NamedTuple) with 2 entries:
  :bar => 3
  :baz => 4
julia> foo(1, b = 2; :bar => 3, :baz => 4)
pairs(::NamedTuple) with 2 entries:
  :bar => 3
  :baz => 4
```

Two syntactic options to pass the keyword arguments are shown here. The first is the usual keyword-argument syntax *variable* = *value*, and the second are pairs of the form `:`*variable* => *value*.

This general facility for passing keyword arguments is useful when the keyword names are computed at runtime or when a number of keyword arguments is assembled and passed through one or more function calls and the receiving functions picks the keyword arguments it needs.

To summarize, keyword arguments are arguments after a semicolon `;` in the argument list of a function definition, while optional arguments are listed before the semicolon.

2.8 Functions with a Variable Number of Arguments

Sometimes, it is convenient for a function to take a variable number of arguments. The syntax to supply the arguments as a tuple is that the last argument is followed by an ellipsis `...`.

```
foo(a, b, c, args...) = args
```

The variable `args` is bound to a tuple of all the trailing values passed to the function.

```
julia> foo(1, 2, 3)
()
julia> foo(1, 2, 3, 4, 5, 6)
(4, 5, 6)
```

Analogously, the ellipsis `...` can be used in a function call to splice the values contained in an iterable collection (see Sect. 4.5.2) into a function call as individual arguments.

```
julia> foo((1, 2, 3, 4, 5, 6)...)
(4, 5, 6)
julia> foo([1, 2, 3, 4, 5, 6]...)
(4, 5, 6)
```

This example shows that the spliced arguments can also take the place of fixed arguments. In fact, the function call taking a spliced argument list does not have to take a variable number of arguments at all.

2.9 `do` blocks

Built-in functions that take a function as one of its arguments usually receive the function argument as the first argument, which is an idiomatic use of function arguments in JULIA. A **do** block is a syntactic expression that supports this idiom. They are useful for passing longer anonymous functions as first arguments to functions. The **do** block

```
function(arguments) do variables
    body
end
```

is equivalent to

```
function(variables -> body, arguments)
```

so that the possibly long function body *body* is written at the end of the **do** block.

Continuing the above example, the `norm` function can equivalently also be defined as follows.

```
norm(v::Vector, p::Number) =
    mapreduce(+, v) do x
        abs(x)^p
    end^(1/p)
```

A prime example where it is convenient to pass long anonymous function bodies to a function is file input and output. The function in the next example ensures that the stream that has been opened to read from or to write to is closed after use. The function takes a variable number of arguments (see Sect. 2.8).

```
function with_stream(fun::Function, args...)
    local stream = open(args...)
    try
        fun(stream)
    finally
        close(stream)
    end
end
```

The function that operates on the input/output stream may be quite complicated. Then it is therefore convenient to use a **do** block. In the following example, `s` is the stream on which the anonymous function body in the **do** block operates. While the end of the file has not been reached, a line is read and printed.

```
with_stream("/etc/passwd", "r") do s
    while !eof(s)
       print(readline(s))
    end
end
```

It should be noted in this context that JULIA comes with the `readlines` function that returns the contents of a file as a vector of strings. `readlines` is often sufficient when the file to be processed is small.

Problems

2.1 Write a function that uses `BigFloat`s and `setprecision` (in conjunction with a `do` block) to calculate large Fibonacci numbers. What is the largest Fibonacci number you can calculate in this manner and what is the limitation you eventually run into?

2.2 Write a function that records the number of calls of `fib4(`m`)` for each $0 \le m < n$ when calculating `fib4(`n`)`.

2.3 Calculating larger and larger Fibonacci numbers using `fib7` is limited by the stack size. However, if you gradually increase the size of the argument, you can circumvent this limitation. Explain why. What is the largest Fibonacci number you can calculate in this manner and what is the limitation you eventually run into?

2.4 The following function is a shorter alternative to `fib7` because of its use of `get!`. However, it does not work. Explain why.

```
global fib_cache = Dict{BigInt, BigInt}(0 => 0, 1 => 1)

function fib(n::Integer)
    global fib_cache
    get!(fib_cache, BigInt(n), fib(n-1) + fib(n-2))
end
```

2.5 (Identities for Fibonacci numbers) Suppose x_n is the n-th Fibonacci number.
(a) Prove d'Ocagne's identity (2.2).
(b) Prove the identity (2.3).
(c) Prove the identity (2.4).

2.6 Suppose x_n is the n-th Fibonacci number. Prove the identity

$$x_{an+b} = \sum_{i=0}^{a} \binom{a}{i} x_{b-i} x_n^i x_{n+1}^{a-i}$$

for all $a \in \mathbb{N}$ and $b \in \mathbb{N}$. Then use it to implement an efficient memoized function to calculate Fibonacci numbers. What is the largest Fibonacci number you can calculate in this manner and what is the limitation you eventually run into?

2.7 Use the macro `@timed` to plot the time and memory consumption of the various functions to calculate Fibonacci numbers.

2.8 (Ackermann function) The Ackermann function is defined for $m \in \mathbb{N}$ and $n \in \mathbb{N}$ as

$$A(m,n) := \begin{cases} n+1, & m = 0, \\ A(m-1,1), & m > 0 \ \land \ n = 0, \\ A(m-1, A(m, n-1)), & m > 0 \ \land \ n > 0. \end{cases}$$

Implement this function and also implement a memoized version. Compare the speed of both versions.

Chapter 3
Variables, Constants, Scopes, and Modules

But I just told you, I don't have a problem with closure.

—Sheldon Lee Cooper in *The Big Bang Theory,* Season 6, Episode 21
The Closure Alternative (2013)

Abstract This chapter discusses how to introduce global and local variables and constants as well as their scopes or visibility. While discussing functions, we have seen that function arguments become local variables in the function body, but there are more ways to introduce variables. Local variables are only visible in (small) parts of a program, which is an important property to structure a program into small, understandable parts. Global variables are only visible in their module. A module is a (usually large) part of a program that contains functions, variables, and constants with a similar purpose. The scopes of variables follow rules that are described in detail.

3.1 Modules and Global Scopes

The scope of a variable is defined as the part of a program where the variable is visible. Modules are a fundamental data structure in this regard, as each module corresponds to a global scope. There is one-to-one correspondence between modules and global scopes in JULIA. The global scope of the REPL is the module called `Main`.

A new module called `Foo` can be defined in a JULIA program or at the REPL like this.

```
module Foo1
global foo = 1
end
```

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_3

Modules usually have names that start with an uppercase letter. Program lines within a module are not indented, since a module almost always comprises a whole file and indenting the whole file would be superfluous. Here we have also defined a variable called `foo`, whose global scope is the module `Foo1`. Although it is good practice to use the keyword **`global`** to define a global variable, it is not necessary to do so.

```
module Foo2
foo = 2
end
```

A module evaluates to itself, and we can access variables within a module using a dot `.`, which signifies a qualified access.

```
julia> Foo1
Main.Foo1
julia> Foo1.foo
1
```

Modules can be replaced by evaluating a module definition for the same name. Modules can be nested, and modules can be imported into other modules, as the next example shows. Here the lines are indented to illustrate the nesting.

```
module Foo3
    module Bar
        global bar = 0
    end
    global foo3 = Bar.bar  # module Bar is visible
    import ..Foo1          # make module Foo1 visible
    global foo4 = Foo1.foo # module Foo1 is visible
end

julia> Foo3.Bar.bar
0
julia> Foo3.foo3
0
julia> Foo3.foo4
1
```

As a safety measure, it is not possible to assign values to variables in the global scope of another module, as illustrated in this example, which raises an error.

```
module Foo4
import ..Foo1
Foo1.foo = 4
end
```

3.2 Dynamic and Lexical Scoping

We have defined the scope of a variable as the part of a program where the variable is visible. In addition to the global scopes of modules, there are also local scopes. For example, a function definition introduces a new local scope.

What happens if there are two variables with same name within a program? If the scopes of the variables do not overlap, there is no ambiguity. If the scopes of the variables overlap, however, then JULIA's scope rules are applied in order to resolve any ambiguities. This section discusses the various scopes of variables and the scope rules.

The scope of a variable is a concept that is familiar from mathematics. An example is given by integrals. In the formula

$$f(x) = \int_a^x f'(\xi)\mathrm{d}\xi,$$

the scope of the integration variable ξ is the integrand, i.e., ξ is only visible within the integrand, which is $f'(\xi)$ here. But how should we interpret the formula

$$f(x) = \int_a^x f'(x)\mathrm{d}x?$$

Is it ambiguous? From dx we know that the integration variable is called x. The integration variable x is certainly different from the upper integration limit x, since the upper integration limit must always be outside of the scope of the integration variable. Since the left-hand side is a function of x, the right-hand side must be one as well, and therefore the upper integration limit x is the function argument x on the left-hand side. But is the x in the integrand $f'(x)$ the function argument x or the integration variable x? Their scopes overlap and therefore we need a scope rule to resolve this ambiguity. A reasonable scope rule is to require that any x in the integrand refers to the innermost definition of x, i.e., the integration variable x, and not to the outer x, i.e., the function argument x.

Another example is given by summation indices, which are only visible within their summands. For example, in the formula

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s},$$

the summation index n is only visible within the summand $1/n^s$, while the function argument s is visible in the whole right-hand side.

Returning from mathematics to computer science, there are two types of scoping, namely dynamic scoping and lexical scoping. In lexical scoping, the scope of a variable is the program text of the scope block where the variable is defined. In dynamic scoping, the scope of a variable is the time period when the code block where the variable is defined is executed. JULIA uses lexical scoping.

Nowadays, lexical scoping is more popular than dynamic scoping, since it makes it easier for the programmer to reason about the variables defined within a code block. The (local) program text is sufficient. Dynamic scoping, on the other hand, requires knowledge about the run-time behavior of the program, which can be arbitrarily complex. COMMON LISP is an example of a programming language that provides both types of scoping for its variables. While lexical scoping is the default, dynamic scoping is advantageous for certain purposes.

To illustrate the difference between dynamic and lexical scoping, suppose that a function called `f` calls another function called `g` and that these two code blocks do not overlap, i.e., none of the two functions is defined within the other one. Under lexical scoping, the function `g` does not have access to the variables introduced by `f`, because its program text does not lie within `f`, whereas under dynamic scoping, `g` does have access to the variables introduced by `f`, since it is executed while `f` is executed.

This is the situation implemented in this example.

```
function f()
    local x = 0
    g()
end

function g()
    x
end
```

Calling `f` results in a call to `g`. Since JULIA uses lexical scoping, evaluating `x` inside the scope block of the function `g` can only refer to variables defined in `g` and the global scope outside of `g` (i.e., the module that contains *g*). Because `x` is defined in neither, an error saying that *x* is not defined is raised.

If we introduce a global variable `x` outside the scope of `g` (i.e., in the module that contains *g*), then evaluation of `x` in `g` will refer to its value.

```
julia> x = 1; f()
1
```

We could have left out the keyword **`global`** here, but it is good style to indicate the definition of variables using the keywords **`global`** or **`local`**.

The next example is concerned with the nesting of local scopes. The function `inner` is defined within the function `outer`.

```
function outer()
    function inner()
        y
    end

    local y = 0
    inner()
end
```

The `inner` function inherits all variables from its outer scope, i.e., the `outer` function, so that the variable `y` inside `inner` refers to the value of `y` in `outer`.

```
julia> outer()
0
```

Even defining a global variable `y` cannot change the return value of `outer`.

```
julia> y = 1; outer()
0
```

Because of lexical scoping, the definition of the `outer` function is sufficient to determine its return value. It is impossible that a global variable overrides a local variable with the same name unless this is the intended behavior by using the keyword **`global`** as we did in Sect. 2.1.

3.3 Local Scope Blocks

In JULIA, there are eight types of local scope blocks, all listed in Table 3.1, which also lists the three ways to introduce global scope blocks for completeness. There are two types of local scope blocks, namely hard local scopes and soft local scopes. On the other hand, **`begin`** blocks and **`if`** blocks are not scope blocks and cannot introduce new variable bindings.

Table 3.1 All global and local scope blocks.

Scope block	Scope type
`module`	global
`baremodule`	global
REPL (`Main` module)	global
`function` bodies and **`do`** blocks	hard local
`macro` bodies	hard local
`struct` blocks	hard local
`for` loops	soft local
`while` loops	soft local
`try` **`catch`** **`finally`** blocks	soft local
`let` blocks	soft local
array comprehensions	soft local

3.3.1 Hard Local Scopes

According to Table 3.1, functions (see Chap. 2), macros (see Chap. 7), and **`struct`** type definitions (see Sect. 5.4) introduce new hard local scopes. The scope rules for hard local scopes are these.

1. All variables are inherited from their parent scope with the following two exceptions.
2. A variable is *not* inherited if an assignment would modify a global variable. (A new binding is introduced instead.)
3. A variable is *not* inherited if it is marked with the keyword **`local`**. (A new binding is introduced instead.)

The second rule means that global variables are inherited only if they are read, not if they are modified. This ensures that a local variable cannot unintentionally modify a global variable with the same name.

```
global x = 0

function foo1()
    x = 1 # introduce new local variable
    x
end

function foo2()
    global x = 2 # assign to global variable
    x
end
```

In the first function, the assignment `x = 1` would modify the global variable `x`, and therefore a new local variable is introduced. In the second function, the **`global`** keyword ensures that the assignment `x = 2` refers to the global variable `x`, whose binding is modified.

```
julia> foo1()
1
julia> x
0
julia> foo2()
2
julia> x
2
```

3.3.2 Soft Local Scopes

According to Table 3.1, **for** loops, **while** loops, **try catch finally** blocks (see Chap. 6), **let** blocks (see Sect. 3.4), and array comprehensions (see Sect. 3.5) introduce new soft local scopes. The scope rules for hard local scopes are these.

1. All variables are inherited from their parent scope with the following exception.
2. A variable is *not* inherited if it is marked with the keyword **local**. (A new binding is introduced instead.)
3. Additional rules for **let** blocks (see Sect. 3.4) and **for** loops and comprehensions (see Sect. 3.5) apply.

Hard and soft local scopes differ in their intended purposes and hence in their scope rules. Hard local scopes, i.e., function, macro, and type definitions, are usually independent entities than can be moved around freely within a program. Modifying global variables within their scopes is possible, but should be done with care, and therefore requires the **global** keyword. On the other hand, soft local scopes such as loops are often used to modify variables that are defined in their parent scopes. Hence the default is to modify variables unless the **local** keywords is used.

The following two examples involving **for** loops illustrate soft local scopes.

```
function sum(args...)
    local sum = 0     # introduce new local variable
    for i in args
        sum = sum + i # inherit
    end
    sum
end
```

Here the assignment `sum = sum + i` does not introduce a new binding for the variable `sum` in the **for** loop, since it is inherited from the parent scope by the first rule.

The situation is different in the following example.

```
function foo(args...)
    local x = 0     # introduce new local variable
    for i in args
        local x = i # introduce new local variable
    end
    x
end
```

Here the **local** keyword always introduces a new variable in the scope of the **for** loop. Therefore this function always returns `0`.

Named functions (in contrast to anonymous functions) are stored as **Function** objects in variables. Therefore a function `f` can be referred to in the definition of a function `g` even if `f` has not been defined yet. An example is given by mutually recursive functions. In JULIA, function definitions can be ordered arbitrarily and no forward function declarations are required as in some other programming languages in such cases.

Whether a variable is defined can be checked using the macro `@isdefined` and the function `isdefined`.

3.4 `let` Blocks and Closures

Closures are a concept in computer science that can be found in many modern programming languages. In general, a closure is a function together with an environment (or set of bindings) of variables. Variables in the enclosing scope are called free variables and can be accessed by the function even when the function is called outside the scope. This behavior is consistent with lexical scoping.

In JULIA, closures are based on **`let`** blocks. A **`let`** block has the syntax

`let` *variable1* [= *value1*], *variable2* [= *value2*], *variable3* [= *value3*]
    body
`end`

and accepts an arbitrary number of assignments separated by commas while the values are optional. The assignments are evaluated in order. A **`let`** block always introduces a new scope block and new local variables each time it is evaluated; this is the additional scope rule for **`let`** blocks.

A closure is created by a **`let`** block containing one or more function definitions. In this example, the local variable `counter` is captured and visible only in the two functions `get_counter` and `increase` due to lexical scoping. Thus the closure serves to encapsulate the data, as the variable `counter` can only be accessed and modified by the functions defined inside the **`let`** block.

```
let counter = 0
    global function get_counter()
        counter
    end

    global function increase()
        counter += 1
    end
end
```

The **`global`** declarations of the functions are necessary. Otherwise the function definitions would only be accessible inside the **`let`** block (and not globally), as function definitions are stored in variables. (In COMMON LISP, the **`global`** declaration would not be needed.)

```
julia> get_counter()
0
julia> increase(), increase(), increase()
(1, 2, 3)
julia> get_counter()
3
```

After reevaluating the **`let`** block above, a new closure is created and the counter is again equal to `0`.

3.5 `for` Loops and Array Comprehensions

Array comprehensions are a convenient way to make (dense) arrays (see Sect. 8.1) while initializing its elements. A multidimensional array can be constructed by `[`*expr* **`for`** *variable1* = *value1*`,` *variable2* = *value2*`]`, where an arbitrary number of iteration variables can be used. The values on the right-hand sides must be iterable objects such as ranges. Then the expression *expr* is evaluated with freshly allocated iteration variables. The dimensions of the resulting array are given by the numbers of the values of the iteration variables in order.

This example shows how an array comprehension is used to make and initialize a two-dimensional array.

```
julia> [10x + y for x in 1:2, y in 1:3]
2×3 Array{Int64,2}:
 11  12  13
 21  22  23
julia> size(ans)
(2, 3)
```

The iteration variables are freshly allocated for each iteration of the comprehension, and hence any previously existing variable bindings with the same name are not affected by the array comprehension.

```
julia> x = 0; y = 0
0
julia> [10x + y for x in 1:2, y in 1:3]
2×3 Array{Int64,2}:
 11  12  13
 21  22  23
julia> x, y
(0, 0)
```

The behavior of **`for`** loops is the same in this regard. We consider two examples. In the first, the iteration variable has not been previously defined. In this case, the iteration variable is local to the **`for`** loop.

```
julia> @isdefined i
false
```

This shows that initially there is no binding for the variable `i`.

```
julia> for i in 1:2 end
```

After the **`for`** loop, there is again no binding for the variable `i`.

```
julia> @isdefined i
false
```

In the second example, a variable with the same name as the iteration variable already exists. After the **`for`** loop has been evaluated, the value of the iteration variable remains unchanged.

```
julia> j = 0
0
julia> for j in 1:2 end
julia> j
0
```

3.6 Constants

Both global and local variables can be declared constant by the **`const`** keyword. Declaring global variables as constant helps the compiler to optimize code. Since the types and values of global variables may change at any time, code involving global variables can generally hardly be optimized by the compiler. If a global variable is declared constant, however, the compile can employ type inference and the performance problem is solved.

The situation is different for local variables in this regard. The compiler can determine whether a local variable is constant or not, and therefore declaring local variables constant does not affect performance.

Finally, we note that declaring a variable constant only affects the variable binding. If the value of a constant variable is a mutable object such as a set, an array, or a dictionary (see Chap. 4), the elements of the mutable object may still be modified as shown in this example.

```
julia> const A = [1, 2]
2-element Array{Int64,1}:
 1
 2
julia> A[1] = 0; A
2-element Array{Int64,1}:
 0
 2
```

3.7 Global and Local Variables in this Book

This book uses the `global` and `local` keywords to explicitly denote global and local variables, although many programmers usually do not do so in practice. There are two reasons for the use of these two keywords here. The first is simply a didactic reason; the keywords clearly indicate where a new variable is defined and of which kind it is.

The second reason is that writing `global` and `local` explicitly to define (and to access) variables can be considered good practice, because it helps spot the declarations of global and local variables at a glance and also where they are used (in the case of global ones). Global variables are always noteworthy, and therefore deserve to be spotted easily.

Using the `global` and `local` keywords is also a matter of style and personal preference. JULIA's syntax is heavily influenced by PASCAL's syntax, and in PASCAL all variables are introduced by the `var` and `const` keywords. In this tradition, the `local` keyword serves the role of `var` in PASCAL. On the other hand, with more experience in spotting variables and recognizing their scopes, the keywords may appear superfluous.

Problems

3.1 Extend the example of a closure in Sect. 3.4 by writing functions for resetting the counter to a given value and for decreasing the counter.

3.2 A Hilbert matrix is a square matrix H with the entries $h_{ij} = 1/(i + j - 1)$. Write a function that returns a Hilbert matrix of given size.

Chapter 4
Built-in Data Structures

Data dominates.
If you've chosen the right data structures and organized things well,
the algorithms will almost always be self-evident.
Data structures, not algorithms, are central to programming.

—Rob Pike

Bad programmers worry about the code.
Good programmers worry about data structures and their relationships.

—Linus Torvalds

Abstract JULIA comes with many useful, built-in data structures that cover many requirements of general-purpose programming. In this chapter, the most important built-in data structures are discussed, including characters, strings, regular expressions, symbols, expressions, and several types of collections. In conjunction with the data structures, the operations on the data structures are introduced as well and examples of their usage are given.

4.1 Characters

One of the simplest, but most fundamental data structures is the character, the type **`Char`**. A character is created by `'`*char*`'`, i.e., using single quotes. Each character corresponds to a Unicode code point, and a character can be converted to its code point, which is an integer value, by calling **`Int`**.

```
julia> 'a', typeof('a'), Int('a'), typeof(Int('a'))
('a', Char, 97, Int64)
```

The type of **`Int`**`('a')` depends on your system architecture.

Vice versa, a code point, i.e., an integer, can be converted to a character by calling **`Char`**.

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_4

```
julia>  Char(97)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

However, not all integers are valid Unicode code points. You can check if an integer is a valid code point by using `isvalid(`**`Char`**`,` *integer*`)`.

Any Unicode character can be input in single quotes using `\u` followed by up to four hexadecimal digits or using `\U` followed by up to eight hexadecimal digits; for example, `'\u61'` is the letter a. Furthermore, some special characters can be escaped using a backslash: the backslash character `'\\'`, the single quote `'\''`, newline (line feed) `'\n'`, carriage return `'\r'`, form feed `'\f'`, backspace `'\b'`, horizontal tab `'\t'`, vertical tab `'\v'`, and alert (bell) `'\a'`. Additionally, the character with octal value *ooo* (three octal digits) can be written as `'\`*ooo*`'`, and the character with hexadecimal value *hh* (two hexadecimal digits) can be written as `'\x`*hh*`'`.

The standard comparison operators `==`, `!=`, `<`, `<=`, `>`, and `>=` are available for characters. Furthermore, the function `-` is defined for characters as well.

```
julia> '0' < 'A' < 'a'
true
julia> @which '0' < 'A'
<(x, y) in Base at operators.jl:268
julia> @which 'z' - 'a'
-(x::AbstractChar, y::AbstractChar) in Base at char.jl:221
julia> 'z' - 'a' + 1
26
```

4.2 Strings

4.2.1 Creating and Accessing

JULIA strings are immutable, i.e., once they have been created, they cannot be changed anymore. Strings are delimited either by double quotes `"` or by triple double quotes `"""`. Characters can be entered as part of a string using the syntax for Unicode characters and the one for special characters mentioned in Sect. 4.1. A double quote needs to be escaped by a backslash if the string is delimited by double quotes, while it can occur unaltered inside a string delimited by triple double quotes.

```
julia> println("123\b\b\b456456\r789")
789456
julia> """This is an "interesting" example."""
"This is an \"interesting\" example."
```

Triple double quotes are useful for creating longer blocks of text such as documentation strings. White space receives special treatment, however, when this syntax is used. When the opening triple quotes are immediately followed by a newline, then the first newline is ignored. Trailing white space at the end of the string remains unchanged, however. The indentation of a triple-quoted string is also changed when it is read. The same amount of white space at the beginning of each line is removed so that the input line with the least amount of indentation is not indented at all in the final string. This behavior is useful when a string occurs as part of indented code.

The elements of a string are characters and can be accessed by their index. Indices in JULIA always start at `1` and end at **`end`**. The **`end`** index can be used in computations as in this example.

```
julia> s = "Hello, world!"
"Hello, world!"
julia> s[1], s[2], s[floor(Int, end/2)], s[end-1], s[end]
('H', 'e', ',', 'd', '!')
```

Substrings can be extracted using any range *start*`:`*step*`:`*end*. The step of a range equals 1 by default, i.e., the two ranges *start*`:1:`*end* and *start*`:`*end* are equivalent.

```
julia> s[8:12]
"world"
julia> s[1:2:length(s)]
"Hlo ol!"
```

Note the difference between accessing an element of a string (which returns a character) and accessing a substring (which returns a string).

```
julia> typeof(s[1]), typeof(s[1:1])
(Char, String)
```

However, strings can consist of arbitrary Unicode characters. Since the Unicode encoding is a variable-length encoding, i.e., not all characters are encoded by the same number of bytes, accessing a string at an arbitrary byte position by `[]` does not necessarily yield a valid Unicode character. The number of Unicode characters in a string is returned by `length`. In this example, the string consists of four Unicode characters, where only the second character occupies one byte.

```
julia> a = "\u2200x\u2208\u211d"
"∀x∈ℝ"
```

Trying to access `a[2]` and `a[3]` results in errors, while `a[4]` evaluates to `'x'`.

It is possible to step through a string using `nextind`, which returns the next valid index after a given index, as this example illustrates.

```
julia> nextind(a, 1)
4
julia> nextind(a, 4)
5
julia> nextind(a, 5)
```

```
8
julia> a[1], a[4], a[5], a[8]
('∀', 'x', '∈', 'ℝ')
```

However, the most convenient way to iterate through a string is using a **`for`** loop.

```
julia> for char in a println(char) end
∀
x
∈
ℝ
```

4.2.2 String Interpolation

Strings can be concatenated using the `string` function. Often it is convenient to substitute a substring given by an expression into a mostly constant string. This procedure is called string interpolation in JULIA. In its most general form, the shortest complete expression after a dollar sign `$` is evaluated, and the resulting printed form of the expression is interpolated into the string. A syntax that always works for an expression *expr* is `$(`*expr*`)` inside a string. In the simplest form of string interpolation, the printed form of a variable is interpolated into the string by using `$`*var*. Finally, a dollar character can be included in a string by escaping it with a backslash like this `\$`.

```
julia> "Command-line arguments are: $ARGS"
"Command-line arguments are: String[]"
julia> "exp(im*pi) = $(exp(im*pi))"
"exp(im*pi) = -1.0 + 1.2246467991473532e-16im"
```

4.2.3 String Operations

A common string operation is sorting. Since the standard comparisons ==, `!=`, <, <=, >, and >= implement lexicographical comparison of strings based on character comparison, the `sort` function can be used to sort strings.

To check whether a string contains a character, the **`in`** function can be used, which also supports infix syntax.

```
julia> in('\u2200', a)
true
julia> '\u2200' in a
true
```

To check whether a string contains another string, the `occursin` function, which returns a Boolean value, can be used.

```
julia> occursin("world", s)
true
```

The `findfirst` function returns more information. Its first argument can be a character, a string, or a regular expression (see Sect. 4.2.5 below). The first argument is searched for in the second argument, a string, and `findfirst` returns the (byte) indices of the matching substring or **`nothing`** if there is no such occurrence.

```
julia> findfirst('x', a)
4
julia> findfirst('x', s)
julia> ans == nothing
true
```

This example shows that printing **`nothing`** prints nothing. The most recently returned value is stored in the variable `ans`, however, and we could check that it is equal to **`nothing`**.

The `replace` function replaces a substring of a string with another one. Since strings are immutable, a new string is always returned. The second argument indices the substitution, and it can be a pair (see Sect. 4.5.4), a dictionary (see also Sect. 4.5.4), or a regular expression (see Sect. 4.2.5). The replacement can be a (constant) string or a function that is applied to the match and that yields a string. The keyword argument `count` indicates how many occurrences are replaced.

```
julia> replace(s, "world" => "World")
"Hello, World!"
julia> replace(s, "w" => uppercase)
"Hello, World!"
julia> replace(s, r"[a-z]" => x -> Char(Int(Char(x[1])) + 1))
"Hfmmp, xpsme!"
```

Here the regular expression `r"[a-z]"` matches all lowercase characters. They are replaced by the character that follows them in the character ordering.

Strings can be assembled from substrings using `*`, `repeat`, and `join`. The function `*` (and not `+`) concatenates two strings. The `repeat` function concatenates a given number of characters or strings. The `join` function concatenates an array of strings, inserting a given delimiter string between adjacent strings. An optional second delimiter may be given; it is then used as the delimiter between the last two substrings.

```
julia> join(["integers", "rationals", "real numbers"],
            ", ", ", and ")
"integers, rationals, and real numbers"
```

4.2.4 String Literals

The syntax *s*"*string*" makes it possible to conveniently create certain objects (whose type is determined by the string *s*) from their string representations. More information about the general mechanism can be found in Sect. 7.6. Here we discuss three built-in types of such string literals. Reading such expressions creates regular expressions using `r` (see Sect. 4.2.5), regular-expression substitution strings using `s` (see also Sect. 4.2.5), and version numbers using `v`.

```
julia> typeof(b"foo")
Base.CodeUnits{UInt8, String}
julia> typeof(r"foo")
Regex
julia> typeof(s"foo")
SubstitutionString{String}
julia> typeof(v"1.2.3")
VersionNumber
```

Strings prefixed by `r` become regular expressions, and substitution strings are prefixed by `s`. They are discussed in Sect. 4.2.5 below.

Strings prefixed by `v` are read as objects of type `VersionNumber`. Version numbers follow the specifications of semantic versioning and hence consist of a major version, a minor version, and a patch version. Everything except the major version number is optional. Version numbers are mostly useful for executing code contingent on the JULIA version and for specifying dependencies on certain version numbers. When comparing two version numbers, a + or – may be appended to indicate a version higher or lower than any of the given version numbers.

```
julia> if v"1.0" <= VERSION < v"1.1-"
           println("Code specific to the version 1.0 release
                   series.")
       end
julia> if v"1" <= VERSION < v"2-"
           println("Code specific to the version 1 release series.")
       end
Code specific to the version 1 release series.
```

Here the version number `v"1.1-"` indicates a version lower than any 1.1 release or pre-release. It is good practice to append a trailing – to version numbers in upper bounds unless there is a specific reason not to do so.

Often the version number should be checked not at run time, but already when the program is parsed into expressions. The `@static` macro makes it possible to perform such a check as in the following example.

```
julia> [@static if v"1" <= VERSION < v"2-" 1 else :unknown end]
1-element Array{Int64,1}:
 1
```

The converse string operation, namely splitting strings, is implemented by the function `split`.

4.2.5 Regular Expressions

Regular expressions are a powerful way to check whether a string matches a given regular pattern and to extract substrings from any matches. For example, regular expressions are convenient for extracting information from unstructured data, for scraping websites, and for parsing large textual input. There are many variants and implementations of regular expressions; JULIA uses the PCRE (PERL compatible regular expressions) library, a fast implementation of a common form of regular expressions. This section gives an introduction to regular expressions and how to use them in JULIA by discussing a few examples, but there are many more options.

Many characters have a special meaning inside a regular expression. Common characters with special meaning are `?`, `+`, and `*`. They allow repetitions of the preceding character or group. The question mark `?` allows no occurrence or exactly one occurrence. For example, the regular expression `r"1?"` matches the empty string or `"1"`. The plus sign `+` allows at least one occurrence. For example, the regular expression `r"1+"` matches `"1"`, `"11"`, `"111"`, and so forth. Third, the asterisk `*` allows any number of occurrences including none. For example, the regular expression `r"1*"` matches the empty string, `"1"`, `"11"`, and so forth.

The `match` function takes a regular expression and a string as its arguments as well as the index where to start the search as an optional argument. If there is a match, a **`RegexMatch`** object with information about the first match is returned; otherwise, **`nothing`** is returned. We have already seen above that **`nothing`** is treated specially by the REPL in the sense that nothing is printed if it is the result of an evaluation.

The example we consider here is extracting information from a text file or the textual output of a program. The lines of a file can conveniently be read by the functions `open` and `readlines` (if it is not too large). Calls to external programs are entered between backquotes, and string interpolation (see Sect. 4.2.2) is performed between backquotes. The output of an external program can then be captured in a string or in an array of strings as in these examples.

```
julia> read(`echo Hello, world!`, String)
"Hello, world!\n"
julia> readlines(`echo Hello, world!`)
1-element Array{String,1}:
 "Hello, world!"
```

We define the string `s` as an example and then check whether it contains at least one digit two or at least two consecutive nines.

```
julia> s = """
    movie name: War Games
    release date: May 7, 1983
    researcher: Stephen Falken
    artificial intelligence: Joshua
    computer name: WOPR (War Operation Plan Response)
    games: Falken's maze, chess, poker, global thermonuclear war
    hero: David L. Lightman
    status: world saved
    phone numbers from: 311_936_0001
    phone numbers to:   311_936_9999""";
julia> match(r"2", s)
julia> match(r"9+", s)
RegexMatch("9")
```

To search for occurrences not only of single characters, but of more complicated patterns, characters and patterns can be grouped by brackets `(`*pattern*`)`. Such a group can also be named so that one can refer to it more conveniently than by index, as we will see later. A named group looks like `(?<`*name*`>`*pattern*`)`. A group consisting of two alternatives *pattern1* and *pattern2* is written as `(`*pattern1*`|`*pattern2*`)`.

```
julia> m = match(r"(?<foo>123|456|789)(a|b|c)", "123abc")
RegexMatch("123a", foo="123", 2="a")
julia> m["foo"], m[2]
("123", "a")
```

We see that a matched group can be accessed by its (numerical) index (here `2`) or by the name of the group (here `"foo"`, using the string `"foo"` as the index). Furthermore, the details of the match `m` can be inspected by `dump(m)`. The fields `match`, `captures`, `offset`, and `offsets` may be useful. In general, `dump` is extremely useful for inspecting any value.

There are also characters and patterns that match only certain characters or locations. A dot `.` matches any character. The characters `^` and `$` match the beginning and end of a string (or line, in multiline mode), respectively.

Character classes are patterns of the form `[`*from*`–`*to*`]`. The negation of such a character class is `[^`*from*`–`*to*`]`. Predefined character classes include `\d` for any decimal digit, `\s` any white-space character, and `\w` for any “word” character. For example, the patterns `[0-9]` and `\d` both match any decimal digit. The negations of these classes are given by uppercase letters.

There are also named character classes such as `[:alnum:]` (letters and digits), `[:alpha:]` (letters), `[:blank:]` (spaces and tabs), `[:digit:]` (digits), `[:lower:]` (lowercase letters), `[:space:]` (white space), `[:upper:]` (uppercase letters), and `[:word:]` (“word” characters).

Named groups and character classes help parse heterogeneous data. In this example, we extract a date from the string using three named groups.

```
julia> m = match(r"(?<month>\w+)\s+(?<day>\d+),\s+(?<year>\d+)", s)
RegexMatch("May 7, 1983", month="May", day="7", year="1983")
julia> m["day"], m["month"], m["year"]
("1", "Jan", "2000")
julia> m["day"], m["month"], m["year"]
("7", "May", "1983")
```

The three parts of the first phone number can be extracted similarly. Groups can even be nested, so that we can defined a group named `number` to contain the whole match.

```
julia> m = match(r"(?<number>(?<area>\d+)_(\d+)_(\d+))", s)
RegexMatch("311_936_0001", number="311_936_0001", area="311",
           3="936", 4="0001")
julia> m["number"], m["area"], m[1], m[2], m[3], m[4]
("311_936_0001", "311", "311_936_0001", "311", "936", "0001")
```

The following example shows how named character classes are used inside brackets.

```
julia> match(r"artificial intelligence: (?<name>[[:alnum:]]+)",
             s)["name"]
"Joshua"
```

In addition to extracting matching substrings, regular expressions are also useful to replace matching parts of a string, where the substituted string may depend on the groups in the match. In the substitution string, which is usually a string literal starting with `s`, `\0` refers to the whole match, `\`*integer* refers to the group with index *integer*, and `\g<`*group*`>` refers to a named group. In this example, we rewrite a date, converting it from American to British format, first using named groups first and then using numbered groups.

```
julia> r = r"(?<month>\w+)\s+(?<day>\d+),\s+(?<year>\d+)"
r"(?<month>\w+)\s+(?<day>\d+),\s+(?<year>\d+)"
julia> replace(s, r => s"\g<day> \g<month> \g<year>")
...release date: 7 May 1983\n...
julia> replace(s, r => s"\2 \1 \3 (\0)")
...release date: 7 May 1983 (May 7, 1983)\n...
```

Another example is parsing floating-point numbers written with a decimal comma, which means that the decimal comma must be replaced by a decimal point. The following regular expression requires at least one digit to the left and to the right of the decimal comma, and the substitution string takes these digits and puts a decimal point between them. The function `parse` takes the desired type as the first argument.

```
julia> replace("2,718281828", r"(\d+),(\d+)" => s"\1.\2")
"2.718281828"
julia> parse(Float64, ans)
2.718281828
julia> typeof(ans)
Float64
```

The behavior of regular expressions can be modified by combinations of the flags `i`, `m`, `s`, and `x` after the closing double quote of its string literal. These flags affect the behavior regarding case sensitivity, the treatment of multiple lines, and white space.

Regular expressions have many more options, and the interested reader is referred to the documentation of the PCRE library for the details. In summary, regular expressions are a fast tool to deal with large bodies of text with a heterogeneous structure.

Whenever you are interested in more than the first match of a regular expression, the function `eachmatch` is useful.

4.3 Symbols

Symbols are an important data structure in LISP like languages, because they serve as variable names and because they are fundamental building blocks of expressions (see Sect. 4.4). A symbol is essentially an interned string identifier. Interning a string means that it is ensured that only one copy of each distinct string is stored, and thus interned strings can be associated with values. This implies that it is not possible for two symbols with the same name to exist simultaneously, i.e., symbols are unique.

There are a few ways to create a symbol. We can call the parser, i.e., the function `parse` in the `Meta` package, to directly create a symbol.

```
julia> Meta.parse("foo")
:foo
julia> typeof(:foo)
Symbol
```

The parser recognizes `foo` as a symbol and returns it (without evaluating it).

Another option to create a symbol is to enter a suitable expression. Entering `foo` at the REPL and hence evaluating it yields the value of the variable called `foo`, however. Therefore we have to protect the expression from evaluation. This is achieved by prepending it with a colon `:`, which adds one layer of protection against evaluation to the expression that follows it. There the colon is also called the quote character in JULIA. Thus `:foo` evaluates to the symbol `foo`.

```
julia> :foo
:foo
```

A more direct way to create a symbol is to use the function **`Symbol`**, which follows the theme in JULIA that functions that have the same name as a type create new values of this type. The function **`Symbol`** creates a new symbol by concatenating the string representations of its arguments.

```
julia> :foo == Symbol("foo") == Symbol('f', "oo")
true
```

This example also shows that symbols are indeed unique.

Sometimes it is necessary to create a new, uninterned symbol, ensuring that its name does not conflict with other symbol names (see Chap. 7). This is achieved by the function `gensym`, which returns a unique symbol whenever it is called. A prefix may be supplied to become part of the symbol name.

```
julia> gensym(), gensym()
(Symbol("##253"), Symbol("##254"))
julia> gensym("foo"), gensym("foo")
(Symbol("##foo#255"), Symbol("##foo#256"))
```

Symbols are used to access variables and evaluate to the values of the variables. Expressions can be evaluated not only in the REPL, but also by calling the function `eval`. In this example, we try to access the value of the undefined variable named `foo`, which raises an error. After defining the variable, however, its value is returned by evaluating `:foo`.

```
julia> eval(:foo)
ERROR: UndefVarError: foo not defined
...
julia> foo = 0
0
julia> eval(:foo)
0
```

We have just seen that JULIA provides access to its parser and its evaluator via `Meta.parse` and `eval`. These functions work not only with symbols, but also with expressions, the building blocks of JULIA programs.

4.4 Expressions

Reading a variable name using `Meta.parse` yields a symbol. The following example shows what happens when we parse more complicated expressions.

```
julia> Meta.parse("0 + 1")
:(0 + 1)
julia> typeof(ans)
Expr
julia> Meta.parse("foo + bar")
```

```
:(foo + bar)
julia> typeof(ans)
Expr
```

In both examples, an object of type **`Expr`**, i.e., an expression, is returned. Hence expressions are first-class objects in JULIA. They store JULIA programs directly in a suitable data structure and not just in a string.

We can evaluate expressions at the REPL just like any other data structure.

```
julia> :(0 + 1)
:(0 + 1)
julia> :(foo + bar)
:(foo + bar)
```

The expressions are returned seemingly unchanged by the REPL, because they have been quoted using the colon `:`. Behind the scenes, the situation is a bit more involved, however. The quote absorbed the evaluation by the REPL, returning the expression `0 + 1`. This expression is printed as `:(0 + 1)` (and not as `1`, which would require another evaluation) so that it remains an expression.

More information about the parts of an expression can be obtained by using the `dump` function.

```
julia> dump(:(0 + 1))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 0
    3: Int64 1
julia> dump(:(foo + 2*bar))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Symbol foo
    3: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol *
        2: Int64 2
        3: Symbol bar
```

We find that an object of type **`Expr`** has two fields, namely `head` and `args`. In both examples, the `head` is `:call`. The arguments are vectors, whose first element is a symbol that names a function. Further arguments can be constants (such as symbols) or further expressions.

In the next example, we deconstruct an expression into the parts we just observed using `dump` and then we make another expression out of these parts. As usual in Julia, the name of a type (here **`Expr`**) is also a function, and calling this function makes a new object of this type.

```
julia> expr = Meta.parse("foo + 2*bar")
:(foo + 2bar)
julia> expr.head
:call
julia> expr.args
3-element Array{Any,1}:
 :+
 :foo
 :(2bar)
julia> Expr(expr.head, expr.args...)
:(foo + 2bar)
julia> Expr(expr.head, expr.args...) == :(foo + 2*bar)
true
```

Evaluating the expression `expr` in this example raises an error, since the variables `foo` and `bar` are undefined. After defining them, however, we can evaluate the expression.

```
julia> foo = 1; bar = 2;
julia> eval(expr)
5
```

We will learn much more about expressions and their evaluation in Chap. 7. The salient point is that Julia code is represented in a canonical form as a Julia data structure, namely as objects of type **`Expr`**. One could argue that any language (that at least has a string data type) can represent programs in this language as a string and therefore using a built-in data type. This is true, of course, but of very limited use, since string data types do not provide the facilities of the **`Expr`** type, of `Meta.parse`, and of `eval`.

4.5 Collections

A collection is the general term for a data type that contain elements in ordered, unordered, indexable, or not indexable form. Various types of collections are discussed in this section. Data types that are collections are listed in Table 4.1. All built-in abstract data types are listed in Table 4.2; several of them are collections, but not all of them. It is not possible to create instances of abstract types, only of concrete subtypes of abstract types.

Table 4.1 Built-in data types that are collections.

Arrays: **`Array`**, **`AbstractArray`**, **`BitArray`**, **`DenseArray`**, **`StridedArray`**, **`SubArray`**;
Dictionaries: **`Dict`**, **`AbstractDict`**, **`WeakKeyDict`**;
Matrices: **`AbstractMatrix`**, **`StridedMatrix`**;
Pairs: **`Pair`**;
Ranges: **`AbstractRange`**, **`OrdinalRange`**, **`StepRange`**, **`UnitRange`**;
Sets: **`Set`**, **`BitSet`**;
Strings: **`String`**, **`AbstractString`**, **`SubString`**;
Tuples: **`Tuple`**, **`NTuple`**, **`NamedTuple`**, **`Vararg`**;
Vectors: **`Vector`**, **`AbstractVector`**, **`BitVector`**, **`StridedVector`**;
Vectors or matrices: **`VecOrMat`**, **`StridedVecOrMat`**.

Table 4.2 All built-in abstract data types.

`AbstractArray`, **`AbstractChannel`**, **`AbstractChar`**, **`AbstractDict`**, **`AbstractDisplay`**, **`AbstractFloat`**, **`AbstractIrrational`**, **`AbstractMatrix`**, **`AbstractRange`**, **`AbstractSet`**, **`AbstractString`**, **`AbstractUnitRange`**, **`AbstractVecOrMat`**, **`AbstractVector`**.

4.5.1 General Collections

The most general operations on collections are summarized in Table 4.3.

Table 4.3 Operations on general collections.

Function	Description
`isempty(`*collection*`)` → **`Bool`**	whether a collection is empty or not
`empty!(`*collection*`)`	destructively modify a collection to be empty
`length(`*collection*`)`	return the number of elements in a collection

Any collection can be queried whether it is empty or not. The following example is an empty array. Arrays are discussed in detail in Chap. 8; for now, it suffices to know that vectors and arrays are denoted by square brackets and that the data type of their elements may be indicated before the opening square bracket.

```
julia> [], typeof([]), isempty([])
(Any[], Vector{Any}, true)
```

Any collection can also be destructively modified to be empty. The examples here are emptying an array and a set (see also Sect. 4.5.5).

```
julia> empty!([1, 2, 3]), empty!(Set([1, 2, 3]))
(Int64[], Set{Int64}())
```

Finally, every collection can be queried how many elements it contains.

```
julia> length(empty!([1, 2, 3])), length(empty!(Set([1, 2, 3])))
(0, 0)
```

4.5.2 Iterable Collections

Iterating over all elements of an iterable collection is a common programming task. Iterable collections are collections whose elements can be iterated over using **for** loops (see Chap. 6.4) or comprehensions (see Sect. 3.5).

Iteration in JULIA is based on the generic function `iterate`. A **for** loop of the form

```
for variable in iterable
    expressions
end
```

executes the *expressions* for all elements of the iterable collection *iterable* bound to *variable* in the order in which they are returned by the `iterate` method. For built-in iterable collections, `iterate` methods have of course already been defined. Furthermore, after defining `iterate` methods for your own data structures, you can iterate over these data structures using **for** loops as well.

There are many useful functions that can be applied to iterable collections; these include functions to extract certain elements, to reduce the collection by applying a function repeatedly, and to map a function over all elements. Hence the functions discussed in this section are important building blocks for functional programming, which often proceeds by combining functions to extract the desired information after starting from a collection.

Table 4.4 gives an overview of basic functions that are defined for iterable collections.

Several functions exist to find extrema in an iterable collection. They are listed in Table 4.5. As usual, destructive versions end in an exclamation mark `!`.

An important set of functions is given in Table 4.6. The most general ones in this table are the ones for reducing and folding an iterable collection. Reducing a collection containing elements $\{a_i\}_{i=1}^n$ using a function or operation $\oplus$ means calculating the expression $a_1 \oplus a_2 \oplus \cdots \oplus a_n$. The operation $\oplus$ must take two arguments and it must be associative. An initial element `init` may also be specified as a keyword argument. The operation is applied repeatedly until the whole expression has been evaluated. If the collection is empty, the initial element must be specified except in special cases where JULIA knows the neutral element of the operation. If the collection is non-empty, it is unspecified whether the initial element is used. If the collection is an ordered one, the elements are not reordered; otherwise the evaluation order is unspecified.

Table 4.4 Basic operations on iterable collections.

Function	Description
`eltype(`*iterable*`)`	return the type of the elements of *iterable*
`in``(`*item*`,` *collection*`)` → **`Bool`**	determine whether *item* is in *collection*
item **`in`** *collection* → **`Bool`**	infix syntax for **`in`**
`issubset(`*a*`,` *b*`)` → **`Bool`**	test whether every element of *a* is also present in *b*
`indexin(`*a*`,` *b*`)`	return an array containing the first index in *b* for each value in *a* which is a member of *b* or **`nothing`** otherwise
`unique([`*f*`,]` *iterable*`)`	returns an array with the unique elements of *iterable* after applying the function *f*
`unique(`*iterable*`;` *dims*`)`	return an array with the unique elements of *iterable* along dimension *dim*
`allunique(`*iterable*`)` → **`Bool`**	determine whether all elements are distinct
`first(`*iterable*`)`	return the first element
`collect(`*collection*`)`	return an **`Array`** containing all elements in *collection*
`collect(`*type*`,` *collection*`)`	return an **`Array`** with element type *type* containing all elements in *collection*
`count([`*f*`,]` *iterable*`)` → **`Integer`**	returns the number of **`true`** elements after applying *f*

Table 4.5 Operations on iterable collections for finding extrema.

Function	Description
`maximum([`*f*`,]` *iterable*`)`	return the maximum element in *iterable* after applying the function *f*
`maximum(`*A*`;` *dims*`)`	return the maximum element of the array *A* over *dims*
`maximum!(`*r*`,` *A*`)`	write the maximum element of the array *A* over the singleton dimensions of the array *r* into *r*
`minimum([`*f*`,]` *iterable*`)`	return the minimum element in *iterable* after applying the function *f*
`minimum(`*A*`;` *dims*`)`	return the minimum element of the array *A* over *dims*
`minimum!(`*r*`,` *A*`)`	write the minimum element of the array *A* over the singleton dimensions of the array *r* into *r*
`extrema([`*f*`,]` *iterable*`)` → **`Tuple`**	return the minimum and maximum elements computed in a single pass
`extrema(`*A*`;` *dims*`)` → **`Array{Tuple}`**	return the min. and max. elements of the array *A* over the dimensions *dims*
`findmax(`*iterable*`)` → **`Tuple`**	return the maximum element of *iterable* and its index
`findmax(`*A*`;` *dims*`)` → **`Tuple`**	return the maximum element of the array *A* and its index over dimensions *dims*
`findmax!(`*vals*`,` *inds*`,` *A*`)` → **`Tuple`**	return the maximum element of the array *A* along the dimensions of *vals* and *inds* and store them there
`findmin(`*iterable*`)` → **`Tuple`**	return the minimum element of *iterable* and its index
`findmin(`*A*`;` *dims*`)` → **`Tuple`**	return the minimum element of the array *A* and its index over dimensions *dims*
`findmin!(`*vals*`,` *inds*`,` *A*`)` → **`Tuple`**	return the minimum element of the array *A* along the dimensions of *vals* and *inds* and store them there

The function `reduce` provides the general form of reduction. Special, often used cases come with the special implementations `maximum`, `minimum`, `sum`, `prod`, `any`, and `all` (and their variants) and should be used instead.

The folding functions `foldl` and `foldr` come with more guarantees. They guarantee left respectively right associativity and use the given initial or neutral element exactly once.

Table 4.6 Reduction and folding operations on iterable collections.

Function	Description
`reduce(`*f*`,` *iterable*`;` *init*`)`	reduce *iterable* using *f* and the initial element *init*
`foldl(`*f*`,` *iterable*`;` *init*`)`	like `reduce`, but guarantee left associativity and use the neutral element *init* exactly once
`foldr(`*f*`,` *iterable*`;` *init*`)`	like `reduce`, but guarantee right associativity and use the neutral element *init* exactly once
`sum(`*iterable*`)`	return the sum of all elements
`sum(`*f*`,` *iterable*`)`	sum the results of calling *f* on each element
`sum(`*A*`;` *dims*`)`	return the sum of all elements over the dimensions *dims*
`sum!(`*r*`,` *A*`)`	store the sum over the singleton dimensions of *r* in *r*
`prod(`*iterable*`)`	return the product of all elements
`prod(`*f*`,` *iterable*`)`	multiply the results of calling *f* on each element
`prod(`*A*`;` *dims*`)`	return the product of all elements over the dimensions *dims*
`prod!(`*r*`,` *A::array*`)`	store the product over the singleton dimensions of *r* in *r*
`any(`*iterable*`)` → **`Bool`**	test whether any element of a collection of **`Bool`**s is **`true`**
`any(`*f*`,` *iterable*`)`	test whether *f* returns **`true`** for any element of *iterable*
`any(`*A*`;` *dims*`)`	test whether any element of an array of **`Bool`**s along the dimensions *dims* is **`true`**
`any!(`*r*`,` *A*`)`	store whether any element of the array *A* along the singleton dimensions of *r* is **`true`** in *r*
`all(`*iterable*`)` → **`Bool`**	test whether all elements of a collection of **`Bool`**s are **`true`**
`all(`*f*`,` *iterable*`)`	test whether *f* returns **`true`** for all elements of *iterable*
`all(`*A*`;` *dims*`)`	test whether all elements of an array of **`Bool`**s along the dimensions *dims* are **`true`**
`all!(`*r*`,` *A*`)`	store whether all elements of the array *A* along the singleton dimensions of *r* are **`true`** in *r*

A simple example of reduction is the summation of the elements of a collection. (Note that in practice the preferred way to sum the elements of a vector is to use the function `sum`.)

```
julia> reduce(+, Int[])
0
julia> reduce(+, [], init = 0)
0
julia> reduce(+, [1])
1
julia> reduce(+, [1, 2])
3
```

```
julia> reduce(+, [1, 2, 3])
6
```

In the first example, we have specified the type (**Int**) of the elements of the vector by using **Int**[]. Since [] may contain elements of any type, which you can check by evaluating eltype([]), and therefore reduce(+, []) raises an error because JULIA cannot determine the neutral element, we have specified the initial element as zero in the second example.

If the collection contains only one element, it is returned. If there are two or more elements in the collection, the operation is applied at least once. In both cases, it is unspecified whether the initial element is used.

A mathematical example that can be implemented using reduce is given by Taylor series.

```
function my_exp(x, n::Integer)
    reduce(+, [x^i/factorial(i) for i in 0:n])
end
```

Here we have used array comprehensions (see Sect. 3.5) to conveniently construct a collection with the appropriate elements.

Analogously, we can multiply all elements in a collection using reduce with the operation $*$ and the initial or neutral element 1. The factorial function can be defined in one line in this manner.

```
my_factorial(n::Integer)::BigInt = reduce(*, BigInt(1):BigInt(n))
```

Here we have used **BigInt**s in order to be able to compute large values. The syntax

start:*step*:*end*

makes a range or more precisely a **UnitRange** of values starting at *start* and ending at *end* with step size *step*. Ranges may be empty.

```
julia> typeof(BigInt(1):BigInt(2))
UnitRange{BigInt}
```

This confirms that the collection which is reduced is indeed a range of **BigInt**s.

Another example is the implementation of the maximum function by reduction. While maximum acts on iterable collections, the max function takes one or more arguments. We can implement maximum using max (called with two arguments) as follows.

```
julia> reduce(max, [], init = -Inf), reduce(max, [1])
(-Inf, 1)
```

Analogously, the effect of minimum can be achieved by reducing min using a suitable initial element.

```
julia> reduce(min, [], init = Inf), reduce(min, [1])
(Inf, 1)
```

Continuing the example of the Taylor series above, the implementation can be made more practical by two improvements. The first improvement is, as indicated above already, to define it even more succinctly using `sum`.

```
my_exp(x, n::Integer) = sum([x^i/factorial(i) for i in 0:n])
```

The second improvement is to replace the array comprehension by a generator. An array comprehension allocates an array (a costly operation if the array is large), but here the array is only used to be reduced or summed. Generators, on the other hand, generate values on demand obviating the need for any allocation. Generators can be used wherever functions take them as arguments. Their syntax is the same as the syntax of array comprehensions (see Sect. 3.5), but without the brackets. Hence the final definition of `my_exp` is the following.

```
my_exp(x, n::Integer) = sum(x^i/factorial(i) for i in 0:n)
```

Similarly, the factorial function can be defined using `prod`.

```
my_factorial(n::Integer) = prod(BigInt(1):BigInt(n))
```

The functions `any` and `all` can also be viewed as reductions of collections. To see this, we define the two helper functions `and` and `or`.

```
julia> and(a, b) = a && b
and (generic function with 1 method)
julia> or(a, b) = a || b
or (generic function with 1 method)
```

Reducing a collection using `or` has the same effect as applying `any` to the collection; the initial or neutral element of the operation `or` is **`false`**. Analogously, reducing a collection using `and` has the same effect as applying `all` to the collection; the initial or neutral element of the operation `and` is **`true`**.

```
my_any(collection) = reduce(or,  collection, init = false)
my_all(collection) = reduce(and, collection, init = true)
```

The following examples show that our definitions work as expected.

```
julia> my_any([]),            any([])
(false, false)
julia> my_any([false]),       any([false])
(false, false)
julia> my_any([true]),        any([true])
(true, true)
julia> my_any([false, true]), any([false, true])
(true, true)
julia> my_all([]),            all([])
(true, true)
julia> my_all([false]),       all([false])
(false, false)
julia> my_all([true]),        all([true])
```

```
(true, true)
julia> my_all([false, true]), all([false, true])
(false, false)
```

A common programming task is to apply a function to each element of a collection or of collections and to collect the results. This operation is called mapping a function over a collection or collections, and variants of mapping are summarized in Table 4.7. The type of the resulting collection is the same as the type of the given collection or collections.

Table 4.7 Mapping operations on iterable collections.

Function	Description
`map(`*f*`, `*iterable*`...)` → *iterable*	return the result of applying *f* elementwise
`map!(`*f*`, `*destination*`, `*iterable*`...)`	like `map`, but store the result in *destination*
`foreach(`*f*`, `*iterable*`...)` → **`Nothing`**	like *map*, but discard the results
`filter(`*f*`, `*iterable*`)`	return a copy of *iterable* without the elements for which *f* returns **`false`**
`filter!(`*f*`, `*iterable*`)`	destructive version of `filter`
`mapreduce(`*f*`, `*op*`, `*iterable*`... ; `*init*`)`	equivalent to `reduce(`*op*`, map(`*f*`, `*iterable*`...); init=`*init*`)`
`mapfoldl`	like `mapreduce`, but use `foldl` instead of `reduce`
`mapfoldr`	like `mapreduce`, but use `foldr` instead of `reduce`

It is often convenient to map anonymous functions (see Sect. 2.5) like in these examples.

```
julia> map(x -> log(10, x), [1, 10, 100])
3-element Vector{Float64}:
 0.0
 1.0
 2.0
```

The same effect can be achieved using an array comprehension (see Sect. 3.5). It is often a matter of style if `map` or a comprehension is used.

```
julia> [log(10, x) for x in [1, 10, 100]]
3-element Vector{Float64}:
 0.0
 1.0
 2.0
```

The function to be mapped may take more than one argument, and then a corresponding number of collections must be supplied to `map` or its cousins, one collection for each function argument.

```
julia> map((x, y) -> x^2 + y^2, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
 101
 404
 909
```

The function `foreach` is the same as `map`, except that it discards the results of applying the function and always returns **`nothing`**, the only value of type **`Nothing`**. It should be used when the function calls are performed to produce side effects only, e.g., to print values.

The functions `filter` and `filter!` are also similar to `map`, but they are used to return only a subset of the collection. Again, a function is applied to each element of a collection. If it returns **`true`**, the element is kept in a copy of the collection, otherwise it is ignored.

The function `mapreduce` and its variants combine `map` and `reduce` as the name indicates. The function call

`mapreduce(`*f*`, `*op*`, `*iterable*`... ; `*init*`)`

is equivalent to evaluating

`reduce(`*op*`, map(`*f*`, `*iterable*`...); init=`*init*`)`

except that the need to allocate any intermediate results is obviated; hence the `mapreduce` version generally runs faster and generates less garbage.

Using `mapreduce`, Taylor series can be implemented perfectly in the spirit of functional programming.

```
function my_exp(x, n::Integer)
    mapreduce(i -> x^i/factorial(BigInt(i)), +, 0:n)
end
```

In order to check the convergence speed for different arguments `x`, we use `map` and an anonymous function to produce tuples that contain the values of `x` and the corresponding residua. Then we filter the tuples to identify the tuples where the residua are above a certain threshold.

```
julia> filter(x_res -> abs(x_res[2]) > 1e-6,
              map(x -> (x, my_exp(x, 20) - exp(x)), 1:5))
1-element Vector{Tuple{Int64,BigFloat}}:
 (5, -1.20351...e-05)
```

Another example of using `mapreduce`, namely the definition of norms, has already been discussed in Sect. 2.5 and Sect. 2.6.

4.5.3 Indexable Collections

Indexable collections are collections whose elements are associated with an index or key. A set (see Sect. 4.5.5) is an example of a collection that is iterable, but not indexable. In JULIA, the syntax `a[i...]`

is just an abbreviation for the function call

`getindex(a, i...)`.

Similarly, assignments of the form

`a[i...]=x`

are equivalent to the expression

`begin` `setindex!(a, x, i...); x` **`end`**.

In the case of **`Dict`**s, the index is called a key (see Sect. 4.5.4).

This example shows that multi-dimensional arrays require a corresponding number of indices.

```
julia> foo = [1 2; 3 4]; setindex!(foo, 5, 1, 1); foo
2\times 2 Matrix{Int64}:
 5  2
 3  4
```

The operations special to indexable collections are summarized in Table 4.8.

Table 4.8 Operations on indexable collections.

Function	Description
`getindex(`*collection*`, `*i*`...)`	return the element stored at the given index or key *i*
`setindex!(`*collection*`, `*value*`, `*i*`...)`	store the *value* at the given index or key *i*

4.5.4 Associative Collections

An associative collection associates keys with values. The standard associative collection is the **`Dict`**, short for dictionary. The type of the keys in a **`Dict`** may be any type for which the hash function `hash` and the comparison function `isequal` are defined. The type of the values may be arbitrary.

A **`Dict`** contains **`Pair`**s of keys and values, and a **`Pair`** is made by the => function, where *key*=>*value* is a syntactic abbreviation for the function call `=>(`*key*`,` *value*).

```
julia> typeof(=>(1, 2)), typeof(1 => 2)
(Pair{Int64, Int64}, Pair{Int64, Int64})
```

There are various ways to create a **`Dict`**. It can be created by passing **`Pair`** objects to the **`Dict`** constructor.

```
julia> Dict("a" => 1, "b" => 2, "c" => 3)
Dict{String, Int64} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1
```

We see that the types of the keys and values (i.e., **`String`** and **`Int64`**) are inferred from the **`Pair`**s, but they can also be specified as parameters to the **`Dict`** function in curly brackets (see Sect. 5.7) by writing **`Dict`**`{`*key-type*`,` *value-type*`}(`*pairs*`...)` as in the following example.

```
julia> Dict{String, Int16}()
Dict{String, Int16}()
julia> Dict{String, Int16}("a" => 1, "b" => 2, "c" => 3)
Dict{String, Int16} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1
```

A **`Dict`** can also be created by a generator as shown here.

```
julia> Dict(i => Char(i+64) for i in 1:3)
Dict{Int64, Char} with 3 entries:
  2 => 'B'
  3 => 'C'
  1 => 'A'
```

Of course, the value associated with a key in a dictionary can be modified. If `d` is a **`Dict`**, then the expression

`d[`*key*`]=`*value*

stores a key-value pair in the dictionary, possibly replacing any existing value for the key. Furthermore, the expression `d[`*key*`]` returns the value of the given key if it exists or throws an error if it does not. The function `haskey` test whether a collection contains a value associated with a given key and returns a **`Bool`** value.

Table 4.9 provides an overview of the operations available on associative collections.

4.5.5 Sets

Sets are among the most fundamental data structures in mathematics. As usual, a set can be constructed by using the name of the data structure, i.e., **`Set`**, as a function or constructor, where the type of the elements can be specified as a type parameter in curly brackets. The initial elements of a set may be passed as an argument that is an iterable object, e.g., a vector.

Table 4.9 Operations on associative collections.

Function	Description
`Dict``(`*pairs*`...)`	create a dictionary with automatic types
`Dict``{`*t1*`,`*t2*`}(`*pairs*`...)`	create a dictionary for the key and value types
`keytype(`*collection*`)`	return the type of the keys
`valtype(`*collection*`)`	return the type of the values
`keys(`*collection*`)`	return an iterator over all keys in a collection, to be used in a **`for`** loop
`values(`*collection*`)`	return an iterator over all values in a collection to be used in a **`for`** loop
`d[`*key*`]`	return the value associated with *key* in `d`
`d[`*key*`]=`*value*	associate *value* with *key* in `d`
`haskey(`*collection*`, `*key*`)`	determine whether a collection contains *key*
`get(`*collection*`, `*key*`, `*default*`)`	return the value for *key* if it exists in *collection* or the *default* value otherwise
`get(`*f*`, `*key*`, `*default*`)`	return the value for *key* if it exists in *collection* or *f*`()` otherwise; to be used with a **`do`** block
`get!(`*collection*`, `*key*`, `*default*`)`	return the value for *key* if it exists in *collection* or otherwise store *key*=>*default* and return *default*
`get!(`*default*`, `*key*`, `*default*`)`	destructive version of `get(`*f*`, `*key*`, `*default*`)`
`getkey(`*collection*`, `*key*`, `*default*`)`	return *key* if it exists in *collection* or *default* otherwise
`delete!(`*collection*`, `*key*`)`	delete the *key* and its value from *collection* and return *collection*
`pop!(`*collection*`, `*key*`[, `*default*`])`	delete *key* and return its value if it exists in *collection*, otherwise return *default* or throw an error if unspecified
`merge(`*collection*`, `*others*`...)`	return a collection merged from all given collections, the value for each key is taken from the last collection that contains it
`merge!(`*collection*`, `*others*`...)`	destructive version of `merge`, modify *collection*
`sizehint!(`*collection*`, `*n*`)`	suggest that *collection* will contain at least *n* elements

```
julia> Set()
Set{Any}()
julia> Set{Int64}()
Set{Int64}()
julia> Set([1, 2, 3]), typeof(Set([1, 2, 3]))
(Set([2, 3, 1]), Set{Int64})
julia> Set{Float16}([1, 2, 3]), typeof(Set{Float16}([1, 2, 3]))
(Set(Float16[2.0, 3.0, 1.0]), Set{Float16})
```

A **`BitSet`** is a sorted set of **`Int`**s implemented as a bit string. While **`Set`**s are suitable for sparse integer sets and generally for arbitrary objects, **`BitSet`**s are especially suited for dense integer sets.

The usual set operations are available in both non-destructive and destructive versions, the latter ending with an exclamation mark `!`. An overview is given in Table 4.10.

Table 4.10 Operations on set-like collections.

Function	Description
`Set``(`[*iterable*]`)`	construct a set with the elements *iterable*
`Set``{`*type*`}(`[*iterable*]`)`	construct a set for elements of *type* with the elements *iterable*
`BitSet``(`[*iterable*]`)`	construct a (dense) set **`Int`**s
`issubset(`*s1*`,` *s2*`)`	determine whether *s1* is a subset of *s2* using **`in`**
`union(...)`	return the union of the given sets
`union!(`*set*`, ...)`	destructively modify *set* to contain the union of all given sets
`intersect(...)`	return the intersection of the given sets
`intersect!(`*set*`, ...)`	destructively modify *set* to contain the intersection of all given sets
`setdiff(`*s1*`,` *sets*`...)`	return the set of elements which are in *s1*, but not in any of the *sets*
`setdiff!(`*s1*`,` *sets*`...)`	destructively delete the elements in *sets* from *s1*
`symdiff(...)`	return the symmetric difference of all given sets
`symdiff!(`*s1*`,` *sets*`...)`	destructive version of `symdiff`, store result in *s1*

Several of the functions in the table can also be applied to arrays with the expected results. If the arguments are arrays, the order of the elements is maintained and an array is returned. These methods of the generic functions are easier to use and run faster when arrays are to be interpreted as sets.

```
julia> a1 = [1, 2]; a2 = [3];
julia> union(Set(a1), Set(a2)) == Set(union(a1, a2))
true
```

4.5.6 Deques (Double-Ended Queues)

Vectors in the context of linear algebra are discussed in detail in Chap. 8. Here we view vectors as deques (double-ended queues) and discuss the operations that implement deques on top of vectors as the underlying data structure. This notion of viewing vectors as collections of items (and not as elements of $\mathbb{R}^d$) and performing operations on them fit well into the theme of the present section.

The operations on deques are summarized in Table 4.11. All of the functions are destructive. The functions in Table 4.11 differ in their return values. Some return the item or items in question, while others return the modified collection.

The two most iconic operations on deques are `push!` and `pop!` for inserting an item or items at the end of a collection and for removing the last item, respectively. These two functions operate on the end of the collection, usually a vector, because the end is where a vector can be modified most easily. If the functions were to operate on the beginning of the vector, then the vector would have to be copied every time.

Table 4.11 Operations on deques (double-ended queues).

Function	Description
`push!(`*collection*`,` *items*`...)`	insert the *items* at the end of *collection*
`pop!(`*collection*`)`	remove the last item of *collection* and return it
`insert!(`*vector*`,` *index*`,` *item*`)`	insert *item* into *vector* at *index*
`deleteat!(`*vector*`,` *index*`)`	remove the item at *index* and return the modified *vector*
`deleteat!(`*vector*`,` *indices*`)`	remove the items at the *indices* and return the modified *vector*
`splice!(`*vector*`,` *index*[`,` *new*]`)`	remove the item at *index* and return it; if specified, the *new* value is inserted instead
`splice!(`*vector*`,` *range*[`,` *new*]`)`	remove the items at the index *range* and return them; if specified, the *new* value is inserted instead
`append!(`*coll1*`,` *coll2*`)`	append the elements of *coll2* at the end of *coll1*
`prepend!(`*vector*`,` *items*`)`	insert the *items* at the beginning of *vector*
`resize!(`*vector*`,` *n*`)`	resize *vector* to contain *n* elements

```
julia> v= []; push!(v, 1, 2, 3)
3-element Vector{Any}:
 1
 2
 3
julia> pop!(v)
3
julia> v
2-element Vector{Any}:
 1
 2
```

The two functions `push!` and `pop!` are inverses of one another.

```
julia> push!(v, pop!(v))
2-element Vector{Any}:
 1
 2
julia> pop!(push!(v, 3))
3
julia> v
2-element Vector{Any}:
 1
 2
```

While both `push!` and `append!` add elements to the end of a collection, `push!` takes a variable number of arguments, while the second argument to `append!` is already a collection. These two function calls have the same effect.

```
julia> push!([], 1, 2, 3) == append!([], [1, 2, 3])
true
```

The `splice!` method acts on a range of indices and can be used to insert new elements without removing any elements by specifying an empty range.

```
julia> v = [1, 2, 3]; splice!(v, 2:1, [20, 30])
Int64[]
julia> v
5-element Vector{Int64}:
  1
 20
 30
  2
  3
```

The range `2:1` is indeed empty.

```
julia> length(2:1)
0
```

Problems

4.1 Write your own REPL.

4.2 Write a function that calculates the natural logarithm using its Taylor series. Compare the speed and memory allocation of five versions using `reduce`, `sum`, `mapreduce`, array comprehensions, and generators.

4.3 Why does `my_factorial(-1)` return `1`?

4.4 Compare the speed and memory allocation of three versions of the `my_factorial` function: (a) using a range, (b) using an array comprehension, and (c) using a generator.

4.5 Write iterative versions of `my_any` and `my_all`. Compare their speed and memory allocation with the versions based on `reduce` and with the built-in functions `any` and `all`.

Chapter 5
User Defined Data Structures and the Type System

It is better to have 100 functions operate on one data structure
than to have 10 functions operate on 10 data structures.

—Alan Jay Perlis

Abstract Data types defined by the programmer are indistinguishable from built-in types in JULIA. In this chapter, it is first explained how variables and return values can be annotated with types and how JULIA's introspective features can be used to inspect the type system. Types that can be defined by the programmer include abstract types, concrete types, composite types, type unions, and tuples. Custom constructors and pretty printers can be defined as well. Furthermore, abstract and composite types can be parameterized. Finally, the operations on types are summarized.

5.1 Introduction

All values in digital, binary computers are stored as vectors of two values, zeros and ones. In order to make memory more useful and accessible to programmers, these vectors or strings of zeros and ones are interpreted as more meaningful objects such as signed and unsigned integers, rational numbers, floating-point numbers as approximations of real numbers, characters, dictionaries, user defined data structures, and many others. The information that enables the interpretation of a vector of zeros and ones as a more meaningful object is its type. In other words, both a vector of zeros and ones and the type information are necessary to interpret a part of memory usefully and correctly.

In computer science, there are traditionally two approaches to implement type systems, namely static type systems and dynamic type systems. In static type systems, each variable and expression in the program has a type that is known or computable before the execution of the program. In dynamic type systems, the

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_5

types of variables and expressions are unknown until run time, when the values stored in variables are available. This means that in a static type system, types are associated with *variables;* variables always contain values of the same type, and the type is known before the execution of the program. In a dynamic type system, types are associated with *values;* variables may contain values of different types, and the type is only known during the execution of the program and may change.

The ability of programs or expressions to operate on different types is called polymorphism. By definition, all programs written in a dynamically typed language are polymorphic; restrictions to the types of values only occur when a type is checked during run time or when an operation is not available for a certain value at run time.

Dynamic and static typing both have their advantages and disadvantages. While JULIA's type system is dynamic, it is also possible to specify the types of certain variables like in a static type system. This helps generate efficient code and allows method dispatch on the types of function arguments. In this manner, JULIA gains some properties and advantages of static type systems.

Since JULIA's type system is dynamic, variables in JULIA may contain values of any type by default. When desired, it is possible to add type annotations to variables and expressions. These type annotations serve a few purposes. They enable multiple dispatch on function arguments, they serve as documentation and can hence make a program more readable and clarify its purpose, and they can serve as safety measures and catch programmer errors.

The type system is an important part of any programming language. Some important properties of JULIA's powerful and expressive type system are the following.

- As usual in a dynamic type system, types are associated with values, and never with variables.
- Some programming languages, especially object oriented ones, discern between object (composite) and non-object (numbers etc.) values. This is not the case in JULIA, where all values are objects and each type is a first-class type.
- It is possible to parameterize both abstract and concrete types, and type parameters are optional when they are not required or not restricted.

In this chapter, the basics of JULIA's type system needed to define own type hierarchies are summarized, and some finer points are discussed as well.

5.2 Type Annotations

The double-colon operator `::` underlies all type annotations. Type annotations can be attached to any variable and expression. As mentioned above, a type annotation conveys type information to the compiler, usually to help it generate faster code, or to programmers, usually to help understand the program and document it. Type annotations also make method dispatch possible. Type annotations may appear in various syntactic locations with slightly different meanings, which are discussed in detail in the following.

5.2.1 Annotations of Expressions

If a type annotates an expression in the form *expr*`::`*type*, the meaning of the `::` operator is that it asserts that the value of the expression must be of the indicated type. If the assertion is true, the value is returned, otherwise an exception is thrown.

```
julia> true::Int
ERROR: TypeError: in typeassert, expected Int64, got a value of
                  type Bool
julia> true::Bool
true
```

5.2.2 Declarations of Variables and Return Values

If the `::` operator appears in a **`local`** variable declaration in the form *variable*`::`*type*, it declares the variable to always have the indicated type. This is the behavior of a statically typed language. Recall from Chap. 3 that such local variable definitions do not require the keyword **`local`** to be written out. Any value assigned to such an annotated variable is converted to the indicated type using the function `convert`, which raises an error if the conversion is not possible.

```
function foo()
    local x::Float64 = 0
    x
end

julia> typeof(foo())
Float64
```

Similarly, it is possible to declare the type of the return value of a function (see Chap. 2). The annotation of a return value (returned by **`return`** or being the last value in the function body) is treated just as the annotation of a local variable as discussed above, and hence the assignment is performed using `convert`. As an example, the following function will always raise an error at run time.

```
function foo()::Int
    "This conversion raises an error."
end
```

5.3 Abstract Types, Concrete Types, and the Type Hierarchy

Abstract types are defined as types that cannot be instantiated, while concrete types can be. In the type graph, the children or subtypes of abstract types are abstract or concrete types, while concrete types cannot have children or subtypes in the type graph. Abstract types are defined via **`abstract type`** *Name* **`end`** or **`abstract type`** *Name* `<:` *Supertype* **`end`**. The first syntax is equivalent to **`abstract type`** *Name* `<:` **`Any end`**, where the type **`Any`** is at the top of the type graph or hierarchy. The names of types are usually capitalized, and the names of abstract types usually start with `Abstract`.

While the type **`Any`** is at the top of the type hierarchy or graph, the **`Union`**`{}` type is at the bottom. While all objects are instances of **`Any`** and all types are subtypes of **`Any`**, no object is an instance of **`Union`**`{}` and all types are supertypes of **`Union`**`{}`.

The whole type graph or hierarchy can be probed easily using the `<:` operator and the functions `subtypes`, `supertype`, and `supertypes`. The expression *type1* `<:` *type2* returns true if *type1* is below *type2* in the type hierarchy; it does not have to be a child.

```
julia> Int8 <: Any
true
```

The function `subtypes` returns a vector with all types that are directly below the given type in the hierarchy, i.e., with all children of the given type.

```
julia> subtypes(Integer)
3-element Vector{Any}:
 Bool
 Signed
 Unsigned
```

The function `supertype` returns the parent of the given type, and the function `supertypes` returns a tuple with all types above the given type in the hierarchy, starting with the parent and always ending with **`Any`**.

```
julia> supertype(Int8)
Signed
julia> supertypes(Int8)
(Int8, Signed, Integer, Real, Number, Any)
```

In the next example, we locate real and complex numbers in the type hierarchy.

```
julia> supertypes(Real)
(Real, Number, Any)
julia> supertypes(typeof(1 + 2im))
(Complex{Int64}, Number, Any)
```

Irrational numbers are also part of the JULIA's numerical tower, as we can see here in the example of Euler's number.

```
julia> Base.MathConstants.e
e = 2.7182818284590...
julia> typeof(Base.MathConstants.e)
Irrational{:e}
julia> supertypes(typeof(Base.MathConstants.e))
(Irrational{:e}, AbstractIrrational, Real, Number, Any)
```

As we have seen in these examples, the built-in functions above suffice to obtain the full type graph consisting of both built-in and user defined types (see Problems Problem 5.1 and Problem 5.2).

5.4 Composite Types

Composite types are the most common types defined by users. In other languages, composite types are also called structs, records, or objects. They consist of named fields of arbitrary types, and therefore usually serve to collect quite distinct objects or values into ensembles, in contrast to vectors, which are usually used to store values of the same type.

A composite type is defined by the keyword **`struct`** followed by the field names, which may be annotated by their types using the usual `::` syntax. If there is no annotation, it defaults to the **`Any`** type. Again, the names of composite types are capitalized in JULIA by convention.

In the first example, there is no type annotation so that both fields can contain values of **`Any`** type. (The types in the examples are numbered, because redefining types is not allowed in JULIA. Numbering the types saves us from restarting JULIA to evaluate the examples.)

```
struct Foo1
    a
    b
end
```

These two field specifications are equivalent to `a::Any` and `b::Any`. In order to create an instance of the newly defined composite type `Foo1`, its name is used as a function; this constructor function was defined by the expression above. By default, a composite object is printed as its name followed by the values of its fields in parentheses.

```
julia> Foo1(1, 2)
Foo1(1, 2)
julia> typeof(ans)
Foo1
julia> methods(Foo1)
# 1 method for type constructor:
[1] Foo1(a, b) in Main at REPL[1]:2
```

The `methods` call reveals that behind the scenes our type definition defined a generic function of the same name. Its method takes the correct number of field values as input.

The value of a field is accessed by a dot `.` followed the field name.

```
julia> foo1 = Foo1(1, 2)
Foo1(1, 2)
julia> foo1.a
1
julia> foo1.b
2
julia> Foo1(1, 2).a
1
```

The fields of a composite object can also be accessed using the function `getfield`, and all field names of a composite type can be obtained by the function `fieldnames` as symbols. This provides a way to iterate over all fields of a composite type, as shown in the next example.

```
julia> fieldnames(Foo1)
(:a, :b)
julia> for name in fieldnames(Foo1)
           @show name, getfield(foo1, name)
       end
(name, getfield(foo1, name)) = (:a, 1)
(name, getfield(foo1, name)) = (:b, 2)
```

Composite types (i.e., **`struct`**s) are defined to be immutable by default, i.e., their field values cannot be changed after they have been constructed. Advantages of immutable types include better efficiency and easier reasoning about the code. However, when required, for example when copying of objects is to be avoided, composite types can also be defined to be mutable by just writing **`mutable`** before **`struct`**.

```
mutable struct Foo2
    a::Int
    b::Float64
end
```

Then field values can be assigned using the equal sign = and the accessor on the left-hand side.

```
julia> foo2 = Foo2(1, 2)
Foo2(1, 2.0)
julia> foo2.a = 3
3
julia> foo2.a
3
```

In our examples so far, the initial field values have matched the field types specified during **struct** definition. What happens if they do not match, though? JULIA tries to `convert` the given values to the types specified in the **struct** definition whenever possible; if it is not possible, an error is raised. An example of such a conversion can be seen above: the **Int** value `2` was convert to a **Float64** value according to the field specification `b::Float64`. The error raised when the conversion is not possible is shown in the following example.

```
julia> Foo2(1, "2")
ERROR: MethodError: Cannot `convert` an object of type String to an
                    object of type Float64
```

The same error is raised when trying to assign a value that cannot be converted to the type indicated in the **struct** definition.

```
julia> Foo2(1, 2).b = "2"
ERROR: MethodError: Cannot `convert` an object of type String to an
                    object of type Float64
```

When an instance of a composite type is constructed using the standard constructor, the field values must usually be supplied. However, as an exception it is also possible to construct instances with uninitialized fields as elements of vectors. We consider two leading examples. In the first one, one of the fields has type **Any**.

```
struct Foo3
    a
    b::Int
end
```

In this case, constructing a **Vector** consisting of elements of type `Foo3` yields a vector consisting of undefined elements.

```
julia> Vector{Foo3}(undef, 3)
3-element Vector{Foo3}:
 #undef
 #undef
 #undef
```

Trying to access any of these undefined elements yields an `UndefRefError` error, saying that an undefined reference was accessed.

The situation is different with numerical fields. We first define a new composite type.

```
struct Foo4
    a::Int8
    b::Int64
    c::Float64
    d::ComplexF64
end
```

In this case, the vector with undefined elements consists of elements whose fields contain random numbers. The numbers are random in the sense that they contain whichever bits were present at their memory location when they were allocated.

```
julia> Vector{Foo4}(undef, 3)
3-element Vector{Foo4}:
 Foo4(1, 2, 1.5e-323, 8.0e-323 + 2.0e-323im)
 Foo4(5, 6, 3.5e-323, 4.4e-323 + 5.0e-323im)
 Foo4(12, 13, 7.0e-323, 7.4e-323 + 5.4e-323im)
```

Constructing a vector of undefined elements in this manner is useful when a large vector is required; of course, care must be taken to never use the values of the uninitialized fields.

5.5 Constructors

Constructors are, in general, functions that create new objects. We have already seen in the previous section that defining a new composite type automatically defines a standard constructor for this type. The standard constructor is a method for the generic function with the same name as the type, taking the initial values for the fields as arguments. Sometimes, however, it is desirable to define custom constructors, for example, to create complex objects in a consistent state, to enforce invariants, or to construct self-referential objects.

There are two types of constructors: inner and outer ones. Outer constructors are just additional methods to the generic function of the same name as the composite type. They usually provide convenience such as constructing objects with default values.

Here we consider the example of (real-valued) intervals. (Again, the types have numbers in their names since JULIA forbids redefining types and we want to avoid restarting JULIA for each example.)

```
struct Interval1
    left::Float64
    right::Float64
end
```

The following method for the generic function `Interval1` is an outer constructor. Its only purpose is to define a default interval.

```
function Interval1()
    Interval1(0, 1)
end
```

Outer constructors bear their name because they are defined outside the scope of the **`struct`** definition. Inner constructors are defined inside the **`struct`** definition and have an additional capability, namely that they can call a function called `new` that creates objects of the composite type being defined. After a composite type has been defined, it is not possible to add any inner constructors. Also, if an inner constructor is defined, no default constructor is defined.

In the following example, the only constructor checks whether a valid interval is being constructed.

```
struct Interval2
    left::Float64
    right::Float64

    function Interval2(l::Real, r::Real)::Interval2
        @assert l <= r "left endpoint must be less than
                        or equal to right endpoint"
        new(l, r)
    end
end
```

In the first call below, the endpoints are valid; in the second one, they are not.

```
julia> Interval2(0, 1)
Interval2(0.0, 1.0)
julia> Interval2(1, 0)
ERROR: AssertionError: left endpoint must be less than or equal to
                       right endpoint
```

A finer point of the `new` function defined in inner constructors is that it can be called with fewer arguments than the number of fields the type has. This feature makes the creation of instances of self-referential types possible. Although this may sound like a situation that is seldom encountered, we do not have to look far for such a data structure; a prime example is lists. In LISP, they consist of a data structure called a `cons` (which is short for construct). For example, the LISP expression

```
(cons 1 (cons 2 (cons 3 nil)))
```

evaluates to the list `(1 2 3)`. `cons` cells consist of two fields, which may contain arbitrary values. When `cons` cells are used to build a list, the first field (traditionally called `car` in LISP, which is short for "contents of the address part of register number" on the IBM 704 computer) holds a value and the second (traditionally called `cdr` in LISP, which is short for "contents of the decrement part of register number" on the IBM 704 computer) holds another `cons` cell or `nil`. Because of their structure, `cons` cells and hence lists are usually traversed recursively.

We start with a first try to define a `cons` cell in JULIA.

```
struct Cons1
    car::Any
    cdr::Cons1
end
```

While this self-referential type definition can be evaluated without any error, we encounter a problem when trying to create an instance. Just saying `Cons1()` or `Cons1(1, Cons1())` does not work, since all fields must be initialized and no instance of this type exists yet. The problem is that it is not possible to create an instance, because the second field must contain an instance of the same type.

The solution is to define an inner constructor that calls `new` with only one argument that initializes the `car` field (in addition to a second, standard constructor).

```
mutable struct Cons
    car::Any
    cdr::Cons

    function Cons(car::Any)::Cons
        new(car)
    end

    function Cons(car::Any, cdr::Cons)::Cons
        new(car, cdr)
    end
end
```

An undefined second field `cdr` is printed as *`#undef`*, and accessing it yields an error.

```
julia> Cons(1)
Cons(1, #undef)
julia> Cons(1).cdr
ERROR: UndefRefError: access to undefined reference
```

With this definition, we can mimic the list above using our `Cons` data type.

```
julia> Cons(1, Cons(2, Cons(3)))
Cons(1, Cons(2, Cons(3, #undef)))
```

Additionally, circular data structures can be defined. JULIA's printer is smart enough to detect them.

```
julia> circular = Cons(1)
Cons(1, #undef)
julia> circular.cdr = circular
Cons(1, Cons(#= circular reference @-1 =#))
```

The last function in this example shows how functions can operate safely on the `Cons` composite type, namely by using `isdefined`.

```
function length(cons::Cons)::Int
    if isdefined(cons, :cdr)
        1 + length(cons.cdr)
    else
```

```
        1
    end
end
```

This recursive definition of the length returns the correct value when the `Cons`es are interpreted as a list.

```
julia> length(Cons(1))
1
julia> length(Cons(1, Cons(2)))
2
julia> length(Cons(1, Cons(2, Cons(3))))
3
```

5.6 Type Unions

A type union is an abstract type that consists of the union of all types given after the **`Union`** keyword. A common example is a type that may take a value or not. Such types are sometimes useful when passing arguments or as return values.

```
julia> MaybeInt = Union{Int, Nothing}
Union{Nothing, Int64}
julia> 0::MaybeInt, nothing::MaybeInt
(0, nothing)
```

5.7 Parametric Types

Types in JULIA can be specified in a more fine-grained manner by parameters. Such types are called parametric types, and the parameters follow the name of the type surrounded by curly brackets. A prime example is vectors: an object of type `Vector{Int}` is a vector that can contain any `Int`, and a `Vector{Bool}` can contain only Boolean values, for example. In the following, parametric composite types and parametric abstract types are discussed in detail.

5.7.1 Parametric Composite Types

In a parametric composite type, the type parameter is used to further specify the types of the fields. For example, suppose we want to define an interval type whose endpoints have the given type `T`.

```
struct Interval3{T}
    left::T
    right::T
end
```

The type `Interval3` has type `UnionAll`. The type `UnionAll` represents the union of all types over all values of the type parameter.

```
julia> typeof(Interval3)
UnionAll
```

The type of the parametric type `Interval3{BigInt}` is `DataType`, as expected.

```
julia> typeof(Interval3{BigInt})
DataType
```

Concrete parametric composite types are subtypes of the underlying composite type (of type `UnionAll`).

```
julia> Interval3{BigInt} <: Interval3
true
julia> Interval3{Rational} <: Interval3
true
```

On the other hand, a concrete parametric composite type is never a subtype of another concrete parametric composite type, even if one type parameter is a subtype of the other.

```
julia> Rational <: Number
true
julia> Interval3{Rational} <: Interval3{Number}
false
```

This is due to the practical reason that composite types should be stored as efficiently in memory as possible. For example, while `Interval3{Int64}` can be stored as two adjacent 64-bit values, this is not true for `Interval3{Real}`, which entails the allocation of two `Real` objects.

The above fact has ramifications for the definition of methods. Suppose we want to define a function that calculates the midpoint of an interval of numbers.

```
function midpoint1(i::Interval3{Number})
    (i.left + i.right) / 2
end
```

This method does *not* work as intended, as the following function call shows.

```
julia> midpoint1(Interval3(-1, 1))
ERROR: MethodError: no method matching midpoint1(::Interval3{Int64})
```

The reason is that `Interval3{Int64}` is not a subtype of `Interval3{Number}` as explained above.

```
julia> Interval3{Int64} <: Interval3{Number}
false
```

There are three ways to define suitable methods, whose syntaxes differ slightly.

```
function midpoint2(i::Interval3{<:Number})
    (i.left + i.right) / 2
end

function midpoint3(i::Interval3{T} where T<:Number)
    (i.left + i.right) / 2
end

function midpoint4(i::Interval3{T}) where T<:Number
    (i.left + i.right) / 2
end
```

To construct objects of a parametric composite type, two default constructors can be used (unless other constructors have been defined, see Sect. 5.5). The first option is to use the default constructor of the concrete parametric composite type as in the following example.

```
julia> Interval3{BigInt}(-1, 1)
Interval3{BigInt}(-1, 1)
julia> typeof(ans)
Interval3{BigInt}
julia> typeof(Interval3{BigInt}(-1, 1).left)
BigInt
```

The second option is to use the default constructor of the underlying parametric composite type (of type `UnionAll`), which is `Interval3` in this example, as long as the implied value of the parameter type `T` is unambiguous. In these two examples, the underlying type is unambiguous.

```
julia> typeof(Interval3(0, 1))
Interval3{Int64}
julia> typeof(Interval3(1//2, 2//3))
Interval3{Rational{Int64}}
```

In the following two examples, the underlying type is ambiguous and errors result.

```
julia> typeof(Interval3(0, 1.0))
ERROR: MethodError: no method matching Interval3(::Int64, ::Float64)
julia> typeof(Interval3(0, 1//2))
ERROR: MethodError: no method matching
                    Interval3(::Int64, ::Rational{Int64})
```

5.7.2 Parametric Abstract Types

Similarly to parametric composite types, parametric abstract types declare a family of abstract types. We continue the interval example.

```
abstract type GeneralInterval{T} end
```

Just as in the case of parametric composite types, each concrete parametric abstract type is a subtype of the underlying abstract type (of type `UnionAll`). Furthermore, a concrete parametric abstract type is never a subtype of another concrete parametric abstract type, even if one type parameter is a subtype of the other.

The notation `GeneralInterval{<:Real}` denotes the set of all types `GeneralInterval{`T`}` where T is a subtype of **`Real`**, and analogously `GeneralInterval{>:Real}` denotes the set of all types `GeneralInterval{`T`}` where T is a supertype of **`Real`**. This is illustrated in the following examples.

```
julia> typeof(GeneralInterval)
UnionAll
julia> typeof(GeneralInterval{<:Real})
UnionAll
julia> typeof(GeneralInterval{>:Real})
UnionAll
julia> GeneralInterval{Int} <: GeneralInterval{<:Real}
true
julia> GeneralInterval{Real} <: GeneralInterval{>:Int}
true
```

The purpose of abstract types is to create type hierarchies over concrete types. This is exactly how the parametric abstract type is used in the following example.

```
struct Interval{T} <: GeneralInterval{T}
    left::T
    right::T
end

struct UnitInterval{T} <: GeneralInterval{T}
    left::T
    right::T

    function UnitInterval{T}(left::T, right::T) where T
        @assert right - left == 1
        new(left, right)
    end
end
```

With these definitions, it is ensured that unit intervals always have length one.

```
julia> UnitInterval{Int}(0, 1)
UnitInterval{Int64}(0, 1)
julia> UnitInterval{Int}(0, 2)
ERROR: AssertionError: right - left == 1
```

By introducing the parametric abstract type `GeneralInterval`, two kinds of inclusions hold. First, each parametric concrete type is a subtype of the corresponding parametric abstract type because their parameter types are the same. These inclusions hold due to our definitions of the concrete types as subtypes of the abstract type.

```
julia> Interval{Int} <: GeneralInterval{Int}
true
julia> Interval{Float64} <: GeneralInterval{Float64}
true
julia> UnitInterval{Int} <: GeneralInterval{Int}
true
julia> UnitInterval{Float64} <: GeneralInterval{Float64}
true
```

Second, further inclusions can be realized using the notation `<:`*type* explained above. In the first example here, `Interval{Int}` is not a subtype of `GeneralInterval{Real}` by the general rule above. However, in the second example, `<:Real` makes it possible to denote such a set of types; `Interval{Int}` is a subtype of `GeneralInterval{<:Real}` because `Interval` is a subtype of the abstract type `GeneralInterval` by its definition and because the parameter type `Int` is a subtype of `Real` (and the usage of `<:Real`).

```
julia> Interval{Int} <: GeneralInterval{Real}
false
julia> Interval{Int} <: GeneralInterval{<:Real}
true
julia> UnitInterval{Int} <: GeneralInterval{Real}
false
julia> UnitInterval{Int} <: GeneralInterval{<:Real}
true
```

Another use of the notation `<:`*type* is to restrict the allowed types. In the following example, we define intervals of characters and of integers.

```
struct CharInterval{T<:AbstractChar} <: GeneralInterval{T}
    left::T
    right::T
end

struct IntInterval{T<:Integer} <: GeneralInterval{T}
    left::T
    right::T
end
```

The parameter types are indeed restricted as shown here.

```
julia> IntInterval{Int8}(0, 100)
IntInterval{Int8}(0, 100)
julia> IntInterval{Rational}(0, 100)
ERROR: TypeError: in IntInterval, in T, expected T<:Integer,
                  got Type{Rational}
```

5.8 Tuple Types

The purpose of tuples is to model the argument lists of functions. Tuple types take multiple type parameters, each corresponding to the type of an argument in order. Because of their purpose, they have the following special properties.

1. Tuples types may be parameterized by an arbitrary number of types.
2. Tuples types are only concrete if their type parameters are.
3. **`Tuple`**$\{S_1, \dots, S_n\}$ is a subtype of **`Tuple`**$\{T_1, \dots, T_n\}$ if each type S_i is a subtype of the corresponding type T_i. This property is, of course, what is needed to determine which methods of a generic function match an argument list.
4. In contrast to composite types, tuples do not have field names, and hence their fields can only be accessed by their index. However, named tuples do have field names.

Because tuples are used for passing arguments to functions and receiving return values from functions and hence are an often used type, the syntax to construct tuple values is very simple. Additionally to the default constructors such as **`Tuple`**`{`**`Int`**`}(1)`, tuple values can be written in parentheses with commas in between, and an appropriate type is automatically constructed as well. It is important to note that a tuple with a single element is still written with a comma at the end in order to make the syntax unambiguous as seen here.

```
julia> typeof((1))
Int64
julia> typeof((1, ))
Tuple{Int64}
julia> typeof((1, 2.0, 3//1, 4im))
Tuple{Int64,Float64,Rational{Int64},Complex{Int64}}
julia> ans <: Tuple{Int, Real, Real, Complex}
true
```

The last line illustrates the second property above.

We already know from Sect. 2.8 that functions can take a variable number of arguments. This corresponds to the case when the last parameter of a tuple type has the type **`Vararg`**, which represents an arbitrary number (including zero) of trailing elements of the given type. The type **`Vararg`** also takes an optional second argument, which indicates the number of elements.

```
julia> varargs = Tuple{Float64, Vararg{Int}}
Tuple{Float64, Vararg{Int64}}
julia> isa((1.0, ), varargs)
true
julia> isa((1.0, 2), varargs)
true
julia> isa((1.0, 2, 3), varargs)
true
julia> isa((1.0, 2, 3, 4//1), varargs)
false
```

While tuples do not have field names, named tuples do. The **`NamedTuple`** type takes two parameters, namely a tuple of symbols indicating the field names and a tuple with the field types. The corresponding parameterized type is constructed automatically when a named tuple is constructed.

```
julia> typeof((arg1 = 1, arg2 = 2.0, arg3 = 3//1))
NamedTuple{(:arg1, :arg2, :arg3), Tuple{Int64, Float64,
                                        Rational{Int64}}}
```

The first argument of the constructor **`NamedTuple`** specifies the names, and the second, optional one specifies the types. If the types are specified, the arguments are converted using `convert`; otherwise, their types are inferred automatically. Note that the values are specified as tuples as well.

```
julia> NamedTuple{(:a, :b)}((1, 1.0))
(a = 1, b = 1.0)
julia> typeof(ans)
NamedTuple{(:a, :b), Tuple{Int64, Float64}}
julia> NamedTuple{(:a, :b), Tuple{Int8, Float32}}((1, 1))
(a = 1, b = 1.0f0)
julia> typeof(ans)
NamedTuple{(:a, :b), Tuple{Int8, Float32}}
```

Another way to construct an instance of **`NamedTuple`** is provided by the macro `@NamedTuple`, which yields a type. Names and types can be indicated, and omitted types default to **`Any`**. In the first example, no types, only names, are specified.

```
julia> @NamedTuple{a, b}
NamedTuple{(:a, :b), Tuple{Any, Any}}
julia> typeof(ans)
DataType
julia> @NamedTuple{a, b}((1, 1.0))
NamedTuple{(:a, :b), Tuple{Any, Any}}((1, 1.0))
```

In the second example, the types are indicated as well, and a **`NamedTuple`** type is returned by the macro accordingly. The arguments to the constructor are converted to the indicated types when the object is created.

```
julia> @NamedTuple{a::Int8, b::Float32}
NamedTuple{(:a, :b), Tuple{Int8, Float32}}
julia> typeof(ans)
DataType
julia> @NamedTuple{a::Int8, b::Float32}((1, 1))
(a = 1, b = 1.0f0)
```

5.9 Pretty Printing

After defining a custom composite type, it is often useful to customize how the objects are printed, if only to make the output more readable, i.e., to pretty print it. This can be accomplished by defining methods for the generic function `Base.show`. We consider the example of intervals again.

```
struct Interval
    left_open::Bool
    left::Number
    right_open::Bool
    right::Number
end
```

The default `Base.show` method prints values such that the resulting string yields a valid object again after parsing.

```
julia> Interval(false, 0, true, 1)
Interval(false, 0, true, 1)
```

This property is illustrated by the following piece of code.

```
let io = IOBuffer()
    Base.show(io, Interval(false, 0, true, 1))
    dump(Meta.parse(String(take!(io))))
end
```

The generic function `Base.show` takes the output stream as its first argument (usually called `io::IO`) and the object to be printed as its second. Note that it is necessary to mention the module name `Base` to add a method to the correct generic function.

```
function Base.show(io::IO, i::Interval)
    print(io,
          i.left_open ? "(" : "[",
          i.left, ", ", i.right,
          i.right_open ? ")" : "]")
end
```

With this method, a standard mathematical notation is achieved.

```
julia> Interval(false, 0, true, 1)
[0, 1)
```

5.10 Operations on Types

Abstract, composite, and a few other types are instances of the type **`DataType`**, as seen in this example.

```
julia> (typeof(Int), typeof(Any))
(DataType, DataType)
```

As we have seen already, ordinary functions can operate on types, since they are objects of type **`DataType`** themselves. In this section, the operations on types are briefly summarized.

The subtype operator `<:` determines whether the type on its left is a subtype of the type on its right. The function `isa` determines whether its first argument is an object of the second argument, a type. The function `typeof` returns the type of its argument.

```
julia> typeof(DataType)
DataType
```

The function `supertype` returns the supertype of its argument, and `supertypes` returns all supertypes of its argument. Finally, the function `subtypes` returns all subtypes.

5.11 Bibliographical Remarks

A comprehensive introduction to type systems and to the basic theory of programming languages is [2]. An excellent review of the IEEE floating-point standard can be found in [1].

Problems

5.1 (Numerical tower) Use the functions in Sect. 5.3 to obtain the numerical tower, i.e., the whole hierarchy below the type **`Number`**. Draw the numerical tower (by hand, but see Problem 5.2). How does the numerical tower correspond to the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$?

5.2 (Visualize type graph) $*$ Write a function that obtains the full type graph. Then write a function that writes an input file for graph visualization software such as Graphviz to plot the type graph.

5.3 (Type hierarchy for intervals) Define a type hierarchy for intervals below an abstract type called `GeneralInterval`. The types in the type hierarchy should provide for intervals with finite and infinite numbers of elements. Define intervals for Unicode characters, for integers, for rational numbers, and for floating-point numbers.

5.4 (Interval arithmetic) Define generic functions for the addition, subtraction, multiplication, and division of numeric intervals. The functions should take numeric intervals of the same type as their inputs and return a newly constructed interval of the same type (and a Boolean value to indicate whether a division by zero occurred in the case of division).

5.5 (Bisection) Bisection is a straightforward algorithm for calculating a root of a continuous function on a given real-valued interval. Starting with an interval on whose endpoints the function to bisect has opposite signs, the interval is split at its midpoint. Depending on the sign of the function at the midpoint, bisection continues recursively.

1. Implement bisection based on the data type for floating-point intervals in Problem 5.3.
2. Find a suitable stopping criterion that works with floating-point numbers.
3. As an example, apply your bisection implementation to finding the root of sin in the interval $[3, 4]$.

References

1. Goldberg, D.: What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys* **23**(1), 5–48 (1991)
2. Pierce, B.: *Types and Programming Languages.* MIT Press (2002)

Chapter 6
Control Flow

Wenn das Wörtchen “wenn” nicht wär,
wär deer Bettelmann Kaiser.

—Proverb

Abstract The control flow in a function or program is not just linear, but is determined by branches, loops, and non-local transfer of control. The standard control-flow mechanisms usually found in high-level programming languages such as compound expressions, conditional evaluation, short-circuit evaluation, repeated evaluation, and exception handling are available in JULIA and are discussed in detail in this chapter. Additionally, tasks are a powerful mechanisms for non-local transfer of control and make it possible to switch between computations. Parallel or distributed computing is discussed in detail as well, presenting various techniques how to distribute computations efficiently and conveniently.

6.1 Compound Expressions

Similar to `progn` in COMMON LISP , **`begin`** blocks and semicolon chains evaluate the constituent expressions in order and return the value of the last expression.

A **`begin`** block begins with the keyword **`begin`** and ends with the keyword **`end`**. Instead of these keywords, parentheses can also be used to the same effect, resulting in a semicolon chain. The expressions in a **`begin`** block are separated by newlines, by semicolons, or by both. The expressions in a semicolon chain are separated by semicolons `;`.

The following examples show the various cases that can occur. Global variables `a` and `b` are defined in each example, i.e., **`begin`** blocks do not introduce a new scope.

© Springer Nature Switzerland AG 2022
C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_6

```
julia> c = begin
    global a = 3
    global b = 2
    a^a // b^b
end
27//4
julia> c = begin
    global a = 3; global b = 2;
    a^a // b^b;
end
27//4
julia> c = begin global a = 3; global b = 2; a^a // b^b end
27//4
julia> c = (global a = 3; global b = 2; a^a // b^b)
27//4
```

Empty expressions such as **`begin end`** and `(;)` return **`nothing`**, which is of type **`Nothing`** and which is not printed by the REPL.

```
julia> begin end
julia> ans == nothing
true
julia> typeof(begin end)
Nothing
```

6.2 Conditional Evaluation

Conditional evaluation means that expressions are evaluated or not depending on the value of a Boolean expression. The syntax of **`if`** expressions is

`if` *condition*$_0$
    expressions$_0$
`elseif` *condition*$_1$
    expressions$_1$
`else`
    expressions
`end`.

All of the **`elseif`** clauses as well as the **`else`** clause are optional. The **`elseif`** clauses **`elseif`** *condition*$_i$ *expressions*$_i$ can be repeated arbitrarily often.

If the Boolean expression *condition*$_0$ in the **`if`** clause is true, then the corresponding expressions *expressions*$_0$ are evaluated; if it is false, then the condition *condition*$_1$ in the first **`elseif`** clause is evaluated. Again, if it is true, then the corresponding expressions *expressions*$_1$ are evaluated; if it is false, then the next

elseif clause is considered, etc. If none of the conditions is true, then the expressions *expressions* after the **else** clause are evaluated.

In other words, the expressions following the first **true** condition are evaluated, and the rest of the conditions are not considered anymore. If none of the conditions is true, the expressions in the **else** clause are evaluated if present.

```
function my_sign(x)
    if x < 0
        typeof(x)(-1)
    elseif x == 0
        typeof(x)(0)
    else
        typeof(x)(1)
    end
end

julia> typeof(my_sign(Int8(2)))
Int8
```

if blocks do not introduce a new scope, implying that local variables that are defined or changed within an **if** block remain visible after the **if** block.

if expressions return a value after having been evaluated, namely the value of the last expression evaluated.

```
julia> foo = if 1 > 0 "yes" else "no" end
"yes"
```

There is an additional syntax, the so-called ternary operator, for **if else end** blocks whose expressions are single expressions. The ternary operator

condition$_0$ **?** *expression*$_0$ **:** *expression*

is equivalent to

if *condition*$_0$
    expression$_0$
else
    expression
end.

It is used to squeeze **if** expressions into a single line.

```
function compare(a, b)
    string(a) * " is " *
        (a < b ? "less than " : "greater than or equal to ") *
        string(b)
end

julia> compare(0, 1)
"0 is less than 1"
```

Ternary operators can be chained, and to facilitate this, the ternary operator associates from right to left.

```
function compare(a, b)
    string(a) * " is " *
        (a < b ? "less than " :
         a == b ? "equal to " : "greater than ") *
        string(b)
end

julia> compare(0, -1), compare(0, 0), compare(0, 1)
("0 is greater than -1", "0 is equal to 0", "0 is less than 1")
```

6.3 Short-Circuit Evaluation

The Boolean operators `&&` and `||` implement the logical "and" and "or" operations. However, not all of their arguments are generally evaluated; only the minimum number of arguments necessary to determine the value of the whole expression is evaluated from the left to the right. This means that the evaluation is short-circuited if the value of the whole expression can be known in advance.

For example, in the expression `a && b`, the second argument `b` is evaluated only if `a` evaluates to **`true`**, since otherwise – if `a` is **`false`** – it is already obvious that the whole expression must be **`false`** after evaluating `a`. Analogously, in the expression `a || b`, the second argument `b` is evaluated only if `a` evaluates to **`false`**.

The `&&` operator has higher precedence than the `||` operator, which sometimes makes it possible to leave out parentheses. However, it is preferable in most cases to write out the parentheses in order to make the intent of the program immediately clear.

Short-circuit evaluation can also be used as an alternative short form for certain short **`if`** expressions. For example, when checking the arguments of a function and acting accordingly, checks may only occupy one line. In this example, the two functions are equivalent. (In general, using the `@assert` macro to check arguments is preferable.)

```
function fib1(n::Int)
    n >= 0 || error("n must be non-negative")
    0 <= n <= 1 && return n
    fib1(n-1) + fib1(n-2)
end

function fib2(n::Int)
    if n < 0 error("n must be non-negative") end
    if 0 <= n <= 1 return n end
    fib2(n-1) + fib2(n-2)
end
```

Whether saving a few characters is worth the terser and slightly obscured appearance of the first version lies in the eye of the beholder.

The condition expressions in **`if`** expressions, in the ternary operator, and the operands of the `&&` and `||` operators must be **`Bool`** values. The only exception are the last arguments in `&&` and `||` chains, whose values may be returned.

```
julia> 0 || true
ERROR: TypeError: non-boolean (Int64) used in boolean context
julia> false || 0
0
```

Note that in contrast to the `&&` and `||` operators, the functions `&` and `|` are just generic functions *without* short-circuit behavior; they are only special in the sense that they support infix syntax.

```
julia> true & true
true
julia> (&)(true, true)
true
julia> true | true
true
julia> |(true, true)
true
```

6.4 Repeated Evaluation

While branching using **`if`** expressions achieves nonlinear control flow, it is possible to evaluate expressions repeatedly using two constructs: **`while`** loops and **`for`** loops.

We discuss **`while`** loops first, as they are more general. The syntax is

`while` *condition*
    expressions
`end`,

where the *condition* must be a Boolean expression. While the *condition* evaluates to **`true`**, the *expressions* in the body of the **`while`** loop are evaluated. In contrast to **`for`** loops, the programmer is responsible for defining and updating an iteration variable if one is needed.

```
global i = 1
while i <= 3
    global i
    @show i
    i += 1
end
```

This example prints three lines as expected. Note that the **`global`** declaration is needed here in order to change the change. An equivalent version of this loop is the following.

```
global j = 0
while j <= 2
    global j += 1
    @show j
end
```

Some programming languages contain `do until` statements. They are equivalent to **`while`** loops as the following example shows.

```
global k = 0
while true
    global k += 1
    @show k
    if k >= 3
        break
    end
end
```

When the number of iterations is known in advance or one needs to iterate over an iterable data structure, it is usually more convenient to use a **`for`** loop. The syntax of a simple **`for`** loop with a single iteration variable is

`for` *i* **`in`** *iterable*
    expressions
`end`.

The keyword **`in`** can be replaced by `=`. Here *i* is the iteration variable, *iterable* is the iterable data structure, and the *expressions* are the body to be evaluated.

This **`for`** loop is equivalent to the following **`while`** loop.

```
global next = Base.iterate(iterable) # e.g. iterable = 1:3
while next !== nothing
    local (iterate, state) = next
    # expressions
    global next = Base.iterate(iterable, state)
end
```

Both loops are linked via the generic function `iterate`, which we have to access as `Base.iterate` when defining additional methods. It must have two methods for the type of `iterable`, namely one taking one arguments and one taking two arguments.

We consider the example of iterating over the coordinates of a three-dimensional point and first define a data structure called `Point` (see Sect. 5.4).

```
struct Point
    x::Float64; y::Float64; z::Float64
end
```

The first method takes one argument (as in the call of `iterate` before the **`while`** loop above) and returns an iterate and a state.

```
function Base.iterate(p::Point)::Tuple
    (p.x, Symbol[:y, :z])
end
```

The state can be any object, but it should be defined in such a way that it is conducive for iterating by the second method. The second method takes the iterable data structure and a state as its two arguments and returns an iterate and a state as well.

```
function Base.iterate(p::Point, state::Vector)::Union{Nothing,
                                                      Tuple}
    if isempty(state)
        nothing
    else
        (getfield(p, state[1]), state[2:end])
    end
end
```

Having defined the two `iterate` methods, we can use the **`while`** loop above to iterate over the coordinates of a `Point`.

```
global point = Point(1, 2, 3)
global next = Base.iterate(point)
while next !== nothing
    local (iterate, state) = next
    @show iterate, state
    global next = Base.iterate(point, state)
end
```

```
(iterate, state) = (1.0, [:y, :z])
(iterate, state) = (2.0, [:z])
(iterate, state) = (3.0, Symbol[])
```

Much more interestingly, however, we have extended the built-in **`for`** loop by defining these two `iterate` methods. This also means that *iterable* data structures are those for which `iterate` methods have been defined.

```
for coord in Point(1, 2, 3)
    @show coord
end
```

```
coord = 1.0
coord = 2.0
coord = 3.0
```

for loops can be nested, but nested loops can be written more succinctly using the syntax

for i_1 **in** *iterable*$_1$, i_2 **in** *iterable*$_2$
    expressions
end,

where an arbitrary number of iteration variables can be given. It is again possibly to replace the keyword **in** by =.

```
for i in 1:2, j in 3:4
    @show i, j
end
```

```
(i, j) = (1, 3)
(i, j) = (1, 4)
(i, j) = (2, 3)
(i, j) = (2, 4)
```

As this example shows, the first iteration variable (`i` here) changes slowest and the last iteration variable (`j` here) changes fastest.

Another extension of the basic syntax of **for** loops is destructuring of the iteration variable. Destructuring of variables is an idea also found in COMMON LISP and CLOJURE. In JULIA, it means that if the iteration variable is a tuple, then its components are bound to the respective components of the elements of the iterable data structure.

```
for (a, b) in ((1, 2), (3, 4))
    @show a, b
end
```

In the first iteration, the two components `a` and `b` of the iteration variable `(a, b)`, a tuple, are bound to the components of the first element `(1, 2)` of the iterable data structure. This loop hence yields the following output.

```
(a, b) = (1, 2)
(a, b) = (3, 4)
```

The data structure to be iterated over may be any iterable data structure. In the next example, it is a set.

```
for (a, b) in Set([(1, 2), (3, 4), (5, 6)])
    @show a, b
end
```

```
(a, b) = (1, 2)
(a, b) = (5, 6)
(a, b) = (3, 4)
```

It is only important that the elements of the iterable data structure are tuples that are compatible with the iteration variable. If the iteration variable is a tuple, it is compatible with the tuples in the iterable data structure if it has the same number of elements or fewer. If the tuple acting as the iteration variable is too long, an error is raised. If it is shorter than the data, then only the given elements are bound as shown in the following example. (Recall that the syntax for a tuple with a single element is `(a,)`, which is necessary to distinguish it from `(a)`, which is the same as just `a`.)

```
for (a,) in Set([(1,2), (3,4), (5,6)])
    @show a
end
```

```
a = 1
a = 5
a = 3
```

Tuples as iteration variables are especially convenient in conjunction with `enumerate`. The function `enumerate` takes an iterable data structure and yields iterates (i, x), a tuple, where i is a counter starting at one and x iterates over the given iterable data structure. This is useful in loops where the number of elements iterated over so far is needed

```
import Primes
import Printf
for (i, p) in enumerate(Primes.PRIMES[1:5])
    Printf.@printf("prime number no. %1d: %2d\n", i, p)
end
```

Here `i` is a counter starting at `1` and `p` iterates over the collection which is the argument of `enumerate`.

```
prime number no. 1:  2
prime number no. 2:  3
prime number no. 3:  5
prime number no. 4:  7
prime number no. 5: 11
```

It is possible to stop **`while`** and **`for`** loops using the **`break`** keyword. This example returns the smallest prime number greater than or equal to 2020.

```
import Primes
global i = 2020
while true
    if Primes.isprime(i)
        break
    else
        global i += 1
    end
end
i
```

The **continue** keyword makes it possible to shortcut an iteration within a **while** or **for** loop and to proceed to the next iteration. This example prints the primes numbers between 2020 and 2050.

```
import Primes
for j in 2020:2050
    if mod(j, 2) == 0 || mod(j, 3) == 0
        continue
    elseif Primes.isprime(j)
        println(j)
    end
end
```

Finally, the classical `go to` statement is available as the `@goto` macro and used in conjunction with the `@label` macro. Although `go to` statements have generally fallen out of favor in modern programming style, they are very useful in certain cases. A prime example is the implementation of finite-state machines, where a state transition table and the `@goto` macro can be used to switch between the states. Examples of finite-state machines are parsers and regular expressions.

```
import Primes
function find_first_prime_after(n::Integer)::Integer
    @assert n >= 2

    @label start
    if Primes.isprime(n)
        return n
    else
        n += 1
        @goto start
    end
end
```

6.5 Exception Handling

The language constructs discussed so far result in local control flow; even conditional and repeated evaluation cannot result in non-local transfer of control. By considering the program text locally, it is clear which expression will be evaluated next.

Throwing an exception is a non-local control flow. Exceptions are useful in situations when an unexpected condition occurs and a function cannot compute and return the value it is meant to return. In these cases, exceptions can be thrown and caught. When the exception is caught, it is decided how to proceed best, e.g., by terminating the program, printing an error message, or by taking a corrective action such as retrying.

6.5.1 Built-in Exceptions and Defining Exceptions

The built-in exceptions are subtypes of the abstract type `Exception` and can be listed by `@doc Exception`. Using the type system, it also is possible to define custom exceptions as in this example.

```
struct MyAwesomeException <: Exception
end
```

Here `<:` indicates that the new type `MyAwesomeException` will be a subtype of `Exception`. The type system and how to define new types are explained in detail in Chap. 5.

6.5.2 Throwing and Catching Exceptions

Exceptions are thrown by the `throw` function. This example throws the fitting `DomainError` for negative arguments when Fibonacci numbers are calculated.

```
function fib(n::Integer)::BigInt
    if n < 0
        throw(DomainError("negative integer"))
    elseif n <= 1
        n
    else
        fib(n-2) + fib(n-1)
    end
end
```

The argument to `throw` must be an exception and not a type of exception. The function call `DomainError(`*arg*`)` yields an exception, while `DomainError` is the type.

```
julia> typeof(DomainError("foo")), typeof(DomainError)
(DomainError, DataType)
julia> typeof(DomainError("foo")) <: Exception
true
```

As usual, the built-in function`<:` tests whether the left-hand side is a subtype of the right-hand side.

Exceptions take arguments to describe the situation as in this example.

```
julia> throw(UndefVarError(:foo))
ERROR: UndefVarError: foo not defined
```

User defined exceptions can also take arguments, which are the fields of the exception type (see Sect. 5.4). In the following example, we define an exception type called `DivisionByZero` with a field called `numerator`, which holds additional information.

```
struct DivisionByZero <: Exception
    numerator::Number
end
```

In order to provide an informative error message, we also define a method for the generic function `Base.showerror`, which is called by JULIA to show an error. (As usual, you can find all methods defined for this generic function using `methods(Base.showerror)`.)

```
function Base.showerror(io::IO, exc::DivisionByZero)
    print(io, exc.numerator, " cannot be divided by zero")
end
```

With these definitions, our new type of exception behaves as intended.

```
julia> throw(DivisionByZero(42))
ERROR: 42 cannot be divided by zero
```

At this point, we know how to throw exceptions. For non-local control flow, we must also be able to catch the exceptions. This facility is provided by the

```
try
    body
catch [exception]
    handler
finally
    cleanup
end
```

expression. It evaluates the expressions in *body* until an exception occurs. If an exception has occurred, the expressions in *handler* are evaluated, where the exception is bound to the optional variable named *exception* so that the exception can be inspected and handled. Finally, the expressions in *cleanup* are evaluated in any case.

The first, simple example is built around the `log` function. The built-in `log` function raises a `DomainError` when called with a negative real-valued argument and returns a complex number only when called with a **`Complex`** argument. In the example, we try the built-in version first; if it does not succeed and an exception is raised, it is retried with a **`Complex`** argument.

```
function my_log(x)
    try
        log(x)
    catch
        log(Complex(x, 0))
    end
end

julia> my_log(Base.MathConstants.e)
1
julia> my_log(-1)
0.0 + 3.141592653589793im
```

In the second example, the exception is bound to the variable `exc` for closer inspection.

```
function my_log(x)
    try
        log(x[2])
    catch exc
        if isa(exc, DomainError)
            log(Complex(x, 0))
        else
            @info "The next exception has been rethrown."
            rethrow(exc) # or just rethrow()
        end
    end
end
```

Whenever there is an exception that cannot or should not be handled in the handler after the **`catch`** keyword after all, it can be rethrown by the `rethrow` function. Within the handler expressions, it suffices to just call `rethrow()`; the exception is the default argument.

```
julia> my_log(-1)
[ Info: The next exception has been rethrown.
ERROR: BoundsError
```

The syntax of the **try** expression requires some care regarding the variable name after the **catch** keyword. It must be on the same line as the **catch** keyword.

Since any symbol after the **catch** keyword is interpreted as the variable name for the exception unless it is written on a new line or separated by a semicolon, one must also be careful when the intention is to return the value of another variable and the whole expression is written on a single line. The following function returns its argument `x` if an exception is raised.

```
function my_log(x)
    try
        log(x)
    catch # no variable name for the exception
        x
    end
end
```

If the **try** expression is written on a single line, it must look like this. Note the semicolon; it ensures that `x` is *not* interpreted as the variable name for the exception, but as the return value of the *handler* expressions.

```
my_log(x) = try log(x) catch; x end

julia> my_log(-1)
-1
```

On the other hand, if there is no semicolon, the function is still syntactically correct. Then, however, `x` is interpreted as the variable name for the exception and the *handler* expressions are empty, resulting in **nothing** as the return value.

```
my_log(x) = try log(x) catch x end

julia> my_log(-1)
julia> my_log(-1) == nothing
true
```

The **catch** clause is optional in **try** expressions. If it is omitted, **nothing** is returned as usual for empty expressions.

The **finally** clause is always evaluated, even if an exception was raised in the *body* or in the *handler* expressions. If an exception is raised in the *handler* expressions, then it is re-raised after evaluating the *cleanup* expressions in the **finally** clause.

```
function my_log(x)
    try
        log(x)
    catch
        error("still something went wrong")
    finally
        println("cleaning up")
    end
end

julia> my_log(-1)
cleaning up
ERROR: still something went wrong
```

A prime example of the usefulness of the **finally** clause is ensuring that an operating-system resource such as a file descriptor or a pipe is freed or closed under all circumstances. In this example, the error is not caught, but the **finally** clause ensures that the stream is closed.

```
function unfortunate_reader(filename::String)
    local stream = open(filename)
    try
        error("something went wrong, e.g., while parsing")
    finally
        close(stream)
        println("stream open: ", isopen(stream))
    end
end

julia> unfortunate_reader("/etc/passwd")
stream open: false
ERROR: something went wrong, e.g., while parsing
```

In practice, however, the `open` function is often used in conjunction with a **do** block (see Sect. 2.9). According to the documentation of `open`, it always closes the file descriptor upon completion.

Finally, the functions for handling exceptions are summarized in Table 6.1. Assertions are discussed in Sect. 6.5.4 below.

Table 6.1 Exception handling.

Function	Description
`error(`*message*`)`	raise an `ErrorException` with *message*
`@assert` *condition message*	throw an `AssertionError` if *condition* is **false**
`throw(`*exception*`)`	throw the *exception*
`rethrow(`[*exception*]`)`	rethrow *exception* from within a **catch** clause
`retry`	retry a function repeatedly
`backtrace`	return the current backtrace object
`catch_backtrace`	return the backtrace object of the current exception, for use in a **catch** clause

6.5.3 Messages, Warnings, and Errors

The general facility to log messages is the `@logmsg` macro. Often the four standard logging macros `@debug`, `@info`, `@warn`, `@error` are used, which are based on `@logmsg` and log messages at the four standard levels `Debug`, `Info`, `Warn`, and `Error`. The first argument to these four macro should be an expression that evaluates to a string that describes the situation. The string is formatted as MARKDOWN when printed. Further, optional arguments can be of the form *key* = *value* or *value* and are attached to the log message.

```
julia> @info "the answer is" answer = 42
Info: the answer is
  answer = 42
julia> @warn "something unexpected occurred" answer =
       log(Complex(-1, 0))
Warning: something unexpected occurred
answer = 0.0 + 3.141592653589793im
julia> @error "cannot divide by zero" denominator = 0
Error: cannot divide by zero
  denominator = 0
```

On the other hand, the `error` function (and not the macro) raises an exception of type `ErrorException`.

```
julia> try error("foo") catch exc exc end
ErrorException("foo")
```

6.5.4 Assertions

Assertions are useful to ensure that certain conditions are always satisfied during the evaluation of a program. For example, an expression may be known to be invariant in a loop; these invariants may be conserved quantities such as energy or angular momentum in a physical simulation. Assertions are also a convenient way to check whether argument values are valid.

Assertions are written using the macro `@assert`, which takes a **`Bool`** expression as its first arguments and an informative messages as its optional second.

```
julia> @assert 1 < 0
ERROR: AssertionError: 1 < 0
julia> @assert 1 < 0 "something is really wrong"
ERROR: AssertionError: something is really wrong
```

In the following example, we know that $\sum_{k=1}^{n} 1/k^2 = \pi^2/6$. Since all terms are positive, all partial sums must be less than $\pi^2/6$, which is checked by an assertion. The argument value is also checked by an assertion.

```
function summation(n::Integer)
    @assert n >= 1

    local s = BigFloat(0)
    for k in 1:n
        s += 1/BigFloat(k)^2
        @assert s < pi^2/6
    end

    s
end
```

6.6 Tasks, Channels, and Events

`Task`s (also known in computer science as symmetric coroutines or cooperative multitasking) are an advanced feature to control the flow of a program that makes it possible to start, suspend, and resume calculations. A `Task` can be viewed as a function call with the additional feature that it can be interrupted and that control is then transferred to another `Task`. The first `Task` can then be resumed where it was interrupted.

There are two main differences compared to usual function calls. First, in contrast to a function call, a `Task` switch does not use space on the call stack so that an arbitrary number of `Task` switches are possible without running out of stack space. Second, `Task`s can be run in any order, again in contrast to function calls, which must be completed before control returns to the caller and another function can be called.

`Task`s are very useful for certain problems without a clear caller-callee structure. A prime example is producer-consumer problems, where a function produces values and another one consumes them. In such problems, the consumer cannot simply call the producer, because calculating the values is a time consuming task and the producer may not be ready to return a value. Using `Task`s, the producer and the consumer can both run as long as required, and the values are passed between them when necessary.

`Task`s typically communicate using `Channel`s, which are first-in first-out queues. Multiple `Task`s can put values into a `Channel` and take values from it. In the following example, we define a producer that `put!`s prime numbers into a channel.

```
import Primes
function my_producer(ch::Channel, n::Integer)
    @assert n >= 1
    for p in Primes.primes(1, n)
        put!(ch, p)
    end
end
```

The producer function must be scheduled to run in a new `Task`. The most convenient way to do so is to use the `Channel` constructor that takes a function as its argument and runs a `Task` associated with the new `Channel`. The function given as the argument to the constructor must take one argument, namely the `Channel`. In our example, a `Channel` and an associated `Task` can be created by `chan = Channel(ch -> my_producer(ch, 9))` for example.

Values can be consumed from a `Channel` by the function `take!`; in our example, evaluating `take!(chan)` yields consecutive prime numbers. Values can also be conveniently consumed in `for` loops by iterating over the `Channel` as in the following example.

```
function my_consumer(n::Integer)
    @assert n >= 1
    for i in Channel(ch -> my_producer(ch, n))
        println(i)
    end
end
```

In the `for` loop, values are consumed as long as they are available from the `Channel`.

```
julia> my_consumer(9)
2
3
5
7
```

You may have expected to close the `Channel`. This is not necessary here, since the `Channel` is associated with the `Task`, and therefore the lifetime of the `Channel` being open is associated with the `Task`. The `Task` terminates when the function returns, at which point the `Channel` is closed automatically.

The operations on `Channel`s are summarized in Table 6.2. When creating a `Channel`, the type of the values to be passed may be specified. If no such type argument is given, the general type `Any` is used by default. If a function argument is given to the constructor, a `Task` is created and associated with the new `Channel` as discussed above. However, a `Channel` may also be created without the function argument. The size of the buffer of the `Channel` may be specified, where the default size zero creates an unbuffered `Channel`. `Channel(Inf)` is equivalent to `Channel{Any}(typemax(Int))`.

Table 6.2 Channel operations.

Function	Description
`Channel`{*T*}(`[`*size*`]`)	create a **`Channel`** for at most *size* objects of type *T*
`Channel`{*T*}(*f*`[`, *size*`]`)	create a **`Task`** and a **`Channel`** for at most *size* objects of type *T*
`put!(`*ch*, *value*`)`	append `value` to the **`Channel`** *ch* and block if it becomes full
`take!(`*ch*`)`	block until a value is available from **`Channel`** *ch*, then remove and return it
`isopen(`*ch*`)`	determine whether the **`Channel`** *ch* is open
`isready(`*ch*`)`	determine whether a value is available from **`Channel`** *ch*
`wait(`*ch*`)`	wait until a value becomes available from **`Channel`** *ch*
`fetch(`*ch*`)`	wait for and return the first value from the `channel` *ch*, but do not remove it from the **`Channel`**
`close(`*ch*`)`	close the **`Channel`** *ch*

Similarly to **`Channel`**s whose constructor takes a function argument, the argument of the **`Task`** constructor must be a function with no arguments. The `@task` macro serves the same purpose. Furthermore, the function `bind` associates a **`Channel`** with a **`Task`**, and the function `schedule` adds a **`Task`** to the queue of the scheduler, causing to task to be run.

The basic building block of symmetric coroutines is the function `yieldto`. It suspends the current task and receive a value at the same time. More precisely, the function call `yieldto(`*task*`,` *value*`)` suspends the current task and switches to *task*; then the last call of `yieldto` in *task* returns the *value*. The first time a task is switched to, the function it is associated with (i.e., the argument that was given to the **`Task`** constructor when it was created) is called with no argument. Therefore `yieldto` achieves both fundamental operations, i.e., suspending the current task on the one hand and transferring control to another task and receiving values on the other hand. Because of the symmetry between the **`Task`**s – they all call `yieldto` –, they are called symmetric coroutines.

While the function `yieldto` is the basic building block, it is not invoked directly in many use cases. Switching between tasks usually requires coordination between the tasks, which means that state must be maintained, e.g., to know which task is a producer and which is a consumer. Therefore it is often more convenient to use the functions `put!` and `take!`.

Table 6.3 summarizes operations on **`Task`**s. A **`Task`** is created by calling the constructor **`Task`** on a function with no arguments or by calling the macro `@task` on an expression. The function `schedule` takes additional arguments that are useful in certain situations. We will see how to use the `@async` and `@sync` macros in an example below. Having starting a **`Task`** with `@async`, the nearest enclosing `@sync` will wait for it to finish. In fact, the `@sync` macro will wait for all lexically enclosed uses of `@async`, `@spawn`, `@spawnat`, and `@distributed` to finish (see also Sect. 6.7).

The scheduler keeps track of the **`Task`**s, executes an event loop, and maintains a queue of runnable tasks. To this end, **`Task`**s have a field called `state` that contains one of three symbols and indicates the execution status of the **`Task`**. The

Table 6.3 Task operations.

Function	Description
`Task(`*f*`)`	create a `Task` to evaluate the function *f*
`@task` *expr*	create a `Task` to evaluate the expression *expr*
`yieldto(`*task*`, `*value*`)`	switch to *task* and return *value* from the last call of `yieldto` in *task*
`yield()`	allow another scheduled task to be run and remain `:runnable`
`current_task()`	return the currently running `Task`
`istaskstarted(`*task*`)` → `**`Bool`**	check whether *task* has been started
`istaskdone(`*task*`)`	check whether *task* has finished
`task_local_storage()`	return the local storage of the current task
`task_local_storage(`*key*`)`	return the value of *key* in the task-local storage
`task_local_storage(`*key*`, `*value*`)`	assign *value* to *key* in the task-local storage
`task_local_storage(`*f*`, `*key*`, `*value*`)`	call the function *f* with a modified task-local storage, which is restored afterward
`schedule(`*task*`)`	add *task* to the queue of the scheduler
`sleep(`*seconds*`)`	block the current task for *seconds* (at least 0.001)
`@async` *expression*	wrap *expression* in a `Task` and schedule it locally
`@sync` *expression*	wait until all uses of `@async` etc. have completed

`Task` states are described in Table 6.4. A newly created `Task` is initially not known to the scheduler and therefore not run.

Table 6.4 Task states.

State	Description
`:runnable`	currently running or able to run
`:done`	successfully finished, i.e., the function has returned
`:failed`	unsuccessfully finished, i.e., an uncaught exception was thrown

In cooperative multitasking, most `Task` switches are the result of waiting for events such as input or output requests. The generic function `wait` is the basic way to wait and includes methods for several types of objects such as `Task`s, `Channel`s, `Distributed.RemoteChannels`, `Base.Events`, and `Base.Processes`. The function `wait` is usually called implicitly; for example, the function `read` uses `wait` to wait for data to become available.

We now discuss how jobs or workloads can be distributed to `Task`s and how `Channel`s can be used for communication between these `Task`s and to collect the results. This example is a leading one, and these techniques are already useful for sequential computing on a single processor. (Parallel computing is discussed in Sect. 6.7 below.) For example, the jobs may be functions that mostly deal with input/output operations and that hence may `wait` for a substantial amount of time.

In the **let** expression below, we will define two variables that are buffered **Channel**s and that hold the jobs and their results. The jobs and the results are **NamedTuple**s, and the buffer sizes of both **Channel**s are only 5. The first function we define in this example creates jobs. We just draw a random number between zero and one, which represents the job. After all jobs have been written to the **Channel**, it is `close`d.

```
function make_jobs(jobs::Channel, n::Integer)
    for i in 1:n
        put!(jobs, (id = i, workload = rand()))
    end
    close(jobs)
end
```

The second function does the work. It uses a **for** loop to get all available jobs and writes the results of performing the jobs into the other channel. The work is trivial, as it is just `sleep`ing, but when running the example, it illustrates nicely how long it takes all jobs to finish.

```
function work(jobs::Channel, results::Channel, id::Integer)
    println("Worker $id started.")
    for j in jobs
        sleep(j.workload)
        put!(results, (id = j.id, worker = id, time = j.workload))
    end
    println("Worker $id finished.")
end
```

Next, we create ten jobs by wrapping the call to `make_jobs` in the `@async` macro. The `@async` macro creates a **Task** and adds it to the queue of the scheduler. It is expedient to use `@async` at this point, since creating the job descriptions may take some time or the number of jobs may exceed the buffer size, but in this way work can start immediately. Having created the jobs, we start three tasks by wrapping calls to `work` in the `@async` macro. Then, we take the results from the buffered **Channel** and print the total elapsed time.

```
let n = 10
    local jobs    = Channel{NamedTuple}(5)
    local results = Channel{NamedTuple}(5)

    @async make_jobs(jobs, n)

    for i in 1:3
        @async work(jobs, results, i)
    end

    @time for i in 1:n
        local r = take!(results)
        println("Job $(r.id) performed by worker $(r.worker) " *
                "took $(round(r.time; digits = 3)) seconds.")
    end
end
```

```
Worker 1 started.
Worker 2 started.
Worker 3 started.
Job 2 performed by worker 2 took 0.224 seconds.
Job 1 performed by worker 1 took 0.229 seconds.
Job 3 performed by worker 3 took 0.533 seconds.
Job 4 performed by worker 2 took 0.755 seconds.
Job 5 performed by worker 1 took 0.818 seconds.
Job 6 performed by worker 3 took 0.936 seconds.
Job 8 performed by worker 1 took 0.552 seconds.
Worker 2 finished.
Job 7 performed by worker 2 took 0.95 seconds.
Worker 3 finished.
Job 9 performed by worker 3 took 0.749 seconds.
Worker 1 finished.
Job 10 performed by worker 1 took 0.796 seconds.
  2.934174 seconds (4.15 M allocations: 180.415 MiB, 3.95% gc time)
```

In the next example, we use an unbuffered **`Channel`** for communication. The function `source` slowly writes values into a **`Channel`**. After all the values have been written, it is `close`d. The function `sink` receives the values. Since **`Channel`**s are iterable, we use a **`for`** loop as usual; the **`for`** loop iterates until the channel is closed.

```
import Primes

function source(ch, n)
    for i in Primes.primes(1, n)
        sleep(rand())
        println("Putting $(i). ")
        put!(ch, i)
    end
    close(ch)
end

function sink(ch)
    for i in ch
        println("Taking  $(i).")
    end
end
```

```
let ch = Channel(0)
    @sync begin
        @async source(ch, 10)
        @async sink(ch)
    end
end
```

The `let` expression runs these two functions. First, an unbuffered `Channel` is recreated. In the `begin` expression, `Task`s for the calls to the `source` and `sink` functions are created and run asynchronously by the macro `@async`. The `@sync` macro outside the `begin` expression waits till both tasks are done; it will return only when the `for` loop in the `sink` function has returned.

The final example in this section shows how `Condition`s can be used. The function `wait_a_bit` is run in a new `Task` after having been started by `schedule` and notifies the `Condition` when it is done. After `notify` has been called, control flow continues after the call to `wait`, which has been waiting for the `Condition`. This construct is more effective than a loop that polls the `Condition` repeatedly.

```
let c = Condition()
    function wait_a_bit()
        println("Waiting for Godot.")
        sleep(1 + rand())
        notify(c)
    end

    schedule(@task wait_a_bit())

    wait(c)
    println("He has arrived.")
end
```

6.7 Parallel Computing

Parallel computing is an important topic on today's and future hardware. As the physical sizes of CMOS transistors have been approaching their physical limits and heat dissipation has become the main limiting factor, the performance of single CPU cores has been stagnating. Therefore modern CPUs consist of multiple cores, single computers may possess multiple CPUs, and computers may be combined into clusters. Distributing the computations however comes with a communication cost.

JULIA's implementation of parallel or distributed computing is based on message passing, but provides higher-level operations than just sending and receiving of messages. Remote references and remote calls are the basic building blocks for parallel computing in JULIA. Remote references refer to objects stored on a certain process and can be used from any process; there are two types of remote references, namely `Futures` and `RemoteChannels`.

A remote call is the request by a process to call a certain function on certain arguments on another (or the same) process. Every remote call returns immediately, and the result of a remote call is a `Future`. The return value of the remote call can be obtained using the function `fetch`, or the function `wait` can be called on the `Future` to wait until the result is available.

6.7.1 Starting Processes

The JULIA system can use multiple processes. The process associated with the REPL always has ID 1, and additional processes with higher IDs can be started and are called workers. If there is only the process with ID 1, it is considered the only worker. The number of workers can be supplied when starting JULIA using the command-line arguments `-p` or `--procs`. The argument should be equal to the number of available (logical) cores or to `auto`, which determines the number of (logical) cores automatically. If the arguments `-p` or `--procs` are supplied on the command line, then the built-in module `Distributed` is loaded automatically.

```
> julia -p auto
julia> length(workers())
24
```

Within JULIA, the workers can be managed using the functions `workers`, `addprocs`, and `rmprocs`. The module `Distributed` must be loaded on the process with ID 1 before using `addprocs` to add workers.

Another option to start worker processes is to use the `--machine-file` command-line option. Then JULIA uses passwordless `ssh` login to start workers on the machines specified in the supplied file.

Worker process differ from the process with ID 1 by not evaluating the `startup.jl` startup file. Their global state, i.e., loaded modules, global variables, and generic functions and methods, is not synchronized automatically between processes. The common way to load modules or program files into all workers is to use the `@everywhere` macro by writing

```
@everywhere import module
```

or

```
@everywhere include("filename").
```

Furthermore, it is possible to write customized `ClusterManagers`, but this option is not discussed in detail here.

6.7.2 Data Movement and Processes

The disadvantage of parallel or distributed computing is that messages and data must be exchanged between processes. Reducing the amount of messages and data sent is of paramount importance for performance and scalability.

The function `remotecall` and the macro `@spawnat` are the basic operations for evaluating a function or an expression, respectively, on a worker process. The first argument of `remotecall` is the function to be called, the second is the process ID of the worker to be used, and the rest of the arguments are passed to the function to be called. The first argument of `@spawnat` is the ID of the worker process to be used or it is equal to `:any`, which lets the scheduler choose the worker process, and the second argument is the expression to be evaluated on the worker process. Both `remotecall` and `@spawnat` return immediately and yield a `Future` as the result.

The generic function `fetch` is the basic operation for moving data. It (or more precisely one of its methods) receives a `Future` as its argument, waits until the worker process has finished evaluating, and returns the value.

Maybe the simplest example is the following. Remember to use the module `Distributed` first, if you have started JULIA without the `-p` or `--procs` command-line options.

```
julia> using Distributed
julia> name = "world"
julia> @time fetch(@spawnat :any begin sleep(3); "Hello, $(name)!"
                                                                   end)
  3.065816 seconds (74.02 k allocations: 3.622 MiB)
"Hello, world!"
```

The example shows that required data are copied to the worker process automatically; here, the global variable `name` is accessed by the expression to be spawned and therefore it is made available to the worker process. After the expression has been evaluated on the worker process, `fetch` fetches its value from the worker process and returns it.

The example could have been written more succinctly using `@fetchfrom`, which is equivalent to `fetch` after `@spawnat`. The function `remotecall_fetch` is equivalent to applying `fetch` to the result of `remotecall`, but it is more efficient. The function `remote_do` evaluates a function on a worker with a given ID, but does not yield the return value of the function. It is also not possible to `wait` for the completion of the function call.

To illustrate the use of worker processes and `RemoteChannels`, we rework the jobs example in Sect. 6.6 to use remote workers and channels. The function `make_jobs` remains essentially unchanged.

```
function make_jobs(jobs::RemoteChannel, n::Integer)
    for i in 1:n
        put!(jobs, (id = i, workload = rand()))
    end
    close(jobs)
end
```

The function `work` to be run on the workers must be made available to all processes. Therefore we wrap the function definition into the `@everywhere` macro.

```
@everywhere function work(jobs::RemoteChannel,
                          results::RemoteChannel)
    while true
        local j
        try
            j = take!(jobs)
        catch exc
            break
        end
        sleep(j.workload)
        put!(results, (id = j.id, worker = myid(),
                       time = j.workload))
    end
    println("Worker $(myid()) has no more jobs to do.")
end
```

Since **`for`** loops over `RemoteChannel`s are not supported directly, we use a **`while`** loop instead. We would like to check whether the `jobs` channel is still open and then take a value from it. However, in the time between checking and taking a value, another worker process may have snatched the last available value, resulting in a race condition. Therefore we just **`try`** to take a value from the channel and **`catch`** any possibly resulting exception. If there is an exception, we know that the channel has been `close`d and we **`break`** the loop and hence end the function.

With these function definition, we can run remote workers and communicate via `RemoteChannel`s. After having defined the channels, we create the jobs asynchronously. On each available worker, we execute the `work` function using `remote_do`. In the final **`while`** loop, we collect all results.

```
let n = 10
    local jobs    = RemoteChannel(() -> Channel{NamedTuple}(5))
    local results = RemoteChannel(() -> Channel{NamedTuple}(5))

    @async make_jobs(jobs, n)

    for w in workers()
        remote_do(work, w, jobs, results)
    end

    @time for i in 1:n
        local r = take!(results)
        println("Job $(r.id) performed by worker $(r.worker) " *
                "took $(round(r.time; digits = 3)) seconds.")
    end
end
```

```
      From worker 11:   Worker 11 has no more jobs to do.
      From worker 9:    Worker 9 has no more jobs to do.
      From worker 4:    Worker 4 has no more jobs to do.
      From worker 19:   Worker 19 has no more jobs to do.
      From worker 23:   Worker 23 has no more jobs to do.
      From worker 25:   Worker 25 has no more jobs to do.
      From worker 6:    Worker 6 has no more jobs to do.
      From worker 14:   Worker 14 has no more jobs to do.
      From worker 21:   Worker 21 has no more jobs to do.
      From worker 24:   Worker 24 has no more jobs to do.
      From worker 15:   Worker 15 has no more jobs to do.
Job 1 performed by worker 2 took 0.052 seconds.
      From worker 2:    Worker 2 has no more jobs to do.
Job 4 performed by worker 8 took 0.045 seconds.
      From worker 8:    Worker 8 has no more jobs to do.
Job 7 performed by worker 17 took 0.069 seconds.
      From worker 17:   Worker 17 has no more jobs to do.
Job 6 performed by worker 12 took 0.137 seconds.
      From worker 12:   Worker 12 has no more jobs to do.
Job 5 performed by worker 22 took 0.312 seconds.
      From worker 22:   Worker 22 has no more jobs to do.
Job 2 performed by worker 3 took 0.338 seconds.
      From worker 3:    Worker 3 has no more jobs to do.
Job 10 performed by worker 16 took 0.628 seconds.
      From worker 16:   Worker 16 has no more jobs to do.
Job 3 performed by worker 10 took 0.781 seconds.
      From worker 10:   Worker 10 has no more jobs to do.
Job 9 performed by worker 18 took 0.809 seconds.
      From worker 18:   Worker 18 has no more jobs to do.
Job 8 performed by worker 5 took 0.969 seconds.
  0.988970 seconds (9.36 k allocations: 426.000 KiB)
      From worker 5:     Worker 5 has no more jobs to do.
```

In this particular example, there are fewer jobs than worker processes. Therefore some of the `work` functions finish immediately. All jobs are finished after about the time it takes the longest job to finish.

The operations for parallel or distributed computing are summarized in Table 6.5. `SharedArrays` are available in the module `SharedArrays`, which must be loaded on all workers, and make it possible for multiple processes to access an entire array.

Table 6.5 Operations for parallel computing.

Function	Description
`workers()`	return an array of all worker process IDs
`addprocs()`	add `Sys.CPU_THREADS` workers
`rmprocs(`*ids*`...)`	remove the workers with the given process IDs
`myid`	return the ID of the current process
`SharedArray`	create an array that can be accessed by multiple workers
`RemoteChannel`	create a channel for communication between processes
`remote_do(`*f*`,` *id*`,` *args*`...)`	evaluate a function asynchronously
`remotecall(`*f*`,` *id*`,` *args*`...)`	evaluate a function asynchronously and return a `Future`
`fetch(`*future*`)`	wait for and return the value of *future*
`remotecall_fetch`	equivalent to `fetch(remotecall(`...`)`, but faster
`remotecall_wait`	equivalent to `wait(remotecall(`...`)`, but faster
`@async` *expression*	wrap *expression* in a **`Task`** and schedule it locally
`@sync` *expression*	wait until all uses of `@async` etc. have completed
`@spawnat` *id expr*	evaluate *expr* asynchronously on process *id* (`:any`)
`@distributed` [*f*] **`for`** ... **`end`**	distributed, parallel version of a **`for`** loop
`@fetchfrom` *id expr*	equivalent to `fetch(@spawnat` *p expr*`)`
`clear!(`*symbols*`,` *ids*`)`	clear global bindings for *symbols* on processes *ids*
`finalize(`*object*`)`	finalize *object*, making it available for garbage collection

We close this section with a comment on garbage collection. JULIA performs garbage collection on local and remote processes as usual so that you are not required to care about allocating and freeing memory. There is one effect, however, that may require you to be more careful. As usual in garbage collection, the time when an object is garbage collected is unspecified and depends on the size of the object and the current memory pressure. Remote garbage collection requires the use of remote references, which are very small, and hence there is little pressure to collect them (often). However, the objects the remote references point to may be quite large and cannot be collected while the remote references exist. To help the garbage collector in such cases, one can explicitly call `finalize` on local instances of a `RemoteChannel` and on unfetched `Futures` that are not needed anymore.

6.7.3 Parallel Loops and Parallel Mapping

Fortunately, many parallel computations can be implemented using parallel **`for`** loops or parallel mapping of functions. When using these built-in facilities, the programmer is not involved in the low-level tasks of moving data and managing worker processes.

We consider a simple mathematical example that can be parallelized in a straightforward manner, namely the approximation of π using random numbers. The area of the circle sector of the unit circle with radius one around the origin within the square $[0,1] \times [0,1]$ is $\pi/4$. If we draw uniformly distributed random numbers X and Y from the interval $[0,1]$, then the fraction of these pairs (X,Y) within this circle sector (i.e., with $X^2 + Y^2 \leq 1$) will thus be $\pi/4$ of the number of all pairs (X,Y) drawn. Hence we have found an algorithm that calculates a Monte Carlo approximation of $\pi/4$.

In the first step, we save the following function `pi` in a file called `pi.jl`.

```
function pi(n::Int)::Float64
    local counter::Int = 0
    for i in 1:n
        if rand()^2 + rand()^2 <= 1
            counter = counter + 1
        end
    end
    4 * counter / n
end
```

To run `pi` on multiple processes (assuming your computer comes with multiple (logical) cores), we start JULIA using `julia -p auto`. Then we use the `@everywhere` macro to make our program available to all workers.

```
julia> @everywhere include("pi.jl")
```

Since we started JULIA with the `-p` command-line option, the `Distributed` module (and hence the macro `Distributed.@spawnat`) is already available; otherwise we would need **`using`** `Distributed`.

The `@spawnat` macro takes two arguments, namely the ID of the process to be used and an expression. The expression is wrapped into a closure and run asynchronously on the specified process, and a `Future` is returned. If the process ID is equal to `:any`, the scheduler picks the process to be used.

```
julia> pi1 = @spawnat 2 pi(1000)
Future(2, 1, 74, nothing)
julia> pi2 = @spawnat :any pi(1000)
Future(2, 1, 75, nothing)
julia> (fetch(pi1) + fetch(pi2)) / 2
3.144
```

Here we have used `fetch` to obtain the return value of the spawned function from the `Future`. The average of the two approximations yields three correct digits of π (at least in this run). This approach is still low level, as some programming work is required to spawn the expressions and to collect their values.

Parallel **`for`** loops provide a convenient way to distribute expressions to processes and to collect the results.

```
function parallel_pi(m::Int, n::Int)::Float64
    (1/m) * @distributed (+) for i in 1:m
        pi(n)
    end
end
```

The `@distributed` macro turns the **for** loop into a parallel **for** loop. Its first (optional) argument is a function that will process the values returned by each iteration; here the argument is supplied as `(+)` (instead of just +) for a syntactic reason. All iterations are performed on the worker processes, and each iteration returns the value of its last expression. The postprocessing is performed on the calling process.

This is all that is needed to run this Monte Carlo algorithm in parallel.

```
julia> @time parallel_pi(length(workers()), 10^9)
 11.515523 seconds (62.33 k allocations: 3.142 MiB)
3.1415952925000004
```

In this run, we obtained six correct decimal digits of π by running $24 \cdot 10^9$ samples in total. By using `length(workers())` as the first argument, this number of iterations is distributed to the same number of workers. If the argument is `10 * length(workers())`, each worker receives ten loop iterations.

When one is interested in receiving the values calculated in all loop iterations, the `vcat` function can be used to return a vector with all values.

```
julia> @distributed vcat for i in 1:length(workers()) pi(10^9) end
24-element Vector{Float64}:
 3.141593724
 ...
```

All variables used inside a parallel **for** loop will be copied to each worker process. On the other hand, any changes to these variables will *not* be visible after the loop has finished, i.e., the values of the variables are *not* copied back. It is also important to note that the order of the iterations is unspecified.

It is possible to omit the function that processes the values when calling `@distributed`. Then the loop iterations are spawned on all available workers and an array of `Futures` is immediately returned without waiting for the iterations to finish. The functions `wait` and `fetch` can be applied to the `Futures` as usual, or it is possible to wait for the completion of all iterations by calling `@sync` on the result, i.e., by writing

```
@sync @distributed for
    ...
end.
```

Using a parallel **for** loop with `vcat` as the postprocessing function is equivalent to using the `pmap` function, which is the parallel version of the mapping function `map`. Using `pmap`, we can rewrite `parallel_pi` above succinctly as follows.

```
function parallel_pi(m::Int, n::Int)::Float64
    (1/m) * sum(pmap(pi, repeat([n], m)))
end
```

The results are the same.

```
julia> @time parallel_pi(length(workers()), 10^9)
 11.433885 seconds (1.77 k allocations: 68.109 KiB)
3.141599547166666
```

What is the difference between a **for** loop and pmap? The pmap function is meant to be used when evaluating the function is computationally expensive. On the other hand, a parallel **for** loop can handle tiny computations in each iteration well.

Problems

6.1 (Fizz-buzz)

Write a function that prints the numbers from 1 to 100. However, for multiples of three, print "fizz" instead of the number; for multiples of five, print "buzz"; and for multiples of both three and five, print "fizz-buzz".

6.2 Define a data structure for a mathematical object as well as corresponding iterate methods. Show two examples of iteration, one using a **for** loop and one using a **while** loop.

6.3 (Sieve of Eratosthenes) Implement the sieve of Eratosthenes for calculating prime numbers.

6.4 Write a parallel program to calculate $f(n) := \sum_{k=1}^{n} 1/k^2$ and determine the speed-up of your program compared to a serial version as a function of the number of workers. Hint: you can check your solution using the formula $\sum_{k=1}^{\infty} 1/k^2 = \pi^2/6$.

6.5 (How to shoot π, continued)

Compare the speed of both versions of the Monte Carlo approximation of π in Sect. 6.7.3 for different m and n. Which version and parameters yield the fastest program?

Chapter 7
Macros

LISP is now the second oldest programming language in present widespread use (after FORTRAN and not counting APT, which isn't used for programming per se). It owes its longevity to two facts. First, its core occupies some kind of local optimum in the space of programming languages given that static friction discourages purely notational changes. Recursive use of conditional expressions, representation of symbolic information externally by lists and internally by list structure, and representation of program in the same way will probably have a very long life.

Second, LISP still has operational features unmatched by other language that make it a convenient vehicle for higher level systems for symbolic computation and for artificial intelligence. These include its run-time system that give good access to the features of the host machine and its operating system, its list structure internal language that makes it a good target for compiling from yet higher level languages, its compatibility with systems that produce binary or assembly level program, and the availability of its interpreter as a command language for driving other programs. (One can even conjecture that LISP owes its survival specifically to the fact that its programs are lists, which everyone, including me, has regarded as a disadvantage. Proposed replacements for LISP [...] abandoned this feature in favor of an Algol-like syntax leaving no target language for higher level systems).

LISP will become obsolete when someone makes a more comprehensive language that dominates LISP practically and also gives a clear mathematical semantics to a more comprehensive set of features.

—John McCarthy, *History of Lisp* (12 February 1979)

Abstract For several decades, Lisp macros have been the state of the art in metaprogramming. Macros are expanded at the time when a program is read, and thus provide a mechanism for defining new language constructs by rewriting expressions at read time and before compile and evaluation time. In this chapter, the concept of macros is explained via the example of macros in COMMON LISP, which is conducive for this purpose due to its uniform syntax. Then JULIA macros and their building blocks are presented in detail. Finally, useful built-in JULIA macros are discussed.

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_7

7.1 Introduction

The lifetime of a program consists of several phases.

1. The first phase is the time when the program is written.
2. The second phase is the read time, when the program is read and translated into the internal data structure the interpreter or compiler uses to perform its task. In the LISP family of languages, the internal representation is a list, and in JULIA, which has more syntax, it is an expression (**`Expr`**). It is a noteworthy feature of these two languages and related ones that the internal representation of the program is exposed to the programmer, much facilitating the definition of macros.
   At read time, any expression that is a macro is expanded and yields a new expression that replaces the macro.
3. At compile time, the expressions are compiled.
4. Finally, at evaluation or run time, the expressions are evaluated and their values are returned.

In LISP and also in JULIA, there are three categories of function-like expressions: macros, special forms, and functions. The differences between these three categories are due to their different purposes.

- The arguments of macros are not evaluated, because the whole purpose of a macro is to translate an expression into a new expression.
- Special forms only evaluate some of their arguments. The canonical example of a special form is the **`if`** expression, which evaluates its first argument and depending on the result its second *or* third argument.
- Functions evaluate all of their arguments.

The discussion of the phases in the lifetime of a program shows that macros are expanded after read time and before any evaluations of the expressions are performed. Therefore macros make it possible to define new language constructs that are syntactically indistinguishable from built-in language constructs in the case of LISP and nearly syntactically indistinguishable in the case of JULIA. In JULIA, the names of macros always start with the at sign `@`.

In the next section, we illustrate the concept of macros via a simple example implemented in COMMON LISP, because its syntax is so uniform that the concept of a macro can be illustrated in this language very easily. Afterwards, we proceed with macros in JULIA, which has more syntactic constructs.

7.2 Macros in COMMON LISP

The simple example we consider is a macro called `do-thrice` that takes an expression and executes it three times. We start gently in COMMON LISP .

```
(in-package :cl-user)
```

The following expression prints a message three times using a `dotimes` loop.

```
(dotimes (i 3)
  (print "Hello, world!"))
```

This expression is a list that contains the three elements `dotimes`, `(i 3)`, and `(print "Hello, world!")`. The first element is the name of the function, macro, or special form to be called. (In fact, `dotimes` is a macro, but it is indistinguishable from a function if all we know is its name.) The second element `(i 3)` defines the iteration variable `i` and specifies how many times the following expression will repeated. The third element is the expression to be repeated.

We are already half way to defining the macro `do-thrice`. The macro `defmacro` defines a new macro. Its first argument is the name of the macro to be defined, its second argument is the argument list, and the remaining arguments are the expressions to be returned by the new macro. Therefore the first version of our simple macro is the following.

```
(defmacro do-thrice-1 (&body body)
  `(dotimes (i 3)
     ,@body))
```

Here the argument list just means that all expressions that will be passed to the new macro will be contained in the local variable `body`. The backquote ` is commonly used in macro definitions to protect its argument from evaluation, just as **`quote`** in JULIA. The syntax `,@` within a backquote means that the elements of its argument, here `body`, are spliced into the surrounding list. If we would not like to use the backquote syntax, we could also construct the expression, i.e., a list, explicitly, but the purpose of the backquote syntax is to facilitate writing macros and therefore we use it.

COMMON LISP offers a simple way to check that our macro does what it is supposed to do: `macroexpand-1` expands a macro only once.

```
(macroexpand-1 '(do-thrice-1 (print "Hello, Julia users!")))
```

This results in the following output.

```
(DOTIMES (I 3) (PRINT "Hello, Julia users!"))
T
```

Symbols are printed in uppercase letters by default. The value `T` is the second return value; it stands for true. We see that the macro `do-thrice-1` does what we intend it to do: it takes an expression and puts it inside a `dotimes` loop.

You might expect that there is also a function called `macroexpand` and you would be right. The function `macroexpand` expands a macro including all nested macro calls. Evaluating the following expression also expands the call of the `dotimes` macro, but the final expression is implementation dependent.

```
(macroexpand '(do-thrice-1 (print "Hello, Julia users!")))
```

Next, we call our macro, which means that the resulting macro expansion is evaluated.

```
(do-thrice-1 (print "Hello, Julia users!"))
```

Three lines are printed as expected. Here `NIL` is the return value.

```
"Hello, Julia users!"
"Hello, Julia users!"
"Hello, Julia users!"
NIL
```

However, there is a problem. Our use of `dotimes` defines the local variable `i` in the macro expansion, and we can access its value.

```
(do-thrice-1 (print i))
```

Evaluating this expression prints the following output.

```
0
1
2
NIL
```

Printing the value is relatively harmless, but we can also – a bit more maliciously – change the value of the iteration variable and hence change the behavior of the macro. (`setq` is short for "set quoted".)

```
(do-thrice-1 (setq i 2) (print "Printed only
                                once."))
```

Now only one line is printed.

```
"Printed only once."
NIL
```

The macro expansion shows why the message is printed once.

```
(macroexpand-1 '(do-thrice-1 (setq i 2) (print "Printed only
                                                once.")))

(DOTIMES (I 3) (SETQ I 2) (PRINT "Printed only once."))
T
```

In the first iteration of the `dotimes` loop, the iteration variable is increased, which prevents any further iterations, and the `print` expression is evaluated.

Such macros are called unhygienic, since the pollute the name space of variables. A hygienic version of the macro is the second version shown here.

```
(defmacro do-thrice-2 (&body body)
  (let ((i (gensym)))
    `(dotimes (,i 3)
       ,@body)))
```

The `let` form assigns a new unique symbol returned by `gensym` to the local variable `i` when the macro is expanded. The name of this new unique symbol is then used as the name of the iteration variable in the `dotimes` loop by splicing it as `,i` into the expression returned by the macro, preventing any unwanted variable capture.

```
(macroexpand-1 '(do-thrice-2 (print "Hello, Julia users!")))

(DOTIMES (#:G456 3) (PRINT "Hello, Julia users!"))
T
```

The macro expansion shows that a variable called `G456` is used as the iteration variable. (More precisely, `#:G456` is an uninterned symbol.)

Finally, we try to change the iteration variable `i` again.

```
(do-thrice-2 (setq i 2) (print "Hello, Julia users!"))

; in: DO-THRICE-2 (SETQ I 2)
;     (SETQ I 2)
;
; caught WARNING:
;   undefined variable: COMMON-LISP-USER::I
;
; compilation unit finished
;   Undefined variable:
;     I
;   caught 1 WARNING condition

"Hello, Julia users!"
"Hello, Julia users!"
"Hello, Julia users!"
NIL
```

Now we only receive a warning that a variable `i` was not declared or defined previously, while the macro still works as intended, printing three strings.

This first example illustrates how expressions can be rewritten and what hygienic macros are. We will encounter the same concepts in JULIA, where only the names are different.

7.3 Macro Definition

In JULIA, macros are defined by **`macro`** analogous to **`function`**. A macro must return an expression, which is easily achieved by wrapping an expression between **`quote`** and **`end`**. Within such a quoted expression, the value of a variable can be substituted by prepending its name with a dollar sign `$`. This syntax is analogous to the syntax for string interpolation (see Sect. 4.2.2). Hence the backquote in Common Lisp corresponds to **quote** in Julia, and the comma corresponds to the dollar sign.

The expansion of a macro is returned by the function `macroexpand` and the macros `@macroexpand` and `@macroexpand1`, which behave slightly differently. The function `macroexpand` takes the module in whose context the macro is expanded as the first argument. The keyword argument `recursive` controls whether it should be expanded recursively or not. The macros `@macroexpand` and `@macroexpand1` both do not take the module as an argument, and they differ in whether the macro is always expanded recursively (in the case of `@macroexpand`) or not (in the case of `@macroexpand1`).

To illustrate the intricacies of defining macros in Julia, we will write three versions of a macro that evaluates an expression a given number of times, just as the `dotimes` macro does in Common Lisp. The first version is the straightforward, unhygienic version.

```
macro unhygienic_dotimes(n::Integer, expr::Expr)
    @assert n >= 0

    quote
        let i = 0
            while i < 3
                $expr
                i += 1
            end
        end
    end
end
```

The assertion in the first line is evaluated at macro-expansion time, and the quoted expression is returned. The value of the local variable `expr` is substituted into the quoted expression because of the dollar sign in `$expr`.

Just like functions, macros are generic in Julia, which means that methods with the same name but with different argument signatures can be defined. The method that best matches the arguments of the function or macro call will be chosen.

Macros are called just like functions, but the arguments are not evaluated at macro-expansion time.

```
julia> @unhygienic_dotimes(3, println("Printed three times."))
Printed three times.
Printed three times.
Printed three times.
```

Whenever a macro takes only one argument, the parentheses around the arguments can be left out. Expressions passed as arguments to macros are created as usual, separating them by semicolons within parentheses or using **begin** and **end**.

The next call illustrates that we can change the value of the iteration variable within the expression passed as an argument, showing that the macro is unhygienic.

```
julia> @unhygienic_dotimes(3, begin i = 2; println("Printed once.")
                                                                 end)
Printed once.
```

It is very instructive to compare the expansions of the macros in this section using the function `macroexpand` or the macro `@macroexpand` in order to understand the details of macro expansion in JULIA (see Problem 7.1).

Although the local variables within a **quote** block always receive unique names in JULIA in a hygiene pass as is easily checked by inspecting the macro expansion, this is not enough to make the macro hygienic.

Analogous to COMMON LISP, we can use the function `gensym` to define a hygienic version of the macro. Now a new variable name is generated at macro-expansion time by `gensym` and used as the iteration variable in the quoted expression.

```
macro hygienic_dotimes(n::Integer, expr::Expr)
    @assert n >= 0

    local var = gensym()

    quote
        let $var = 0
            while $var < 3
                $expr
                $var += 1
            end
        end
    end
end
```

We check that the expression is indeed evaluated three times.

```
julia> @hygienic_dotimes(3, begin i = 2; println("Printed thrice.")
                                                                 end)
Printed thrice.
Printed thrice.
Printed thrice.
```

The mechanism built into JULIA to facilitate the definition of hygienic macros is called `esc`. It is best illustrated by a simple example. We first define a global variable and a local variable in the expression returned by the macro, which both have the same name `foo` but different values. The macro returns `foo` and `$(esc(foo))`.

```
global foo = 0

macro escaped()
    quote
        local foo = 1
        (foo, $(esc(foo)))
    end
end
```

After calling the macro, we observe that `foo` evaluates to `1` and the escaped variable evaluates to `0`. (When calling a macro with no arguments, no parentheses are required.)

```
julia> @escaped
(1, 0)
```

This result shows that `foo` refers to the local variable of the same name as expected, while `$(esc(foo))` escapes the **`quote`** block and refers to the global variable.

Next, we have a look at the macro expansion, which is very instructive. (Comments have been deleted.)

```
julia> @macroexpand @escaped
quote
    local var"#10#foo" = 1
    (var"#10#foo", 0)
end
```

The expansion shows JULIA's hygiene mechanism at work. All local variables are renamed to new unique names in order to prevent unintended variable capture. In escaped expressions, these substitutions are not performed, however. Therefore `$(esc(foo))` can escape the **`quote`** block, and the value `0` of the global variable called `foo` at macro-expansion time is used. This explains the output `(1, 0)`.

In summary, `esc` is only valid in expressions returned from a macro and prevents renaming embedded variables into hygienic variables generated by `gensym`.

Knowing `esc`, we return to the `dotimes` macro and present its idiomatic version in JULIA. (Using a **`for`** loop is possible as well, of course, but would not allow us to explain variable capture.)

```
macro escaped_dotimes(n::Integer, expr::Expr)
    @assert n >= 0

    quote
        let i = 0
            while i < 3
                $(esc(expr))
                i += 1
            end
        end
    end
end
```

Calling the macro shows that it works correctly.

```
julia> @escaped_dotimes(3, begin i = 2; println("Printed thrice.")
                                                              end)
Printed thrice.
Printed thrice.
Printed thrice.
```

Expanding the macro illustrates the substitution of variables in the quoted expression by `gensym` versions except for the escaped variables. (Comments have been deleted.)

```
julia> @macroexpand @escaped_dotimes(3,
                                     begin
                                         i = 2
                                         println("Printed thrice.")
                                     end)
quote
    let var"#13#i" = 0
        while var"#13#i" < 3
            begin
                i = 2
                println("Printed thrice.")
            end
            var"#13#i" += 1
        end
    end
end
```

This is all there is to defining macros in JULIA.

7.4 Two Examples: Repeating and Collecting

In this section, we discuss two more examples of macro definitions. The first example is called `@repeat`. Its purpose is to take an expression and a condition and to repeat the expression until the condition is satisfied just as repeat statements in other programming languages. Since JULIA usually has more syntactic sugar than LISP, we require the second argument of our `@repeat` macro to be the symbol `until`. This also allows us to illustrate that the corresponding check is evaluated when the macro is expanded. We substitute the values variables `expr` and `condition` into the right places in a **`while`** loop, and we employ the escape mechanism to make the macro hygienic. Therefore our `@repeat` macro looks like this.

```
macro repeat(expr::Expr, until::Symbol, condition::Expr)
    if until != :until
        error("malformed call of @repeat")
    end

    quote
        while true
            $(esc(expr))
            if $(esc(condition))
                break
            end
        end
    end
end
```

Next we use the macro by defining a local variable to serve as an iteration counter. Evaluating the following expression returns an error, since the check at the beginning fails when the macro is expanded.

```
julia> let i = 0
           @repeat begin
               i += 1
               @show i
           end untill i >= 3
       end
ERROR: LoadError: malformed call of @repeat
```

It works as expected if we spell correctly.

```
julia> let i = 0
           @repeat begin
               i += 1
               @show i
           end until i >= 3
       end
i = 1
i = 2
i = 3
```

Again, it is constructive to inspect the macro expansion. When using the function `macroexpand`, we must specify the module (here `Main`) in whose context the macro is evaluated and quote the expression we want to expand. When using `@macroexpand` or `@macroexpand1`, the argument expression is not quoted. (Comments in the macro expansion have been deleted.)

```
julia> macroexpand(Main, quote
                                let i = 0
                                    @repeat begin
                                        i += 1
                                        @show i
                                    end until i >= 3
                                end
                            end)
quote
    let i = 0
        begin
            while true
                begin
                    i += 1
                    begin
                        Base.println("i = ", Base.repr(begin
                                    var"#1#value" = i
                                end))
                        var"#1#value"
                    end
                end
                if i >= 3
                    break
                end
            end
        end
    end
end
```

The second example in this section is called `@collect`. It wraps an expression, often a loop, in which `remember` can be used to collect values, which are returned by `@collect`. Such a feature is available in the `loop` macro in COMMON LISP and is useful, for example, when the number of values collected in each iteration of a loop is unknown beforehand. A simple example of its usage is the following.

```
julia> import Primes
julia> @collect for i in 1:10
           if Primes.isprime(i)
               remember(i)
           end
       end
4-element Vector{Any}:
 2
 3
 5
 7
```

The definition of `@collect` looks like this.

```
macro collect(expr::Expr)
    quote
        let v = Vector()
            function $(esc(:remember))(x)
                push!(v, x)
            end

            $(esc(expr))

            v
        end
    end
end
```

Note that the function `remember` is accessible only within the argument expression that is passed to `@collect`; it is not a globally defined function and does not pollute the global variable bindings.

It is instructive to macroexpand the example above once.

```
julia> @macroexpand1 @collect for i in 1:10
           if Primes.isprime(i)
               remember(i)
           end
       end
quote
    let var"#2#v" = Main.Vector()
        function remember(var"#4#x")
            Main.push!(var"#2#v", var"#4#x")
        end
        for i = 1:10
            if Primes.isprime(i)
                remember(i)
            end
        end
        var"#2#v"
    end
end
```

7.5 Memoization

In this section, we implement the optimization technique of memoization as a macro called `@memoize`. Memoization trades run time for memory by storing the results of (computationally expensive) function calls and returning the cached results whenever possible.

The `@memoize` macro is defined in such a way that it is straightforward to use. In order to memoize a function, we only have to write `@memoize` in front of its definition; in other words, the only argument of the `@memoize` macro is a function definition (which is an expression).

```
macro memoize(fun::Expr)
    local call = fun.args[1]
    local name = call.args[1].args[1]
    local arg1 = call.args[1].args[2]
    local arg1_name = arg1.args[1]
    local arg1_type = arg1.args[2]
    local return_type = call.args[2]
    local body = fun.args[2]

    quote
        let cache = Dict{$(esc(arg1_type)), $(esc(return_type))}()
            global function $(esc(name))($(esc(arg1_name))::
                                $(esc(arg1_type)))::$(esc(return_type))
                if haskey(cache, $(esc(arg1_name)))
                    cache[$(esc(arg1_name))]
                else
                    cache[$(esc(arg1_name))] = $(esc(body))
                end
            end
        end
    end
end
```

About half of the work that the macro performs is spent on parsing the function definition. We look for the name of the function, its first argument, the type of its first argument, the return type, and the body of the function. You can use `dump` to view all these expressions and make sense of the meaning of their parts.

The macro returns a **`quote`**d expression as usual. The definition of the memoized function is encapsulated within a closure (see Sect. 3.4) created by **`let`**. The `cache` variable contains a **`Dict`** with keys that have the type of the function argument and with values that have the type of the return value. Within this closure, the memoized function is defined. We have to write **`global`** before the **`function`** definition, because otherwise the function would only be defined locally within the closure and thus inaccessible and useless.

The function signature consists of the parsed function name, argument, argument type, and return type. The memoized function itself is simple. It checks whether the cache contains the argument of the memoized function as a key. If it does, the cached value is returned. If it does not, the `esc`aped `body` of the function is evaluated and the resulting value is stored in the cache. Since the **`if`** expression is the last expression in the function, one of these two values is returned.

The Fibonacci sequence serves as a good example. We have already seen in Sect. 2.1 that caching the return values is a key strategy to the fast calculation of Fibonacci numbers, but now we can fully automate this idea by just prepending the function definition with `@memoize`. The function definition we use here takes a **`BigInt`** and returns a **`BigInt`**.

```
@memoize function fib(n::BigInt)::BigInt
    if n <= 1
        n
    else
        fib(n-2) + fib(n-1)
    end
end
```

Of course, the cache is reset to an empty one whenever this definition is evaluated. It is very instructive to view the expansion of the macro using `@macroexpand1`, for example.

Just after defining the memoized function, we can easily observe the speedup by calculating the same Fibonacci number twice.

```
julia> @time fib(BigInt(10_000));
  0.007028 seconds (80.01 k allocations: 5.826 MiB)
julia> @time fib(BigInt(10_000));
  0.000004 seconds (2 allocations: 40 bytes)
```

Some extensions of this macro are the subject of Problems 7.6, 7.7, and 7.8.

7.6 Built-in Macros

Tables 7.1 and 7.2 summarize the built-in macros. Since macros are code transformations, some of the more advanced or extravagant features of the JULIA language can be found in these two tables, and some of them are explained in the following in more detail.

The first group of macros to be discussed in more detail are the ones whose names end in `_str`. These macros create string literals (already mentioned in Sect. 4.2.4), which are a mechanism to create objects from a textual representation. The part of the name before `_str` indicates the type of the object to be created. For example, the macros `@int128_str` and `@uint128_str` return an **`Int128`** and an **`UInt128`**, respectively, and the `big_str` macro returns a **`BigFloat`** or **`BigInt`** depending on whether the string contains a decimal point or not. For example, `big"1.2"` returns a **`BigFloat`** and `big"1"` returns a **`BigInt`**, as is easily checked by `typeof(big"1.2")` and `typeof(big"1")`.

The usefulness of these macros is much increased by the syntactic rule that `@`*name*`_str "…"` is equivalent to *name*`"…"`. For example, `v"1.2.3"` returns the same `VersionNumber` object as `@v_str "1.2.3"`.

Table 7.1 Built-in macros: parsing, documentation, output, profiling, tasks, metaprogramming, and performance annotations.

Macro	Description
`@__DIR__`	directory of the file containing the macro call or the current working directory
`@__FILE__`	file containing the macro call or an empty string
`@__LINE__`	line number of the location of the macro call or `0`
`@__MODULE__`	module of the toplevel eval
`@cmd` *string*	generate a `Cmd` object from *string*
`@int128_str` *string*	parse *string* into an **`Int128`**
`@uint128_str` *string*	parse *string* into an **`UInt128`**
`@big_str` *string*	parse *string* into a **`BigInt`** or a **`BigFloat`**
`@b_str` *string*	create an immutable **`UInt8`** vector
`@r_str` *string*	create a **`Regex`** (regular expression)
`@s_str` *string*	create a substitution string for regular expressions
`@v_str` *string*	parse *string* into a `VersionNumber`
`@raw_str` *string*	create a raw string without interpolation and unescaping
`@MIME_str` *string*	parse *string* into a MIME type
`@text_str` *string*	parse *string* into a `Text` object
`@html_str` *string*	parse *string* into an HTML object
`@doc`	retrieve documentation for a function, macro, or other object
`@show` *expr*	print and return the expression *expr*
`@time` *expr*	return the value of *expr* after printing timing and allocation
`@timed` *expr*	return the value of *expr* together with allocation information
`@timev` *expr*	verbose version of the `@time` macro
`@elapsed` *expr*	return the number of seconds it took to evaluate *expr*
`@allocated` *expr*	return the total number of bytes allocated while evaluating *expr*
`@sync`	wait until all lexically enclosed **`Task`**s have completed
`@async`	wrap an expression in a **`Task`** and add it to the scheduler
`@task` *expr*	create a **`Task`** from *expr*
`@threadcall`	similar to **`ccall`**, but in a different thread
`@macroexpand` *expr*	fully (recursively) expand the macros in *expr*
`@macroexpand1` *expr*	expand *expr* non-recursively (only once)
`@generated`	annotate a function which will be generated
`@gensym`	generate a symbol for a variable
`@eval` [*mod*] *expr*	evaluate `expr` (in `Module` *mod* if given)
`@deprecate` *old new*	mark function as deprecated
`@boundscheck` *expr*	annotate the expression allowing it to be elided by `@inbounds`
`@inbounds` *expr*	eliminate checking of array bounds within *expr*
`@fastmath` *expr*	use fast math operations, strict IEEE semantics may be violated
`@simd` **`for`** ... **`end`**	annotate a **`for`** loop to allow more re-ordering
`@inline`	hint that the function is worth inlining
`@noinline`	prevent the compiler from inlining a function
`@nospecialize`	hint that the method should not be specialized for different types
`@specialize`	reset specialization hint for an argument back to the default
`@polly`	tell the compiler to apply the optimizer Polly to a function

Table 7.2 Built-in macros: errors etc., compiler, and miscellaneous macros.

Macro	Description
`@debug`	create a log record with a debug message
`@info`	create a log record with an informational message
`@warn`	create a log record with a warning message
`@error`	create a log record with an error message
`@logmsg`	general way to create a log record
`@code_llvm`, `@code_lowered`, `@code_native`, `@code_typed`, and `@code_warntype`	evaluate the arguments of the function or macro call, determine their types, and call the corresponding function on the resulting expression (see text for more explanations)
`@__dot__` *expr* and `@.` *expr*	convert every function call, operator, and assignment into a "dot call" (`f` into `f.` etc.)
`@assert` *cond*	throw an `AssertionError` if *cond* is **`false`**
`@cfunction`	generate a C-callable function pointer from a JULIA function
`@edit`	call the `edit` function
`@enum`	create an enum subtype
`@evalpoly`	evaluate a polynomial efficiently using Horner's method
`@functionloc`	return the location of a method definition
`@goto` *name*	unconditionally jump to the location denoted by `@label` *name*
`@label` *name*	label a destination for `@goto`
`@isdefined` *var*	tests whether a variable *var* is defined in the current scope
`@less`	shows source code (using `less`) for a function or macro call
`@static` *expr*	partially evaluate the expression *expr* at parse time
`@view` *A*`[...]`	create a **`SubArray`** from the indexing operation
`@views` *expr*	convert all array-slicing operations in *expr* to return a view
`@which`	return the `Method` that would be called for a given function or macro call with given arguments

If you are familiar with the finer points of COMMON LISP macros, you will have noticed that this mechanism plays the role of reader macros in COMMON LISP. The mechanism in JULIA that translates macro calls of the form *name*`"..."` to macro calls of the form `@`*name*`_str "..."` is general in the sense that you can define your own translations. This is useful when you want to construct objects from a textual representation, for example while reading constants in a program or while parsing data files. An example is the following. We first define a data structure and then a macro to convert strings into such a data structure.

```
struct Interval
    a::Float64
    b::Float64
end

macro i_str(s::String)
    local comma = findfirst(",", s)[1]
    local a = parse(Float64, s[1:comma-1])
    local b = parse(Float64, s[comma+1:end])
    @assert a <= b

    Interval(a, b)
end
```

Now we can create intervals easily from a straightforward string interpretation, while the input is checked as well.

```
julia> i"1.0, 2.0"
Interval(1.0, 2.0)
```

The group of macros `@time`, `@timed`, `@timev`, `@allocated`, and `@elapsed` return information about memory usage and evaluation time of an expression. The `@allocated` macro discards the resulting value and returns the total number of bytes allocated during evaluation. Analogously, the `@elapsed` macro discards the resulting value and returns the number of seconds the evaluation took. `@time` evaluates an expression and returns its value after printing the time it took to evaluate, the number of allocations, and the total number of bytes allocated. `@timed` returns multiple values (that can be used in the program instead of just being printed): the return value of the expression, the elapsed time, the total number of bytes allocated, the garbage collection time, and an object with various memory allocation counters. `@timev` is a more verbose version of `@time`.

The group of macros `@async`, `@sync`, and `@task` as well as `@distributed`, `@spawn`, and `@spawnat` are discussed in Sect. 6.6 and Sect. 6.7.

The next two macros we discuss in more detail are `@boundscheck` and `@inbounds`. The `@inbounds` macro skips range checks in its argument expression in order to improve performance when referencing array elements. The user must guarantee that all bounds checks after a call to `@inbounds` are satisfied. The canonical example of its usage is within a **`for`** loop when many array elements are referenced. One should be careful when using it; if an illegal array reference is made, incorrect results, corrupted memory, or program crashes may result.

The `@boundscheck` macro makes it possible to use `@inbounds` in your own functions, but you can use `@boundscheck` only within inlined functions. The `@boundscheck` macro marks the following expression as a bounds check, which is elided when the inlined function is called after `@inbounds`.

The next family of macros consists of `@code_llvm`, `@code_lowered`, `@code_native`, `@code_typed`, and `@code_warntype`. These five macros make it easy to watch the compiler at work and are useful when you want to optimize a function at the assembler level. The first macro, `@code_llvm`, shows the compiler output. It evaluates the arguments of a function or macro call, determines the types of the arguments, and calls the function `code_llvm` on the resulting expression. The `@code_llvm` macro also takes a few keyword arguments.

Maybe the simplest example is the following. What happens when we ask JULIA to evaluate `2+2`?

```
julia> @code_llvm 2+2
;  @ int.jl:87 within `+`
define i64 @"julia_+_117"(i64 signext %0, i64 signext %1) #0 {
top:
  %2 = add i64 %1, %0
  ret i64 %2
}
```

The output shows the assembler code for the method specialized for two arguments of type **Int64** (i64) for the generic function +. In assembler, the method for this particular argument signature consists of a call of add and a call of ret (return). This example shows that JULIA generates highly efficient code for known argument types.

It is often more interesting to disassemble your own function. We define a simple (generic) function first without specifying any types.

```
function times_two(x)
    2*x
end
```

Then we apply @code_llvm to a call of our function with an **Int64** argument. This means that the JULIA compiler generates code for a method specialized for this particular argument type.

```
julia> @code_llvm times_two(1)
;  @ REPL[2]:1 within `times_two`
define i64 @julia_times_two_137(i64 signext %0) #0 {
top:
;  @ REPL[2]:2 within `times_two`
; ┌ @ int.jl:88 within `*`
   %1 = shl i64 %0, 1
; └
  ret i64 %1
}
```

We see that the assembler code consists of two instructions for an **Int64** (i64) argument. The first is a call to shl (shift left), the fastest way to multiply an integer given in its binary representation by two. The second is a call to ret (return).

Next we apply @code_llvm to a call of our function with a **Float64** (double) argument.

```
julia> @code_llvm times_two(1.0)
;  @ REPL[2]:1 within `times_two`
define double @julia_times_two_139(double %0) #0 {
top:
;  @ REPL[2]:2 within `times_two`
; ┌ @ promotion.jl:380 within `*` @ float.jl:405
   %1 = fmul double %0, 2.000000e+00
; └
  ret double %1
}
```

The assembler code for this method consists again of two instructions, but the multiplication instruction is different now. The instruction `fmul` (floating-point multiplication) is applied to the argument and the constant **`Float64`** value `2.000000e+00`. The resulting value is returned by `ret`.

These two examples show that the generic function `x -> 2x` is compiled into a single assembly instruction in both cases and that the special instruction for the argument type is used. Therefore JULIA is capable of compiling programs into highly efficient code. The JULIA compiler also inlines functions automatically (see the macros `@inline` and `@noinline` below). The only drawback of generating specialized code for all argument signatures (and of inlining functions) is increased code size, which makes cache misses more likely, which slows down modern processors. But in summary, it is very unlikely that you will need to resort to lower-level languages than JULIA for performance reasons.

The next macro, `@code_lowered`, returns arrays of `CodeInfo` objects containing the lowered forms for the methods matching the given method and its type signature.

The third macro in this family, `@code_native`, is similar to `@code_llvm`, but instead of showing the instructions used by the LLVM compiler framework, the native instructions of the processor you are using are shown.

The next macro, `@code_typed`, is similar to `@code_lowered`, but shows type inferred information.

The last macro in this family, `@code_warntype`, prints lowered and type inferred abstract syntax trees for the given method and its type signature. The output is annotated in color (if available) to give warnings of potential type instabilities, i.e., variables whose types may change during evaluation are marked. These annotations may be related to operations for which the generated code is not optimal. This macro is especially useful for optimizing functions.

These five macros have sister functions: `code_llvm`, `code_lowered`, `code_native`, `code_typed`, and `code_warntype`.

The four macros `@debug`, `@info`, `@warn`, and `@error` are the recommended way to communicate debugging output, informational messages, warning messages, and errors to users of your program (see Sect. 6.5.3).

The `@doc` macro, already mentioned in Sect. 1.3.3, is highly useful at the REPL to retrieve documentation not only about built-in functions, macros, and types, about also about user defined ones if a documentation string was included.

The `@enum` macro makes it possible to define enumeration types.

The `@generated` macro, used before a function definition, defines so-called generated functions. Generated functions are a generalization of the multiple dispatch we know from generic functions. The body of a generated function has access only to the types of the arguments, but not to their values, and a generated function must return a quoted expression like a macro. They differ from macros, because generated functions are expanded after the types of the arguments are known, but before the function is compiled, while macros are expanded at read time and cannot access the types of their arguments. Generated functions are a seldom used feature.

Hints about inlining functions can be given to the compiler using the two macros `@inline` and `@noinline`. Inlining is a form of optimization that replaces calls of the function to be inlined with the code of the function itself within its caller. The advantage is that the overhead of passing and returning arguments is eliminated; on the other hand, the disadvantage is increased code size. Usually, only small and often called functions are inlined by the compiler. The macros `@inline` and `@noinline` are written just before **`function`**.

The `@less` macro is very useful to show the source code of a method. For example, `@less 2+2` shows the file `int.jl`, which is part of the implementation of JULIA.

The `@simd` macro annotates a **`for`** loop and allows the compiler to perform more loop reordering, although the compiler already is able to automatically vectorize inner **`for`** loops. The **`for`** loop must satisfy a few conditions when `@simd` is to be used. SIMD (single instruction multiple data) instructions are available on most modern processors; a SIMD instruction is executed in parallel on multiple data as opposed to executing multiple instructions.

The macros `@specialize` and `@nospecialize` make it possible to exert some control whether the compiler should generate code for methods with certain argument signatures or not. If `@nospecialize` appears in front of an argument in a method, it gives a hint to the compiler that the method should not be specialized for different types of this argument, thus avoiding excess code generation. The `@nospecialize` macro can also appear in the function body before any other code. Furthermore, it can be used without arguments in the local scope of a function and then applies to all arguments of the function. It can also be used without arguments in the global scope and then applies to all methods subsequently defined in the module. The `@specialize` macro resets the hint back to the default when used in the global scope.

The `@static` macro partially evaluates the following expression at read time. This is useful, for example, to define functions or values that are system specific. A simple example is the following.

```
@static if Sys.isapple() || Sys.islinux()
    function isunix()
        true
    end
end
```

A more interesting example is calling functions (for example using **`ccall`**) that only exist on certain systems.

The `@view` macro creates a **`SubArray`** from an array and an indexing expression into the array. A simple example of its usage is the following.

```
julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
julia> b = @view A[1, :]
2-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
julia> b[1] = 0; A
2×2 Matrix{Int64}:
 0  2
 3  4
```

The macro `@views` applies `@view` to every array indexing expression in the given expression.

7.7 Bibliographical Remarks

Homoiconicity and macros have been part of LISP [3] since its inception. The metaprogramming and macro features of COMMON LISP, the most modern and standardized [1] LISP dialect, are second to none and provided inspiration to the metaprogramming facilities in JULIA.

Problems

7.1 Use `@macroexpand` or `macroexpand` to expand all three versions of the `dotimes` macro in Sect. 7.3 and discuss the significance of all local variables.

7.2 Modify the `@collect` macro such that the user can specify the element type of the vector that is returned.

7.3 (Unless) Macros make it possible to define new control structures that are indistinguishable from built-in ones apart from the at sign. Write a macro called `@unless` which takes two arguments, namely a condition and an expression, and that evaluates the expression only if the condition is false.

7.4 (Anaphoric macro) Anaphoric macros [2, Chapter 14] are macros that deliberately capture an argument of the macro which can later be referred to by an anaphor. (In linguistics, an anaphor is the use of an expression whose interpretation depends on another (usually previous) expression.)

Write a macro called `@if_let` that takes four arguments, namely a **`Dict`**, a key (of arbitrary type), and two expressions. If the **`Dict`** contains the key, its value is bound to the local variable `it` within the first expression, which is evaluated; if the key cannot be found, the second expression is evaluated.

7.5 (Case) JULIA does not come with a `switch` expression that is common in other programming languages. Implement four macros modeled after their counterparts in COMMON LISP.

1. The macro `@case` takes a value and evaluates the clause that matches the value. A default clause may be given.
2. The macro `@ecase` is analogous to `@case`, but it takes no default clause and it raises an error if no clause matches.
3. The macro `@typecase` takes a value and evaluates the clause that matches the type of the given value. A default clause may be given.
4. The macro `@etypecase` is analogous to `@typecase`, but it takes no default clause and it raises an error if no clause matches.

7.6 (Memoization for unspecified types)

In the definition of the `@memoize` macro in Sect. 7.5, the argument and return types are parsed in order to be able to use a **`Dict`** for these types.

Generalize the `@memoize` macro and make it more robust (still for functions with a single argument) by checking whether the argument and return types are specified or not. If they are not, use the **`Any`** type instead of the argument or return type (or both) in the **`Dict`** that acts as the cache.

7.7 (Empty memoization cache)

Extend the `@memoize` macro such that it additionally defines a function called *name*`_empty_cache!` (when defining a memoized function called *name*) that empties the cache.

7.8 (Memoization for two arguments)

Generalize the `@memoize` macro to functions with two arguments.

7.9 (Quine) Write a quine in JULIA.

References

1. American National Standards Institute (ANSI), Washington, DC, USA: *Programming Language Common Lisp, ANSI INCITS 226-1994 (R2004)* (1994)
2. Graham, P.: *On Lisp.* Prentice Hall (1993)
3. McCarthy, J.: *LISP 1.5 Programmer's Manual.* The MIT Press (1962)

Chapter 8
Arrays and Linear Algebra

Should array indices start at 0 or 1?
My compromise of 0.5 was rejected
without, I thought, proper consideration.

—Stan Kelly-Bootle

Abstract Arrays are a multi-dimensional data structure and hence encompass the mathematical structures of vectors, matrices, and tensors. An important application of arrays is the numerical implementation of linear algebra. In certain applications, the majority of the entries of vectors or matrices are zero; in these situations, the sparse versions should be used instead of the dense ones. Operations on dense and sparse arrays are discussed in detail in this chapter, including those from linear algebra such as solving systems of linear equations, calculating the eigenvalues and eigenvectors of matrices, singular-value decomposition, and matrix factorizations.

8.1 Dense Arrays

8.1.1 Introduction

An array is a collection of objects called elements that can be referenced according to a rectilinear coordinate system. All elements have the same type, but the type of the elements may be arbitrary; in the most general case, the elements are of type `Any`. Usually, the type of the elements is more specific and better suited for efficient numerical calculations. When performing calculations known from linear algebra, the elements are usually floating-point numbers.

When arrays are passed as function arguments (see Sect. 2.2), they are passed by reference. This is due to performance reasons, as passing by reference is much

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_8

more efficient than passing by value, since the arrays passed as arguments do not have to be copied. The disadvantage is that any modifications made by the function called persist and are then seen by the caller.

It is important when designing programs that one keeps these advantages and disadvantages in mind. Destructively modifying arrays often results in large performance gains at the expensive of programs whose control and data flow is harder to understand. The convention that the names of functions that destructively modify any of their arguments end in an exclamation mark `!` is particularly useful in this context. On the other hand, functions that never modify their (array) arguments make it easier to reason about the data flow, but are generally less efficient when large data structures must be copied.

8.1.2 Construction, Initialization, and Concatenation

The basic syntax to construct arrays are square brackets. The type of the array elements may be specified before the opening square bracket; if it is not, it is inferred from the elements given.

```
julia> [1]
1-element Vector{Int64}:
 1
julia> Int8[1]
1-element Vector{Int8}:
 1
julia> [1.0]
1-element Vector{Float64}:
 1.0
julia> [1, 2.0]
2-element Vector{Float64}:
 1.0
 2.0
julia> Float32[1]
1-element Vector{Float32}:
 1.0
```

Furthermore, within the square brackets, the elements may be separated by spaces, commas, or semicolons. These separators have different meanings and result in different types of arrays being constructed.

In mathematics, there is a difference between vectors (elements of $\mathbb{R}^n$), column vectors (elements of $\mathbb{R}^{n\times 1}$), and row vectors (elements of $\mathbb{R}^{1\times n}$). Mathematical vectors are implemented as **`Vector`**s, i.e., one-dimensional **`Array`**s. Mathematical column and row vectors are two-dimensional **`Array`**s, whose second or first dimension has length one.

The following examples illustrate the cases that may occur. One-dimensional **Array**s, i.e., **Vector**s, are constructed if the elements are separated by commas only or by semicolons only.

```
julia> [1, 2]
2-element Vector{Int64}:
 1
 2
julia> isa([1, 2], Vector)
true
julia> [1; 2]
2-element Vector{Int64}:
 1
 2
julia> isa([1; 2], Vector)
true
```

If the elements are separated by spaces only, a row vector is constructed.

```
julia> [1 2]
1×2 Matrix{Int64}:
 1  2
```

If the elements are separated by spaces and semicolons, then a two-dimensional array is constructed, whereby the semicolons separate the rows.

```
julia> [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4
```

Vectors, i.e., one-dimensional **Array**s, are treated as column vectors whenever an implicit conversion is expedient. The first example is taking the conjugate transpose (adjoint) of a vector, which yields a row vector.

```
julia> adjoint([1, 2])
1×2 adjoint(::Vector{Int64}) with eltype Int64:
 1  2
```

The conjugate transpose can be written more conveniently using the postfix operator `'`.

```
julia> [1im, 2im]'
1×2 adjoint(::Vector{Complex{Int64}}) with eltype Complex{Int64}:
 0-1im  0-2im
julia> [1im, 2im]' * [1im, 2im]
5 + 0im
```

The conjugate transpose of the 1×2 array `[1im 2im]`, i.e., a row vector, is a 2×1 array, i.e., a column vector, as expected.

```
julia> [1im 2im]'
2×1 adjoint(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 0 - 1im
 0 - 2im
```

The usefulness of treating one-dimensional **`Array`**s, i.e., **`Vector`**s as column vectors is also seen in the second example, matrix-vector multiplication. Recall that `[1, 2]` and `[1; 2]` are **`Vector`**s.

```
julia> [1 2; 3 4] * [1, 2]
2-element Vector{Int64}:
  5
 11
julia> [1 2; 3 4] * [1; 2]
2-element Vector{Int64}:
  5
 11
```

On the other hand, trying to multiply the matrix by the 1×2 array `[1 2]` yields an error as expected.

In summary, the vector and matrix operations in JULIA follow the conventions of linear algebra, and one-dimensional arrays of length n are interpreted as column vectors of size $n \times 1$ for convenience.

Basic operations to query multi-dimensional arrays about their properties are summarized in Table 8.1.

Table 8.1 Basic operations on arrays.

Function	Description
`eltype(`A`)`	return the type of the elements of A
`length(`A`)`	return the number of elements in A
`ndims(`A`)`	return the number of dimensions of A
`size(`A`)`	return a tuple with all the dimensions of A
`size(`A`, `n`)`	return the size of A along dimension n
`eachindex(`A`)`	return an iterator for iterating over each element of A
`stride(`A`, `n`)`	return the number of elements (in memory) between adjacent elements along dimension n
`strides(`A`)`	return a tuple with the strides along all dimensions of A

In addition to the syntax using square brackets to construct vectors and arrays, there is a number of functions to construct, initialize, and fill multi-dimensional arrays. Table 8.2 provides an overview of the functions to construct and initialize arrays.

The function `reshape` is useful to interpret the elements of a given array as an array of a different shape, i.e., with different dimensions or with a different number of dimensions. The elements of the underlying array are not changed by

Table 8.2 Operations for constructing and initializing dense arrays.

Function	Description
`Array{`*type*`}(undef,` *dims*`...)`	construct an uninitialized *n*-dimensional array containing elements of *type*
`Matrix(I,` *n*`,` *m*`)`	construct a matrix with ones in the main diagonal
`zeros([`*type*`,]` *dims*`...)`	construct an array of zeros of *type* (default `Float64`)
`ones(`*type*`,` *dims*`...)`	construct an array of ones of *type* (default `Float64`)
`falses(`*A*`)`	construct a `BitArray` containing only `false`
`trues(`*A*`)`	construct a `BitArray` containing only `true`
`copy(`*A*`)`	create a shallow copy of *A*
`deepcopy(`*A*`)`	create a deep copy of *A*
`similar(`*A*`[,` *type*`][,` *dims*`])`	construct an uninitialized array similar to *A* with the specified element *type* and dimensions *dims*
`reshape(`*A*`,` *dims*`)` and `reshape(`*A*`,` *dims*`...)`	return the elements of *A* reshaped into the dimensions *dims*
`reinterpret(`*type*`,` *A*`)`	return an array with the same binary data as *A*, but with element type *type*
`fill(`*a*`,` *dims*`)` and `fill(`*a*`,` *dims*`...)`	construct an array with dimensions *dims* filled with the values *a*
`fill!(`*A*`,` *a*`)`	destructively fill the array *A* with the value *a*
`rand([`*type*`,]` *dims*`)` and `rand([`*type*`,]` *dims*`...)`	return an array with i.i.d., uniformly (in $[0,1)$) distributed random values of *type* (default `Float64`)
`randn([`*type*`,]` *dims*`)` and `randn([`*type*`,]` *dims*`...)`	return an array with i.i.d., standard normally distributed random values of *type* (default `Float64`)

`reshape` itself; however, the two arrays share the same elements, so that changing the elements of one array also affects the elements of the other. The new dimensions are specified as a tuple, whereby one dimension may be specified as `:` indicating that this dimension should be calculated to match the total number of the elements. In this example, we define a magic square.

```
julia> AD = [16, 5, 9, 4, 3, 10, 6, 15, 2, 11, 7, 14, 13, 8, 12, 1];
julia> M = reshape(AD, (4, :))
4×4 Matrix{Int64}:
 16   3   2  13
  5  10  11   8
  9   6   7  12
  4  15  14   1
julia> AD[1] = -16; sum(M)/4
26.0
julia> AD[1] = 16; sum(M)/4
34.0
```

The functions and the syntax for the concatenation of arrays are summarized in Table 8.3. The syntactic expressions in the second column are just a convenient way to call the functions in the first column.

Table 8.3 Operations for concatenating arrays.

Function	Syntax	Description
`cat(`*A*`...; dims=`*dims*`)`		concatenate the arrays along the dimensions *dims*
`vcat(`*A*`...)`	`[a; b; c; ...]`	vertical concatenation (along the first dimension)
`hcat(`*A*`...)`	`[a b c ...]`	horizontal concatenation (along the second dimension)
`hvcat(`*rows*`, `*elems*`...)`	`[a b; c d; ...]`	horizontal and vertical concatenation

8.1.3 Comprehensions and Generator Expressions

Another way to construct arrays are comprehensions (cf. Sect. 3.5), whose syntax

type`[`*expr* **`for`** *var1* **`in`** *iterable1*`,` *var2* **`in`** *iterable2*`,` ...`]`

is similar to the array syntax and to mathematical set notation. The dots indicate an arbitrary number of iteration variables. The values of the iteration variables may be given by any iterable collection (see Sect. 4.5.2) such as ranges or vectors. The resulting array has the dimensions given by the dimensions of the collections specifying the iteration variables in order and the elements are found by evaluating the expression *expr*, which often depends on the iteration variables. Specifying the element type *type* of the resulting array by prepending it to the array comprehension is optional. If the element type is not specified, it is determined automatically.

```
julia> sum([M[i, i] for i in 1:4])
34
julia> sum([M[5-i, i] for i in 1:4])
34
julia> sum([M[i, j] for i in 1:2, j in 1:2])
34
```

(This particular magic square has even more remarkable properties.)

Comprehensions are a convenient way to construct arrays with elements computed in a non-trivial manner. However, in situations when not the constructed array itself, but further computations on the array are of interest, generator expressions are advantageous, since they do not require the allocation of the intermediate array. Generator expressions are syntactically just comprehensions without the square brackets, although it is sometimes necessary to enclose the generator expression in parentheses in order to avoid syntactic ambiguity. Since all comma separated expressions after the **`for`** keyword are interpreted as iteration variables, parentheses are sometimes required to distinguish the iteration variables from other arguments.

```
julia> (i for i in 1:10)
Base.Generator{UnitRange{Int64}, typeof(identity)}(identity, 1:10)
```

The following example shows how a generator expression is used inside `sum`. Changing the number of terms in the sum does not change the amount of memory allocated when using a generator.

```
julia> @time sqrt(6*sum(1/i^2 for i in 1:100_000_000)) - pi
  0.139757 seconds (97.91 k allocations: 5.038 MiB)
-8.607403234606181e-9
```

On the other hand, the amount of allocated memory grows linearly with the number of iterations when using a comprehension.

```
julia> @time sqrt(6*sum([1/i^2 for i in 1:100_000_000])) - pi
 0.300991 seconds (124.37 k allocations: 769.142 MiB, 3.04% gc time)
-9.549294688326881e-9
```

The iterations both in comprehensions and in generator expressions can be nested, i.e., the collections to be iterated over can depend on previous ones. If this is the case, the result is always a one-dimensional **`Array`**.

```
julia> [10i + j for i in 1:3 for j in 1:i]
6-element Vector{Int64}:
 11
 21
 22
 31
 32
 33
```

Another extension that is sometimes useful is filtering by using the **`if`** keyword. The expression is collected into the one-dimensional output array only if the condition after the **`if`** keyword is true.

```
julia> [10i + j for i in 1:3 for j in 1:i if mod(i + j, 2) == 0]
4-element Vector{Int64}:
 11
 22
 31
 33
```

8.1.4 Indexing and Assignment

There are various ways to retrieve a certain element or certain elements from an array by indexing or to assign elements of an array by indexing. One indexing syntax is to supply n indices in square brackets after an n-dimensional array. Another option is to index a (multi-dimensional) array by a single index, which is then interpreted as a linear index. Each index may be

- a positive integer,
- a range of the form *from*`:`*to* or *from*`:`*step*`:`*to*,
- a colon `:`, which is the same as `Colon()`, to select the whole dimension,
- an array of positive integers including the empty array `[]`, or
- an array of **`Bool`**s.

The indexing syntax denotes an array or a single element of an array if all the indices are scalar integers. As part of the indexing syntax, the last valid index of each dimension can be specified by the keyword **`end`**. Hence, a colon is equivalent to `1:`**`end`**, as the first index is always `1`.

If any of the indices `I`j, $j \in \{1, \dots, n\}$, is not a scalar integer, but an array, then `B = A[I1, I2, ...]` becomes an array. The dimensions of the resulting array `B` are given by the dimensions of the indices `I`j. Suppose that the index `I`j is a d_j-dimensional array and drop the empty, 0-dimensional arrays that correspond to the scalar indices. Then the dimensions of `B` are `size(I1, 1)`, …, `size(I1,` d_1`)`, `size(I2, 1)`, …, `size(I2,` d_2`)`, …, `size(I`n`, 1)`, …, `size(I`n`,` d_n`)`. The resulting element

`B[i11, ..., i1`d_1`, i21, ..., i2`d_2`, ..., i`n`1, ..., i`nd_n`]`

is the element

`A[I1[i11, ..., i1`d_1`], I2[i21, ..., i2`d_2`], ..., I`n`[i`n`1, ..., i`nd_n`]]`

of the original array `A`.

In this example, we extract the lower left subsquare from the magic square above.

```
julia> M[[3, 4], [1, 2]]
2×2 Matrix{Int64}:
 9   6
 4  15
julia> sum(ans)
34
```

We can extract the two innermost elements from the first and fourth row in this manner.

```
julia> M[[1, 4], [2, 3]]
2×2 Matrix{Int64}:
  3   2
 15  14
julia> sum(ans)
34
```

The shape of the resulting array is determined by the shape of the indices.

```
julia> M[3, [2 3; 1 4]]
2×2 Matrix{Int64}:
 6   7
 9  12
```

Indexing using a Boolean array `B` is also called logical indexing. If it is used, each Boolean array used as an index must have the same length as the dimension of `A` it corresponds to or it must be the only index provided and have the same shape as `A`. In the second case, a one-dimensional array is returned. A logical index `B` acts as a mask and chooses the elements of `A` that correspond to **`true`** values in the index `B`.

This example uses two logical indices to select the four corner elements of the magic square.

```
julia> M[[true, false, false, true], [true, false, false, true]]
2×2 Matrix{Int64}:
 16  13
  4   1
julia> sum(ans)
34
```

The next example illustrates using a single logical index that has the same shape as the array.

```
julia> B = [false true true false; true false false true;
            true false false true; false true true false]
4×4 Matrix{Bool}:
 0  1  1  0
 1  0  0  1
 1  0  0  1
 0  1  1  0
julia> M[B]
8-element Vector{Int64}:
  5
  9
  3
 15
  2
 14
  8
 12
julia> sum(M[B])/2
34.0
```

Logical indexing is also useful in expressions such as `AD[AD .<= 8]`. First, the index `AD .<= 8` is a **`BitArray`**, which is then used to extract the subset of the one-dimensional array `AD` for which the condition holds.

The indexing syntax using square brackets is nearly equivalent to calling the function `getindex`. The only difference is that the **`end`** keyword, representing the last index in each dimension, can only be used inside square brackets. The **`end`** keyword can also be part of an expression as in this example.

```
julia> M[end-1:end, 1:div(end, 2)]
2×2 Matrix{Int64}:
 9   6
 4  15
```

The syntax for assigning values to elements of an n-dimensional array `A` is to use the indexing syntax on the left-hand side of an assignment, i.e.,

`A[I1, I2, …, I`n`] = B`.

Again, each index can be one of the five items mentioned at the beginning of this section.

If the right-hand side `B` is an array, the number of elements on the left- and on the right-hand sides must match. Then the element

`A[I1[i11, …, i1`d_1`], I2[i21, …, i2`d_2`], …, I`n`[i`n`1, …, i`nd_n`]]`

on the left-hand side is overwritten with the element

`B[i11, …, i1`d_1`, i21, …, i2`d_2`, …, i`n`1, …, i`nd_n`]`

on the right-hand side. If the right-hand side `B` is not an array, then its value is written to all elements of `A` referenced on the left-hand side.

Analogously to `getindex`, the assignment `A[I1, I2, …, I`n`] = B` is equivalent to the function call `setindex!(A, B, I1, I2, …, I`n`]`.

8.1.5 Iteration and Linear Indexing

Linear indexing means that the elements of an array are indexed by a single index that runs from one to the total number of elements in the array. As a linear index increases, the first dimension (i.e., the row) changes faster than the second dimension, and so forth. Fast linear indexing is generally available if the elements of an array are contiguous in memory. Linear indexing into an array is not always available, e.g., if the array is a view into another array.

```
julia> M[1], M[end]
(16, 1)
```

Iterating over all elements of an array `A` is simple.

```
for m in M
    @show m
end
```

It is also possible to use an index `i` to iterate over all elements of an array `A`.

```
for i in eachindex(M)
    @show i, M[i]
end
```

Here the index `i` is an **`Int`** if fast linear indexing is available for the type of `A`. If linear indexing is not available, the index `i` is, e.g., a `CartesianIndex` as in this example, which also shows how to create a view into an array (see Sect. 8.3). The row index changes faster than the column index.

```
julia> A = view(M, 3:4, 1:2)
2×2 view(::Matrix{Int64}, 3:4, 1:2) with eltype Int64:
 9   6
 4  15
julia> sum(A)
34
julia> for i in eachindex(A) @show (i, A[i]) end
(i, A[i]) = (CartesianIndex(1, 1), 9)
(i, A[i]) = (CartesianIndex(2, 1), 4)
(i, A[i]) = (CartesianIndex(1, 2), 6)
(i, A[i]) = (CartesianIndex(2, 2), 15)
```

8.1.6 Operators

Table 8.4 summarizes the most important operations on arrays. Operators without a dot `.` are operations on (whole) arrays or matrices, while operators with a dot `.` always act elementwise. For example, the equality operator == compares two arrays and returns a single **`Bool`** value, while the elementwise equality operator `.==` returns an array of the same shape as its arguments that contains the results of the elementwise comparisons.

In addition to this general rule, multiplication $*$ acts elementwise when one argument is a scalar value, and the division operators `/` and `\` act elementwise when the denominator is a scalar value.

The left-division operator `\` is popular for solving systems

$$A\mathbf{x} = \mathbf{b}$$

of linear equations. Depending on the structure of the first argument, namely the matrix A, it chooses a suitable linear solver and returns the solution x.

```
julia> A = randn(2, 2); b = randn(2);
julia> A\b
2-element Vector{Float64}:
 3.6219606957004062
 2.2365962069070697
julia> A * (A\b) - b
2-element Vector{Float64}:
 -6.938893903907228e-17
  0.0
```

Table 8.4 Operations on arrays.

Type	Operator	Description
unary arithmetic	`+`,`-`	addition, subtraction
binary arithmetic	`+`,`-`	addition, subtraction
binary arithmetic	`*`	matrix multiplication
binary arithmetic	`.*`	elementwise multiplication
binary arithmetic	`/`	right division, $A/B = AB^{-1}$
binary arithmetic	`\`	left division, $A\backslash B = A^{-1}B$
binary arithmetic	`./`	elementwise right division, $(A\texttt{./}B)_{ij} = a_{ij}b_{ij}^{-1}$
binary arithmetic	`.\`	elementwise left division, $(A\texttt{.\textbackslash}B)_{ij} = a_{ij}^{-1}b_{ij}$
binary arithmetic	`^`	matrix exponentation
binary arithmetic	`.^`	elementwise exponentation
assignment	`=`	assignment
assignment	`.=`	elementwise assignment
comparison	`==`	equality
comparison	`.==`	elementwise equality
comparison	`!=`	inequality
comparison	`.!=`	elementwise inequality
comparison	`.<`, `.>`	elementwise <, >
comparison	`.<=`, `.>=`	elementwise $\leq$, $\geq$
unary Boolean	`.!`	elementwise Boolean not
binary Boolean	`.&`	elementwise Boolean and
binary Boolean	`.|`	elementwise Boolean or
unary bitwise	`.~`	elementwise bitwise not
binary bitwise	`.&`	elementwise bitwise and
binary bitwise	`.|`	elementwise bitwise or

8.1.7 Broadcasting and Vectorizing Functions

Vectorizing a function means to apply it to each element of an array to yield a new array. Hence vectorizing is thus just a synonym for applying the function elementwise or for mapping the function. The syntax for vectorizing any function f and applying it elementwise to any collection A is just `f.(A)`.

```
julia> cos.((0:4) * pi/2)
5-element Vector{Float64}:
  1.0
  6.123233995736766e-17
 -1.0
 -1.8369701987210297e-16
  1.0
```

The effect of the syntax `f.(A)` for vectorizing can also achieved by defining a method such as

```
f(A::AbstractArray) = map(f, A)
```

but it is more convenient to use the built-in syntax for vectorizing than to define methods for each generic function to be vectorized.

Broadcasting is a generalization of vectorization and is supported by the syntax `f.(args...)`, which is equivalent to `broadcast(f, args...)`. Broadcasting means that singleton dimensions in array arguments are expanded in order to match the corresponding dimensions in the other, larger array and that the function is then applied elementwise. No additional memory is required; eliminating the allocation of intermediary arrays is important for performance.

A leading example is adding a vector to the columns of a matrix, e.g., adding $(17, 17, 17, 17)^\top$ to minus the magic square `M`. Just using the substraction `-` results in a dimension-mismatch error. Using the function `broadcast` is more convenient and efficient.

```
julia> broadcast(-, [17, 17, 17, 17], M)
4×4 Matrix{Int64}:
  1  14  15   4
 12   7   6   9
  8  11  10   5
 13   2   3  16
```

Broadcasting is performed by elementwise operations such as `.+`, `.-`, and `.*` automatically if necessary, so that we can also write `[17, 17, 17, 17] .- M`.

The `broadcast` function is even more general and also works on tuples. Any argument that is not an **`Array`** or a **`Tuple`** is treated as a scalar and broadcast.

```
julia> convert.(Int32, (0xf, 0xff, 0xfff, 0xffff, 0xff_fff))
(15, 255, 4095, 65535, 1048575)
```

Furthermore, the compiler guarantees to fuse nested vectorized function calls into a single `broadcast` loop, i.e., `f.(g.(A))` is equivalent to

```
broadcast(a -> f(g(a)), A).
```

This implies that there is only a single loop iterating over the collection A and that only a single array is allocated for the result. The significance of this guarantee is that allocations of temporary arrays for intermediate results are avoided. Fusion of vectorized calls is not possible if non-vectorized function calls happen in between.

The destructive version of `broadcast` is called `broadcast!` as usual. Avoiding allocations of intermediary results is always important for performance, since it eliminates time for memory allocation and reduces the work load of the garbage collector. To avoid allocations, the output of a vectorized operation can be pre-allocated, which can be achieved by *A* `.=` *RHS*, which is equivalent to `broadcast!(identity, A, RHS)`. Additionally, the outer call to `broadcast!` is fused with any vectorized calls in *RHS* if possible. The `broadcast!` function, i.e., the destructive version of `broadcast`, overwrites the first array with the result in place and thus eliminates an allocation for the result and possibly any intermediate results. The left-hand side of `.=` may also be an indexing expression; then `broadcast!` acts on a view.

8.2 Sparse Vectors and Matrices

Sparse vectors and sparse matrices are important types of vectors and matrices. Their defining characteristic is that sufficiently many elements are zero so that storing them in a special data structure is advantageous regarding execution time and memory consumption. Special data structures and algorithms for sparse matrices make calculations possible that could not be performed within reasonable time or space requirements using dense vectors or matrices. An important example is given by discretizations of partial differential equations, especially in higher spatial dimensions (see Chap. 10).

To use sparse vectors or matrices, the built-in module `SparseArrays` must be imported or used first.

```
julia> using SparseArrays
```

The two types `SparseVector` and `SparseMatrixCSC` have two parameters, namely the type of the (non-zero) elements and the integer type of column and row indices.

Sparse matrices are stored in the compressed-sparse-column (CSC) format. This format is especially efficient for calculating matrix-vector products and column slicing. On the other hand, accessing a sparse matrix stored in this format by rows is much slower. Furthermore, inserting non-zero values one at a time is slow, since all elements beyond the insertion point must be moved over.

Many functions pertaining to sparse vectors or matrices start with the prefix `sp` added to the names of the functions dealing with their dense counterparts. The simplest example is `spzeros` for creating empty sparse vectors and matrices, where the type of the elements can optionally be supplied.

```
julia> spzeros(1000)
1000-element SparseVector{Float64, Int64} with 0 stored entries
julia> spzeros(1000, 1000)
1000×1000 SparseMatrixCSC{Float64, Int64} with 0 stored entries
julia> spzeros(BigInt, 1000, 1000)
1000×1000 SparseMatrixCSC{BigInt, Int64} with 0 stored entries
```

As mentioned above, inserting elements into a sparse vector or matrix one element at a time is slow due to the bookkeeping that must be performed. The recommended way to create a `SparseVector` or a `SparseMatrixCSC` with a sizeable number of non-zero elements is to use the functions `sparsevec` (to create a `SparseVector`) or `sparse` (to create a `SparseMatrixCSC`). Calling `sparsevec` as

```
s = sparsevec(i, v)
```

creates a `SparseVector` named `s` such that

s[i[k]] = v[k]

for all indices k. Analogously, calling `sparse` as

```
S = sparse(i, j, v)
```

creates a `SparseMatrixCSC` named `S` such that

`S[i[`k`], j[`k`]] = v[`k`]`

for all indices k. Here the vectors `i` and `j` contain the row and column indices of the non-zero elements and the vector `v` contains the non-zero elements themselves.

```
julia> sparsevec([1, 10, 100], [1.0, 2.0, 3.0])
100-element SparseVector{Float64, Int64} with 3 stored entries:
  [1  ]  =  1.0
  [10 ]  =  2.0
  [100]  =  3.0
julia> S = sparse([1, 10, 100],
                  [1000, 10_000, 100_000],
                  [1.0, 2.0, 3.0])
100×100000 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
  [1  ,   1000]  =  1.0
  [10 ,  10000]  =  2.0
  [100, 100000]  =  3.0
julia> findnz(S)
([1, 10, 100], [1000, 10000, 100000], [1.0, 2.0, 3.0])
```

As this example shows, the function `findnz` retrieves the indices and the non-zero elements of a `SparseVector` or a `SparseMatrixCSC`.

Another use of the function `sparse` is to create the sparse counterpart of a dense vector or matrix. The function `issparse` tests whether its argument is sparse or not.

```
julia> issparse(sparse([0, 1, 2]))
true
julia> sparse([0, 1, 2]) == [0, 1, 2]
true
```

The technique of using `sparsevec` or `sparse` and `findnz` to construct and decompose sparse vectors or matrices is critical for performance when the sizes of the vectors or matrices become large. An example is constructing the vectors and matrices for finite-difference, finite-volume, or finite-element calculations [1] (see Chap. 10).

Finally, Table 8.5 summarizes the functions related to sparse vectors and matrices.

8.3 Array Types

Since many types of arrays and matrices occur in mathematics and in applications, the part of the type system that deals with arrays and matrices is quite

Table 8.5 Functions that operate on sparse vectors or matrices.

Function	Description
`SparseVector`	type for sparse vectors
`SparseMatrixCSC`	type for sparse matrices
`Array`	create a dense version
`issparse`	check whether a vector or matrix is sparse
`sparsevec`	create a sparse vector
`sparse`	create a sparse matrix
`spzeros`	create an empty sparse vector or sparse matrix
`spdiagm`	create a sparse diagonal matrix
`sprand`	create a random sparse matrix of given density;[a] non-zero elements are sampled from given distribution
`sprandn`	create a random sparse matrix of given density;[a] non-zero elements are sampled from given distribution
`nnz`	return the number of stored elements
`nonzeros`	return a vector of the structural non-zero elements
`findnz`	return the indices and values of the non-zero elements
`rowvals`	return a vector with the row indices
`nzrange`	return the column indices of non-zero elements, useful for iterating with a **`for`** loop
`blockdiag`	concatenate matrices block-diagonally
`dropzeros`	remove zero elements
`dropzeros!`	destructive version of `dropzeros`
`droptol!`	drop elements whose absolute value is smaller than a tolerance
`permute`	permute rows and columns
`permute!`	destructive version of `permute`

[a] The density of a sparse matrix is the probability that any element is non-zero.

extensive in order to accommodate various types of arrays. The subtree below **`AbstractArray`** in the tree of types is discussed in this section to elucidate the relationship between the various subtypes and to indicate how special types of matrices or arrays can be implemented idiomatically.

The most general array type is the abstract type **`AbstractArray`**`{T, n}`. An abstract type is a type that cannot be instantiated, i.e., no objects of this type can be created; abstract types usually have subtypes that can be instantiated. The first parameter `T` is the element type and the second parameter is the number `n` of dimensions. The subtypes **`AbstractVector`** and **`AbstractMatrix`** are just aliases for the one- and two-dimensional cases.

```
julia> AbstractVector
AbstractVector (alias for AbstractArray{T, 1} where T)
julia> AbstractVector <: AbstractArray
true
julia> AbstractMatrix
AbstractMatrix (alias for AbstractArray{T, 2} where T)
julia> AbstractMatrix <: AbstractArray
true
```

The essential properties of a specific array type should be implemented by a concrete subtype of **`AbstractArray`**. A concrete type is a type that can be instantiated. Essential properties are `size`, `getindex`, and `setindex!` (in the case of a mutable array). These functions should have a computational complexity that is constant in time, since a defining feature of arrays is that the time to access or change an element is constant. (On the other hand, accessing and changing elements of lists, for example, is an operation whose complexity is usually linear as a function of the number of elements.) Concrete types should also implement the function `similar`, which is used by `copy`, for example. Furthermore, an object of the element type `T` must always be returned when indexing the array using integers (cf. Sect. 8.1.4), and the length of the **`Tuple`** returned by `size` must be the number `n` of dimensions of the array.

The type **`DenseArray`** is an abstract subtype of **`AbstractArray`**.

```
julia> DenseArray <: AbstractArray
true
```

The defining characteristic of a **`DenseArray`** is that it each element occupies memory (in contrast to a sparse vector or sparse array, see Sect. 8.2) and that the elements are laid out in a regular pattern in memory. The memory layout is compatible with C and FORTRAN so that arrays can easily be passed to external C and FORTRAN functions. Concrete subtypes should define a method for the function `stride` such that `stride(A, k)` returns the distance in the memory layout, i.e., the difference in linear indices, between two elements that are adjacent in dimension `k`.

This is illustrated by the following example. We can use a linear index to iterate over all elements of the square matrix `M` in the order in which they are laid out in memory. (The recommended and simpler way to iterate over all elements of an array is **`for`** `a` **`in`** `A`, see Sect. 8.1.5.)

```
for i in 1:prod(size(M))
    print(M[i], " ")
end
# 16 5 9 4 3 10 6 15 2 11 7 14 13 8 12 1
```

This prints the elements in the same order as **`for`** `m` **`in`** `M`.

Another way to iterate over all elements is to use strides. We can calculate a linear index from the indices over each dimension. In **`for`** loops, the iteration variable given last changes fastest, and we want the iteration variable over the first dimension to change fastest. Therefore the dimensions are listed in descending order when specifying the loop variables in this example.

```
for j in 1:size(M, 2), i in 1:size(M, 1)
    println((i, j, M[1 + (i-1)*stride(M, 1) + (j-1)*stride(M, 2)]))
end
```

This prints the elements (and their indices) in the same linear order as above.

The use strides can be illustrated by iterating over a three-dimensional array in two different ways as well.

```
global A = rand(2, 2, 2)

for i in 1:prod(size(A))
    println(A[i])
end

for k in 1:size(A, 3), j in 1:size(A, 2), i in 1:size(A, 1)
    println((i, j, k,
             A[1 + (i-1) * stride(A, 1) +
                   (j-1) * stride(A, 2) +
                   (k-1) * stride(A, 3)]))
end
```

The type **Array** is a subtype of **DenseArray** and ensures that elements are stored in column-major order.

```
julia> Array <: DenseArray
true
julia> isa(M, Array)
true
```

Vectors and matrices as we know them in mathematics are subtypes of the **Array** type: **Vector** is an alias for a one-dimensional **Array** and **Matrix** is an alias for a two-dimensional **Array**.

```
julia> Vector <: Array
true
julia> Matrix <: Array
true
julia> Vector
Vector (alias for Array{T, 1} where T)
julia> Matrix
Matrix (alias for Array{T, 2} where T)
```

A **SubArray** is a subtype of **AbstractArray** that performs indexing by reference, and not by copying, which can be useful for performance reasons. A **SubArray** is created by a call to the `view` function, which is similar to `getindex`, but returns a view into the parent array instead of copying the elements. **SubArray**s are a convenient way to reference a part of an array, which means that modifying the elements of a **SubArray** also modifies the elements of the original array as illustrated in this example.

```
julia> N = copy(M); column1 = view(N, :, 1)
4-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 16
  5
  9
  4
```

```
julia> isa(column1, SubArray)
true
julia> column1 .= 0; N
4×4 Matrix{Int64}:
 0   3   2  13
 0  10  11   8
 0   6   7  12
 0  15  14   1
```

Two more subtypes of **`AbstractArray`** are the types **`StridedVector`** and **`StridedMatrix`**, whose purpose is to interface to BLAS and LAPACK functions efficiently regarding memory allocation and copying.

Finally, sparse vectors and sparse matrices are subtypes of their abstract supertypes `AbstractSparseVector` and `AbstractSparseMatrix`. They are (of course) not subtypes of **`DenseArray`**, and hence also not of **`Array`**, **`Vector`**, **`Matrix`**, and **`SubArray`**.

```
julia> using SparseArrays
julia> SparseVector <: AbstractSparseVector <: AbstractVector
true
julia> SparseMatrixCSC <: AbstractSparseMatrix <: AbstractMatrix
true
```

8.4 Linear Algebra

In this section, major concepts from linear algebra are summarized and their implementation in JULIA is discussed.

8.4.1 Vector Spaces and Linear Functions

Linear algebra is the branch of mathematics concerning linear functions and their properties as functions between vector spaces. Before we discuss functions on vector spaces, it is useful to briefly recall the definition of a vector space. A vector space V over a field F is a set equipped with two binary operations, namely vector addition and scalar multiplication, satisfying the following axioms. The elements of V are called vectors and the elements of F are called scalars. Vector addition is the function $+: V \times V \to V, (\mathbf{u}, \mathbf{v}) \mapsto \mathbf{u} + \mathbf{v}$. Scalar multiplication is the function $\cdot: F \times V \to V, (a, \mathbf{u}) \mapsto a\mathbf{u}$.

The eight axioms defining a vector space are

1. the associativity $\mathbf{u} + (\mathbf{v} + \mathbf{w}) = (\mathbf{u} + \mathbf{v}) + \mathbf{w}$ of addition,
2. the commutativity $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ of addition,

3. the existence of an identity element $\mathbf{0} \in V$ (the zero vector) of addition, i.e., $\mathbf{v} + 0 = \mathbf{v}$,
4. the existence of inverse elements $-\mathbf{v} \in V$ of addition, i.e., $\mathbf{v} + (-\mathbf{v}) = 0$,
5. the distributivity $a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$ of scalar multiplication with respect to vector addition,
6. the distributivity $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$ of scalar multiplication with respect to field addition,
7. the compatibility $a(b\mathbf{v}) = (ab)\mathbf{v}$ of scalar multiplication with field multiplication, and
8. the multiplicative identity 1 of the field F is the identity element of scalar multiplication, i.e., $1\mathbf{v} = \mathbf{v}$.

Here all equations must hold for all $\mathbf{u} \in V$, for all $\mathbf{v} \in V$, for all $\mathbf{w} \in V$, for all $a \in F$, and for all $b \in F$.

The first four axioms are equivalent to V being an Abelian group under vector addition. The last four axioms concern the interaction of vector addition and scalar multiplication with the underlying field F.

The historically leading example of a vector space is of course the set of points in a Euclidean space equipped with the well-known geometric operations. The underlying field of a vector space is often the real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. (The reader may want to recall the definition of a field.) Further examples of vector spaces are sequences, polynomials, functions, and matrices, all equipped with suitable operations.

The significance of the vector data structure is that it provides a short and convenient representation of the elements of a vector space. Suppose that $\mathbf{B} := \{\mathbf{b}_1, \dots, \mathbf{b}_l\} \subset U$ is a finite subset of the vector space U over the field F. The set $\mathbf{B}$ is called a basis of U if the elements of $\mathbf{B}$ are linearly independent and span the whole vector space. The vectors in $\mathbf{B}$ are defined to be linearly independent if the condition

$$\forall a_1, \dots, a_l \in F: \quad \sum_{i=1}^{l} a_i \mathbf{b}_i = 0 \implies a_1 = \cdots = a_l = 0$$

holds. They span the whole vector space if every element $\mathbf{u}$ of U can be written as a linear combination of the basis vectors $\mathbf{B}$, i.e.,

$$\forall \mathbf{u} \in U: \quad \exists u_1, \dots, u_l \in F: \quad \mathbf{u} = \sum_{i=1}^{l} u_i \mathbf{b}_i.$$

The coefficients $u_i \in F$ are the coordinates of the vector $\mathbf{u}$ with respect to the basis $\mathbf{B}$ and they are uniquely determined because of the linear independence of the basis vectors. Furthermore, the dimension $\dim U$ of U is l.

Therefore every vector $\mathbf{u} \in U$ can be represented by its coordinates u_i written in the form

$$\mathbf{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix}_{\mathbf{B}} = \begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix}. \tag{8.1}$$

The basis **B** has been indicated here for the sake of completeness; in most cases, it is known from the context and omitted. The coefficients u_i are called the elements (of the representation) of the vector.

It is customary to write vectors as column vectors (and not as row vectors) for most purposes in linear algebra for a reason that will become clear soon.

In JULIA, vectors are of course represented by the data structure `Vector`{*type*}, where the *type* of the elements plays the role of the underlying field F in mathematics.

The significance of matrices is that every linear function between two given vector spaces can be represented as a matrix and, vice versa, every matrix gives rise to a linear function (again between two given vector spaces). To see this, we consider linear functions $f : U \to V$ between two vector spaces U and V. We also choose a basis $\mathbf{B} := \{\mathbf{b}_1, \dots, \mathbf{b}_l\}$ of the l-dimensional vector space U and a basis $\mathbf{C} := \{\mathbf{c}_1, \dots, \mathbf{c}_m\}$ of the m-dimensional vector space V. Since f is linear, i.e., it is compatible with the vector addition and scalar multiplication via

$$\begin{aligned} f(\mathbf{u}_1 + \mathbf{u}_2) &= f(\mathbf{u}_1) + f(\mathbf{u}_2) \qquad && \forall \mathbf{u}_1 \in U \quad \forall \mathbf{u}_2 \in U, \\ f(a\mathbf{u}) &= af(\mathbf{u}) && \forall a \in F \quad\;\; \forall \mathbf{u} \in U, \end{aligned}$$

it suffices to know or to store the images of the basis vectors $\mathbf{b}_i$. This fact follows immediately from the linearity of f, since

$$f(\mathbf{u}) = f\left(\sum_{i=1}^{l} u_i \mathbf{b}_i\right) = \sum_{i=1}^{l} u_i f(\mathbf{b}_i) \tag{8.2}$$

holds. Representing the vector $\mathbf{u}$ by the coefficients $u_i \in F$ and knowing the images $f(\mathbf{b}_i)$, the right-hand side is calculated in a straightforward manner in order to obtain the image $f(\mathbf{u}) \in V$ of $\mathbf{u} \in U$.

To record the images $f(\mathbf{b}_i) \in V$ in an orderly fashion, we arrange them as column vectors in a two-dimensional array of numbers, the matrix A, and write

$$A := (f(\mathbf{b}_1), \dots, f(\mathbf{b}_l)) = \begin{pmatrix} f(\mathbf{b}_1)_1 & \cdots & f(\mathbf{b}_l)_1 \\ \vdots & \ddots & \vdots \\ f(\mathbf{b}_1)_m & \cdots & f(\mathbf{b}_l)_m \end{pmatrix},$$

where the element $a_{ji} := f(\mathbf{b}_i)_j \in F$ of the matrix A is the j-th element of the vector $f(\mathbf{b}_i) \in W$. Since U is l-dimensional, i runs from 1 to l, and since V is m-dimensional, j runs from 1 to m. Therefore the matrix A contains m rows and l columns, and we say it has dimension $m \times l$. We denote the set of all $(m \times l)$-dimensional matrices over the field F by $F^{m\times l}$.

Since the matrix A contains all the information about the function f, it is certainly possible to calculate the image $f(\mathbf{u})$. How can we calculate it easily using A? The answer is given by the matrix-vector multiplication defined by

$$A\mathbf{u} = \begin{pmatrix} a_{11} & \cdots & a_{1l} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{ml} \end{pmatrix}_{\mathbf{BC}} \begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix}_{\mathbf{B}} := \begin{pmatrix} a_{11}u_1 + \cdots + a_{1l}u_l \\ \vdots \\ a_{m1}u_1 + \cdots + a_{ml}u_l \end{pmatrix}_{\mathbf{C}}$$

$$= \begin{pmatrix} \sum_{i=1}^{l} a_{1i}u_i \\ \vdots \\ \sum_{i=1}^{l} a_{mi}u_i \end{pmatrix}_{\mathbf{C}}, \qquad (8.3)$$

which is a function $F^{m\times l} \times U \to V$. For the sake of completeness, the bases over which the elements of the matrix and the vectors are to be understood are indicated here, but are usually omitted.

The right-hand side is just the linear combination of the images $f(\mathbf{b}_i)$ with the coefficients u_i, i.e.,

$$A\mathbf{u} = \sum_{i=1}^{l} u_i f(\mathbf{b}_i).$$

Recalling (8.2), this equation implies

$$f(\mathbf{u}) = A\mathbf{u}.$$

We have seen that every linear function $f : U \to V$ gives rise to a matrix A after fixing a basis $\mathbf{B}$. This matrix contains all the information of the linear function f. Vice versa, any matrix A gives rise to a linear function f via matrix-vector multiplication by defining $f(\mathbf{u}) := A\mathbf{u}$. It is straightforward to see that the function f defined in this manner is indeed linear by the definition of matrix-vector multiplication.

Therefore we have constructed a bijection between the linear functions $f : U \to V$ from the l-dimensional vector space U to the m-dimensional vector space V (both over the field F) and the matrices in $F^{m\times l}$. In other words, the matrix A is a representation of the function f in the basis $\mathbf{B}$.

Now the reason why vectors are usually written as column vectors in linear algebra becomes clear. If the matrix A represents the linear function f, the matrix-vector product $A\mathbf{u}$ is equivalent to the function evaluation $f(\mathbf{u})$ and both the function and the matrix are written before the vector $\mathbf{u}$. Being able to choose a notation between matrix-vector products $A\mathbf{u}$ and column vectors on the one hand and vector-matrix productions $\mathbf{u}A$ and row vectors on the other hand, the first choice resembles the function evaluation $f(\mathbf{u})$ and is therefore the more natural choice (cf. Sect. 8.1.2).

Because of this bijection between linear functions f and matrices A, there is always a correspondence between the properties of linear functions and matrices. For example, if f is bijective, then A is called regular. Furthermore, properties of and operations on linear functions and matrices correspond to types and functions in JULIA. These JULIA types and functions are described in detail in the following. For example, matrix-vector multiplication is performed by the generic function $*$. Many symbols concerning linear algebra are placed in the built-in module `LinearAlgebra`, and it is assumed in the following that you have evaluated **`using`** `LinearAlgebra`.

How can we represent the composition of linear functions using matrices? Suppose we want to compose two linear functions $f : U \to V$ (represented by the matrix $A \in F^{m\times l}$) and $g : V \to W$ (represented by the matrix $B \in F^{n\times m}$). Here W is an n-dimensional vector space. It is straightforward to show that the resulting composition $g\circ f : U \to W$ is again a linear function, and we intend to find its representation $C \in F^{n\times l}$.

Using (8.3), we know that $f(\mathbf{u}) = A\mathbf{u}$. Using (8.3) again for multiplying the matrix $B = (b_{kj})$ by the vector $A\mathbf{u} \in V$, we find that

$$g(f(\mathbf{u})) = B(A\mathbf{u}) = B\begin{pmatrix} \sum_{i=1}^{l} a_{1i}u_i \\ \vdots \\ \sum_{i=1}^{l} a_{mi}u_i \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{m} b_{1j}\sum_{i=1}^{l} a_{ji}u_i \\ \vdots \\ \sum_{j=1}^{m} b_{nj}\sum_{i=1}^{l} a_{ji}u_i \end{pmatrix}$$

$$= \begin{pmatrix} \sum_{j=1}^{m}\sum_{i=1}^{l} b_{1j}a_{ji}u_i \\ \vdots \\ \sum_{j=1}^{m}\sum_{i=1}^{l} b_{nj}a_{ji}u_i \end{pmatrix} = \underbrace{\begin{pmatrix} \sum_{j=1}^{m} b_{1j}a_{j1} & \cdots & \sum_{j=1}^{m} b_{1j}a_{jl} \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^{m} b_{nj}a_{j1} & \cdots & \sum_{j=1}^{m} b_{nj}a_{jl} \end{pmatrix}}_{C:=}\begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix} = C\mathbf{u}.$$

The last equation yields the matrix $C \in F^{n\times l}$, whose entries are

$$c_{ki} = \sum_{j=1}^{m} b_{kj}a_{ji}.$$

In this calculation, we have only used matrix-vector multiplication defined above and we found that $B(A\mathbf{u}) = C\mathbf{u}$ holds for all vectors $\mathbf{u} \in U$. This implies that the composition $g\circ f$ is represented by the matrix C. The matrix-matrix multiplication defined by

$$BA := C$$

is a function $F^{n\times m} \times F^{m\times l} \to F^{n\times l}$ and corresponds to the composition of two linear functions.

Matrix-matrix multiplication is associative, i.e., $(CB)A = C(BA)$ for all matrices whose products exist. This can be checked directly using the definition above. It also follows from the associative property $(h\circ g)\circ f = h\circ(g\circ f)$ of the corresponding linear functions. Matrix-matrix multiplication is not commutative, since $g\circ f \neq f\circ g$ in general. In JULIA, matrix-matrix multiplication is performed by the generic function $*$.

8.4.2 Basis Change

In our discussion of the relationships between linear functions and matrices, we have used fixed bases so far. The canonical basis or standard basis of an l-dimensional vector space U is the basis $\mathbf{E} := \{\mathbf{e}_1, \dots \mathbf{e}_l\}$, where

$$\mathbf{e}_1 := \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \mathbf{e}_2 := \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \dots, \quad \mathbf{e}_l := \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

i.e., the i-th element of $\mathbf{e}_i$ is equal to $1 \in F$ and the other elements vanish. The matrix

$$I := (\mathbf{e}_1, \dots, \mathbf{e}_l) = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & & & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

containing all standard basis vectors as columns is called the identity matrix. Obviously, $AI = IA = A$ holds for all square matrices $A \in F^{l \times l}$. In JULIA, identity matrices are constructed by the (generic) function `eye`.

Sometimes it is useful to change the basis in which a vector is represented in (8.1). We call the old basis $\mathbf{B}_1 := \{\mathbf{b}_1^1, \dots, \mathbf{b}_l^1\}$ and the new basis $\mathbf{B}_2 := \{\mathbf{b}_1^2, \dots, \mathbf{b}_l^2\}$. Analogously to the identity matrix, we define $B_1 := (\mathbf{b}_1^1, \dots, \mathbf{b}_l^1)$ to be the matrix that contains all basis vectors of $\mathbf{B}_1$ as columns, and we also define $B_2 := (\mathbf{b}_1^2, \dots, \mathbf{b}_l^2)$.

We denote the function that performs a basis change from the old basis $\mathbf{B}_1$ to the new basis $\mathbf{B}_2$ by

$$g\colon U \to U, \quad \mathbf{u}_{\mathbf{B}_1} \mapsto \mathbf{u}_{\mathbf{B}_2}.$$

It maps the coefficients of the vector $\mathbf{u}$ with respect to the old basis $\mathbf{B}_1$ to its coefficients with respect to the new basis $\mathbf{B}_2$. Since $g(\mathbf{b}_i^1) = \mathbf{b}_i^2$ holds for all $i \in \{1, \dots, l\}$, it is obvious that g is a linear function. We denote the matrix representation of g by G. Since every basis change g is a bijection, its inverse function g^{-1} exists and is represented by the inverse matrix G^{-1} of G (see Sect. 8.4.8).

Therefore we have

$$\begin{aligned} \mathbf{u}_{\mathbf{B}_2} = \begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix}_{\mathbf{B}_2} &= \sum_{i=1}^{l} u_i \mathbf{b}_i^2 = \sum_{i=1}^{l} u_i g(\mathbf{b}_i^1) = \sum_{i=1}^{l} u_i G \mathbf{b}_i^1 = G \sum_{i=1}^{l} u_i \mathbf{b}_i^1 \\ &= G \begin{pmatrix} u_1 \\ \vdots \\ u_l \end{pmatrix}_{\mathbf{B}_1} = G \mathbf{u}_{\mathbf{B}_1}. \end{aligned}$$

An important example of a basis change is rotation. In two-dimensional Euclidean geometry, counter-clockwise rotation by the angle ϕ is expressed by multiplication with the matrix

$$R(\phi) := \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}.$$

In JULIA, this matrix is calculated as follows.

```
function rotate(phi)
    [cos(phi) -sin(phi)
     sin(phi)  cos(phi)]
end
```

Note that a newline character can be used instead of a semicolon to indicate the start of another row.

So far we seen how to change the basis over which a vector as written in (8.1) is to be understood. We can also change the bases over which a matrix as written in (8.3) is to be understood. This is useful in situations when a linear function is known or more easily investigated in a certain basis. Linear functions also come with basis vectors which are helpful to understand their action (see Sect. 8.4.9 and Sect. 8.4.10).

Suppose that $A_{\mathbf{B}_1\mathbf{C}_1}$ is the representation of a linear function f in the old bases $\mathbf{B}_1$ of U and $\mathbf{C}_1$ of V. How can we find the representation $A_{\mathbf{B}_2\mathbf{C}_2}$ of f in the new bases $\mathbf{B}_2$ and $\mathbf{C}_2$? We start from the two basis changes

$$\mathbf{u}_{\mathbf{B}_2} = G\mathbf{u}_{\mathbf{B}_1},$$
$$\mathbf{v}_{\mathbf{C}_2} = H\mathbf{v}_{\mathbf{C}_1}$$

and the representation of A over the old bases, i.e., from the equation

$$V \ni \mathbf{v}_{\mathbf{C}_1} = A_{\mathbf{B}_1\mathbf{C}_1}\mathbf{u}_{\mathbf{B}_1} \in U.$$

Multiplying this equation from the left by H and using $\mathbf{u}_{\mathbf{B}_1} = G^{-1}\mathbf{u}_{\mathbf{B}_2}$ yields

$$\mathbf{v}_{\mathbf{C}_2} = H\mathbf{v}_{\mathbf{C}_1} = HA_{\mathbf{B}_1\mathbf{C}_1}\mathbf{u}_{\mathbf{B}_1} = \underbrace{HA_{\mathbf{B}_1\mathbf{C}_1}G^{-1}}_{A_{\mathbf{B}_2\mathbf{C}_2}:=}\mathbf{u}_{\mathbf{B}_2}$$

so that the sought matrix $A_{\mathbf{B}_2\mathbf{C}_2}$ is given by

$$A_{\mathbf{B}_2\mathbf{C}_2} = HA_{\mathbf{B}_1\mathbf{C}_1}G^{-1}. \tag{8.4}$$

If $U = V$, the two matrices $A_{\mathbf{B}_1\mathbf{C}_1}$ and $A_{\mathbf{B}_2\mathbf{C}_2}$ are called similar or conjugate (see Definition 8.34). Similarity is an equivalence relation.

The last equation implies

$$A_{\mathbf{B}_1\mathbf{C}_1} = H^{-1}A_{\mathbf{B}_2\mathbf{C}_2}G, \tag{8.5}$$

$$\begin{tikzcd}
\mathbf{B}_2 \arrow[r, "A_{\mathbf{B}_2\mathbf{C}_2}"] & \mathbf{C}_2 \arrow[d, "H^{-1}"'] \\
\mathbf{B}_1 \arrow[u, "G"] \arrow[r, "A_{\mathbf{B}_1\mathbf{C}_1}"] & \mathbf{C}_1
\end{tikzcd}$$

Fig. 8.1 Commutative diagram for changing the bases of a matrix.

which is easily interpreted in the commutative diagram Fig. 8.1. In the old bases $\mathbf{B}_1$ and $\mathbf{C}_1$, $A_{\mathbf{B}_1\mathbf{C}_1}$ maps vectors represented using $\mathbf{B}_1$ to those represented using $\mathbf{C}_1$; this is the left-hand side of the equation and the arrow at the bottom in the diagram. The same effect is achieved by changing the argument vector from the old basis $\mathbf{B}_1$ to the new basis $\mathbf{B}_2$, then applying the linear function via its new representation $A_{\mathbf{B}_2\mathbf{C}_2}$, and finally changing from the new basis $\mathbf{C}_2$ back to the old basis $\mathbf{C}_1$; this is the right-hand-side of the equation and the other three arrows in the diagram.

Next, we consider an example. We seek the representation of a geometric transformation in the canonical basis. The transformation is stretching the entire two-dimensional plane by a factor of 2 only in the direction of the x-axis rotated by $\pi/4$. We define the two bases $\mathbf{C}_1 := \mathbf{B}_1 := \mathbf{E}$ and the basis change $H := G := R(-\pi/4)$ such that

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}_{\mathbf{B}_2} = R(-\pi/4)\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}_{\mathbf{B}_1}.$$

In the new bases $\mathbf{B}_2$ and $\mathbf{C}_2$, the stretching transformation is easily expressed as

$$A_{\mathbf{B}_2\mathbf{C}_2} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}.$$

The matrix representing the transformation in the canonical basis is therefore

$$A_{\mathbf{EE}} = A_{\mathbf{B}_1\mathbf{C}_1} = H^{-1}A_{\mathbf{B}_2\mathbf{C}_2}G = R(\pi/4)\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}R(-\pi/4).$$

In JULIA, the transformation is

```
julia> A = rotate(pi/4) * [2 0; 0 1] * rotate(-pi/4)
2×2 Matrix{Float64}:
 1.5  0.5
 0.5  1.5
```

Finally, we check that it computes the desired transformation. The vector $(1,1)^\top$ should be stretched by a factor of two, and the vector $(-1,1)^\top$, which is orthogonal to it, should remain unchanged.

```
julia> A * [1, 1]
2-element Vector{Float64}:
 2.0
 2.0
julia> A * [-1, 1]
2-element Vector{Float64}:
 -1.0
  1.0
```

8.4.3 Inner-Product Spaces

Many vector spaces can be equipped with an inner product. Inner products give vector spaces geometric structure by making it possible to define lengths and angles. An inner product $\langle .,. \rangle$ of the vector space V is a function

$$\langle .,. \rangle \colon V \times V \to F$$

that satisfies – for all vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w} \in V$ and for all scalars $a \in F$ – the three conditions of conjugate symmetry $\langle \mathbf{u}, \mathbf{v} \rangle = \overline{\langle \mathbf{v}, \mathbf{u} \rangle}$, linearity in the first argument $\langle a\mathbf{u}, \mathbf{v} \rangle = a\langle \mathbf{u}, \mathbf{v} \rangle$ and $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$, and positive-definiteness $\langle \mathbf{v}, \mathbf{v} \rangle \geq 0$ with equality if and only if $\mathbf{v} = 0$.

Every inner product induces a norm on its vector space V by defining

$$||\mathbf{v}||^2 := \langle \mathbf{v}, \mathbf{v} \rangle.$$

In JULIA, the function `LinearAlgebra.dot` computes the canonical inner product

$$\mathbf{u} \cdot \mathbf{v} := \sum_{i=1}^{\dim V} \overline{u_i} v_i,$$

where the first vector is conjugated.

An important inequality that involves the inner product is the following.

Theorem 8.1 (Cauchy–Bunyakovsky–Schwarz inequality) *The inequality*

$$|\langle \mathbf{u}, \mathbf{v} \rangle|^2 \leq \langle \mathbf{u}, \mathbf{u} \rangle \langle \mathbf{v}, \mathbf{v} \rangle \qquad \forall\forall \mathbf{u}, \mathbf{v} \in \mathbb{R}^d$$

holds, while equality holds if and only if $\mathbf{u}$ *and* $\mathbf{v}$ *are linearly dependent.*

Equivalently, we can also write

$$|\langle \mathbf{u}, \mathbf{v} \rangle| \le \|\mathbf{u}\|\|\mathbf{v}\| \qquad \forall\forall \mathbf{u}, \mathbf{v} \in \mathbb{R}^d.$$

The cosine of the angle $\phi(\mathbf{u}, \mathbf{v})$ between two vectors $\mathbf{u}$ and $\mathbf{v}$ is defined as

$$\cos \phi(\mathbf{u}, \mathbf{v}) := \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|},$$

and the inequality $|\cos\phi| \le 1$ holds because of the Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1. Two vectors are called orthogonal if $\langle \mathbf{u}, \mathbf{v} \rangle = 0$, which corresponds to $\phi(\mathbf{u}, \mathbf{v}) \in \{\pi/2, 3\pi/2\}$ in the case of the canonical inner product.

A basis of a vector space is called orthogonal if all basis vectors are orthogonal to one another. It is called orthonormal if all basis vectors are orthogonal and have length one.

An inner product also gives rise to the conjugate transpose or Hermitian conjugate of a matrix A. It is denoted by A^* and is defined as the matrix that satisfies

$$\langle A\mathbf{u}, \mathbf{v} \rangle = \langle u, A^*\mathbf{v} \rangle$$

for all vectors $\mathbf{u}$ and $\mathbf{v} \in V$. This means that the elements of A^* are given by $\overline{a_{ji}}$, if the elements of A are denoted by a_{ij}. If the underlying field of the vector space V are the real numbers $\mathbb{R}$, then the complex transpose A^* is the transpose of A and denoted by $A^\top$; its elements are a_{ji}.

In JULIA, the conjugate transpose or Hermitian conjugate of a matrix is calculated by the functions `adjoint` and `adjoint!` or the postfix operator `'`.

```
julia> A = [1+2im 3+4im; 5+6im 7+8im]; A'
2×2 adjoint(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 1-2im  5-6im
3-4im  7-8im
```

The transpose is calculated by `transpose`.

```
julia> transpose(A)
2×2 transpose(::Matrix{Complex{Int64}}) with eltype Complex{Int64}:
 1+2im  5+6im
 3+4im  7+8im
```

A matrix A is called self-adjoint or Hermitian if $A = A^*$. If the underlying field is $\mathbb{R}$ and A is self-adjoint, i.e., $A = A^\top$, then A is called symmetric.

If V is a vector space over the complex numbers $\mathbb{C}$, there is a bijection between the inner products and the sesquilinear forms $(\mathbf{u}, \mathbf{v}) \mapsto \overline{\mathbf{v}}A\mathbf{u}$, where A is a self-adjoint positive-definite matrix.

8.4.4 The Rank-Nullity Theorem

Before we can state the rank-nullity theorem, some definitions are required. The kernel or nullspace of a function $f : U \to V$ is the set of all elements $\mathbf{u} \in U$ whose image vanishes, i.e.,

$$\ker(f) := \{\mathbf{u} \in U \mid f(\mathbf{u}) = 0\}.$$

It is straightforward to show that the kernel of a linear function is a linear subspace of the domain U. The nullity $\mathrm{nul}(f)$ of a linear function f or a matrix is the dimension of its kernel. Furthermore, the rank $\mathrm{rk}(f)$ of a linear function f or a matrix is the dimension of its image $f(U)$, which can be shown to be a linear subspace as well.

The rank-nullity theorem is an important relationship between the dimensions of the kernel, the image, and the preimage space of a linear function or matrix.

As we have seen, matrices are representations of linear functions. Therefore we can state the rank-nullity theorem in both the language of linear functions and in the one of matrices.

In the language of linear functions, the following theorem holds.

Theorem 8.2 (rank-nullity theorem for linear functions) *Let $f : U \to V$ be a linear function. Then the equation*

$$\dim(\ker f) + \dim(\mathrm{im}\, f) = \dim(U)$$

holds.

In the language of matrices, the theorem can be stated as follows. The nullity and the rank of a matrix are the nullity and rank of the corresponding linear function.

Theorem 8.3 (rank-nullity theorem for matrices) *Let $A \in F^{n \times l}$ be an $n \times l$-dimensional matrix. Then the equation*

$$\mathrm{nul}(A) + \mathrm{rk}(A) = l$$

holds.

In JULIA, the function `LinearAlgebra.nullspace` calculates a basis of the nullspace of a matrix and the function `LinearAlgebra.rank` computes its rank.

```
julia> A = [1 1 0 0; 0 1 1 0; 0 1 -1 0]
3×4 Matrix{Int64}:
 1  1   0  0
 0  1   1  0
 0  1  -1  0
julia> nullspace(A)
```

```
4×1 Matrix{Float64}:
 0.0
 0.0
 0.0
 1.0
julia> rank(A)
3
julia> size(nullspace(A), 2) + rank(A) == size(A, 2)
true
```

8.4.5 Matrix Types

In applications, matrices with special structures often arise. The special properties of these matrices can often be exploited by specialized operations and algorithms such as matrix factorizations (see Sect. 8.4.11). Therefore important matrix types are discussed in the following.

The simplest matrix type are diagonal matrices which have the form

$$\begin{pmatrix} * & & \\ & \ddots & \\ & & * \end{pmatrix}.$$

We simplify notation by not showing zero entries. Furthermore, the symbol $*$ denotes any element of the underlying field, and two entries $*$ are not necessarily equal.

The function `LinearAlgebra.Diagonal` constructs a diagonal matrix from a vector by placing its elements in the diagonal or from a matrix taking only its diagonal elements. The result is of type `LinearAlgebra.Diagonal`.

```
julia> Diagonal([1, 2])
2×2 Diagonal{Int64, Vector{Int64}}:
 1  .
 .  2
julia> Diagonal <: AbstractMatrix
true
```

Upper and lower bidiagonal matrices are of the forms

$$\begin{pmatrix} * & * & & \\ & \ddots & \ddots & \\ & & \ddots & * \\ & & & * \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} * & & & \\ * & \ddots & & \\ & \ddots & \ddots & \\ & & * & * \end{pmatrix},$$

respectively. They are constructed by the function `Bidiagonal`. Again, `Bidiagonal` is a subtype of **`AbstractMatrix`**.

Tridiagonal matrices are of the form

$$\begin{pmatrix} * & * & & \\ * & \ddots & \ddots & \\ & \ddots & \ddots & * \\ & & * & * \end{pmatrix},$$

and their type is called `Tridiagonal`.

Symmetric tridiagonal matrices are represented by the type `SymTridiagonal`. They are tridiagonal matrices whose first sub- and super-diagonals are equal.

Self-adjoint or Hermitian matrices, i.e., matrices over $\mathbb{C}$ with the property $A = A^*$, are represented by the type `Hermitian` and are constructed by the function `Hermitian` from the upper or lower triangle of a given array.

```
julia> A = [1 3+4im; 5+6im 7]
2×2 Matrix{Complex{Int64}}:
 1+0im  3+4im
 5+6im  7+0im
julia> Hermitian(A, :U)
2×2 Hermitian{Complex{Int64}, Matrix{Complex{Int64}}}:
 1+0im  3+4im
 3-4im  7+0im
julia> Hermitian(A, :L)
2×2 Hermitian{Complex{Int64}, Matrix{Complex{Int64}}}:
 1+0im  5-6im
 5+6im  7+0im
```

Analogously, symmetric matrices, i.e., matrices over $\mathbb{R}$ and with the property $A = A^\top$, are represented by the type `Symmetric` and are constructed by the function `Symmetric`.

Upper-triangular and lower-triangular matrices are matrices of the forms

$$\begin{pmatrix} * & * & * \\ & \ddots & * \\ & & * \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} * & & \\ * & \ddots & \\ * & * & * \end{pmatrix},$$

respectively. They are important for solving linear systems (see Sect. 8.4.8) and in matrix factorizations (see Sect. 8.4.11). They are represented by the types `UpperTriangular` and `LowerTriangular`, and they are constructed by functions of the same name.

The function `UniformScaling` returns a multiple of the identity matrix I, which is generally sized so that it can be multiplied by any matrix.

```
julia> UniformScaling(2) * [1 2; 3 4]
2×2 Matrix{Int64}:
 2  4
 6  8
```

Table 8.6 gives an overview of the matrix types.

Table 8.6 Matrix types.

Function or type	Description
`LinearAlgebra.Diagonal`	diagonal matrix
`LinearAlgebra.Bidiagonal`	bidiagonal matrix
`LinearAlgebra.Tridiagonal`	tridiagonal matrix
`LinearAlgebra.SymTridiagonal`	symmetric tridiagonal matrix
`LinearAlgebra.Symmetric`	symmetric matrix
`LinearAlgebra.Hermitian`	self-adjoint or Hermitian matrix
`LinearAlgebra.UpperTriangular`	upper-triangular matrix
`LinearAlgebra.LowerTriangular`	lower-triangular matrix
`LinearAlgebra.UniformScaling`	constant times identity matrix

In general, matrices can be converted from the general **`AbstractMatrix`** type to a special type by calling the constructor of the special type on the matrix. Vice versa, a special type can be converted to the general **`Array`** type by calling the constructors **`Matrix`** or **`Array`** on the special matrix.

```
julia> Tridiagonal(M)
4×4 Tridiagonal{Int64, Vector{Int64}}:
 16   3   .   .
  5  10  11   .
  .   6   7  12
  .   .  14   1
julia> Matrix(Tridiagonal(M))
4×4 Matrix{Int64}:
 16   3   0   0
  5  10  11   0
  0   6   7  12
  0   0  14   1
```

8.4.6 The Cross Product

The cross product or vector product $\times\colon \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ is defined on the three-dimensional real vector space $\mathbb{R}^3$ and has useful geometric meaning. Its three defining, geometric properties are that

1. $\mathbf{a} \times \mathbf{b}$ is orthogonal to $\mathbf{a}$ and $\mathbf{b}$,
2. the three vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{a} \times \mathbf{b}$ are a right-handed system, and
3. the (Euclidean) length of $\mathbf{a} \times \mathbf{b}$ is the area of the parallelogram spanned by $\mathbf{a}$ and $\mathbf{b}$.

It can be shown that these three defining properties imply the definition

$$\mathbf{a} \times \mathbf{b} := ||\mathbf{a}|| ||\mathbf{b}|| \sin(\angle(\mathbf{a}, \mathbf{b}))\mathbf{n},$$

where the angle $\angle(\mathbf{a}, \mathbf{b}) \in [0, \pi]$ is the angle between $\mathbf{a}$ and $\mathbf{b}$ in the plane containing both and $\mathbf{n}$ is a unit vector normal to the same plane. Its direction is given by the right-hand rule: if $\mathbf{a}$ points along the thumb and $\mathbf{b}$ along the forefinger of the right hand, then $\mathbf{n}$ and thus $\mathbf{c}$ point along the middle finger.

The cross product is anticommutative, i.e., $\mathbf{a} \times \mathbf{b} = -(\mathbf{b} \times \mathbf{a})$ holds for all vectors $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^3$. It is also bilinear, i.e., $(\lambda\mathbf{a} + \mu\mathbf{b}) \times \mathbf{c} = \lambda(\mathbf{a} \times \mathbf{c}) + \mu(\mathbf{b} \times \mathbf{c})$ holds for all λ and $\mu \in \mathbb{R}$ and for all vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c} \in \mathbb{R}^3$. Furthermore, two vectors $\mathbf{a} \neq 0$ and $\mathbf{b} \neq 0$ are parallel if and only if $\mathbf{a} \times \mathbf{b} = 0$. Finally, the Lagrangian identity $||\mathbf{a} \times \mathbf{b}||^2 = ||\mathbf{a}||^2||\mathbf{b}||^2 - (\mathbf{a} \cdot \mathbf{b})^2$ holds for all vectors $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^3$. The cross product is not associative, i.e., in general $\mathbf{a}\times(\mathbf{b}\times\mathbf{c}) \neq (\mathbf{a}\times\mathbf{b})\times\mathbf{c}$.

We can use the three defining properties to determine the cross products $\mathbf{e}_i \times \mathbf{e}_j$ for all i and $j \in \{1, 2, 3\}$ of all combinations of vectors in the standard basis $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$. Then multiplying out the product $(a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + a_3\mathbf{e}_3)\times(b_1\mathbf{e}_1 + b_2\mathbf{e}_2 + b_3\mathbf{e}_3)$ and using the bilinearity yields the formula

$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} = \begin{pmatrix} \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix} \\ -\begin{vmatrix} a_1 & b_1 \\ a_3 & b_3 \end{vmatrix} \\ \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} \end{pmatrix}.$$

In JULIA, the function `cross` calculates the cross product of two three-dimensional vectors.

```
julia> cross([1, 0, 0], [0, 1, 0])
3-element Vector{Int64}:
 0
 0
 1
```

8.4.7 The Determinant

The geometric meaning of the determinant $\det(A)$ of a square matrix $A \in F^{n\times n}$ is that it is equal to the signed volume of the n-dimensional parallelepiped spanned by the column or the row vectors of the matrix. It is positive when the corresponding linear function preserves the orientation of the vector space and negative otherwise. A matrix is singular, i.e., it cannot be inverted, if and only if its determinant is zero; conversely, a matrix is regular, i.e., it can be inverted, if and only if its determinant is non-zero. Determinants occur in different contexts in mathematics and have many important properties. In JULIA, the determinant of a matrix is calculated by the function `LinearAlgebra.det`.

8.4.8 Linear Systems

Systems of linear equations can always be written in the form

$$A\mathbf{x} = \mathbf{b}, \tag{8.6}$$

where the matrix $A \in F^{n \times m}$ and the vector $\mathbf{x}$ contains the unknowns x_i. Linear systems are among the most common equations to be solved. Linear systems also arise from the linearization of systems of nonlinear equations; the linearized system is then used as an approximation of the more complicated and usually harder to solve nonlinear system. Because linear systems are ubiquitous, JULIA provides advanced algorithms for solving them.

If you only remember one JULIA function from this section, it should be the backslash function `\`. It usually solves a linear system very reliably.

```
julia> A = [1 2; 4 5]; b = [15; 42];
julia> x = A \ b
2-element Vector{Float64}:
 3.0
 6.0
julia> A * x == b
true
julia> A * x
2-element Vector{Float64}:
 15.0
 42.0
```

The rest of this section is concerned with what happens under the hood.

8.4.8.1 Solvability

Cramer's rule, an explicit formula for solving systems of linear equations with an equal number n of equations and unknowns, has been known since the mid-18th century. It is named after Gabriel Cramer, who published the rule for systems of arbitrary size in 1750. Cramer's rule is based on determinants and its naive implementation has a time complexity of $O((n+1)n!)$, although it can be implemented with a time complexity of $O(n^3)$ [2].

However, before we solve linear systems, it is important to consider their solvability. There may be no solution, a unique solution, or multiple solutions; if the underlying field F of the preimage vector space is infinite, the third case of multiple solutions means that there are infinitely many solutions. In the following, we assume that the underlying field F is $\mathbb{R}$ or $\mathbb{C}$.

A system with fewer equations than unknowns is called an underdetermined system. In general, such a system has infinitely many solutions, but it may have no solution. A system with the same number of equations and unknowns usually has a unique solution. A system with more equations than unknowns is called an overdetermined system. In general, such a system has no solution.

The reasons why a certain system may behave differently from the general case is that the equations may be linearly dependent, i.e., one or more equations may be redundant, or that two or more of the equations may be inconsistent, i.e., contradictory.

Equations of the form (8.6) can be interpreted – as all of linear algebra – within the context of linear functions f and within the context of matrices A. There is also a geometric interpretation: each linear equation (or row) in (8.6) determines a hyperplane in F^m and the set of solutions is the intersection of these hyperplanes.

To characterize the three cases for the number of solutions that can occur, it is useful to start with homogeneous systems. A system of linear equations is called homogeneous if the constant terms in each equation vanish, i.e., if (8.6) has the form

$$A\mathbf{x} = 0. \tag{8.7}$$

Each homogeneous equation has at least one solution, namely the trivial solution $\mathbf{x} = 0$.

If the matrix A is regular, which is equivalent to f being a bijection, then the trivial solution is the unique solution; the set of solutions is the kernel $\ker(A) = \{0\}$.

If the matrix A is singular, the set of solutions is the kernel $\ker(A)$ and it contains infinitely many solutions. It is straightforward to see that if $\mathbf{x}$ and $\mathbf{y}$ are two solutions, then the linear combination $\alpha\mathbf{x} + \beta\mathbf{y}$ is a solution as well. This implies that the set of solutions is a linear subspace of F^m.

A linear function f can only be a bijection if the dimensions m and n of the preimage and image spaces are equal. Therefore a regular matrix A must be a square matrix. A square matrix A is regular if and only if $\det(A) \neq 0$.

An example of a singular matrix is the following.

```
julia> A = [1 0; 1 0]; det(A)
0.0
julia> rank(A)
1
julia> nullspace(A)
2×1 Matrix{Float64}:
 0.0
 1.0
```

An example of a regular matrix is the following.

```
julia> A = [1 2; 3 4]; det(A)
-2.0
julia> rank(A)
2
julia> nullspace(A)
2×0 Matrix{Float64}
```

With this knowledge about the solution set of homogeneous systems (8.7), we can now consider general, inhomogeneous systems (8.6). An inhomogeneous system has (at least) one solution if the inhomogeneity $\mathbf{b}$ lies in the image of f or A, i.e., if $\mathbf{b} \in \operatorname{im}(A)$. If $\mathbf{z} \in F^m$ is any particular solution of (8.6), then all solutions of (8.6) are given by the set

$$\mathbf{z} + \ker(A) = \{\mathbf{z} + \mathbf{v} \mid \mathbf{v} \in \ker(A)\}.$$

Geometrically, this means that the solution set of an inhomogeneous system is a translation (by a particular solution $\mathbf{z}$) of the solution set of the corresponding homogeneous equation, which is $\ker(A)$.

These facts underline the importance of the two formulations of the rank-nullity theorem, Theorem 8.2 and Theorem 8.3. It is important to note that most of the algorithms we will encounter in the following yield the rank or the nullity of a matrix as an important byproduct so that it is often not necessary and even inefficient to calculate them separately.

In the following, algorithms for solving two important types of linear systems are discussed in some detail. The two types are square linear systems and overdetermined linear systems. Underdetermined linear systems are of less practical importance.

8.4.8.2 Square Linear Systems

In this section, we consider the important special case when the system matrix A in (8.6) is square, i.e., $A \in F^{n\times n}$. In other words, the preimage and the image spaces have the same dimension n. A square linear system has the form

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1,\\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2,\\
&\vdots\\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n.
\end{aligned}$$

The next theorem records basic facts about regular matrices.

Theorem 8.4 (regular matrices) *The following statements about a matrix $A \in F^{n\times n}$ are equivalent.*

1. *The linear function associated with A is bijective.*
2. *A is regular.*
3. *The inverse matrix A^{-1} of A exists and $(A^{-1})^{-1} = A$ holds.*
4. $\det(A) \neq 0$.
5. *The equation $A\mathbf{x} = \mathbf{0}$ implies that $\mathbf{x} = \mathbf{0}$.*
6. *The equation $A\mathbf{x} = \mathbf{b}$ has a unique solution $\mathbf{x} = A^{-1}\mathbf{b}$ for all $\mathbf{b} \in F^n$.*

Furthermore, the inverse matrix A^{-1} always commutes with A, i.e., $AA^{-1} = A^{-1}A = I$ holds.

It is important to note that inverse matrices are usually not calculated explicitly when solving a system of linear equations – unless the system is very small –, since there are much more efficient computational alternatives. On the other hand, if we have a fast algorithm for solving a square linear system, we can always calculate the inverse A^{-1} explicitly if desired by solving the equations $A\mathbf{x}_i = \mathbf{e}_i$ and collecting the solutions $\mathbf{x}_i$ columnwise into a matrix because of $AA^{-1} = I$.

We start with triangular matrices. If the square linear system has the special form

$$L\mathbf{x} = \mathbf{b},$$

where L is a lower-triangular matrix, then the elements of the solution $\mathbf{x}$ are found as

$$x_i := \frac{b_i - \sum_{j=1}^{i-1} l_{ij}x_j}{l_{ii}} \qquad \forall i \in \{1, \dots, n\}$$

starting with the first equation. This algorithm is called forward substitution.

Analogously, if the system has the special form

$$U\mathbf{x} = \mathbf{b},$$

where U is an upper-triangular matrix, then the elements of the solution $\mathbf{x}$ are found as

$$x_i := \frac{b_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}} \qquad \forall i \in \{n, \dots, 1\}$$

starting with the last equation. This algorithm is called backward substitution.

Both forward and backward substitution fail if and only if one of the diagonal elements is zero, but this is equivalent to the system matrix L or U being singular.

In the next step we use Gaussian elimination to factor the matrix A into $A = P^\top LU$. This factorization is called LU factorization; the permutation matrix P is only a complication of the basic idea. LU factorization then immediately results in an algorithm for solving the linear system

$$A\mathbf{x} = P^\top L \underbrace{U\mathbf{x}}_{=:\mathbf{y}} = \mathbf{b}.$$

Algorithm 8.5 (solve linear system)

1. Factor A into $A = P^\top LU$ by Gaussian elimination.
2. Solve $P^\top L\mathbf{y} = \mathbf{b}$, which is equivalent to $L\mathbf{y} = P\mathbf{b}$, for $\mathbf{y}$ using forward substitution.
3. Solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ using backward substitution.

But how can we factor the matrix A into $A = LU$ using Gaussian elimination? The idea of Gaussian elimination is that row operations are used on the augmented matrix $\hat{A} := (A, \mathbf{b})$ such that the matrix A becomes an upper triangular matrix. The row operations that yield an equivalent system are operations on the equations of the system: it is possible

1. to multiply a row or equation by a non-zero scalar,
2. to add a multiple of a row or equation to another one, and
3. to swap the position of two rows or equations.

Algorithm 8.6 (Gaussian elimination, LU factorization)

1. Set $j := 1$. The counter j indicates the current column.
2. Repeat:

   a. Find the row index i of the element in $\{a_{jj}, a_{j+1,j}, \dots, a_{nj}\}$ with the largest absolute value, i.e.,

   $$i := \operatorname*{arg\,max}_{k \in \{j,\dots,n\}} |a_{kj}|.$$

   This element a_{ij} is called the pivot element. If the pivot element is equal to zero, the matrix A is singular and the algorithm stops.
   Swap the i-th and the j-th rows such that the pivot element is now located at a_{jj} in the j-th row. Record swapping the two rows as left-multiplication by a permutation matrix P_j.

   b. For all rows $i \in \{j + 1, \dots, n\}$ below the pivot element, add the pivot row multiplied by $-a_{ij}/a_{jj}$ to the i-th row such that all matrix elements below the pivot element vanish.
   Record these row operations by left-multiplication with a lower-triangular matrix L_j.

   c. Increase j by one and repeat while $j \le n$.

Swapping the rows in the second step is called row pivoting or partial pivoting. Choosing the element with the largest absolute value improves the numerical stability of the algorithm, although other variants are used as well.

If it is not possible to find a non-zero pivot element in the second step, we have shown that the matrix is singular. In fact, Gaussian elimination is a decision procedure for inverting a square matrix A: if it succeeds, the inverse matrix A has been calculated; if it does not succeed, it has constructed a proof that the matrix is singular.

The following theorem states that Gaussian elimination indeed yields a factorization of the desired shape.

Theorem 8.7 (*LU* factorization) *Gaussian elimination of a regular matrix A yields a factorization that is of the form*

$$A = P^\top LU.$$

Proof Gaussian elimination performed on a regular matrix A yields the equation

$$L_n P_n \cdots L_1 P_1 A = U,$$

where the matrices L_j and P_j record the row operations performed and the right-hand side is upper triangular by construction.

The lower-triangular matrices L_j record adding the j-th row multiplied by λ_{ij} all to i-th rows, where $i > j$, and therefore they have the form

$$L_j = I + \sum_{i=j+1}^{n} \lambda_{ij} \mathbf{e}_i \mathbf{e}_j^\top,$$

which is indeed lower triangular. The only non-zero element of the product $\mathbf{e}_i \mathbf{e}_j^\top$ is in row i and column j.

The permutation matrices P_j record swapping the i-th row with the j-th one, where $i > j$ again. It is straightforward to show that the inverse of a permutation matrix P is $P^{-1} = P^\top$.

The goal is to reorder the terms in the product $L_n P_n \cdots L_1 P_1$ such that it becomes $K_n \cdots K_1 P_n \cdots P_1$. This can be achieved by moving the matrices P_j to the right repeatedly by replacing the products $P_k L_j$ with $k > j$ by products $K_j P_k$. We have $K_j = P_k L_j P_k^\top$. Since $k > j$, multiplication of L_j on the right by $P_k^\top$ only swaps two columns whose only non-zero element is equal to one. Since $k > j$, multiplication of $L_j P_k^\top$ on the left by P_k swaps these two ones back into the main diagonal and leaves the structure of the matrix unchanged otherwise. Therefore K_j has the same, lower-triangular structure as L_j, and all the terms can be reordered such that

$$L_n P_n \cdots L_1 P_1 A = K_n \cdots K_1 P_n \cdots P_1 A = U.$$

The last equation yields

$$PA = (K_n \cdots K_1)^{-1} U = K_1^{-1} \cdots K_n^{-1} U.$$

We now show that the inverse of

$$K_j = I + \sum_{i=j+1}^{n} \kappa_{ij} \mathbf{e}_i \mathbf{e}_j^\top$$

is simply

$$K_j^{-1} = I - \sum_{i=j+1}^{n} \kappa_{ij} \mathbf{e}_i \mathbf{e}_j^\top.$$

Multiplying out the products $K_j K_j^{-1}$ and $K_j^{-1} K_j$ yields

$$\begin{aligned} K_j K_j^{-1} &= K_j^{-1} K_j \\ &= I + \sum_{k=j+1}^{n} \kappa_{kj} \mathbf{e}_k \mathbf{e}_j^\top - \sum_{l=j+1}^{n} \kappa_{lj} \mathbf{e}_l \mathbf{e}_j^\top + \Bigg(\sum_{k=j+1}^{n} \kappa_{kj} \mathbf{e}_k \mathbf{e}_j^\top\Bigg)\Bigg(\sum_{l=j+1}^{n} \kappa_{lj} \mathbf{e}_l \mathbf{e}_j^\top\Bigg) = I. \end{aligned}$$

The last of the four terms vanishes because of the inner products $\mathbf{e}_j^\top \mathbf{e}_l$ with $j \neq l$.

Therefore the inverses K_j^{-1} have the same lower-triangular structure as the matrices L_j. Since the product of lower-triangular matrices is again lower-triangular, we have thus shown that

$$PA = LU,$$

which completes the proof. □

LU factorization is also useful to compute the determinant $\det(A)$ of a square matrix, as the next theorem shows.

Theorem 8.8 (determinant by *LU* factorization) *The determinant of a regular square matrix A with LU factorization $A = P^\top LU$ is given by*

$$\det(A) = (-1)^p \Bigg(\prod_{i=1}^{n} l_{ii}\Bigg)\Bigg(\prod_{i=1}^{n} u_{ii}\Bigg),$$

where p is the number of row exchanges in the factorization.

Proof The formula follows from $\det(A) = \det(P)\det(L)\det(U)$. □

In Julia, *LU* factorization is implemented for dense and sparse matrices by the functions `LinearAlgebra.lu` and `LinearAlgebra.lu!`. In the case of a dense matrix, the components of the factorization are the fields `L` for the lower-triangular matrix, `U` for the upper-triangular matrix, `P` for the right permutation matrix, and `p` for the right permutation vector, which is a space saving and convenient representation of the permutation.

```
julia> A = randn(4, 4); f = lu(A);
julia> f.L * f.U - A[f.p, :]
4×4 Matrix{Float64}:
 0.0          0.0           0.0          0.0
 0.0          0.0          -1.11022e-16  0.0
 2.22045e-16  0.0           8.32667e-17  0.0
 0.0          1.11022e-16  -1.66533e-16  0.0
```

Furthermore, the resulting factorization can be used as an argument to the functions `/`, `\`, `det`, `inv`, `logdet`, `logabsdet`, and `size`.

```
julia> size(f)
(4, 4)
julia> det(f)
4.504801047286308
julia> A * inv(f)
4×4 Matrix{Float64}:
  1.0          7.05739e-17  1.25627e-17  -1.0213e-16
  3.81014e-17  1.0          1.23241e-16   1.6374e-17
 -2.50489e-16  3.05704e-17  1.0           1.67098e-16
  3.03655e-16  1.52831e-16  1.36405e-16   1.0
```

The functions `LinearAlgebra.lu` and `LinearAlgebra.lu!` also work on sparse matrices. Then the additional fields are `q` for the left permutation vector, and `Rs` for the vector of scaling factors.

```
julia> A = sprandn(4, 4, 1/2); f = lu(A);
julia> f.L * f.U - (f.Rs .* A)[f.p, f.q]
4×4 SparseMatrixCSC{Float64, Int64} with 0 stored entries
```

Choosing the element with the maximal absolute value as the pivot element is important for the precision of the algorithm. To see this, we consider the example

$$A := \begin{pmatrix} \epsilon & 1 \\ 1 & -1 \end{pmatrix}, \qquad \mathbf{b} := \begin{pmatrix} 1+\epsilon \\ 0 \end{pmatrix}$$

with $\epsilon \ll 1$. If we choose $a_{11} = \epsilon$ as the pivot element, already in place, then Gaussian elimination results in

$$\begin{pmatrix} \epsilon & 1 & 1+\epsilon \\ 0 & -1-1/\epsilon & -\frac{1+\epsilon}{\epsilon} \end{pmatrix}.$$

The second row yields $x_2 = 1$, and the first row yields

$$x_1 = \frac{(1+\epsilon)-1}{\epsilon}.$$

Symbolic evaluation of this expression results in the correct solution $\mathbf{x} = (1,1)^\top$. Numerical evaluation, however, requires to subtract 1 from $1+\epsilon$, which are two close numbers. This leads to cancellation and the loss of digits in floating-point arithmetic. The problem is aggravated by the division by ϵ. This effect makes the solution unstable.

```
julia> e = 1.5*eps(1.0); @show e; @show (1+e)-1; @show ((1+e)-1)/e;
e = 3.3306690738754696e-16
(1 + e) - 1 = 4.440892098500626e-16
((1 + e) - 1) / e = 1.3333333333333333
```

The function `eps` returns the epsilon of the given floating-point type, which is defined as the gap between `1` and the next largest value representable by this type.

On the other hand, row pivoting yields

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1+\epsilon & 1+\epsilon \end{pmatrix}$$

and hence $x_2 = (1+\epsilon)/(1+\epsilon) = 1$ and

$$x_1 = \frac{0-(-1)}{1}.$$

In this quotient, no such problem occurs.

An special type of matrices of importance is the following.

Definition 8.9 (positive-definite matrix) A Hermitian matrix $A \in F^{n\times n}$ is called *positive definite* if

$$\mathbf{x}^* A\mathbf{x} > 0 \qquad \forall \mathbf{x} \in F^n \backslash \{\mathbf{0}\}$$

holds.

It is easy to construct positive-definite matrices. If $A \in \mathbb{C}^{n\times n}$ is regular, then A^*A is a positive-definite matrix. A^*A is obviously Hermitian; furthermore, $\mathbf{x}^*A^*A\mathbf{x} = \|A\mathbf{x}\|_2^2 > 0$, since $A\mathbf{x} \neq 0$ due to the regularity of A.

If A is a positive-definite matrix, then Cholesky factorization, which is a specialization of LU factorization for this type of matrices, can be used and is roughly twice as efficient as LU factorization.

Theorem 8.10 (Cholesky factorization) *A matrix $A \in F^{n\times n}$ is positive definite if and only if there exists a unique lower-triangular matrix L with real and strictly positive diagonal elements such that*

$$A = LL^*,$$

which is called the Cholesky factorization of A.

Cholesky factorization is a decision procedure for positive definiteness: it either succeeds in constructing such a factorization or it shows that the matrix is not positive definite.

In JULIA, Cholesky factorization is implemented by the functions `LinearAlgebra.cholesky` and `LinearAlgebra.cholesky!` analogously to `LinearAlgebra.lu`. Testing whether a matrix is Hermitian is implemented by `LinearAlgebra.ishermitian`, and testing for positive definiteness is implemented by the functions `LinearAlgebra.isposdef` and `LinearAlgebra.isposdef!`.

```
julia> A = [1 2; 2 1]; ishermitian(A), isposdef(A)
(true, false)
julia> A = [2 -1+1im; -1-1im 2]; ishermitian(A), isposdef(A)
(true, true)
julia> f = cholesky(A); f.L * f.U - A
2×2 Matrix{ComplexF64}:
 4.44089e-16+0.0im  0.0+0.0im
         0.0+0.0im  0.0+0.0im
```

8.4.8.3 Overdetermined Systems

Overdetermined systems of linear equations are systems of the form (8.6) where

$$n > m$$

holds for the system matrix $A \in F^{n\times m}$, i.e., the number n of equations is larger than the number m of unknowns. The unknown vector $\mathbf{x}$ is an element of F^m and the inhomogeneity $\mathbf{b}$ is an element of F^n. In general, overdetermined systems do not have a solution, since the equations are contradictory.

Instead of solving the system, it is expedient to minimize a norm of the residuum $\mathbf{b} - A\mathbf{x}$, i.e., to find

$$\operatorname*{arg\,min}_{\mathbf{x}\in F^m} ||\mathbf{b} - A\mathbf{x}||.$$

Any choice of norm is possible. If the infinity norm is chosen, then this type of problem is often called a minimax problem. In the case of the 2-norm, the problem is called a linear least-squares problem. The name stems from the form

$$||\mathbf{b} - A\mathbf{x}||_2^2 = \sum_{i=1}^{m} |b_i - (A\mathbf{x})_i|^2 \tag{8.8}$$

of the 2-norm of the residuum.

This problem has a long history and is of great importance for regression or fitting functions to data. The choice of the 2-norm is also distinguished by its relationship to linear systems. Since the expression in (8.8) to be minimized is quadratic, its derivative is linear and vanishes at a minimum. This fact results in the close relationship between least-squares problems and linear systems, as we will see in more detail.

From now on, we use the 2-norm and introduce the notation

$$A\mathbf{x} \approx \mathbf{b}$$

for the linear least-squares problem

$$\arg\min_{\mathbf{x}\in F^m} \|\mathbf{b} - A\mathbf{x}\|_2.$$

Regression or data-fitting problems result in overdetermined linear systems as follows. Suppose that there are n measurements $(b_1, \dots, b_n)$ of a dependent variable $b(t)$ at n values $(t_1, \dots, t_n)$ of the independent variable t. We want to model the dependent variable $b(t)$ as a linear combination

$$b(t) = x_1 f_1(t) + \cdots + x_m f_m(t) \tag{8.9}$$

of certain m functions $(f_1(t), \dots, f_m(t))$ with the coefficients $(x_1, \dots, x_m)$. Since we know the values $b_i = b(t_i)$, we obtain the equations

$$b(t_i) = x_1 f_1(t_i) + \cdots + x_m f_m(t_i) = b_i \qquad \forall i \in \{1, \dots, n\},$$

which can be written as the system

$$\begin{pmatrix} f_1(t_1) & \cdots & f_m(t_1) \\ \vdots & & \vdots \\ f_1(t_n) & \cdots & f_m(t_n) \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}.$$

After setting

$$a_{ij} := f_j(t_i),$$

we have thus found a linear system $A\mathbf{x} = \mathbf{b}$ or a linear least-squares problems $A\mathbf{x} \approx \mathbf{b}$.

Furthermore, substitutions can be used to write nonlinear relationships between the dependent and independent variables in the form (8.9) of a linear combination. For example, taking the logarithm of both sides of the power law $c(s) := \alpha s^\beta$, we find $\ln c_i = \ln \alpha + \beta \ln s_i$ and hence define $t_i := \ln s_i$ and $b_i := \ln c_i$. This yields the linear relationship $b_i = \ln \alpha + \beta t_i$ and the two functions $f_1(t) := 1$ and $f_2(t) := t$. Then $x_1 = \ln \alpha$ and $x_2 = \beta$.

The following theorem gives the already expected relationship between linear least-squares problems and linear systems.

Theorem 8.11 (least-squares problem) *If* $\mathbf{x} \in F^m$ *solves the linear system*

$$A^*(A\mathbf{x} - \mathbf{b}) = \mathbf{0}, \tag{8.10}$$

then $\mathbf{x}$ *solves the least-squares problem* $A\mathbf{x} \approx \mathbf{b}$.

Proof (elementary) Let $\mathbf{y} \in F^m$ be an arbitrary vector. Then the inequality

$$\begin{aligned} \|A(\mathbf{x} + \mathbf{y}) - \mathbf{b}\|_2^2 &= ((A\mathbf{x} - \mathbf{b}) + A\mathbf{y})^*((A\mathbf{x} - \mathbf{b}) + A\mathbf{y}) \\ &= (A\mathbf{x} - \mathbf{b})^*(A\mathbf{x} - \mathbf{b}) + 2(A\mathbf{y})^*(A\mathbf{x} - \mathbf{b}) + (A\mathbf{y})^*(A\mathbf{y}) \\ &= \|A\mathbf{x} - \mathbf{b}\|_2^2 + 2\mathbf{y}^* \underbrace{A^*(A\mathbf{x} - \mathbf{b})}_{=\mathbf{0}} + \|A\mathbf{y}\|_2^2 \\ &\geq \|A\mathbf{x} - \mathbf{b}\|_2^2 \end{aligned}$$

holds. Here we have used the fact that if $\mathbf{u}$ and $\mathbf{v}$ are two vectors, then the equation $(\mathbf{u} + \mathbf{v})^*(\mathbf{u} + \mathbf{v}) = \mathbf{u}^*\mathbf{u} + 2\mathbf{v}^*\mathbf{u} + \mathbf{v}^*\mathbf{v}$ holds due to $\mathbf{u}^*\mathbf{v} = \mathbf{v}^*\mathbf{u}$ being a scalar.

The inequality shows that any other vector $\mathbf{x} + \mathbf{y}$ cannot be a solution of the least-squares problem. □

Proof (using calculus) The gradient of the expression to be minimized is

$$\nabla_{\mathbf{x}} \|\mathbf{b} - A\mathbf{x}\|_2^2 = \nabla_{\mathbf{x}} \left(\sum_{i=1}^{n} \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right)^2 \right).$$

The k-th element of the gradient is

$$\partial_{x_k} = -2 \sum_{i=1}^{n} a_{ik} \left(b_i - \sum_{j=1}^{n} a_{ij} x_j \right),$$

which yields

$$\nabla_{\mathbf{x}} \|\mathbf{b} - A\mathbf{x}\|_2^2 = 2A^*(A\mathbf{x} - \mathbf{b}) = \mathbf{0}.$$

The last equation holds due to (8.10). □

The condition (8.10) is equivalent to

$$A^* A\mathbf{x} = A^*\mathbf{b}. \tag{8.11}$$

The matrix A^*A on the left-hand side is $(m \times m)$-dimensional, and therefore this system is a square linear system for $\mathbf{x}$. These equations are called the normal equations. This form of the condition (8.10) motivates the following definition.

Definition 8.12 (pseudoinverse) The matrix

$$A^+ := (A^*A)^{-1}A^* \in F^{m \times n}$$

is called the *pseudoinverse* of $A \in F^{n \times m}$.

With this definition, we can at least formally write the solution of the least-squares problem $A\mathbf{x} \approx \mathbf{b}$ as

$$\mathbf{x} = A^+\mathbf{b}.$$

It is straightforward to see that A^*A is Hermitian. In general, A^*A is not regular, however. The properties of the pseudoinverse are studied in more detail in Sect. 8.4.8.4.

Every matrix $A \in F^{n \times m}$ can also be factored as

$$A = QR,$$

where $Q \in F^{n \times n}$ is an orthogonal matrix and $R \in F^{m \times n}$ is an upper-triangular matrix. This factorization is called QR factorization. Before discussing its details, we define orthogonal vectors and matrices.

Definition 8.13 (orthogonal and orthonormal vectors) A set $\{\mathbf{q}_1, \dots, \mathbf{q}_n\}$ of vectors is called *orthogonal*, if

$$\forall i \in \{1, \dots, n\}: \quad \forall j \in \{1, \dots, n\}: \quad i \neq j \implies \mathbf{q}_i^* \mathbf{q}_j = 0$$

holds. If $\mathbf{q}_i^* \mathbf{q}_i = 1$ additionally holds for all $i \in \{1, \dots, n\}$, the vectors are called *orthonormal*.

Definition 8.14 (orthogonal matrix) A square matrix $Q \in \mathbb{R}^{n \times n}$ is called *orthogonal* if its columns are orthonormal vectors.

Definition 8.15 (unitary matrix) A square matrix $Q \in \mathbb{C}^{n \times n}$ is called *unitary* if its columns are orthonormal vectors.

Theorem 8.16 (inverse of orthogonal/unitary matrix) *The inverse of an orthogonal/unitary matrix Q is $Q^{-1} = Q^*$.*

Proof By definition, $Q^*Q = I$ and $QQ^* = I$. Therefore the left and right inverses of Q are equal to Q^*. □

Theorem 8.17 (orthogonal matrix) *A matrix $Q \in F^{n \times m}$ is orthogonal if and only if*

$$\|Q\mathbf{x}\|_2 = \|\mathbf{x}\|_2 \qquad \forall \mathbf{x} \in F^m$$

holds.

Geometrically, this property means that an orthogonal matrix represents a linear isometry. An isometry is a function that does not change the lengths of its arguments.

Orthogonal vectors are important both theoretically and computationally. In theoretic calculations, they have the convenient property that their inner products $\mathbf{q}_i^* \mathbf{q}_j$ vanish (if $i \neq j$), which implies that norms of their difference are calculated easily as

$$\|\mathbf{q}_i - \mathbf{q}_j\|^2 = \|\mathbf{q}_i\|^2 + \|\mathbf{q}_j\|^2 \tag{8.12}$$

if $i \neq j$. In computations, orthogonality has the advantageous property that it avoids multidimensional subtractive cancellation. If $\mathbf{x} \approx \mathbf{y}$, then $\|\mathbf{x} - \mathbf{y}\| \ll \|\mathbf{x}\|$ and $\|\mathbf{x} - \mathbf{y}\| \ll \|\mathbf{y}\|$ and subtractive cancellation may occur. Orthogonal vectors cannot suffer from this problem due to (8.12). In other words, a basis consisting of orthogonal vectors is expedient for computations.

The importance of QR factorization stems from the fact that it yields an orthogonal basis Q and a basis change R in convenient upper-triangular form.

QR factorizations can be calculated by Gram–Schmidt orthogonalization, by Householder transformations, and by Givens rotations. These methods have their advantages and disadvantages. The advantage of the Gram–Schmidt algorithm is that it is implemented easily, while its disadvantage is that it is numerically unstable. Givens rotations can be parallelized better than the other methods, but their implementation is more involved. Householder transformations cannot be parallelized, but they are the simplest of the numerically stable *QR* factorization algorithms. We therefore discuss Householder transformations, also called Householder reflections, for *QR* factorization in the following.

The defining properties of a Householder transformation or reflection are that it is represented by an orthogonal/unitary matrix and that it maps a vector $\mathbf{x}$ to a vector whose only non-zero element is the first one, i.e.,

$$P\mathbf{x} = \begin{pmatrix} \pm\|\mathbf{x}\|_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

The first element of $P\mathbf{x}$ must be $\pm\|\mathbf{x}\|_2$ because of Theorem 8.17. The Householder reflection P reflects through the line bisecting the angle between $\mathbf{x}$ and $\mathbf{e}_1$. The advantageous numerical property is that the maximum bisected angle is 45°. On the other hand, orthogonal projection of the vector $\mathbf{x}$ onto $\mathbf{e}_1$ as used in the Gram–Schmidt algorithm is numerically unstable whenever $\mathbf{x}$ and $\mathbf{e}_1$ are approximately orthogonal/unitary.

The following two theorems show how to construct Householder reflections in the real case $F = \mathbb{R}$ and in the complex case $F = \mathbb{C}$.

Theorem 8.18 (Householder reflection ($F = \mathbb{R}$)) *Suppose* $\mathbf{x} \in \mathbb{R}^n$ *and choose an* $\alpha \in \mathbb{R}$ *such that* $|\alpha| = \|\mathbf{x}\|_2$. *Define* $\mathbf{u} := \mathbf{x} - \alpha\mathbf{e}_1$ *and*

$$P := \begin{cases} I - \dfrac{2}{\mathbf{u}^\top\mathbf{u}}\mathbf{u}\mathbf{u}^\top, & \mathbf{u} \neq \mathbf{0}, \\ I, & \mathbf{u} = \mathbf{0}. \end{cases}$$

Then $P \in \mathbb{R}^{n\times n}$ *is symmetric and orthogonal, and the equation* $P\mathbf{x} = \alpha\mathbf{e}_1$ *holds.*

Theorem 8.19 (Householder reflection ($F = \mathbb{C}$)) *Suppose* $\mathbf{u} \in \mathbb{C}^n$ *and define* $\alpha := -\mathrm{e}^{\mathrm{i}\arg u_k}\|\mathbf{u}\|$. *Define* $\mathbf{u} := \mathbf{x} - \alpha\mathbf{e}_1$, $\mathbf{v} := \mathbf{u}/\|\mathbf{u}\|_2$ *unless* $\mathbf{u} = \mathbf{0}$, *and*

$$P := \begin{cases} I - \left(1 + \dfrac{\mathbf{x}^*\mathbf{v}}{\mathbf{v}^*\mathbf{x}}\right)\mathbf{v}\mathbf{v}^*, & \mathbf{u} \neq \mathbf{0}, \\ I, & \mathbf{u} = \mathbf{0}. \end{cases}$$

Then $P \in F^{n\times n}$ *is Hermitian and unitary, and the equation* $P\mathbf{x} = \alpha\mathbf{e}_1$ *holds.*

When using floating-point numbers, the scalar α in the real case in Theorem 8.18 should be chosen so that it has the opposite sign to the k-th coordinate u_k of $\mathbf{u}$, where u_k is the pivot element in Theorem 8.21, in order to avoid cancellation. The choice of α in Theorem 8.19 also achieves this in the complex case [3, Section 4.7].

We can now state the algorithm and prove the theorem for *QR* factorization. Since several Householder reflections are used, we use a more precise notation now and denote the Householder reflection for the vector $\mathbf{x}$ by $P(\mathbf{x})$.

Algorithm 8.20 (QR factorization)

1. Set $j := 0$ and $R_0 := A$.
2. Repeat:

   a. Increase j.
   b. Let $\mathbf{x}_j$ be the j-th column of R_{j-1} rows j to n, i.e., $\mathbf{x}_j := R_{j-1}[j : n, j]$. (Here the notation $j : n$ indicates the integers $j, j+1, \ldots, n$.) Set $P_j := P(\mathbf{x}_j)$, the Householder reflection constructed for $\mathbf{x}_j$ as in Theorem 8.18 or 8.19.
   c. Set
   $$Q_j := \begin{pmatrix} I_{j-1} & \mathbf{0}^* \\ \mathbf{0} & P_j \end{pmatrix}.$$
   The matrix P_j has size $(n+1-j) \times (n+1-j)$, and I_{j-1} is the identity matrix of size $(j-1) \times (j-1)$. Therefore Q_j has size $n \times n$.
   d. Set $R_j := Q_j R_{j-1}$. The elements of the matrix R_j in the first j columns below the main diagonal are all zero because of the actions of the Householder reflections P_k, $k \le j$.
   e. Repeat while $j < \min(n-1, m) =: N$. If $m < n$, i.e., the matrix A is tall, then a Householder reflection for each of the m columns must be used, resulting in m steps. If $m \ge n$ on the other hand, then $n-1$ Householder reflections are necessary to zero all elements below the main diagonal, which comprises n elements.

3. After the loop, the product
$$Q_N \cdots Q_1 A = R_N =: R$$
is upper triangular by construction. Set
$$Q := Q_1^* \cdots Q_N^*.$$
The matrix Q is orthogonal/unitary because of Problem 8.11 and Problem 8.12. Because of Theorem 8.16, we have
$$A = Q_1^* \cdots Q_N^* R = QR.$$

This algorithm always computes the factorization, which implies the following theorem.

Theorem 8.21 (QR factorization) *Every matrix $A \in F^{n\times m}$, $F \in \{\mathbb{R}, \mathbb{C}\}$, can be factored as*
$$A = QR,$$
where $Q \in F^{n\times n}$ is orthogonal/unitary and $R \in F^{n\times m}$ is upper triangular.

In JULIA, QR factorization is implemented by the two functions `LinearAlgebra.qr` and `LinearAlgebra.qr!`, whose optional argument `pivot` indicates if pivoting is to be used; it is not used by default. These functions behave similarly to `LinearAlgebra.lu` and `LinearAlgebra.lu!` and return an object that contains the components of the factorization. The field `Q` contains the orthogonal/unitary matrix Q, `R` contains the upper triangular matrix R, and `p` and `P` contain the permutation vector and matrix, respectively.

```
julia> A = randn(3, 4); f = qr(A);
julia> f.Q * f.R - A
3×4 Matrix{Float64}:
 -1.11022e-16  8.88178e-16  -4.44089e-16  -1.11022e-16
  0.0          2.22045e-16  -2.22045e-16   0.0
  0.0          2.22045e-16  -2.77556e-17   0.0
```

Furthermore, the resulting factorization can be used as an argument to the functions `inv`, `size`, and `\`. If the matrix A is not square, then the function `\` returns the least-squares solution with minimal norm.

```
julia> A \ [1; 2; 3]
4-element Vector{Float64}:
 -0.9553736238383352
  1.4629912433582717
  0.4012402900419049
  0.016950787356409203
```

These two functions also works on sparse matrices. In this case, row and column permutations are provided in the fields `prow` and `pcol` such that the number of non-zero entries is reduced.

```
julia> A = sprandn(4, 4, 1/2); f = qr(A);
julia> f.Q * f.R - A[f.prow, f.pcol]
4×4 SparseMatrixCSC{Float64, Int64} with 0 stored entries
```

QR factorization is useful to solve least-squares problems $A\mathbf{x} \approx \mathbf{b}$. In overdetermined systems, i.e., when $n > m$, QR factorization yields a factorization of the form

$$A = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_{n-1} & \mathbf{q}_n \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ 0 & r_{22} & \cdots & r_{2m} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{mm} \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix},$$

where the columns of the matrix Q are denoted by the vectors $\mathbf{q}_j$. We notice that the vectors $\mathbf{q}_{m+1}, \dots, \mathbf{q}_n$ are always multiplied by zero. Neglecting the superfluous parts, the product becomes

$$A = \begin{pmatrix}\mathbf{q}_1 & \mathbf{q}_2 & \cdots & \mathbf{q}_{m-1} & \mathbf{q}_m\end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ 0 & r_{22} & \cdots & r_{2m} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{mm} \end{pmatrix} =: \tilde{Q}\tilde{R}, \qquad (8.13)$$

where the columns $\mathbf{q}_j$, $j \in \{1, \ldots, m\}$, of $\tilde{Q} \in F^{n\times m}$ are still orthonormal and $\tilde{R} \in F^{n\times n}$ is upper triangular. While $\tilde{Q}^*\tilde{Q} = I_m$, in general $\tilde{Q}\tilde{Q}^* \neq I_n$.

To solve the least-squares problem $A\mathbf{x} \approx \mathbf{b}$, we solve the normal equations (8.11). Using (8.13), they become

$$\begin{aligned} A^*A\mathbf{x} &= A^*\mathbf{b}, \\ \tilde{R}^*\tilde{Q}^*\tilde{Q}\tilde{R}\mathbf{x} &= \tilde{R}^*\tilde{Q}^*\mathbf{b}, \\ \tilde{R}^*\tilde{R}\mathbf{x} &= \tilde{R}^*\tilde{Q}^*\mathbf{b}. \end{aligned}$$

If A has full rank, then $\tilde{R}$ is regular. Therefore $\tilde{R}^*$ is regular as well and the last equation becomes

$$\tilde{R}\mathbf{x} = \tilde{Q}^*\mathbf{b}.$$

Thus we can find $\mathbf{x}$ by calculating $\tilde{Q}$ and $\tilde{R}$ and then using backward substitution (see Sect. 8.4.8.2).

8.4.8.4 The Moore–Penrose Pseudoinverse

A pseudoinverse of a matrix A is generalization of the inverse matrix. The most common pseudoinverse is the Moore–Penrose pseudoinverse, which we will refer to as the pseudoinverse here. If $A \in F^{n\times m}$, then the pseudoinverse $A^+ \in F^{m\times n}$ is defined as the matrix that satisfies the four conditions

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^* &= AA^+, \\ (A^+A)^* &= A^+A. \end{aligned}$$

The first and second conditions means that A and A^+ are weak inverses of A^+ and A, respectively, in the multiplicative semigroup. The third and fourth conditions mean that AA^+ and A^+A are self-adjoint.

The pseudoinverse as defined by these four conditions exists uniquely. Important properties of the pseudoinverse are the following. As expected, if the matrix A is regular, then the pseudoinverse is its inverse, i.e., $A^+ = A^{-1}$. The pseudoinverse of the pseudoinverse is again the original matrix, i.e., $(A^+)^+ = A$. Pseudoinversion commutes with complex conjugation, taking the conjugate transpose, and with transposition. The pseudoinverse of a scalar multiple of a matrix A is the reciprocal multiple of the pseudoinverse, i.e., $(\alpha A)^+ = \alpha^{-1}A^+$ for all $\alpha \neq 0$ in the underlying field. Finally, the pseudoinverse of a zero matrix is its (conjugate) transpose.

In Julia, the pseudoinverse of a matrix is calculated using the function `LinearAlgebra.pinv`, which uses the singular-value decomposition $A = U\Sigma V^*$ (see Sect. 8.4.10) to find $A^+ = V\Sigma^+ U^*$ in the general case, although there are special methods for special matrix types as well. For example, the pseudoinverse Σ^+ of the rectangular diagonal matrix Σ is simply found by replacing the elements on the diagonal whose absolute value is above a certain tolerance by their reciprocals, leaving the rest of the elements, close to zero, unchanged, and finally transposing the matrix. The value of the tolerance that determines non-zero elements is important for ill-conditioned matrices and can be supplied as an optional second argument to `pinv`.

```
julia> pinv([1.0 0.0; 0.0 1.0e-17])
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  0.0
julia> pinv([1.0 0.0; 0.0 1.0e-17], 1e-18)
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  1.0e17
julia> pinv(Diagonal([1.0, 1.0e-17]))
2×2 Diagonal{Float64, Vector{Float64}}:
 1.0   .
  .   1.0e17
```

There are applications of the pseudoinverse related to the solution of linear systems where the pseudoinverse plays the role of a generalization of the inverse matrix and earns its name.

As the solution of a system of linear equations $A\mathbf{x} = \mathbf{b}$, $A \in F^{n\times m}$, does not necessarily exist uniquely, the pseudoinverse still always solves such a system in the least-squares sense. More precisely, this means that

$$\mathbf{y} := A^+\mathbf{b}$$

satisfies

$$||A\mathbf{z} - \mathbf{b}||_2 \geq ||A\mathbf{y} - \mathbf{b}||_2 \qquad \forall \mathbf{z} \in F^m,$$

i.e., the vector $\mathbf{y}$ provides the smallest error in the least-squares sense. Equality holds if and only if

$$\mathbf{z} = \underbrace{A^+\mathbf{b}}_{=\mathbf{y}} + (I - A^+A)\mathbf{u} \qquad \exists \mathbf{u} \in F^m, \tag{8.14}$$

meaning that there are infinitely many minimizing solutions $\mathbf{z}$ unless A has full rank, i.e., $\text{rk}(A) = m$. If A has full rank, then $I = A^+A$ and thus $\mathbf{z} = \mathbf{y} = A^+\mathbf{b}$.

Solutions of the system $A\mathbf{x} = \mathbf{b}$ exist if and only if $A\mathbf{y} = AA^+\mathbf{b} = \mathbf{b}$. If this last equation holds, the solution is unique if and only if A has full rank, i.e., $\mathrm{rk}(A) = m$ and thus $I = A^+A$. If the system has any solutions, they are all given by $\mathbf{z}$ in (8.14).

Furthermore, if the system $A\mathbf{x} = \mathbf{b}$ has multiple solutions, then the pseudoinverse yields the solution $\mathbf{y}$ of minimal Euclidean norm, i.e., $\|\mathbf{y}\|_2 \le \|\mathbf{x}\|_2$ holds for all solutions $\mathbf{x}$.

We consider two short examples. If the linear system has no solution, the vector with the smallest least-squares error is found.

```
julia> A = [1 0; 1 0]; b = [-1; 1]; pinv(A) * b
2-element Vector{Float64}:
 1.1102230246251565e-16
 0.0
```

In geometric terms, the point $(0,0)^\top$ on the line $\lambda(1,1)^\top$, $\lambda \in \mathbb{R}$, has the smallest Euclidean distance from the point $(-1,1)^\top$.

If the linear system has multiple solutions, the vector with the minimal Euclidean norm is found.

```
julia> A = [1 0; 1 0]; b = [1; 1]; pinv(A) * b
2-element Vector{Float64}:
 0.9999999999999997
 0.0
julia> pinv(A) * A
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  0.0
```

In this example, all solutions are given by $(1,u_2)^\top$, $u_2 \in \mathbb{R}$.

8.4.9 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors[1] of matrices have important meaning in applications such as geometric transformations, differential equations, stability analysis, quantum mechanics (the Schrödinger equation and molecular orbitals), vibration analysis, principal-component analysis, and graph theory. They are defined for linear transformation from a vector space into itself.

Definition 8.22 (eigenvalue and eigenvector) Suppose $A \in \mathbb{C}^{n\times n}$ is a square matrix. If

$$A\mathbf{v} = \lambda\mathbf{v} \tag{8.15}$$

and $\mathbf{v} \ne \mathbf{0}$ hold, then $\lambda \in \mathbb{C}$ is called an *eigenvalue* of A and $\mathbf{v} \in \mathbb{C}^n$ is called an *eigenvector* of A.

[1] German “eigen-” means “self-”.

Since the zero vector always satisfies (8.15) trivially, it is excluded from the definition of an eigenvector. The geometric interpretation of a pair of eigenvalues and eigenvectors is that an eigenvector is a direction in which the transformation stretches by a factor that is the eigenvalue.

How can we find the eigenvalues and eigenvectors of a square matrix? Determining the eigenvalues of a square matrix in theory is not hard. Equation (8.15) has a non-zero solution if and only if the determinant of $A - \lambda I$ is zero. This observation yields the equation

$$\det(A - \lambda I) = 0 \tag{8.16}$$

for any eigenvalue $\lambda \in \mathbb{C}$. Expanding the determinant shows that the left-hand side of this equation is a polynomial of degree n in λ (and that the coefficient of λ^n is $(-1)^n$).

Definition 8.23 (characteristic polynomial) The polynomial

$$\chi_A(\lambda) := \det(A - \lambda I)$$

in λ is called the *characteristic polynomial* of the square matrix A.

The fundamental theorem of algebra implies that the characteristic polynomial has n roots over the complex numbers $\mathbb{C}$. This is the reason why the underlying field in Definition 8.22 is $\mathbb{C}$.

Once we have determined the eigenvalues, e.g., by finding all roots of the characteristic polynomial, the eigenvector $\mathbf{v}$ that corresponds to the eigenvalue λ can be found by solving the linear system

$$(A - \lambda I)\mathbf{v} = \mathbf{0}, \tag{8.17}$$

that stems from (8.15), for $\mathbf{v}$.

In JULIA, the two functions `LinearAlgebra.eigvals` and `LinearAlgebra.eigvals!` compute eigenvalues and the function `LinearAlgebra.eigvecs` computes eigenvectors.

```
julia> A = randn(3, 3);
julia> lambda = eigvals(A)
3-element Vector{ComplexF64}:
 -0.8670151992707238 + 0.0im
  0.8351764501056504 - 0.578399928311388im
  0.8351764501056504 + 0.578399928311388im
```

The functions `LinearAlgebra.eigen` and `LinearAlgebra.eigen!` compute factorization objects `f` of type `Eigen`, which contain the eigenvalues in the field `values` and the eigenvectors in the field `vectors` such that the k-th eigenvector is stored in the column `f.vectors[:, k]`. The functions `det`, `inv`, and `isposdef` are also defined for `Eigen` objects.

```
julia> f = eigen(A);
```

We check that the results satisfy the definition.

```
julia> A * f.vectors - f.vectors * diagm(f.values)
3×3 Matrix{ComplexF64}:
  2.77556e-16+0.0im   1.11022e-15-4.996e-16im     1.11022e-15+4.996e-16im
 -1.66533e-16+0.0im  -3.88578e-16+5.55112e-17im  -3.88578e-16-5.55112e-17im
 -4.44089e-16+0.0im   1.11022e-16-1.11022e-16im   1.11022e-16+1.11022e-16im
```

Equation (8.16) is also satisfied.

```
julia> [det(A - f.values[i] * Matrix(I, 3, 3))
        for i in 1:size(A, 1)]
3-element Vector{ComplexF64}:
 -1.2470931052324467e-15 + 0.0im
  -1.108463404734065e-15 - 3.8148117819699507e-16im
  -1.108463404734065e-15 + 3.8148117819699507e-16im
```

The constructor **`Matrix`** called with `I` as the first argument constructs matrices with ones in the main diagonal such as identity matrices.

Eigenvalues may be repeated. There are two ways to define the multiplicity of an eigenvalue: the first is called algebraic multiplicity and the second is called geometric multiplicity. The algebraic multiplicity stems from the multiplicity of the eigenvalue as a root of the characteristic polynomial, while the geometric multiplicity stems from the number of corresponding eigenvectors or the size of the solution space of the linear system (8.17).

Definition 8.24 (algebraic multiplicity) Let λ_i be an eigenvalue of the square matrix A. Then the *algebraic multiplicity* $\mu_A(\lambda_i)$ is the multiplicity of the eigenvalue λ_i as a root of the characteristic polynomial.

In other words, if $A \in \mathbb{C}^{n\times n}$ has d distinct eigenvalues, then the characteristic polynomial can be written as the product

$$\chi_A(\lambda) = \prod_{i=1}^{d} (\lambda_i - \lambda)^{\mu_A(\lambda_i)}.$$

Clearly, the inequality $1 \le \mu_A(\lambda_i) \le n$ holds for all $i \in \{1, \dots, d\}$, and the sum of all algebraic multiplicities is equal to the dimension of the vector space, i.e.,

$$\sum_{i=1}^{d} \mu_A(\lambda_i) = n. \tag{8.18}$$

Before we can define the geometric multiplicity of an eigenvalue, we define its eigenspace.

Definition 8.25 (eigenspace) Suppose λ_i is an eigenvalue of $A \in \mathbb{C}^{n\times n}$. Then the *eigenspace associated with* λ_i is the set

$$E(\lambda_i) := \{\mathbf{v} \in \mathbb{C}^n \mid (A - \lambda_i I)\mathbf{v} = \mathbf{0}\}.$$

The eigenspace of the eigenvalue λ_i is, of course, the kernel of the matrix $A - \lambda_i I$ and can be calculated using the function `nullspace`. It is spanned by all eigenvectors associated with λ_i. It is straightforward to show that an eigenspace is always a linear subspace (see Problem 8.16). Using the notion of an eigenspace, we can now define the geometric multiplicity.

Definition 8.26 (geometric multiplicity) Let λ_i be an eigenvalue of the square matrix A. Then the *geometric multiplicity* $\gamma_A(\lambda_i)$ is the dimension of its eigenspace, i.e.,

$$\gamma_A(\lambda_i) := \dim E(\lambda_i).$$

In other words, the geometric multiplicity of the eigenvalue λ_i is the nullity of the matrix $A - \lambda_i I$. By Theorem 8.3, the geometric multiplicity is equal to

$$\gamma_A(\lambda_i) = n - \operatorname{rk}(A - \lambda I).$$

By equation (8.16), the inequality

$$\operatorname{rk}(A - \lambda I) < n$$

holds and therefore the geometric multiplicity is at least one, i.e.,

$$\gamma_A(\lambda_i) \geq 1.$$

How are the algebraic and the geometric multiplicities related? The answer is given by the following theorem, whose proof is nontrivial (see Problem 8.17). It means that the geometric multiplicity of an eigenvalue is always smaller than its algebraic one.

Theorem 8.27 (geometric and algebraic multiplicities) *Suppose λ_i is an eigenvalue of $A \in \mathbb{C}^{n\times n}$. Then the inequality*

$$1 \leq \gamma_A(\lambda_i) \leq \mu_A(\lambda_i) \leq n$$

holds.

This theorem also implies the inequality

$$d \leq \gamma_A := \sum_{i=1}^{d} \gamma_A(\lambda_i) \leq n$$

by summing over all eigenvalues and using (8.18). Here we have defined γ_A as the sum of all geometric multiplicities.

If the geometric multiplicities of all eigenvalues are equal to their algebraic multiplicities and thus are maximal, then the eigenvectors have the important and useful property that a basis of $\mathbb{C}^n$, the eigenbasis, can be chosen from the set of eigenvectors. This property is recorded in the following theorem.

Theorem 8.28 (eigenbasis) *Suppose $A \in \mathbb{C}^{n \times n}$. If the equation*

$$\sum_{i=1}^{d} \gamma_A(\lambda_i) = n \tag{8.19}$$

holds, then

$$\operatorname{span}\left(\bigcup_{i=1}^{d} E(\lambda_i)\right) = \mathbb{C}^n$$

and a basis of $\mathbb{C}^n$, the eigenbasis, can be formed from n linearly independent eigenvectors.

Further important properties of eigenvalues are collected in the following theorem.

Theorem 8.29 (properties of eigenvalues) *Suppose $A \in \mathbb{C}^{n \times n}$ is a square matrix with the n eigenvalues $(\lambda_1, \dots, \lambda_n)$. (Each eigenvalue λ_i appears $\mu_A(\lambda_i)$ (algebraic multiplicity) times in this vector.) Then the following statements hold true.*

1. *The determinant of A is equal to the product of all its eigenvalues, i.e.,*

$$\det(A) = \prod_{i=1}^{n} \lambda_i.$$

2. *The trace of A, which is defined as the sum of all diagonal elements, is equal to the sum of all its eigenvalues, i.e.,*

$$\operatorname{tr}(A) := \sum_{i=1}^{n} A_{ii} = \sum_{i=1}^{n} \lambda_i.$$

3. *The matrix A is regular if and only if all of its eigenvalues are non-zero.*
4. *If A is regular, then the eigenvalues of the inverse A^{-1} are $(1/\lambda_1, \dots, 1/\lambda_n)$ with the same algebraic and geometric multiplicities.*
5. *The eigenvalues of A^k, $k \in \mathbb{N}$, are $(\lambda_1^k, \dots, \lambda_n^k)$.*
6. *If A is unitary, then the absolute value of all its eigenvalues is 1, i.e., $|\lambda_i| = 1$ for all $i \in \{1, \dots, n\}$.*
7. *If A is Hermitian, then all its eigenvalues are real.*
8. *If A is Hermitian and positive definite, positive semidefinite, negative definite, or negative semidefinite, then all its eigenvalues are positive, nonnegative, negative, or nonpositive, respectively.*

As we have seen in Theorem 8.28, a matrix $A \in \mathbb{C}^{n \times n}$ has n linearly independent eigenvectors if the geometric multiplicities of all eigenvalues are maximal, i.e., if (8.19) holds. In this case, it can be represented very simply by a diagonal matrix in a suitable basis. The converse statement is also true, as the following theorem shows.

Theorem 8.30 (diagonalization) *A matrix $A \in \mathbb{C}^{n \times n}$ has n linearly independent eigenvectors if and only if A can be factorized as*

$$A = Q\Lambda Q^{-1}, \tag{8.20}$$

where $Q \in \mathbb{C}^{n \times n}$ is a regular matrix and Λ is a diagonal matrix whose entries are the eigenvalues of A.

Proof By definition and assumption, we can find eigenvalues and n eigenvectors such that

$$A\mathbf{v}_{i,j_i} = \lambda_i \mathbf{v}_{i,j_i},$$

where the indices i and j_i enumerate the n eigenvectors. There are d eigenvalue such that $i \in \{1, \dots, d\}$ and $j_i \in \{1, \dots, \mu_A(\lambda_i)\}$. Collecting all the n eigenvectors $\mathbf{v}_{i,j_i}$ as columns in a matrix Q, we hence find $AQ = Q\Lambda$ and therefore $A = Q\Lambda Q^{-1}$.

To prove the converse statement, the existence of a factorization $A = Q\Lambda Q^{-1}$ (with Q being regular) implies $AQ = Q\Lambda$ and therefore there are d eigenvalues and n eigenvectors. It remains to show that the eigenvectors $\mathbf{v}_k$ found in this manner are linearly independent.

Suppose they are linearly dependent. Then, by definition, there exists a vector $\mathbf{a} \neq \mathbf{0}$ such that the linear combination $\sum_{k=1}^{n} a_k \mathbf{v}_k$ is zero or, equivalently, $Q\mathbf{a} = \mathbf{0}$. Therefore $\operatorname{nul}(A) > 0$ and, by Theorem 8.3, $\operatorname{rk}(A) < n$, which is a contradiction to the regularity of the matrix Q. □

A matrix is called diagonalizable if it admits such a factorization. If not, it is called defective. Maybe the simplest example of a defective matrix is

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{8.21}$$

(see Problem 8.20).

It is clear that if the characteristic polynomial has no repeated roots, then the matrix is diagonalizable.

If it exists, the diagonalization of a matrix A is very useful, since many important properties of A can immediately be seen from the factorization (8.20) (see Theorem 8.29). In particular, if A is regular in addition, then its inverse is immediately given by $A^{-1} = Q\Lambda^{-1}Q^{-1}$. The geometric action of the linear function represented by the matrix A is also obvious.

Certain matrices, called normal matrices, have especially simple factorizations, as the following definition and theorem show.

Definition 8.31 (normal matrix) A matrix $A \in \mathbb{C}^{n \times n}$ is called *normal* if

$$A^* A = A A^*.$$

Theorem 8.32 (normal matrices) *A matrix $A \in \mathbb{C}^{n\times n}$ is normal if and only if there exist a diagonal matrix Λ and a unitary matrix U such that*

$$A = U\Lambda U^*.$$

This factorization is especially useful, since the basis change given by a unitary matrix is numerically stable and the representation as a diagonal matrix in this basis is especially simple.

Theorem 8.30 shows that if and only if there are n linearly independent eigenvectors, then the matrix has an eigenfactorization and hence can be represented very simply by a diagonal matrix in an eigenbasis. This naturally leads to the question what happens when there are fewer than n independent eigenvectors. Can we still find factorization? We expect that such a factorization would be more complicated than the representation by a diagonal matrix; it is hardly conceivable that a simpler factorization exists.

The answer is given by the following theorem, which also draws the complete picture. In order to formulate the theorem, we need a definition first.

Definition 8.33 (Jordan matrix) A square, complex matrix $A \in \mathbb{C}^{n\times n}$, which has the block diagonal form

$$J = \begin{pmatrix} J_1 & & \\ & \ddots & \\ & & J_d \end{pmatrix},$$

where each block J_i is a square matrix of the form

$$J_i = \begin{pmatrix} \lambda_i & 1 & & \\ & \lambda_i & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_i \end{pmatrix},$$

is called a *Jordan matrix*.

This means that a Jordan matrix J is a square matrix whose only non-zero entries are in the main diagonal and the first superdiagonal. The blocks J_i are called Jordan blocks. In each block, all entries in the first superdiagonal are equal to one.

Definition 8.34 (similar matrix) Two square matrices $A \in \mathbb{C}^{n\times n}$ and $B \in \mathbb{C}^{n\times n}$ are called *similar* if there exists a regular matrix $P \in \mathbb{C}^{n\times n}$ such that

$$A = PBP^{-1}.$$

Theorem 8.35 (eigenfactorization, Jordan normal form) *Every square matrix $A \in \mathbb{C}^{n\times n}$ is similar to a Jordan matrix J, called the Jordan normal form of A, i.e., there exists a regular matrix $P \in \mathbb{C}^{n\times n}$ and a Jordan matrix J such that*

$$A = PJP^{-1}.$$

The theorem shows that matrices are not diagonalizable in general, but that every matrix is similar to Jordan matrix, which still contains the eigenvalues in the main diagonal, but may contain additional ones in the first superdiagonal.

Is the Jordan normal form of a matrix unique? Of course, it is also possible to rearrange the similarity matrix P such that the Jordan blocks are reordered. Apart from these rearrangements, however, the Jordan normal form is unique, as the following theorem records.

Theorem 8.36 (uniqueness of Jordan normal form) *The Jordan normal form of a matrix $A \in \mathbb{C}^{n\times n}$ is unique up to the order of the Jordan blocks.*

The Jordan normal form J and its blocks have the following properties. The geometric multiplicity $\gamma_A(\lambda_i)$ is the number of Jordan blocks corresponding to the eigenvalue λ_i, and the sum of the sizes of all Jordan blocks corresponding to λ_i is its algebraic multiplicity $\mu_A(\lambda_i)$. In terms of the sizes of the Jordan blocks, we thus see that a matrix A is diagonalizable if and only if the algebraic and geometric multiplicities of every eigenvalue λ_i coincide.

The sizes of the Jordan blocks corresponding to an eigenvalue help to solve the mystery of the missing eigenvectors whenever the geometric multiplicity is smaller than the algebraic one. In this case, there are fewer than n linearly independent eigenvectors, and we would like to find more vectors in order to complete the set of eigenvectors and obtain a basis of the whole vector space.

We consider a three-dimensional example and define

$$J := \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 1 \\ 0 & 0 & \lambda_2 \end{pmatrix}$$

consisting of two Jordan blocks. The Jordan normal form $A = PJP^{-1}$ implies $AP = PJ$, and we denote the three columns of P by $\mathbf{p}_i$, $i \in \{1, 2, 3\}$. Then we have the equation

$$A\left(\mathbf{p}_1 \;\; \mathbf{p}_2 \;\; \mathbf{p}_3\right) = \left(\mathbf{p}_1 \;\; \mathbf{p}_2 \;\; \mathbf{p}_3\right) \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 1 \\ 0 & 0 & \lambda_2 \end{pmatrix} = \left(\lambda_1\mathbf{p}_1 \;\; \lambda_2\mathbf{p}_2 \;\; \mathbf{p}_2 + \lambda_2\mathbf{p}_3\right),$$

whose columns yield the equations

$$\begin{aligned} (A - \lambda_1 I)\mathbf{p}_1 &= \mathbf{0}, \\ (A - \lambda_2 I)\mathbf{p}_2 &= \mathbf{0}, \\ (A - \lambda_2 I)\mathbf{p}_3 &= \mathbf{p}_2. \end{aligned}$$

The first equation means that $\mathbf{p}_1 \in \ker(A - \lambda_1 I)$ is an eigenvector for the eigenvalue λ_1 and the second equation means that $\mathbf{p}_2 \in \ker(A - \lambda_2 I)$ is an eigenvector for the eigenvalue λ_2.

The second Jordan block

$$\begin{pmatrix} \lambda_2 & 1 \\ 0 & \lambda_2 \end{pmatrix}$$

reduces the geometric multiplicity $\gamma_A(\lambda_2)$, so that we would like to construct an additional vector corresponding to λ_2 to adjoin to the eigenvectors to find a basis of the whole vector space.

We can do so as follows. Multiplying the last equation by $A - \lambda_2 I$ yields

$$(A - \lambda_2 I)^2 \mathbf{p}_3 = (A - \lambda_2 I)\mathbf{p}_2 = \mathbf{0},$$

meaning that $\mathbf{p}_3 \in \ker((A - \lambda_2 I)^2)$. Vectors such as $\mathbf{p}_3$ are called generalized eigenvectors of A.

Definition 8.37 (generalized eigenvector) A vector $\mathbf{v} \in \mathbb{C}^n$ is called a *generalized eigenvector of rank k of the matrix $A \in \mathbb{C}^{n \times n}$ corresponding to the eigenvalue λ* if

$$(A - \lambda I)^k \mathbf{v} = \mathbf{0},$$
$$(A - \lambda I)^{(k-1)} \mathbf{v} \neq \mathbf{0}.$$

An eigenvector is, of course, a generalized eigenvector of rank 1.

Just as in this example, it is generally possible to construct a Jordan chain of generalized eigenvectors for a given Jordan block starting from an eigenvector. The eigenvector is an element of $\ker(A - \lambda_i I)$, while the generalized eigenvectors are elements of $\ker((A - \lambda_i I)^k)$, $k \in \{2, \dots, K\}$, where K is the size of the Jordan block for which the Jordan chain is calculated.

These generalized eigenvectors are the vectors needed in Theorem 8.35 to complete the eigenvectors to find a basis of the whole vector space. It can be shown that this construction is possible for all Jordan blocks and that the generalized eigenvectors are indeed linearly independent. The space spanned by all generalized eigenvectors corresponding to an eigenvalue is called the generalized eigenspace of this eigenvalue.

It is now clear how the simple example of a defective matrix in (8.21) relates to the general case and the Jordan normal form. This defective matrix is just a single Jordan block for the eigenvalue 1.

We now perturb this matrix and consider the matrix

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix}$$

with $\epsilon \neq 0$. Its eigenvalues are $1 \pm \sqrt{\epsilon}$, and therefore its Jordan normal form is

$$\begin{pmatrix} 1+\sqrt{\epsilon} & 0 \\ 0 & 1-\sqrt{\epsilon} \end{pmatrix}.$$

This means that a slight perturbation of a matrix with multiple eigenvalues can completely change the structure of its Jordan normal form. The numerical problem of calculating the Jordan normal form of a matrix is therefore ill-conditioned and depends critically on the criterion whether two eigenvalues are considered equal. Hence, the Jordan normal form of a matrix is usually avoided in computations, although it is of great theoretical importance, and alternatives such as the Schur decomposition are employed. The Schur factorization or Schur decomposition always exists, as the following theorem shows.

Theorem 8.38 (Schur factorization) *Suppose $A \in \mathbb{C}^{n\times n}$. Then there exist a unitary matrix Q and an upper-triangular matrix T such that*

$$A = QTQ^{-1}.$$

The Schur factorization means that every complex, square matrix is similar to an upper-triangular matrix. The Schur factorization is not unique. The advantage of such a factorization is that the basis change Q is given by a unitary matrix, and we already know that multiplication by an orthogonal or by a unitary matrix is well-conditioned.

How does the Schur factorization relate to eigenvalues? If a Schur factorization is known, then A and T have the same eigenvalues by Problem 8.25. The eigenvalues of an upper-triangular matrix are just its diagonal elements by Problem 8.26. This means that a Schur factorization of a matrix A immediately yields its eigenvalues.

In JULIA, Schur factorization is available as the two functions `LinearAlgebra.schur` and `LinearAlgebra.schur!`. Just as the other functions for matrix factorization in JULIA, it returns an object. The information in the `Schur` object can be accessed as the fields `T` or `Schur` for the quasi upper-triangular matrix, as the fields `Z` or `vectors` for the unitary matrix, and as the index `values` for the eigenvalues. This built-in implementation only calculates a quasi upper-triangular matrix and not an upper-triangular matrix as in Theorem 8.38.

```
julia> A = randn(3, 3); f = schur(A);
julia> A - f.Z * f.T * f.Z'
3×3 Matrix{Float64}:
 9.99201e-16  -6.38378e-16  -1.11022e-15
 0.0          -2.22045e-16   6.66134e-16
 1.63064e-16  -1.66533e-16  -3.33067e-16
```

How is the Schur factorization of a matrix $A \in \mathbb{C}^{n\times n}$ calculated? The answer is provided by QR iteration, also called the QR algorithm. The basic version is the following.

Algorithm 8.39 (*QR* iteration)

1. Set $k := 1$ and define $A_0 := A$.
2. Repeat:

   a. Define

$$\begin{aligned} Q_k R_k &:= A_{k-1}, \\ A_k &:= R_k Q_k. \end{aligned}$$

   The first definition means that Q_k and R_k are obtained by a QR factorization of A, whose factors are multiplied in the second definition in reverse order.

   b. Increase $k := k + 1$ and repeat until a termination criterion is satisfied.

The orthogonal similarity transformations Q_k obtained by QR factorization and employed in the iteration make QR iteration numerically stable.

We find that

$$A_k = R_k Q_k = Q_k^{-1} Q_k R_k Q_k = Q_k^* A_{k-1} Q_k, \tag{8.22}$$

since the factor Q_k in the QR factorization is unitary. This equation shows by induction that all matrices $A = A_0, A_1, A_2, \ldots$ are similar and hence have the same eigenvalues (by Problem 8.25).

Furthermore, it can be shown that the matrices A_k converge to an upper-triangular matrix as k tends to infinity. Then, by (8.22), we find that

$$A_k = Q_k^* Q_{k-1}^* \cdots Q_1^* A Q_1 Q_2 \cdots Q_k = (Q_1 Q_2 \cdots Q_k)^* A (Q_1 Q_2 \cdots Q_k) = Q^* A Q,$$

where we have defined

$$Q := Q_1 Q_2 \cdots Q_k$$

and $Q^{-1} = Q^*$ holds. Since A_k converges to an upper-triangular matrix U, we obtain $U \approx Q^* A Q$ and hence

$$A \approx Q U Q^*,$$

which is a Schur factorization as in Theorem 8.38. Since the eigenvalues of a triangular matrix are just the diagonal elements (by Problem 8.26), the eigenvalues have been approximated by calculating U.

We consider a numerical example next. In order to observe convergence, we construct an eigenvalue problem with a known solution. We start from a matrix U with known eigenvalues, construct a similar matrix A, and then implement QR iteration.

```
julia> U = diagm([0.2, -0.1, 0.3, -0.5, 0.4]);
julia> S = randn(size(U));
julia> A = S * U / S;
julia> Ak = A;
julia> for i in 1:10 (Q, R) = qr(Ak); Ak = R*Q end; Ak
```

```
5×5 Matrix{Float64}:
 -0.531019    -0.470487      0.0271631    -0.325596    -0.808545
  0.061126     0.435922      0.0383569     0.321512    -1.41668
 -0.00602601  -0.0159307     0.296267      0.0870228    1.98122
 -1.26847e-6  -0.000144795  -0.000895731   0.199251     1.87909
  4.86472e-9  -2.29092e-8   -7.59125e-7   -6.64697e-5  -0.10042
julia> Ak = A;
julia> for i in 1:100 (Q, R) = qr(Ak); Ak = R*Q end; Ak
5×5 Matrix{Float64}:
 -0.5          -0.531455     -0.0335534    -0.345635     -0.701827
  1.18005e-11   0.4           0.0504886     0.28702      -1.70175
 -3.62799e-25  -3.91756e-15   0.3           0.126148      1.77161
 -1.89986e-46  -1.1034e-34   -2.15349e-21   0.2           1.89475
  5.94044e-79  -1.54211e-68  -1.51952e-54  -5.22028e-35  -0.1
```

We observe that the eigenvalue closest to zero is approximated in the lower right corner. After 100 steps, an upper-triangular matrix with the sought eigenvalues is obtained. Even after 10 steps, three correct digits of the eigenvalue closest to zero are found in the lower right corner.

The convergence rate of QR iteration depends on the separation between the eigenvalues. The Gershgorin circle theorem, a bound on the spectrum (i.e., the set of all eigenvalues) of a square matrix, is useful for testing for convergence.

It is known that an eigenvalue close to zero improves convergence. How can we move an eigenvalue closer to zero? It is straightforward to see that if λ is an eigenvalue of A, then $\lambda - s$ is an eigenvalue of $A - sI$. Assuming that we can find suitable shifts s_k that approximate the eigenvalues closest to zero, we hence define shifted QR iteration as

$$Q_k R_k := A_{k-1} - s_k I, \tag{8.23a}$$

$$A_k := R_k Q_k + s_k I. \tag{8.23b}$$

We must check that (8.22) still holds, which is done by calculating

$$A_k = R_k Q_k + s_k I = Q_k^{-1}(A_{k-1} - s_k I)Q_k + s_k I = Q_k^* A_{k-1} Q_k,$$

and therefore all matrices A_k are again similar to A.

We still have to find suitable shifts s_k. Since approximations of the eigenvalues are calculated during the iteration, these approximations suggest themselves for this task. The most obvious approximation would be to use the lower right element of A_{k-1} as the shift s_k in (8.23). This choice, however, may cause the iteration to fail. A much better choice is the eigenvalue of the 2×2 lower right submatrix of A_{k-1} that is closest to the lower right element of A_{k-1}. This choice is called the Wilkinson shift.

Another technique to improve the convergence of QR iteration is called deflation. Once the eigenvalue closest to zero has been approximated satisfactorily, a smaller matrix without this eigenvalue is constructed, i.e., the matrix is deflated, and the next eigenvalue is calculated. Deflation is based on the following theorem.

Theorem 8.40 (deflation) *Suppose $A \in \mathbb{C}^{n\times n}$ has the form*

$$A = \begin{pmatrix} B & \mathbf{u} \\ \mathbf{0}^* & \lambda \end{pmatrix},$$

where $B \in \mathbb{C}^{(n-1)\times(n-1)}$, $\mathbf{u} \in \mathbb{C}^{(n-1)\times 1}$, $\mathbf{0}^* \in \mathbb{C}^{1\times(n-1)}$, and $\lambda \in \mathbb{C}$. Then the eigenvalues of A are λ and the eigenvalues of B.

Further improvements are still possible; a modern variant of QR iteration is implicit QR iteration, which simplifies the use of multiple shifts.

In the QR iterations discussed so far, each iteration requires a QR factorization, which requires $O(n^3)$ floating-point operations in the general case, as can be shown. In order to avoid this computational expense, practical algorithms transform the matrix into almost triangular form at the beginning, which results in large savings in each QR factorization.

A factorization which achieves this goal is Hessenberg factorization, and it is the final factorization relating to eigenvalues we discuss here. Hessenberg factorization can be shown to reduce the computational cost of both LU factorization and QR factorization to $O(n^2)$. Although the computational cost of Hessenberg factorization is $O(n^3)$, it is usually worth the effort in the beginning by reducing the computational cost of the subsequent steps; the constants in the operation count are important, not only its asymptotic behavior as n tends to infinity.

We first define the form of the final result, which is almost triangular: in an upper-Hessenberg matrix, all elements below the first subdiagonal are zero.

Definition 8.41 (upper-Hessenberg matrix) A matrix $H \in \mathbb{C}^{n\times n}$ is called *upper Hessenberg* if $h_{ij} = 0$ whenever $i > j + 1$.

The next theorem shows that it is always possible to find a matrix H in upper-Hessenberg form that is similar to a given matrix A. The fact that the basis change Q is unitary is advantageous and again ensures numerical stability. We use Householder reflections (see Sect. 8.4.8.3) in the algorithm and its proof.

Theorem 8.42 (Hessenberg factorization) *Suppose $A \in \mathbb{C}^{n\times n}$. Then there exist a unitary matrix Q and an upper-Hessenberg matrix H such that*

$$A = QHQ^{-1}.$$

Proof The proof is constructive. The following algorithm computes unitary transformations such that the resulting matrix is upper Hessenberg. □

Algorithm 8.43 (Hessenberg factorization)

1. Set $A_0 := A$.
2. For j from 1 to $n-2$, perform these steps:

a. Set $\mathbf{x}_j$ to be the j-th column of A_j rows $j+1$ to n, i.e., $\mathbf{x}_j := A_j[(j+1):n, j]$. Set $P_j := P(\mathbf{x}_j)$, the Householder reflection constructed for $\mathbf{x}_j$ as in Theorem 8.19.
b. Set
$$Q_j^* := \begin{pmatrix} I_j & \mathbf{0}^* \\ \mathbf{0} & P_j \end{pmatrix}.$$
The matrix P_j has size $(n-j)\times(n-j)$, and I_j is the identity matrix of size $j \times j$. Therefore Q_j^* has size $n \times n$. The matrix Q_j^* is Hermitian and unitary by Theorem 8.19.
c. Set
$$A_j := Q_j^* A_{j-1} Q_j.$$
In the first step, the form of the product $Q_1^* A_1$ is
$$Q_1^* A_1 = \begin{pmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{1n} \\ \pm\|\mathbf{x}\|_2 & * & * & \cdots & * \\ 0 & * & * & \cdots & * \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & * & * & \cdots & * \end{pmatrix},$$
because of the definition of P_1 as a Householder reflection. Furthermore, the form of the product $Q_1^* A_1 Q_1$ is
$$A_2 = Q_1^* A_1 Q_1 = \begin{pmatrix} a_{11} & * & * & \cdots & * \\ \pm\|\mathbf{x}\|_2 & * & * & \cdots & * \\ 0 & * & * & \cdots & * \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & * & * & \cdots & * \end{pmatrix}.$$
In all later steps, analogous calculations show that zeros are created by left-multiplication by Q_j^* and all zeros thus created, also in the previous steps, remain after right-multiplication by Q_j.

3. After the $n-2$ steps of the loop, we set
$$H := A_{n-2} = Q_{n-2}^* A_{n-1} Q_{n-2} = Q_{n-2}^* \cdots Q_1^* A Q_1 \cdots Q_{n-2}.$$
We also set
$$Q := Q_1 \cdots Q_{n-2}$$
to obtain
$$H = Q^* A Q.$$
The matrix Q is unitary as a product of unitary matrices, and the matrix H is upper Hessenberg.

In JULIA, Hessenberg factorization is implemented by the functions `LinearAlgebra.hessenberg` and `LinearAlgebra.hessenberg!`. The unitary matrix stored in the resulting object of type `Hessenberg` in the field `Q` and the upper-Hessenberg matrix in the field `H`.

```
julia> A = randn(5, 5); f = hessenberg(A);
```

Having calculated the Hessenberg factorization, we check the result.

```
julia> maximum(abs.(f.Q * f.Q' - Matrix(I, 5, 5)))
4.440892098500626e-16
julia> maximum(abs.(f.Q * f.H / f.Q - A))
8.881784197001252e-16
julia> maximum(abs.(f.Q * f.H * f.Q' - A))
1.5543122344752192e-15
```

8.4.10 Singular-Value Decomposition

The singular-value decomposition (SVD) is another factorization that is as fundamental as the eigenfactorization in Theorem 8.35 or the Schur factorization in Theorem 8.38. In contrast to these two factorization, SVD provides a factorization of any complex $n \times m$ matrix and not only of square matrices. The following theorem shows the form of the factorization.

Theorem 8.44 (singular-value decomposition) *Suppose $A \in \mathbb{C}^{n\times m}$. Then A can be factored as*

$$A = U\Sigma V^*,$$

where $U \in \mathbb{C}^{n\times n}$ and $V \in \mathbb{C}^{m\times m}$ are unitary matrices and $\Sigma \in \mathbb{R}^{n\times m}$ is a rectangular diagonal matrix with nonnegative elements on the diagonal. If A is a real matrix, then U and V can be chosen real as well.

The matrices U and V in an SVD are not unique, as $U\Sigma V^* = (-U)\Sigma(-V)^*$, for example. The

$$s := \min(m, n)$$

diagonal entries of Σ are called the singular values of A and denoted by $\sigma_1, \ldots, \sigma_s$. The convention is to order the singular values such that

$$\sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_s \ge 0.$$

With this convention, the matrix Σ is uniquely determined.

The columns of U and V are called the left and right singular vectors, respectively. Writing the SVD as $AV = U\Sigma$ yields the equations

$$A\mathbf{v}_k = \sigma_k \mathbf{u}_k \qquad \forall k \in \{1, \ldots, s\}, \tag{8.24}$$

which explains the names of the singular vectors. Geometrically, these equations mean that the function represented by A maps each right singular value $\mathbf{v}_k$ to the corresponding left singular vector $\mathbf{u}_k$ stretched by the corresponding singular value σ_k.

The SVD and the eigenfactorization of a matrix are related. The SVD $A = U\Sigma V^*$ of a matrix $A \in \mathbb{C}^{n\times m}$ yields the two equations

$$A^*A = V\Sigma^*U^*U\Sigma V^* = V(\Sigma^*\Sigma)V^*, \tag{8.25a}$$

$$AA^* = U\Sigma V^*V\Sigma^*U^* = U(\Sigma\Sigma^*)U^*. \tag{8.25b}$$

The first equation means that the right singular vectors (i.e., the columns of V) are eigenvectors of A^*A, while the second equation means that the left singular vectors (i.e., the columns of U) are eigenvectors of AA^*. Furthermore, the non-zero singular values are the square roots of the non-zero eigenvalues of A^*A or AA^*. If A is normal, then it can be diagonalized and written as $A = U\Lambda U^*$ by Theorem 8.32. If A is positive semidefinite in addition, then this factorization $A = U\Lambda U^*$ is also an SVD.

This observation also yields a numerical algorithm for the calculation of the singular values and singular vectors of a matrix A, namely to apply QR iteration (see Sect. 8.4.9) to the matrix in (8.25a) to first find the singular values and the right singular vectors of A and then to use (8.24) to find its left singular vectors. Practical methods, however, are based on the matrix

$$\begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix}$$

(see Problem 8.36).

The SVD has many useful properties and can be used to calculate important properties of a matrix, as we will see in the next few theorems. The first theorem in this series shows that the SVD gives an representation of the range and the null space of a matrix.

Theorem 8.45 (SVD and rank, nullity) *Suppose $A \in \mathbb{C}^{n\times m}$. Then the left singular values corresponding to non-zero singular values of A span the range of A and the right singular vectors corresponding to zero singular values of A span the null space of A. Furthermore, the rank of A equals the number of non-zero singular values.*

Knowing a SVD of a matrix, its pseudoinverse (see Sect. 8.4.8.4) is easily found, as the following theorem shows.

Theorem 8.46 (SVD and pseudoinverse) *Suppose $A = U\Sigma V^*$ is a SVD of a matrix $A \in \mathbb{C}^{n\times m}$. Then its pseudoinverse is*

$$A^+ = V\Sigma^+U^*.$$

The following theorem means that the principal singular value σ_1 is equal to the operator 2-norm of the matrix A. The p-norm of a matrix is defined as

$$||A||_p := \sup_{\mathbf{x}\neq\mathbf{0}} \frac{||A\mathbf{x}||_p}{||\mathbf{x}||_p}.$$

Therefore the SVD is the usual means for calculating the 2-norm of a matrix.

Theorem 8.47 (SVD and norm) *Suppose $A \in \mathbb{C}^{n\times m}$. Then $||A||_2 = \sigma_1$.*

The following criterion for determining whether a square matrix is regular or singular follows from Theorem 8.45.

Theorem 8.48 (SVD and regularity) *Suppose $A \in \mathbb{C}^{n\times n}$. Then A is regular if and only if $\sigma_n \neq 0$.*

Another important application of the SVD is the approximation of a matrix A by a simpler and – as the following theorem shows – truncated version of A. The idea is to use only the first k singular values, which are also the largest ones by convention. Depending on how fast the singular values decrease, these first singular values may already capture a significant portion of the behavior of the linear function. As the approximation is of lower rank than the original matrix A, it is called an low-rank approximation of A.

Theorem 8.49 (SVD and low-rank approximation, Eckart–Young–Mirsky Theorem) *Suppose $A \in \mathbb{C}^{n\times m}$ and define*

$$A_k := U\Sigma_k V^*,$$

where Σ_k is the copy of Σ only containing the first k singular values of Σ. Then the equation

$$||A_k - A||_2 = \sigma_{k+1} \qquad \forall k \in \{1, \dots, n-1\}$$

holds.

For computing the low-rank approximation A_k of rank k, only the first k left and right singular values, i.e., the first k columns of U and V, are needed, as all singular values after the first k ones are replaced by zero in Σ_k.

In JULIA, the SVD of a matrix is calculated by the two functions `LinearAlgebra.svd` and `LinearAlgebra.svd!`. The singular values are computed by `LinearAlgebra.svdvals` and `LinearAlgebra.svdvals!`. All these functions follow the convention of sorting the singular values in descending order. The functions `svd` and `svd!` return objects of type `SVD` with the fields `U`, `S`, and `Vt`. There is also a field `V`, but since V^* is calculated and accessible by `Vt`, it is more efficient to use than `V`.

```
julia> A = randn(5, 5); f = svd(A);
julia> f.V' == f.Vt
true
```

We check that the resulting factorization is correct.

```
julia> norm(f.U * diagm(f.S) * f.Vt - A)
2.3308177396304765e-15
```

If `svd` or `svd!` are called with two matrix arguments, they compute the generalized SVD of two matrices.

8.4.11 Summary of Matrix Operations and Factorizations

Tables 8.7 and 8.8 give an overview of the vector and matrix operations available in JULIA excluding matrix factorizations.

A multitude of low-level functions are available as well; for example, the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) functions are available in the modules `LinearAlgebra.BLAS` and `LinearAlgebra.LAPACK`.

Table 8.9 summarizes the various types of matrix factorizations available in JULIA. The functions in Table 8.9 whose names ends with an exclamation mark are destructive versions of their counterparts without the exclamation mark and hence save memory. The function `factorize` acts as a general interface to the various matrix factorizations. It recognizes the matrix types listed in Table 8.10, determines the most specific type a given matrix has, and then calculates the factorization indicated in the table. The return value can be used as an argument to the left-division operator `\`.

Problems

8.1 Prove the Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1.

8.2 Show that A^*A is a Hermitian matrix.

8.3 Prove Theorem 8.2 and Theorem 8.3.

8.4 (Properties of the cross product)
(a) Show that the cross product is anticommutative.
(b) Show that the cross product is bilinear.
(c) Show that two vectors $\mathbf{a} \neq 0$ and $\mathbf{b} \neq 0$ are parallel if and only if $\mathbf{a} \times \mathbf{b} = 0$.
(d) Show that the Lagrangian identity

$$||\mathbf{a} \times \mathbf{b}||^2 = ||\mathbf{a}||^2 ||\mathbf{b}||^2 - (\mathbf{a} \cdot \mathbf{b})^2$$

holds for all vectors $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^3$.

Table 8.7 Vector and matrix operations (in the module `LinearAlgebra`).

Function	Description
`*`	matrix multiplication
`\`	left division
`/`	right division
`dot`	compute the inner product
`cross`	compute the cross product
`transpose`	compute the transpose
`transpose!`	compute the transpose (destructive version)
`adjoint`	compute the conjugate transpose
`adjoint!`	compute the conjugate transpose (destructive version)
`'`	postfix operator, same as `adjoint`
`det`	compute the determinant
`inv`	compute the inverse (using left division)
`kron`	compute the Kronecker tensor product
`logdet`	compute the logarithm of determinant
`logabsdet`	compute the logarithm of absolute value of determinant
`nullspace`	compute a basis of the nullspace
`rank`	compute the rank by counting the non-zero singular values
`pinv`	compute the Moore–Penrose pseudoinverse
`tr`	compute the trace (sum of diagonal elements)
`norm`	compute the norm of a vector or the operator norm of a matrix
`normalize`	normalize so that the norm becomes one
`normalize!`	destructive version of `normalize`
`diag`	return the given diagonal of the given matrix as a vector
`diagind`	return an **`AbstractRange`** with the indices of the given diagonal
`diagm`	construct a matrix with the given vector as a diagonal
`repeat`	construct an array by repeating the elements of a given one
`tril`	return the lower triangle of a matrix
`tril!`	destructive version of `tril`
`triu`	return the upper triangle of a matrix
`triu!`	destructive version
`cond`	compute the condition number of a matrix
`condskeel`	compute Skeel condition number of a matrix
`givens`	compute a Givens rotation
`lyap`	solve a Lyapunov equation
`sylvester`	solve a Sylvester equation
`peakflops`	compute the peak flop rate of the computer using matrix multiplication

(e) Show that the identity

$$||\mathbf{a} \times \mathbf{b}||^2 = ||\mathbf{a}||^2||\mathbf{b}||^2 - (\mathbf{a} \cdot \mathbf{b})^2$$

holds for all vectors $\mathbf{a}$ and $\mathbf{b} \in \mathbb{R}^3$.

Table 8.8 Functions for checking properties of matrices (in the module `LinearAlgebra`).

Function	Description
`isbanded`	determine whether a matrix is banded
`isdiag`	determine whether a matrix is diagonal
`ishermitian`	determine whether a matrix is Hermitian
`isposdef`	determine whether a matrix is positive definite
`isposdef!`	destructive version of `isposdef`
`issuccess`	determine whether a matrix factorization succeeded
`issymmetric`	determine whether a matrix is symmetric
`istril`	determine whether a matrix is lower triangular
`istriu`	determine whether a matrix is upper triangular

Table 8.9 Functions for matrix factorizations (in the module `LinearAlgebra`).

Function	Description
`factorize`	compute a convenient factorization, general interface to factorizations
`bunchkaufman`	compute the Bunch-Kaufman fact. of a symmetric/Hermitian matrix
`bunchkaufman!`	destructive version of `bunchkaufman`
`cholesky`	compute Cholesky factorization of positive definite matrix
`cholesky!`	destructive version of `cholesky`
`eigen`	compute eigenfactorization
`eigen!`	destructive version of `eigen`
`eigvals`	compute the eigenvalues
`eigvals!`	destructive version of `eigvals`
`eigvecs`	compute the eigenvectors
`eigmin`	compute the smallest eigenvalue if all eigenvalues are real
`eigmax`	compute the largest eigenvalue if all eigenvalues are real
`hessenberg`	compute Hessenberg factorization
`hessenberg!`	destructive version of `hessenberg`
`ldlt`	compute $LDL^\top$ factorization
`ldlt!`	destructive version of `ldlt`
`lq`	compute LQ factorization
`lq!`	destructive version of `lq`
`lu`	compute LU factorization
`lu!`	destructive version of `lu`
`qr`	compute QR factorization
`qr!`	destructive version of `qr`
`schur`	compute Schur factorization
`schur!`	destructive version of `schur`
`ordschur`	reorder Schur factorization
`ordschur!`	destructive version of `ordschur`
`svd`	compute SVD
`svd!`	destructive version of `svd`
`svdvals`	compute singular values and return them in descending order
`svdvals!`	destructive version of `svdvals`

Table 8.10 Forms of matrices recognized by the function `factorize`. The type of the return object and the function called are shown in the second and third columns.

Form	Function
Diagonal	none
Bidiagonal	none
Tridiagonal	`lu`
Lower/upper triangular	none
Positive definite	`cholesky`
Dense symmetric/Hermitian	`bunchkaufman`
Sparse symmetric/Hermitian	`ldlt`
Symmetric real tridiagonal	`ldlt`
General square	`lu`
General non-square	`qr`

8.5 (Volume of parallelepiped) Show that the signed volume V of the parallelepiped with the edges $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ is given by

$$V = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a}) = \mathbf{c} \cdot (\mathbf{a} \times \mathbf{b}).$$

8.6 Prove Theorem 8.4.

8.7 Implement LU factorization, Algorithm 8.6, without row pivoting.

8.8 Implement LU factorization, Algorithm 8.6, with row pivoting.

8.9 Prove Theorem 8.10.

8.10 Prove Theorem 8.17.

8.11 Prove that the product of orthogonal matrices is again an orthogonal matrix.

8.12 Prove that Q^* is orthogonal if Q is an orthogonal matrix.

8.13 Prove Theorem 8.18.

8.14 Prove Theorem 8.19.

8.15 Implement QR factorization, Algorithm 8.20.

8.16 Prove that every eigenspace is a linear subspace, i.e., it is closed under addition and scalar multiplication.

8.17 Prove Theorem 8.27.

8.18 Prove Theorem 8.28.

8.19 Prove Theorem 8.29.

8.20 Show that the matrix

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

is defective by calculating (algebraically, not numerically) all eigenvalues, their algebraic and geometric multiplicities, and all eigenvectors.

8.21 Prove Theorem 8.32.

8.22 Prove Theorem 8.35.

8.23 Prove Theorem 8.36.

8.24 Prove Theorem 8.38.

8.25 Show that two similar matrices A and B have the same eigenvalues. How do the eigenvectors of B relate to those of A?

8.26 Show that the eigenvalues of an upper-triangular or lower-triangular matrix are its diagonal elements.

8.27 Implement QR iteration, Algorithm 8.39.

8.28 Solve several eigenvalue problems numerically by QR iteration after constructing problems with known eigenvalues. Use problems with different eigenvalues close to zero. Plot the error for the eigenvalue closest to zero. Which type of plot is most useful? Which convergence rate to you observe?

8.29 Perform numerical experiments to investigate how convergence is influenced by the absolute value of the eigenvalue closest to zero and by the separation between the two eigenvalues closest to zero. What do you observe?

8.30 Implemented shifted QR iteration using the Wilkinson shift. Use

```
norm(A[n, 1:n-1]) < sqrt(n) * eps(Float64)
```

as the stopping criterion, where `n = size(A, 1)`.

8.31 Compare the convergence rates of standard QR iteration (Problem 8.27) and shifted QR iteration (Problem 8.30).

8.32 Prove Theorem 8.40.

8.33 Implement shifted QR iteration with deflation to find all eigenvalues of a given matrix. Use recursion.

8.34 Implement Hessenberg factorization, Algorithm 8.43.

8.35 Prove Theorem 8.44.

8.36 Suppose $A \in \mathbb{C}^{n\times n}$ has a SVD $A = U\Sigma V^*$ and define

$$B := \begin{pmatrix} 0 & A^* \\ A & 0 \end{pmatrix}.$$

Show that the eigenvalues of B are $\{\pm\sigma_k \mid k \in \{1, \dots, n\}\}$.

8.37 Prove Theorem 8.45.

8.38 Prove Theorem 8.46.

8.39 Prove Theorem 8.47.

8.40 Prove Theorem 8.48.

8.41 Prove Theorem 8.49. Hint: Show that

$$A_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^*,$$

where $\mathbf{u}_k$ and $\mathbf{v}_k$ are the k-th column of U and V, respectively. Then use Theorem 8.47.

8.42 Use Theorem 8.49 to compress an image. Choose a sample image, represent it by a matrix, and use different numbers of singular values to compress the image.

8.43 Find an example of each of the types of matrices listed in Table 8.10, check its type in JULIA, and determine the type of the return value after factorization.

References

1. Cuvelier, F., Japhet, C., Scarella, G.: An efficient way to assemble finite element matrices in vector languages (2014). URL http://arxiv.org/abs/1401.3301. *arXiv:1401.3301 [cs.MS]*
2. Habgood, K., Arel, I.: A condensation-based application of Cramer's rule for solving large-scale linear systems. *Journal of Discrete Algorithms* **10**, 98–109 (2012)
3. Stoer, J., Bulirsch, R.: *Introduction to Numerical Analysis*, 3rd edn. Springer (2002)

Part II
Algorithms for Differential Equations

Chapter 9
Ordinary Differential Equations

Differo, distuli, dilatum (latin, from *dis-* (apart) and *fero* (carry, bear)):
to carry different ways, to spread, to scatter, to disperse, to separate

Differens (present participle of *differo*):
carrying different ways, spreading, scattering, dispersing, separating

Abstract Differential equations are among the most successful models in physics, chemistry, biology, engineering, and many other fields. This chapter is concerned with solving systems of ordinary differential equations. Ordinary differential equations are equations that contain derivatives with respect to only one independent variable. The main result that a system of ordinary differential equations has a unique solution under certain reasonable assumptions is presented. In order to solve the equations numerically, RUNGE–KUTTA formulas are discussed in detail, since they yield excellent results for a wide range of equation types. Finally, the formulas are implemented in an idiomatic manner in JULIA.

9.1 Introduction

The unknown in an ordinary differential equation (ODE) is a function $y : \mathbb{R} \to \mathbb{R}$ of a single independent variable, often of time t or position x, in contrast to partial differential equations, whose unknown functions depend on two or more independent variables (see Chap. 10).

The order of an ODE is the order of the highest derivative of the unknown function that appears in the equation. In abstract terms, any ordinary differential equation of order $n \in \mathbb{N}_0$ can be written in the form

$$G(y^{(n)}(t), y^{(n-1)}(t), \dots, y'(t), y(t)) = 0 \qquad \forall t \in I, \tag{9.1}$$

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_9

where G is a function of all derivatives that occur in the equation and I usually is an interval and possibly all of $\mathbb{R}$.

Since this is a very abstract way of writing an ODE, we now derive an important time dependent ODE that models exponential growth. The example stems from modeling bacterial growth. We denote the amount of bacteria in a Petri dish by $y(t)$ and the known amount of bacteria at the initial time $t = 0$ by y_0, i.e.,

$$y(0) = y_0 \in \mathbb{R}^+.$$

To derive a differential equation, we start with a finite, small time interval of length $\Delta t \in \mathbb{R}^+$. Our modeling assumption is that the equation

$$y(t + \Delta t) = y(t) + \alpha\, \Delta t\; y(t)$$

holds, which is reasonable because it means that the number of bacteria at the end of the small time interval is equal to their number at the beginning of the interval plus a constant $\alpha \in \mathbb{R}$ times the length of the interval times the number of bacteria present (at the beginning of the interval). In other words, the change in the number of bacteria is proportional to the length of the interval and the number of bacteria provided that the time interval is small enough.

By considering the units of the terms in the equation, it becomes clear that the last term must contain a constant factor, because if it did not, the units could not match. More precisely, if we denote the unit of y by $[y]$, comparing the three terms yields $[y] = [\alpha][t][y]$, and thus the unit of the constant factor α is $[\alpha] = 1/[t]$; it is a growth rate. Such considerations are a general principle and they are very useful when assessing constants or parameters in differential equations.

Rearranging the terms in the equation yields

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} = \alpha y(t),$$

which holds for all small time intervals. Therefore we can take the limit as Δt goes to zero on both sides of the equation to obtain the first-order ODE

$$y'(t) = \alpha y(t).$$

In order to fully specify the problem, we also have to give the initial value $y(0) = y_0$ in addition to the equation that holds at all later times. Therefore we arrive at the initial-value problem

$$\begin{aligned} y'(t) &= \alpha y(t) \qquad \forall t \in (0, \infty),\\ y(0) &= y_0, \end{aligned}$$

where we have also noted the time interval.

9.2 Existence and Uniqueness of Solutions *

Do solutions of a given ODE exist? Is the solution unique? These questions are not only of theoretical importance, but they are also valid questions to ask from the modeling and numerical points of view. An ODE that is supposed to model a physical, chemical, biological, or engineering process gains credibility when it is known that it has a unique solution. The existence and uniqueness of a solution is also important whenever the solution of an ODE is to be approximated numerically. What should be approximated unless there is a unique solution?

Here we answer the questions of existence and uniqueness for general first-order initial-value problems under very general assumptions [2, Section 2.8]. We can always write a first-order initial-value problem in the form

$$y'(t) = f(t, y(t)) \qquad \forall t \in (0, \infty), \tag{9.2a}$$

$$y(0) = 0. \tag{9.2b}$$

Here we have assumed that the initial point is the origin $(0, 0)$, but this can always be achieved by a simple substitution. The main result is the following.

Theorem 9.1 (Picard's existence and uniqueness theorem) *Suppose $f : R \to \mathbb{R}$ and $\partial f/\partial y$ are continuous functions in a rectangle $R := [-a, a] \times [-b, b]$ containing the origin. Then there exists a unique solution $y : [-h, h] \to \mathbb{R}$ defined on the interval $[-h, h] \subset [-a, a]$ of the first-order initial-value problem (9.2).*

The solution referred to in this theorem is a classical solution, i.e., a function that is differentiable, that thus can be substituted into equation (9.2), and that satisfies it for every point in the solution interval.

Proof To be able to apply the method used in this proof, we transform the differential equation into an integral equation. This can always be achieved by integrating the equation. If y is a solution of (9.2), then $f(t, y(t))$ is a continuous function by assumption and hence integrable. Integrating (9.2) from the initial point 0 to an arbitrary point t yields the integral equation

$$y(t) = \int_0^t f(s, y(s)) \mathrm{d}s, \tag{9.3}$$

where the initial condition $y(0) = 0$ was used.

Conversely, if y is a solution of the integral equation, the integrand is continuous, and therefore y is differentiable by the fundamental theorem of calculus. Differentiating the integral equation yields (9.2) again, and the initial condition is also satisfied.

In summary, the differential equation (9.2) and the integral equation (9.3) are equivalent.

The proof method used here is called the method of successive approximation or Picard's iteration method. We start with the initial approximation

$$y_0(t) := 0,$$

which satisfies the initial condition. Further approximations are found by using the previous approximations on the right-hand side of the integral equation (9.3) and using it as the definition of the next approximation, i.e., by defining

$$y_{n+1}(t) := \int_0^t f(s, y_n(s))\mathrm{d}s. \tag{9.4}$$

Each function in the sequence $\langle y_n \rangle$ satisfies the initial condition. If there is an $n \in \mathbb{N}_0$ such that $y_n = y_{n+1}$, then y_n is a solution of the differential equation and hence the integral equation, but in general this does not happen.

We therefore consider the limit function of this sequence and will establish that it solves the equation by proceeding in the following steps.

1. Are all elements of the sequence well-defined, differentiable functions?
2. If yes, does the sequence converge?
3. If yes, does the limit function satisfy the integral equation (9.3)?
4. If yes, is the solution unique?

If the last equation can be answered positively, the proof is complete.

1. So far, the approximations y_n have not been fully defined. In addition to (9.4), the domains of definition (and the images) of the functions y_n must also be specified. Here, in particular, the domains of definition must be specified such that $f(s, y_n(s))$ in the integrand of y_{n+1} can be evaluated. Since f is only known to be defined when its second argument is in the interval $[-b, b]$, the domains of definition must be chosen sufficiently small such that y_n lies in the interval $[-b, b]$.

Since f is a continuous function on a closed bounded domain, it is bounded, i.e.,

$$\exists M \in \mathbb{R}_0^+ : \qquad \forall (t, y) \in R : \qquad |f(t, y)| \le M. \tag{9.5}$$

Because $y_n'(t) = f(t, y_{n-1}(t))$, the absolute slope of y_n' is also bounded by M. Hence, because $y_n(0) = 0$, we have $-Mt \le y_n(t) \le Mt$. This consideration implies that the condition that y_n lies in the interval $[-b, b]$ is ensured if $t \le b/M$.

Therefore we define

$$h := \min\left(a, \frac{b}{M}\right)$$

and use the rectangle

$$D := [-h, h] \times [-b, b]$$

as the domain of definition of the functions y_n. The y_n are thus functions $y_n : D \to \mathbb{R}$ and well-defined.

2. The second question is whether the sequence $\langle y_n \rangle$ converges. We start by showing the estimate

$$|y_n(t) - y_{n-1}(t)| \le \frac{ML^{n-1}|t|^n}{n!} \qquad \forall t \in [-h,h] \qquad \forall n \in \mathbb{N} \tag{9.6}$$

by induction. If $n = 1$, then $|y_1(t)| \le M|t|$ follows from the definition (9.4) of y_1 and (9.5). If $n > 1$, we use the Lipschitz condition (see Problem 9.2) to calculate

$$\begin{aligned}
|y_{n+1}(t) - y_n(t)| &\le \int_0^t \Big|f(s, y_n(s)) - f(s, y_{n-1}(s))\Big| \mathrm{d}s \\
&\le L \int_0^t |y_n(s) - y_{n-1}(s)| \mathrm{d}s \\
&\le L \int_0^t \frac{ML^{n-1}|s|^n}{n!} \mathrm{d}s \\
&= \frac{ML^n |s|^{n+1}}{(n+1)!}.
\end{aligned}$$

Since $t \in [-h,h]$, the estimate (9.6) implies

$$|y_n(t) - y_{n-1}(t)| \le \frac{ML^{n-1}h^n}{n!} \qquad \forall t \in [-h,h] \qquad \forall n \in \mathbb{N}, \tag{9.7}$$

whose right-hand side is independent of t.

Next, we write $y_n(t)$ as the telescoping sum

$$y_n(t) = y_0(t) + (y_1(t) - y_0(t)) + \cdots + (y_n(t) - y_{n-1}(t)),$$

which implies

$$|y_n(t)| \le |y_0(t)| + |y_1(t) - y_0(t)| + \cdots + |y_n(t) - y_{n-1}(t)|. \tag{9.8}$$

Using inequality (9.7), we thus find

$$|y_n(t)| \le 0 + \sum_{k=1}^{n} \frac{ML^{k-1}h^k}{k!} = \frac{M}{L} \sum_{k=1}^{n} \frac{Lh^k}{k!} \qquad \forall t \in [-h,h] \qquad \forall n \in \mathbb{N}.$$

The right-hand side converges to

$$\lim_{n\to\infty} \frac{M}{L} \sum_{k=1}^{n} \frac{Lh^k}{k!} = \frac{M}{L}(\mathrm{e}^{Lh} - 1)$$

from below, which implies that

$$|y_n(t)| \le \frac{M}{L}(\mathrm{e}^{Lh} - 1) \qquad \forall t \in [-h,h] \qquad \forall n \in \mathbb{N}.$$

We have hence shown that the sum in (9.8) converges as $n \to \infty$. Therefore the sequence $\langle y_n(t)\rangle$ converges for all $t \in [-h, h]$ as it is a sequence of partial sums of a convergent infinite series.

The bound in (9.7) does not depend on t and hence the bounds in the inequalities in the preceding argument also hold independently of t. Therefore the sequence $\langle y_n\rangle$ even converges uniformly.

Having shown that the sequence $\langle y_n\rangle$ converges uniformly, we denote its limit by

$$y(t) := \lim_{n\to\infty} y_n(t).$$

3. Does the limit function y satisfy the integral equation (9.3)?

We start by taking the limit as n goes to ∞ on both sides of the iteration (9.4), yielding

$$y(t) = \lim_{n\to\infty} \int_0^t f(s, y_n(s))\mathrm{d}s.$$

Since the sequence $\langle y_n\rangle$ converges uniformly, we can interchange taking the limit and integration (see Problem 9.3) to obtain

$$y(t) = \int_0^t \lim_{n\to\infty} f(s, y_n(s))\mathrm{d}s.$$

Since the function f is continuous in its second argument, we can take the limit inside its second argument to find

$$y(t) = \int_0^t f(s, \lim_{n\to\infty} y_n(s))\mathrm{d}s = \int_0^t f(s, y(s))\mathrm{d}s.$$

The last equation means that y solves the integral equation and hence the differential equation by the discussion at the beginning of the proof.

4. Is the solution unique? Suppose there is another solution z. Then

$$y(t) - z(t) = \int_0^t \big(f(s, y(s)) - f(s, z(s))\big)\mathrm{d}s \qquad \forall t \in [0, a]$$

holds for their difference, and furthermore we have

$$|y(t) - z(t)| \le \int_0^t \big|f(s, y(s)) - f(s, z(s))\big|\mathrm{d}s \qquad \forall t \in [0, a].$$

Using Problem 9.2, the last inequality implies that

$$|y(t) - z(t)| \le L \underbrace{\int_0^t |y(s) - z(s)|\mathrm{d}s}_{U(t):=} \qquad \forall t \in [0, a].$$

We denote the integral on the right-hand side by $U(t)$. The function U is differentiable, and we obviously have

$$U(0) = 0 \tag{9.9}$$

and

$$U(t) \geq 0 \qquad \forall t \in [0, a]. \tag{9.10}$$

Using U, the last inequality becomes

$$U'(t) - LU(t) \leq 0 \qquad \forall t \in [0, a].$$

By multiplying this inequality by e^{-Lt}, we find that it is equivalent to

$$\left(\mathrm{e}^{-Lt} U(t)\right)' \leq 0 \qquad \forall t \in [0, a].$$

Integrating this inequality from zero to t and using (9.9), we find the inequality

$$\mathrm{e}^{-Lt} U(t) \leq 0 \qquad \forall t \in [0, a].$$

The last inequality and (9.10) imply $U(t) = 0$ for all $t \in [0, a]$ and hence $U'(t) = |y(t) - z(s)| = 0$. In other words, any two solutions y and z are identical for $t \in [0, a]$. An analogous argument shows that the solution is unique for all $t \in [-a, 0]$.

This completes the proof. □

An alternative proof is based on observing that the operator given by the Picard iteration (9.4) is a contraction and then using the Banach fixed-point theorem (see Problem 9.4).

The result of the theorem is that the solution exists in a finite (and possibly very small) time interval. Can we do better? In general, a stronger result cannot be expected. The most prominent and simple counterexample is the equation $y'(t) = y(t)^2$ with the initial condition $y(0) = y_0$. Separation of variables shows that its solution is $y(t) = 1/(y_0 - t)$ if $y_0 \neq 0$ and $y = 0$ if $y_0 = 0$. If $y_0 > 0$, then the solution exists only in the interval $t \in [0, y_0)$ and even becomes unbounded even within a finite amount of time.

9.3 Systems of Ordinary Differential Equations

A linear ODE is a special case of the general form (9.1) and has the form

$$a_n(t) y^{(n)}(t) + a_{n-1}(t) y^{(n-1)}(t) + \cdots + a_1 y'(t) + a_0 y(t) = b(t) \qquad \forall t \in I,$$

whose defining feature is that all terms that contain the unknown y are linear in y.

A linear ODE of order n can always be written as a linear system of n first-order ODEs. The idea is to introduce new variables for the higher-order derivatives. We hence define

$$\begin{aligned} z_0 &:= y, \\ z_1 &:= y', \\ &\vdots \\ z_{n-1} &:= y^{(n-1)} \end{aligned}$$

and can now write the linear equation as the linear system

$$\begin{aligned} z_0' &= z_1, \\ z_1' &= z_2, \\ &\vdots \\ z_{n-2}' &= z_{n-1}, \\ a_n z_{n-1}' &= b - a_{n-1}z_{n-1} - a_{n-2}z_{n-2} - \cdots - a_1 z_1 - a_0 z_0 \end{aligned}$$

in the interval I. There are n equations for the n variables $z_0, \dots, z_{n-1}$. The last equation stems from the original equation, while the other equations connect the new variables.

This consideration underlines the importance of (linear) systems of first-order ODEs. Any linear ODE of order n for a single unknown function can be written in this form, and linear problems with more unknowns can also be written in this form. Therefore most numerical programs for ODEs have been developed for systems of first-order equations.

9.4 Euler Methods

In the rest of this chapter, numerical methods for the approximation of solutions of ODEs are presented. Although many sophisticated methods for finding solutions of ODEs in closed form have been developed, the solutions can generally not be written in closed form. A simple counterexample is the ODE $y'(t) = f(t)$ with the initial condition $y(0) = 0$. Its solution is the integral $y(t) = \int_0^t f(s)\mathrm{d}s$. But the antiderivative of an elementary function (in the sense of differential algebra this is a function that can be written in closed algebraic form) is not necessarily elementary; the most prominent example is the function

$$f(t) := \mathrm{e}^{-t^2}.$$

The Risch algorithm is a decision procedure that answers the question whether an elementary function has an elementary antiderivative or not [6, 7].

Therefore calculating precise approximations of the solutions in an efficient manner is an important practical task.

9.4.1 Forward and the Backward Euler Methods

The most straightforward idea to solve any differential equation is to use the definition of the derivative and to replace it by its difference quotient. We start from the general first-order equation

$$y'(t) = f(t, y), \qquad y(t_0) = y_0$$

and assume that it has a unique solution (see Sect. 9.2). We also define a sequence of points t_n such that $t_0 < t_1 < \cdots < t_n < t_{n+1} < \cdots$ and denote the approximation of $y(t_n)$ by y_n. Replacing the derivative by its forward difference quotient yields

$$\frac{y_{n+1} - y_n}{t_{n+1} - t_n} \approx y'(t_n) = f(t_n, y(t_n))$$

necessitating that $t_{n+1} - t_n$ is small. This motivates the definition

$$y_{n+1} := y_n + (t_{n+1} - t_n) f(t_n, y_n), \tag{9.11}$$

which is called the forward Euler method.

Algorithm 9.2 (forward Euler method) Input: the right-hand side f, the initial value y_0, and points $t_0 < t_1 < \cdots < t_N$ or a step size h. If the step size h is given, the points are $t_n := t_0 + nh$.

1. Loop for n from 1 to N: set

$$y_{n+1} := y_n + (t_{n+1} - t_n) f(t_n, y_n).$$

2. Return the values y_n.

Alternatively, we can also use the backward difference quotient instead of the forward difference quotient in the derivation above. Then we approximate the derivative by

$$\frac{y_n - y_{n-1}}{t_n - t_{n-1}} \approx y'(t_n) = f(t_n, y(t_n)),$$

which motivates to define the approximation y_{n+1} as the solution of the (algebraic) equation

$$y_{n+1} = y_n + (t_{n+1} - t_n) f(t_{n+1}, y_{n+1})$$

after shifting the index n by one. This is called the backward Euler method. In contrast to the forward Euler method, this formula is not an explicit formula for y_{n+1}, but defines y_{n+1} only implicitly. Therefore an algebraic equation must be solved in each time step.

Algorithm 9.3 (backward Euler method) Input: the right-hand side f, the initial value y_0, and points $t_0 < t_1 < \cdots < t_N$ or a step size h.

1. Loop for n from 1 to N: set y_{n+1} to be the solution of the (algebraic) equation

$$y_{n+1} = y_n + (t_{n+1} - t_n) f(t_{n+1}, y_{n+1}).$$

2. Return the values y_n.

9.4.2 Truncation Errors of the Forward Euler Method

The difference between the exact solution of the ODE and its numerical approximation is called the global truncation error. It stems from two causes (ignoring the round-off error). The first cause is the use of an approximate formula to calculate y_{n+1} from the previous approximation y_n (assuming that the previous approximation was exact, i.e., $y(t_n) = y_n$). This cause of errors is called the local truncation error; it is the error due to the use of an approximate formula only. The second cause is the fact that the input used in each step is only approximatively correct since $y(t_n)$ is not equal to y_n in general, also because the previous errors accumulate.

Another fundamental source of errors arises from performing the computations in arithmetic with only a finite number of digits. This error is called the round-off error and is not considered here.

In the following, we focus on the local truncation error

$$e_{n+1} := y_{n+1} - y(t_{n+1}),$$

i.e., the difference between the approximation y_{n+1} at t_{n+1} and the value $y(t_{n+1})$ of the exaction solution y at t_{n+1} *while assuming that* $y(t_n) = y_n$.

Theorem 9.4 (local truncation error of the forward Euler method) *Suppose the exact solution y exists uniquely and that it is twice differentiable in the open interval* (t_n, t_{n+1}) *and continuously differentiable in the closed interval* $[t_n, t_{n+1}]$. *Then the local truncation error* e_{n+1} *of the forward Euler method is given by*

$$e_{n+1} = -\frac{1}{2} h^2 y''(\tilde{t}_n) \qquad \exists \tilde{t}_n \in (t_n, t_{n+1}).$$

Proof Taylor expansion of the exact solution y at $t_{n+1} = t_n + h$ around the point t_n and using the Lagrange form of the remainder term yields

$$y(t_{n+1}) = y(t_n) + h y'(t_n) + \frac{h^2}{2} y''(\tilde{t}_n),$$

where $\tilde{t}_n \in (t_n, t_{n+1})$. Subtracting the Taylor expansion from the forward Euler method (9.11) yields

$$e_{n+1} = y_n - y(t_n) + h(f(t_n, y_n) - y'(t_n)) - \frac{h^2}{2} y''(\tilde{t}_n). \tag{9.12}$$

Recalling that we assume that $y(t_n) = y_n$ when considering the local truncation error, we also have $y'(t_n) = f(t_n, y(t_n)) = f(t_n, y_n)$. This simplifies the local truncation error to

$$e_{n+1} = -\frac{h^2}{2} y''(\tilde{t}_n)$$

as claimed. □

Hence the local truncation error is proportional both to the square of the step size h and to the second derivative of the solution somewhere in the interval $[t_n, t_{n+1}]$. If a bound M of the absolute value of the second derivative is known on the whole interval where the solution is approximated, we can write

$$|e_n| \le \frac{h^2 M}{2}.$$

Hence it can be ensured that the local truncation error is less than or equal to ϵ if the inequality

$$h \le \sqrt{\frac{2\epsilon}{M}}$$

holds. It is also clear from the proof that a bound on the global truncation error will require a bound on the second derivative of the solution.

Analyzing the global truncation error, which is defined as

$$E_n := y_n - y(t_n)$$

while not assuming that $y(t_n) = y_n$, is more involved.

Theorem 9.5 (global truncation error of the forward Euler method) *Suppose that $t_n := t_0 + nh$ ($h \in \mathbb{R}^+$), that f is continuous, that f is Lipschitz continuous with respect to its second argument with Lipschitz constant L, and that the exact solution y is twice differentiable in the open interval $(0, t_n)$ and continuously differentiable in the closed interval $[0, t_n]$. Then the global truncation error E_n of the forward Euler method is bounded by*

$$|E_n| \le \frac{\mathrm{e}^{(t_n - t_0)L} - 1}{L} \beta h,$$

where $\alpha := 1 + hL$ *and* $\beta := (1/2) \max_{t \in (t_0, t_n)} |y''(t)|$.

If $\partial f / \partial t$ is continuous in the interval $[t_0, t_n]$, then the solution y has a continuous second derivative on this interval and hence the assumptions on the smoothness of y are satisfied.

Proof Equation (9.12) and the Lipschitz continuity of f with respect to its second argument imply that

$$|E_{n+1}| \le |E_n| + h|f(t_n, y_n) - f(t_n, y(t_n))| + \frac{h^2}{2}|y''(\tilde{t}_n)| \le \alpha|E_n| + \beta h^2.$$

It is straightforward to show by induction that $E_0 = 0$ and the last inequality $|E_{n+1}| \le \alpha|E_n| + \beta h^2$ imply that

$$|E_n| \le \frac{\alpha^n - 1}{\alpha - 1}\beta h^2.$$

Note that $\alpha > 1$ can be assumed without loss of generality.

Substituting the definition of α into the last estimate yields

$$|E_n| \le \frac{(1 + hL)^n - 1}{L}\beta h.$$

The Taylor expansion of the exponential function shows that $1 + hL \le \mathrm{e}^{hL}$ and hence $(1 + hL)^n \le \mathrm{e}^{nhL}$.

In summary, we find that

$$|E_n| \le \frac{\mathrm{e}^{nhL} - 1}{L}\beta h = \frac{\mathrm{e}^{(t_n - t_0)L} - 1}{L}\beta h,$$

which concludes the proof. □

Since the global truncation error has order one in the step size h, the forward Euler method is called a first-order method. Much effort has been devoted to the development of higher-order methods, and we discuss such methods in the rest of this chapter.

9.4.3 Improved Euler Method

Another view towards deriving formulas for numerical approximations is not to approximate the derivative as we have done until now, but to approximate the integral in the equivalent formulation as an integral equation (see Sect. 9.2). The improved Euler method can be derived in this manner.

We start by considering the initial-value problem

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

and recall the equivalent integral-equation formulation

$$y(t_{n+1}) = y(t_n) + \int_{t_n}^{t_{n+1}} f(s, y(s))\mathrm{d}s$$

from Sect. 9.2, now written for arbitrary initial points.

Next, we approximate the integral. We recover the forward Euler method by using the approximation

$$\int_{t_n}^{t_{n+1}} f(s, y(s))\mathrm{d}s \approx hf(t_n, y_n)$$

and the backward Euler formula by using

$$\int_{t_n}^{t_{n+1}} f(s, y(s))\mathrm{d}s \approx hf(t_{n+1}, y_{n+1}).$$

We have replaced the integrand by its value $f(t_n, y_n)$ on the left interval endpoint in the case of the forward Euler formula (recall the forward difference) and by its value $f(t_{n+1}, y_{n+1})$ on the right interval endpoint in the case of the backward Euler formula (recall the backward difference).

Both choices seem to be one-sided and arbitrary. It is more prudent to use the approximation

$$\int_{t_n}^{t_{n+1}} f(s, y(s))\mathrm{d}s \approx \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1})),$$

approximating the integral by the area of a trapezoid, which motivates to define y_{n+1} as the solution of the equation

$$y_{n+1} = y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_{n+1})).$$

Unfortunately, this is only an implicit definition of y_{n+1}. We can arrive at an explicit formula if we replace the occurrence of y_{n+1} in the last term by its approximation $y_n + hf(t_n, y_n)$ according to the forward Euler formula.

In summary, we define

$$y_{n+1} := y_n + \frac{h}{2}(f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n)), \tag{9.13}$$

which is the improved Euler method. Its advantage is that its local truncation error has order three as we will show next. Its disadvantage is that the evaluation of f on the right-hand side proceeds in two steps and requires two evaluations of f, which is more costly.

Theorem 9.6 (local truncation error of improved Euler method) *Suppose the exact solution y exists uniquely and that it is three times differentiable in the open interval (t_n, t_{n+1}) and twice continuously differentiable in the closed interval $[t_n, t_{n+1}]$. Then the local truncation error of the improved Euler method has order three in the step size h.*

Proof Recall that the local truncation error is given by

$$e_{n+1} = y_{n+1} - y(t_{n+1})$$

while assuming that $y_n = y(t_n)$.

Taylor expansion of the exact solution y at $t_{n+1} = t_n + h$ around the point t_n and using the Lagrange form of the remainder term yields

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2!}y''(t_n) + \frac{h^3}{3!}y'''(\tilde{t}_n),$$

where $\tilde{t}_n \in (t_n, t_{n+1})$. Subtracting the Taylor expansion from the improved Euler method (9.13) yields

$$\begin{aligned} e_{n+1} = y_n &+ \frac{h}{2}\big(f(t_n, y_n) + f(t_{n+1}, y_n + hf(t_n, y_n))\big) \\ &- \Big(y(t_n) + hy'(t_n) + \frac{h^2}{2!}y''(t_n) + \frac{h^3}{3!}y'''(\tilde{t}_n)\Big) \\ = -\frac{h}{2}f(t_n, y_n) &+ \frac{h}{2}f(t_{n+1}, y_n + hf(t_n, y_n)) - \frac{h^2}{2!}y''(t_n) - \frac{h^3}{3!}y'''(\tilde{t}_n). \end{aligned}$$

The two-dimensional Taylor expansion of the term $f(t_{n+1}, y_n + hf(t_n, y_n))$ around the point (t_n, y_n) is

$$f(t_{n+1}, y_n + hf(t_n, y_n)) = f(t_n, y_n) + hf_t(t_n, y_n) + hf(t_n, y_n)f_y(t_n, y_n) + O(h^2).$$

Using the ODE and the chain rule, the second derivative of y can be written as

$$y''(t_n) = f_t(t_n, y(t_n)) + f_y(t_n, y(t_n))y'(t_n) = f_t(t_n, y_n) + f_y(t_n, y_n)f(t_n, y_n).$$

Substituting these last two equations into the expression for e_{n+1} shows that

$$e_{n+1} = O(h^3),$$

which concludes the proof. □

9.5 Variation of Step Size

It is often conducive to adjust the step size in order to maintain the local truncation error at a nearly constant level. Not only can computational work be saved in this manner, but it is also possible to control the accuracy of the approximation.

The most straightforward way to control the local truncation error would be to calculate the difference between the approximation and the exact solution. While this approach is a good idea in test problems, where the exact solution is known, it is obviously not possible to do so in the general setting; if the exact solution were known, we would not approximate it. Therefore we use a more accurate numerical method as a substitute for the exact solution and compute two different approximations using two different methods.

For example, we can use the forward Euler method (as the less accurate method) and the improved Euler method (as the more accurate method). Then the difference between the two approximate solutions is used as the estimate

$$e_{n+1}^{\text{est}} := \left|y_{n+1}^{\text{Euler}} - y_{n+1}^{\text{improved}}\right|$$

of the error of the less accurate method.

If the estimated error e_{n+1}^{est} does not match a given error tolerance ϵ, then the step size h is adjusted and the calculations are repeated. It is important to know how the local truncation error e_{n+1} depends on the step size h so that it can be adjusted efficiently. In the case of the Euler method, used as the less accurate method in this example, the local truncation error e_{n+1} is proportional to h^2 (see Theorem 9.4), which means that multiplying the step size h by

$$\sqrt{\frac{\epsilon}{e_{n+1}^{\text{est}}}}$$

adjusts the local truncation error (up or down) to the given error tolerance ϵ.

In this way, the local truncation error can be kept approximately constant throughout the approximation of a solution. Small step sizes, which increase computation time, are only used where needed so that the resulting algorithm is both efficient and accurate (see Problem 9.5).

9.6 RUNGE–KUTTA Methods

Two of the most often executed programs in the history of ordinary differential equations are probably the functions called `ode23` and `ode45` in MATLAB. These two functions use adaptive RUNGE–KUTTA methods [4, 1, 8], and therefore we have a closer look at these methods in the rest of this chapter.

We still consider the initial-value problem

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0 \tag{9.14}$$

and start by defining the (classical) RUNGE–KUTTA method

$$k_1 := f(t_n, y_n), \tag{9.15a}$$

$$k_2 := f\Big(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\Big), \tag{9.15b}$$

$$k_3 := f\Big(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\Big), \tag{9.15c}$$

$$k_4 := f(t_n + h, y_n + hk_3), \tag{9.15d}$$

$$y_{n+1} := y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{9.15e}$$

$$t_{n+1} := t_n + h. \tag{9.15f}$$

It is called a four-stage method, since the four stages k_1, k_2, k_3, and k_4 are needed to proceed from time t_n to time t_{n+1}. This classical RUNGE–KUTTA method is therefore often abbreviated as RK4.

If f does not depend on y, then the method (9.15) simplifies to

$$y_{n+1} = y_n + \frac{h}{6}\Big(f(t_n) + 4f\Big(t_n + \frac{h}{2}\Big) + f(t_n + h)\Big),$$

which is Simpson's rule for approximating the integral of $y'(t) = f(t)$. This consideration is analogous to the interpretation of the improved Euler method as an application of the trapezoid rule to an integral in Sect. 9.4.3.

Generalizing (9.15), RUNGE–KUTTA methods with s stages can be written in the form

$$k_i := f\Big(t_n + hc_i, y_n + h\sum_{j=1}^{s} a_{ij}k_j\Big), \qquad 1 \le i \le s, \tag{9.16a}$$

$$y_{n+1} := y_n + h\sum_{i=1}^{s} b_i k_i, \tag{9.16b}$$

$$t_{n+1} := t_n + h. \tag{9.16c}$$

The following two theorems mean that the RK4 method has order four; more precisely, its local truncation error has order five and its global truncation error has order four.

Theorem 9.7 (local truncation error of the RK4 method) *Suppose that the fourth partial derivatives of f in the ordinary differential equation (9.14) exist in the open interval (t_n, t_{n+1}) and that its third partial derivatives exist and are continuous in the closed interval $[t_n, t_{n+1}]$. Then the local truncation error of the* RK4 *method (9.15) has order five, i.e.,*

$$e_{n+1} = y_{n+1} - y(t_{n+1}) = O(h^5).$$

Theorem 9.8 (global truncation error of the RK4 method) *Under the assumptions of Theorem 9.7, the global truncation error of the* RK4 *method (9.15) has order four.*

These two theorems are shown in Problems 9.6 and 9.7.

RUNGE–KUTTA methods can be classified into explicit and implicit methods. The system (9.16) of equations is explicit if the coefficients a_{ij} vanish for $j < i$, corresponding to an explicit method. In practice, explicit methods are used because calculating the stages k_i is faster compared to implicit methods and because explicit methods already enable a large choice of coefficients.

The RK4 method in (9.15) is a four-stage method and has order four. How do the number of stages s and the order p relate in explicit RUNGE–KUTTA methods? In general, it can be shown that the inequality

$$p \le s$$

holds for any explicit RUNGE–KUTTA method; if $p \ge 5$, then the stronger inequality

$$p < s$$

holds [3, Paragraph 324].

It is not known, however, whether these inequalities are sharp. It is an open problem what the minimum number of stages s of an explicit RUNGE–KUTTA method with order p is in the cases where no methods are already known that satisfy $p + 1 = s$. The following table summarizes what is known about the known minimum number of stages for orders one to ten [3, Chapter 32].

Order p	1	2	3	4	5	6	7	8	9	10
Number s of stages	1	2	3	4	6	7	9	11	?	17

9.7 Butcher Tableaux

The coefficients a_{ij}, b_i, and c_i in the general form (9.16) of a RUNGE–KUTTA method can be arranged in so-called Butcher tableaux. The Butcher tableau of an explicit RUNGE–KUTTA method has the form

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s,
\end{array}$$

while the tableau of an implicit RUNGE–KUTTA method would have nonzero a_{ij} entries in or above the diagonal. In the following, we only consider explicit RUNGE–KUTTA methods.

Calculations such as the ones in Problem 9.6 show that an explicit RUNGE–KUTTA method is consistent if

$$\sum_{j=1}^{i-1} a_{ij} = c_i \qquad \forall i \in \{2, \dots, s\}$$

holds.

In the following, in this section and the next, the Butcher tableaux of important RUNGE–KUTTA methods are given. The forward Euler method (see Sect. 9.4.1) is the simplest RUNGE–KUTTA method and has the Butcher tableau

$$\begin{array}{c|c} 0 & \\ \hline & 1. \end{array}$$

It is the only consistent explicit one-stage RUNGE–KUTTA method. (The backward Euler method (see Sect. 9.4.1 is an implicit method and therefore not considered here.)

A family of second-order two-stage RUNGE–KUTTA methods has the Butcher tableau

$$\begin{array}{c|cc} 0 & & \\ \alpha & \alpha & \\ \hline & (1-1/(2\alpha)) & 1/(2\alpha). \end{array}$$

The case $\alpha = 1/2$ is called the midpoint method. In the case $\alpha = 1$, we recover the improved Euler method (see Sect. 9.4.3).

Finally, the classical RUNGE–KUTTA or RK4 method has the Butcher tableau

$$\begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6. \end{array}$$

9.8 Adaptive RUNGE–KUTTA Methods

The basic idea of adaptive RUNGE–KUTTA methods was already discussed in Sect. 9.5. In an adaptive RUNGE–KUTTA method, two methods, one of order p and one of order $p-1$, are used to obtain an estimate of the local truncation error. In order to keep the computational cost as small as possible, the stages of both RUNGE–KUTTA methods are identical; only the linear combinations of the k_i, i.e., the last lines in the Butcher tableaux, differ. The orders of the methods are known and therefore the algorithm for adjusting the step size in Sect. 9.5 can be used. In this way, the step size h is always almost optimal.

More concretely, we can write the two RUNGE–KUTTA methods as

$$\begin{aligned} y_{n+1}^* &:= y_n^* + h\sum_{i=1}^{s} b_i^* k_i, \\ y_{n+1} &:= y_n + h\sum_{i=1}^{s} b_i k_i, \\ t_{n+1} &:= t_n + h, \end{aligned}$$

where the asterisk indicates the method with order $p-1$ and the other method has order p. Then the local truncation error e_{n+1} is estimated by

$$e_{n+1} \approx e_{n+1}^{\text{est}} := y_{n+1}^* - y_{n+1} = h\sum_{i=1}^{s}(b_i^* - b_i)k_1 = O(h^p).$$

The local truncation error of the lower-order method is proportional to h^p, since its order is $p-1$. Therefore the step size h is multiplied by

$$\left(\frac{\epsilon}{e_{n+1}^{\text{est}}}\right)^{1/p}$$

in order to adjust the local truncation error (up or down) to the given error tolerance ϵ.

The last two lines of the Butcher tableau

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & b_1^* & b_2^* & \cdots & b_{s-1}^* & b_s
\end{array}$$

of an adaptive RUNGE–KUTTA method contain the coefficients b_i and b_i^*.

The simplest adaptive RUNGE–KUTTA method combines the forward Euler and the improved Euler methods as already discussed in Sect. 9.5. Its Butcher tableau is

$$\begin{array}{c|cc}
0 & & \\
1 & 1 & \\
\hline
 & 1/2 & 1/2 \\
 & 1 & 0.
\end{array}$$

A well-known example of an adaptive RUNGE–KUTTA method is the so-called RUNGE–KUTTA–FEHLBERG method [5], which uses two methods of orders four and five. Its Butcher tableau is

$$\begin{array}{c|cccccc}
0 & & & & & & \\
1/4 & 1/4 & & & & & \\
3/8 & 3/32 & 9/32 & & & & \\
12/13 & 1932/2197 & -7200/2197 & 7296/2197 & & & \\
1 & 439/216 & -8 & 3680/513 & -845/4104 & & \\
1/2 & -8/27 & 2 & -3544/2565 & 1859/4104 & -11/40 & \\
\hline
 & 16/135 & 0 & 6656/12825 & 28561/56430 & -9/50 & 2/55 \\
 & 25/216 & 0 & 1408/2565 & 2197/4104 & -1/5 & 0.
\end{array}$$

These ideas can immediately be applied to systems of ordinary differential equations (see Sect. 9.3, Problem 9.10, and Problem 9.11).

9.9 Implementation of Runge–Kutta Methods

After discussing the theory of Runge–Kutta methods, two different implementations are presented in this section. Both implementations take a Butcher tableau as input, but they differ in how they use it. `NaN`, short for "not a number," is a floating-point number, and we use it here for values that should never be accessed.

```
const RK1 = [0 NaN;
             NaN 1]

const RK4 = [0    NaN  NaN  NaN  NaN;
             1/2  1/2  NaN  NaN  NaN;
             1/2  0    1/2  NaN  NaN;
             1    0    0    1    NaN;
             NaN  1/6  1/3  1/3  1/6]
```

The first implementation is a straightforward use of the general Runge–Kutta equations (9.16) with a constant step size `h`.

```
function RK(T::Array{Float64, 2}, f::Function,
            t_start::Float64, t_end::Float64,
            y_start::Float64, h::Float64)::NamedTuple
    @assert T[1, 1] == 0
    @assert h > 0

    local A = T[1:end-1, 2:end-1]
    local b = T[end,     2:end]
    local c = T[1:end-1, 1]
    local s = size(b, 1)
    local N = ceil(Int, (t_end - t_start) / h) + 1

    @assert c[1] == 0
    @assert all(isapprox(c[i], sum(A[i, j] for j in 1:i-1))
                for i in 2:size(A, 1))

    local k = fill(NaN, s)
    local t = LinRange(t_start, t_start + (N-1)*h, N)
    local y = fill(NaN, N)
    y[1] = y_start

    for n in 1:N-1
        k[1] = f(t[n], y[n])
        for i in 2:s
            k[i] = f(t[n] + h * c[i],
                     y[n] + h * sum(A[i, j] * k[j] for j in 1:i-1))
        end
        y[n+1] = y[n] + h * sum(b[i] * k[i] for i in 1:s)
    end

    (t = t, y = y)
end
```

After some assertions to check the consistency of the input and after extracting the matrix `A` and the vectors `b` and `c` from the Butcher tableau, the coefficient vector `k` and the output vectors `t` and `y` are allocated and initialized. In the **`for`** loop, the equation is solved using (9.16). (Unfortunately, `sum` does not work on empty generators so that `k[1]` is defined separately.)

This implementation is a straightforward implementation of equations (9.16). It is important to note, however, that the Butcher tableau and the number of stages are known and constant. Therefore it seems wasteful in the inner loop to use a **`for`** loop to iterate over the stages, to access the coefficients stored in a matrix and in vectors, and to use `sum` to sum a few terms instead of writing out the expressions explicitly. But writing out the expressions explicitly would have to be done for every Butcher tableau, and we still want a general, yet efficient code.

The solution is to not write a program to solve the equation, but to write a program that writes programs to solve the equation. In other words, we will write a macro (see Chap. 7) that generates the code specialized for a given Butcher tableau and right-hand side.

The first version of the `@RK` macro is more straightforward and easier to understand, while the second version is an optimized one. We start with the first version, called `@RK0`.

```
macro RK0(T, f, t_start::Float64, t_end::Float64, y_start::Float64,
          h::Float64)
    local TT = eval(T)

    @assert isa(TT, Array{Float64, 2})
    @assert TT[1, 1] == 0
    @assert h > 0

    local A = TT[1:end-1, 2:end-1]
    local b = TT[end,     2:end]
    local c = TT[1:end-1, 1]
    local s = size(b, 1)
    local N = ceil(Int, (t_end - t_start) / h) + 1
    local k = [gensym("k") for i in 1:s]

    @assert c[1] == 0
    @assert all(isapprox(c[i], sum(A[i, j] for j in 1:i-1))
                for i in 2:size(A, 1))
    local y_update = :(0)
    for i in 1:s
        y_update = :($y_update + $(b[i]) * $(esc(k[i])))
    end

    local ks = :()
    for i in 1:s
        local sum = :(0)
        for j in 1:i-1
            sum = :($sum + $(h * A[i, j]) * $(esc(k[j])))
        end
        ks = :($ks; local $(esc(k[i])) = $f(t[n] + $(h * c[i]),
                                            y[n] + $sum))
    end

    quote
        local t = LinRange($t_start, $(t_start + (N-1)*h), $N)
        local y = fill(NaN, $N)
        y[1] = $y_start

        for n in 1:$(N-1)
            $ks
            y[n+1] = y[n] + $h * $y_update
        end

        ($(esc(:t)) = t, $(esc(:y)) = y)
    end
end
```

After some assertions and the definitions of local variables such as `A`, `b`, and `c`, the first **for** loop builds the expression `y_update` that is used in the macro expansion where `y[n+1]` is updated. The local variable `y_update` is initialized as the expression `0` and then the terms $b_i k_i$ are added in the **for** loop. The vector `k` contains already unique symbols generated by `gensym`, and therefore `esc` is used.

The expressions for the stages k_i are built in a similar manner. The outer **for** loop adds definitions of local variables to the initially empty expression `ks`. The names of the local variables are the elements of the vector `k`. Each symbol stored in `k[i]` has a unique name that starts with a `k`. The inner **for** loop adds terms to the expression stored in `sum`, analogous to the generation of `y_update`.

All this work is performed during macro-expansion time. The advantage is that the expressions contain the entries of the Butcher tableau and do not have to access vectors or arrays during run time.

At the end of the macro, a **quote** expression returns the code that is executed. First, the local variables `t` and `y` are initialized and will contain the results. Then, in the **for** loop, the solution is calculated: the expression `ks` calculates the stages and then the next element of `y` is calculated. Finally, `t` and `y` are returned. Because of all the preparatory work before the **`quote`** expression, the whole **`quote`** expression is rather short.

It is instructive to use `@macroexpand1` to see what the code that solves the equation looks like. You will notice that the entries of the Butcher tableau have been substituted into the code.

The first version of the macro is already faster than the `RK` function, but some improvements are still possible. This leads us to the second version, called `@RK`.

```
macro RK(T, f, t_start::Float64, t_end::Float64, y_start::Float64,
         h::Float64)
    local TT = eval(T)

    @assert isa(TT, Array{Float64, 2})
    @assert TT[1, 1] == 0
    @assert h > 0

    local A = TT[1:end-1, 2:end-1]
    local b = TT[end,     2:end]
    local c = TT[1:end-1, 1]
    local s = size(b, 1)
    local N = ceil(Int, (t_end - t_start) / h) + 1
    local k = [gensym("k") for i in 1:s]

    @assert c[1] == 0
    @assert all(isapprox(c[i], sum(A[i, j] for j in 1:i-1))
                for i in 2:size(A, 1))

    local y_update = :($(h * b[1]) * $(esc(k[1])))
    for i in 2:s
        y_update = :($y_update + $(h * b[i]) * $(esc(k[i])))
    end

    local ks = :(local $(esc(k[1])) = $f(t[n], y[n]))
    for i in 2:s
        local sum = :($(h * A[i, 1]) * $(esc(k[1])))
        for j in 2:i-1
            sum = :($sum + $(h * A[i, j]) * $(esc(k[j])))
        end
        ks = :($ks; local $(esc(k[i])) = $f(t[n] + $(h * c[i]),
                                            y[n] + $sum))
    end

    quote
        local t = LinRange($t_start, $(t_start + (N-1)*h), $N)
        local y = fill(NaN, $N)
        y[1] = $y_start

        for n in 1:$(N-1)
            $ks
            y[n+1] = y[n] + $y_update
        end

        ($(esc(:t)) = t, $(esc(:y)) = y)
    end
end
```

In this second version, subexpressions of the form `0 + ...` have been eliminated and the value of `ks` has been streamlined. Furthermore, the multiplications with `h` are already performed in the preamble of the macro and not at run time. These changes result in faster code, and it is instructive to compare the two macro expansions using `@macroexpand1`.

Finally in this section, we use a small benchmark to compare the speeds and accuracies of the forward Euler method and the classical RUNGE–KUTTA method, both solved using the `RK` function and the `@RK` macro. The very simple ODE used here is the initial-value problem $y'(t) = y(t)$, $y(0) = 1$, whose solution is the function $y(t) = \mathrm{e}^t$, whose value at $t = 10$ is e^{10}.

```
function benchmark()
    local sol1 = @time   RK(RK1, (t, y) -> y, 0.0, 10.0, 1.0, 1e-6)
    local sol2 = @time  @RK(RK1, (t, y) -> y, 0.0, 10.0, 1.0, 1e-6)
    local sol3 = @time   RK(RK4, (t, y) -> y, 0.0, 10.0, 1.0, 1e-6)
    local sol4 = @time  @RK(RK4, (t, y) -> y, 0.0, 10.0, 1.0, 1e-6)

    @show sol1[:y][end]
    @show sol2[:y][end]
    @show sol3[:y][end]
    @show sol4[:y][end]
    @show exp(10.0)

    nothing
end
```

```
julia> benchmark()
 0.176908 seconds (20.00 M allocations: 534.058 MiB, 15.12% gc time)
 0.027769 seconds (2 allocations: 76.294 MiB)
 0.749787 seconds (80.00 M allocations: 1.863 GiB, 21.85% gc time)
 0.091455 seconds (2 allocations: 76.294 MiB, 4.83% gc time)
(sol1[:y])[end] = 22026.355662833706
(sol2[:y])[end] = 22026.355662833706
(sol3[:y])[end] = 22026.465794806456
(sol4[:y])[end] = 22026.465794806456
exp(10.0) = 22026.465794806718
```

The first-order method yields five correct digits, while in the fourth-order method all digits but the last three are correct.

The results obtained by the function and the macros are identical. In this particular, but typical run, the macro is six to eight times faster than the function. The allocations are also in favor of the macro implementation; the macro allocates memory only twice, while the function performs 20 million allocations (first-order method) or 80 million allocations (fourth-order method) and thus spends significant time in garbage collection.

Problems 9.8, 9.9, 9.10, and 9.11 are concerned with the implementation of the numerical methods presented in this chapter.

These ideas are applicable and useful also when implementing other numerical methods in a generic way while emphasizing performance. For example, specialized code for graphics processing units (GPU) can be written in this manner.

9.10 JULIA Packages

The package `DifferentialEquations` contains a comprehensive suite for numerically solving differential equations. It can solve ODEs, stochastic ODEs, random ODEs, differential algebraic equations, delay differential equations, and discrete stochastic equations.

The very simple ODE used in the following example is the initial-value problem $y'(t) = y(t)$, $y(0) = 1$, whose solution is the function $y(t) = \mathrm{e}^t$. We define the right-hand side first and then the problem specifying the initial value and the interval. An approximation is calculated by the `solve` function, which also makes it possible to choose from many algorithms. Finally the solution is plotted.

```
julia> using DifferentialEquations
julia> f(u, p, t) = u
f (generic function with 1 method)
julia> problem = ODEProblem(f, 1.0, (0.0, 10.0))
...
julia> sol = solve(problem)
...
julia> using Plots
julia> plot(sol)
```

This pattern of defining problem and solution objects can be followed for all equation types that are supported by the package. Each supported equation has a problem type and a solution type that are understood by the generic functions `solve` and `plot`.

Finally, it is mentioned that the package `OrdinaryDiffEq` is a component package of `DifferentialEquations` and holds the solvers and utilities for ODEs. It is completely independent and usable on its own, which is expedient when a light-weight package is sufficient.

9.11 Bibliographical Remarks

Both the theory of ordinary differential equations and their numerical methods are large fields. A very accessible and comprehensive text book on differential equations is [2]. A detailed treatment of numerical methods for ordinary differential equations can be found in [3].

Problems

9.1 (Modeling) Find an ODE in a subject of your interest and derive it similarly to the example in Sect. 9.1.

9.2 (Lipschitz condition) Suppose that $\partial f/\partial y$ is continuous in the rectangle $D := [-a,a] \times [-b,b]$. Prove that

$$\exists L \in \mathbb{R}^+ : \quad \forall t \in [-a,a] : \quad \forall\forall y_1, y_2 \in [-b,b] : \\ |f(t,y_1) - f(t,y_2)| \le L|y_1 - y_2|$$

holds.

Hint: The Lipschitz constant L is the maximum value of $|\partial f/\partial y|$ in D. Apply the mean-value theorem to f as a function of y only.

9.3 (Interchanging taking the limit and integration) ∗ Suppose that the sequence $\langle f_n \rangle$ of Riemann integrable functions defined on a compact interval I converges uniformly to f. Show that then the limit function f is Riemann integrable and that the equality

$$\lim_{n\to\infty} \int_I f_n = \int_I \underbrace{\lim_{n\to\infty} f_n}_{=f}$$

holds.

9.4 (Picard and Banach) ∗ Show that the operator given by the Picard iteration (9.4) is a contraction and use the Banach fixed-point theorem to show Theorem 9.1.

9.5 (Variation of step size) Implement variation of step size as described in Sect. 9.5 for the Euler and improved Euler methods. Compare the accuracies and the computational expenses of the Euler method and of the method with variable step size in an example whose exact solution is known.

9.6 (Local truncation error of the RK4 method) ∗ Show Theorem 9.7 by following these steps.

1. Calculate the Taylor expansions of k_2, k_3, and k_4 in (9.16) based on k_1 up to $O(h^4)$. The coefficients are treated as unknowns. (The expressions become lengthy and contain partial derivatives of f up to third order.)
2. Substitute the expressions for k_1, k_2, k_3, and k_4 into the RK4 method (9.15).
3. Write the Taylor expansion

$$\begin{aligned}
y(t_n+h) &= y(t_n) + h\frac{\mathrm{d}y(t_n)}{\mathrm{d}t} + \frac{h^2}{2!}\frac{\mathrm{d}^2y(t_n)}{\mathrm{d}t^2} + \cdots \\
&= y(t_n) + hf(t_n, y(t_n)) + \frac{h^2}{2!}\frac{\mathrm{d}f(t_n, y(t_n))}{\mathrm{d}t} + \frac{h^3}{3!}\frac{\mathrm{d}^2f(t_n, y(t_n))}{\mathrm{d}^2t} \\
&\quad + \frac{h^4}{4!}\frac{\mathrm{d}^3f(t_n, y(t_n))}{\mathrm{d}^3t} + O(h^5)
\end{aligned}$$

   of the solution y of the differential equation in terms of partial derivatives of f (up to third order).
4. Compare the two Taylor expansions to find a system of algebraic equations for the unknown coefficients in (9.16).

9.7 (Global truncation error of the RK4 method) $*$ Show Theorem 9.8.

9.8 (Adaptive Runge–Kutta methods) $*$ Extend (a) the function and (b) the macro to implement adaptive Runge–Kutta methods as described in Sect. 9.8.

9.9 (Plot and compare) Choose the solution of an initial-value problem first and then calculate the right-hand side f. Plot and compare the numerical solutions with the exact solution for different Runge–Kutta methods, for different step sizes, and using adaptive Runge–Kutta methods (building on Problem 9.8).

9.10 (Systems of equations) $*$ Write a function to numerically solve systems of first-order ODEs.

9.11 (Systems of equations) $*$ Write a macro to numerically solve systems of first-order ODEs.

References

1. Bogacki, P., Shampine, L.: A 3(2) pair of Runge–Kutta formulas. *Appl. Math. Lett.* **2**(4), 321–325 (1989)
2. Boyce, W., DiPrima, R.: *Elementary Differential Equations and Boundary Value Problems*, 9th edn. John Wiley and Sons, Inc. (2009)
3. Butcher, J.: *Numerical Methods for Ordinary Differential Equations*, 2nd edn. John Wiley & Sons, Ltd., Chichester, England (2008)
4. Dormand, J., Prince, P.: A family of embedded Runge–Kutta formulae. *J. Comp. Appl. Math.* **6**(1), 19–26 (1980)

5. Fehlberg, E.: Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme. *Computing* **6**(1–2), 61–71 (1970)
6. Risch, R.: The problem of integration in finite terms. *Trans. Amer. Math. Soc.* **139**, 167–189 (1969)
7. Risch, R.: The solution of the problem of integration in finite terms. *Bull. Amer. Math. Soc.* **76**(3), 605–608 (1970)
8. Shampine, L., Reichelt, M.: The MATLAB ODE suite. *SIAM Journal on Scientific Computing* **18**(1), 1–22 (1997)

Chapter 10
Partial-Differential Equations

Hydrodynamics procreated complex analysis, partial differential equations, Lie groups and algebra theory, cohomology theory, and scientific computing.

—Vladimir Arnold

Abstract Partial-differential equations are equations that contain partial derivatives of the unknown function. The field of partial-differential equations is large and diverse, as these equations describe many different phenomena and systems, and hence many theories for the existence and uniqueness of their solutions as well as many numerical methods have been developed. In this chapter, we explain fundamental concepts and numerical methods using the example of three important classes of partial-differential equations, namely elliptic, parabolic, and hyperbolic. These types of equations describe diverse physical phenomena such as diffusion, thermal conduction, electromagnetism, and wave propagation. Finite differences, finite volumes, and finite elements are used to calculate approximations of the solutions.

10.1 Introduction

While ordinary differential equations are equations that contain derivatives of the unknown univariate function $y(x)$ (or $y(t)$) with respect to its only independent variable x (or t), the unknown function u in a partial-differential equation (PDE) is multivariate, and a PDE contains derivatives of the unknown with respect to more than one independent variable. The unknown function is commonly denoted by u, and the independent variables are often called t, x, y, and z, or t and $\mathbf{x}$ with $\mathbf{x} := (x, y, z)$ or $\mathbf{x} := (x_1, x_2, x_3)$. Once the context has been established, it is common to leave out the independent variables entirely to simplify the notation.

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_10

In the derivatives, one often writes the independent variable as an index as in

$$u_x = \frac{\partial u}{\partial x}, \qquad u_{xy} = \frac{\partial^2 u}{\partial x \partial y}, \qquad u_{x_i x_j} = \frac{\partial^2 u}{\partial x_i \partial x_j}$$

to simplify notation.

The order of a PDE is the order of the highest derivative that occurs in the equation. PDEs of order higher than second are much rarer than those of first and second order.

What do the elliptic, parabolic, and hyperbolic equations look like? Second-order linear PDEs are classified into three types: elliptic, parabolic, and hyperbolic equations. All second-order linear PDEs in two independent variables x and y can be written in the form

$$Au_{xx} + Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0 \qquad \forall (x,y) \in U,$$

where we have already dropped the dependence of the unknown $u = u(x,y)$, of its partial derivatives, and of the coefficient functions $A = A(x,y)$ to $G = G(x,y)$ on the independent variables x and y to shorten the notation. Note that the terms that contain the unknown u or its derivatives are all linear, as they must be in a linear equation. The domain $U \subset \mathbb{R}^2$ is the domain where the equation holds.

To complete the specification of a PDE that can be solved, it is also necessary to provide boundary conditions, initial conditions, or both. The types and amounts of such conditions depend on the equation type.

A second-order linear PDE is called elliptic if the condition

$$B(x,y)^2 - 4A(x,y)C(x,y) < 0 \qquad \forall (x,y) \in U$$

holds, it is called parabolic if the condition

$$B(x,y)^2 - 4A(x,y)C(x,y) = 0 \qquad \forall (x,y) \in U$$

holds, and it is called hyperbolic – you guessed it – if the condition

$$B(x,y)^2 - 4A(x,y)C(x,y) > 0 \qquad \forall (x,y) \in U$$

holds.

This naming convention is an analogy to conic sections. If we replace u_{xx} by x^2, u_{xy} by xy, and u_{yy} by y^2, we see that the same three conditions hold for ellipses $x^2 + y^2 = a^2$, parabolas $y^2 = 4ax$, and hyperbolas $x^2/a^2 - y^2/b^2 = 1$, respectively. More precisely, after replacing u_{xx} by x^2, u_{xy} by xy, and u_{yy} by y^2 and only considering the first three terms, which are of second order in x and y, we find the equation $Ax^2 + Bxy + Cy^2 = 0$. Dividing it by y (or x) and defining $z := x/y$ (or $z := y/x$) yields the second-order polynomial equation $Az^2 + Bz + C = 0$ (or $Cz^2 + Bz + A = 0$), whose discriminant is the expression $B^2 - 4AC$.

For the purpose of classifying second-order PDEs, only the second-order terms responsible for the expression $B^2 - 4AC$ are important.

More generally, in higher dimensions, when there are d independent variables $x_1, \dots, x_d$ and $D \subset \mathbb{R}^d$, the general second-order linear PDE has the form

$$\sum_{i=1}^{d} \sum_{j=1}^{d} a_{ij} u_{x_i x_j} + \text{lower-order terms} = 0. \tag{10.1}$$

Elliptic, parabolic, and hyperbolic equations are then characterized by the properties of the matrix A whose entries are the coefficients a_{ij}.

Various methods for solving PDEs analytically have been developed. They include separation of variables, the method of characteristics, integral transforms, change of variables, use of fundamental solutions, the superposition principle, and Lie groups. As we are interested in computational methods in this book, we discuss the three main methods for solving PDEs numerically: finite differences, finite volumes, and finite elements.

But before doing so, we take a closer look at elliptic equations in the next section including their theory, before we briefly discuss parabolic and hyperbolic equations in Sections 10.3 and 10.4. We focus on elliptic equations in this chapter, since they are amenable to all three methods, finite differences, finite volumes, and finite differences, which are discussed in the subsequent sections, but the three methods are generally applicable to all kinds of PDEs.

10.2 Elliptic Equations

In this section we take a closer look at elliptic equations. Three physical phenomena, namely electrostatics, diffusion processes, and thermal conduction, and how elliptic equations arise in these three applications are presented. The final part of this section is more advanced and summarizes the theory of weak solutions of elliptic equations.

We start with convenient notation first. When formulating PDEs, the so-called nabla operator

$$\nabla := \begin{pmatrix} \dfrac{\partial}{\partial x_1} \\ \vdots \\ \dfrac{\partial}{\partial x_d} \end{pmatrix}$$

is commonly used. It is used to write the gradient of a scalar multivariate function $f\colon \mathbb{R}^d \to \mathbb{R}$ as

$$\nabla f = \begin{pmatrix} \dfrac{\partial f}{\partial x_1} \\ \vdots \\ \dfrac{\partial f}{\partial x_d} \end{pmatrix},$$

the divergence of a vector-valued multivariate function $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$ as

$$\nabla \cdot \mathbf{f} = \sum_{i=1}^{d} \frac{\partial f_i}{\partial x_i},$$

and the rotation of a vector-valued multivariate function $\mathbf{f} : \mathbb{R}^3 \to \mathbb{R}^3$ as

$$\nabla \times \mathbf{f} = \begin{pmatrix} \frac{\partial f_3}{\partial x_2} - \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_1}{\partial x_3} - \frac{\partial f_3}{\partial x_1} \\ \frac{\partial f_2}{\partial x_1} - \frac{\partial f_1}{\partial x_2} \end{pmatrix}.$$

Using the gradient, the divergence, and a matrix-valued function $A : \mathbb{R}^d \to \mathbb{R}^{d \times d}$, we can write second- and first-order derivatives of a function $u : \mathbb{R}^d \to \mathbb{R}$ that appear in the general form (10.1) in the compact form

$$\nabla \cdot (A(\mathbf{x})\nabla u(\mathbf{x})) = \sum_{i=1}^{d} \sum_{j=1}^{d} \left(a_{ij} u_{x_i x_j} + \frac{\partial a_{ij}}{\partial x_i} u_{x_j} \right) \tag{10.2}$$

if the matrix-valued function A is smooth enough (see Problem 10.1). With these preliminaries, we can write elliptic equations in convenient and compact forms.

10.2.1 Three Physical Phenomena

The first example of an elliptic equation is the derivation of the Poisson equation for electrostatic problems from the Maxwell equations, which are the fundamental equations for electromagnetism. An alternative, but more limited relationship between Coulomb's law and elliptic equations is also discussed.

The second and third example are diffusion and thermal conduction. Although these are also elliptic model equations, additional modeling assumptions are necessary, and hence these equations are not as fundamental as the first example and variants are possible.

10.2.1.1 Electrostatics and Derivation from the Maxwell Equations

We derive the Poisson equation from the Maxwell equations. The Poisson equation is contained in the Maxwell equations and it is retrieved by considering the electrostatic case. An electrostatic system is a system whose magnetic field does not vary with time.

The Maxwell equations are the four PDEs

$$\begin{aligned}
\nabla \cdot \mathbf{D} &= \rho && \text{(Gauss's law)},\\
\nabla \cdot \mathbf{B} &= 0 && \text{(Gauss's law for magnetism)},\\
\nabla \times \mathbf{E} &= -\frac{\partial}{\partial t}\mathbf{B} && \text{(Faraday's law of induction)},\\
\nabla \times \mathbf{H} &= \mathbf{J} + \frac{\partial}{\partial t}\mathbf{D} && \text{(Ampère's circuital law)}
\end{aligned}$$

in three spatial dimensions and time, where $\mathbf{E}(t,\mathbf{x})$ is the electric field, $\mathbf{B}(t,\mathbf{x})$ is the magnetic field, $\mathbf{D}(t,\mathbf{x})$ is the electric displacement field, $\mathbf{H}(t,\mathbf{x})$ is the magnetizing field, $\rho(\mathbf{x})$ the charge density, and $\mathbf{J}(t,\mathbf{x})$ is the (applied) current density. Here bold letters (irrespective of being upper- or lowercase letters) denote vector valued functions. The electric displacement field $\mathbf{D}$ and the magnetizing field $\mathbf{H}$ satisfy the constitutive relations

$$\begin{aligned}
\mathbf{D}(t,\mathbf{x}) &= \epsilon(\mathbf{x})\mathbf{E}(t,\mathbf{x}),\\
\mathbf{H}(t,\mathbf{x}) &= \mu(\mathbf{x})^{-1}\mathbf{B}(t,\mathbf{x}),
\end{aligned}$$

where $\epsilon(\mathbf{x})$ is the permittivity and $\mu(\mathbf{x})$ is the permeability, which are both matrix-valued functions from $\mathbb{R}^3$ to $\mathbb{R}^{3\times 3}$.

The fields $\mathbf{E}$ and $\mathbf{H}$ satisfy the physical interface or jump conditions

$$\begin{aligned}
[\mathbf{E}\times\mathbf{n}] &= \mathbf{0}, & [\epsilon\mathbf{E}\cdot\mathbf{n}] &= \rho|_\Gamma,\\
[\mathbf{H}\times\mathbf{n}] &= \mathbf{0}, & [\mu\mathbf{H}\cdot\mathbf{n}] &= 0
\end{aligned}$$

across an interface Γ. The interface Γ is the interface between two domains Ω_1 and Ω_2 consisting of different materials, and the vector $\mathbf{n}$ is the outward unit normal vector of $\partial\Omega_1$. The jump $[f]$ of any function f across the interface Γ is defined as

$$[f] := f|_{\Gamma\cap\bar{\Omega}_2} - f|_{\Gamma\cap\bar{\Omega}_1},$$

i.e., it is the difference between two one-sided limits of f, the first within Ω_2 and the second within Ω_1. Furthermore, $\rho|_\Gamma \colon \Gamma \to \mathbb{R}$ is the restriction of the charge density ρ to the interface Γ.

If the magnetic field $\mathbf{B}$ is constant with respect to time, the rotation $\nabla\times\mathbf{E}$ vanishes by Faraday's law of induction and therefore the electric field $\mathbf{E}$ is conservative and can be expressed as the gradient of minus a scalar potential $\phi \colon \mathbb{R} \to \mathbb{R}$, i.e.,

$$\begin{aligned}
\mathbf{E} &= -\nabla\phi,\\
\mathbf{D} &= -\epsilon\nabla\phi.
\end{aligned}$$

The minus sign only serves cosmetic purposes here. Substitution of the last equation into the first the Maxwell equations, namely Gauss's law, now yields the Poisson equation

$$-\nabla\cdot(\epsilon\nabla\phi) = \rho. \tag{10.3}$$

Here ϵ is a matrix-valued function and ρ is a scalar function, as already noted above, and it becomes clear why the gradient and the divergence are so convenient to express these equations.

10.2.1.2 Electrostatics and Derivation from Coulomb's Law

We know that Coulomb's law holds for electric fields in homogeneous materials with no magnetic field present, and therefore the question naturally arises how it relates to the Poisson equation under these two assumptions. It should be possible to arrive at the Poisson equation from Coulomb's law, and we now discuss how this is indeed possible. The derivation from Coulomb's law governs only the electrostatic case; no magnetic field ever enters the picture. A homogeneous material means that the permittivity $\epsilon_0 \in \mathbb{R}$ is simply a real constant.

According to Coulomb's law, the force $\mathbf{F}_{ij} \in \mathbb{R}^3$ that a particle at position $\mathbf{r}_j \in \mathbb{R}$ with charge $q_j \in \mathbb{R}$ exerts on a particle at position r_i with charge q_i is given by

$$\mathbf{F}_{ij} = \frac{q_i q_j}{4\pi\epsilon_0} \frac{\mathbf{r}_i - \mathbf{r}_j}{|\mathbf{r}_i - \mathbf{r}_j|^3}.$$

It implies that the force is proportional to $1/|\mathbf{r}_i - \mathbf{r}_j|^2$, one over the distance between the particles squared. Coulomb's law is an atomistic model, since the charges are point like.

In a continuum model, on the other hand, all the point charges q_j give rise to a charge density $\rho : \mathbb{R}^3 \to \mathbb{R}$. The force $\mathbf{F}$ that acts on a particle with charge q at position $\mathbf{x}$ can be written as

$$\mathbf{F} = q\mathbf{E}$$

using the electric field $\mathbf{E}$, which is obtained from Coulomb's law by integration over all other charges q_j at positions $\mathbf{y}$ as

$$\mathbf{E}(\mathbf{x}) := \frac{1}{4\pi\epsilon_0} \iiint \frac{\rho(\mathbf{y})(\mathbf{x} - \mathbf{y})}{|\mathbf{x} - \mathbf{y}|^3} d\mathbf{y}.$$

Next we check by a simple calculation that the electric field is irrotational, i.e., that

$$\nabla \times \mathbf{E} = \mathbf{0} \tag{10.4}$$

holds (see Problem 10.2). Hence the electric field $\mathbf{E}$ is a gradient field again and thus can be written as

$$\mathbf{E} = -\nabla\phi \tag{10.5}$$

as the gradient of minus a potential, where ϕ is called the electrostatic potential and the minus sign serves a cosmetic purpose the next calculation reveals. It is straightforward to check by differentiating that

$$\phi(\mathbf{x}) := \frac{1}{4\pi\epsilon_0} \iiint \frac{\rho(\mathbf{y})}{|\mathbf{x}-\mathbf{y}|} \mathrm{d}\mathbf{y} \tag{10.6}$$

yields the field **E** above (see Problem 10.3).

Since the function

$$G(\mathbf{x}) := -\frac{1}{4\pi}\frac{1}{|\mathbf{x}|} \tag{10.7}$$

is a fundamental solution or a Green function of the Laplace operator

$$\Delta := \nabla \cdot \nabla$$

on the whole space $\mathbb{R}^3$, i.e.,

$$\Delta G(\mathbf{x}-\mathbf{y}) = \delta(\mathbf{x}-\mathbf{y}) \qquad \forall \mathbf{x} \in \mathbb{R}^3 \quad \forall \mathbf{y} \in \mathbb{R}^3 \tag{10.8}$$

(see Problem 10.4), integration of the last equation against the charge density ρ yields

$$\iiint_{\mathbb{R}^3} \Delta G(\mathbf{x}-\mathbf{y})\rho(\mathbf{y})\mathrm{d}\mathbf{y} = \iiint_{\mathbb{R}^3} \delta(\mathbf{x}-\mathbf{y})\rho(\mathbf{y})\mathrm{d}\mathbf{y} = \rho(\mathbf{x}) \qquad \forall \mathbf{x} \in \mathbb{R}^3$$

and further

$$\Delta\Big(\iiint_{\mathbb{R}^3} G(\mathbf{x}-\mathbf{y})\rho(\mathbf{y})\mathrm{d}\mathbf{y}\Big) = \Delta(-\epsilon_0\phi(\mathbf{x})) = \rho(\mathbf{x}) \qquad \forall \mathbf{x} \in \mathbb{R}^3.$$

In other words, the potential ϕ given by (10.6) solves the Poisson equation

$$-\epsilon_0 \Delta\phi = \rho.$$

This equation is a special case of (10.3), as here the permittivity is a real constant ϵ_0 and could be pulled out of the divergence.

10.2.1.3 Diffusion

We can describe any stationary diffusion process by a PDE using two considerations. The first is fundamental, while the second one requires a physical model. Transient diffusion processes lead to parabolic equations (see Sect. 10.3).

The first step is to note that the total flux of particles out of any subdomain $\Omega \subset D$ of the domain D equals the amount of particles produced by all sources in Ω due to mass conservation. This yields

$$\oiint_{\partial\Omega} \mathbf{n} \cdot \mathbf{J}\mathrm{d}S = \iiint_{\Omega} f\mathrm{d}V.$$

In the surface integral on the left-hand side, the flux density of the particles is denoted by **J** and the **n** are outward unit normal vectors. The volume integral on the right-hand side is over the function f that describes the sources.

Using the divergence theorem, the boundary integral on the left-hand side becomes a volume integral, and hence the equation becomes

$$\iiint_\Omega \nabla \cdot \mathbf{J} \mathrm{d}V = \iiint_\Omega f \mathrm{d}V.$$

It holds true for all subdomains Ω. After assuming that the integrand is compactly supported and smooth, we can therefore apply the fundamental lemma of variational calculus to the equation

$$\iiint_\Omega (\nabla \cdot \mathbf{J} - f)\mathrm{d}V = 0 \qquad \forall \Omega \subset D$$

to find

$$\nabla \cdot \mathbf{J}(\mathbf{x}) = f(\mathbf{x}) \qquad \forall \mathbf{x} \in D.$$

There are various versions of the fundamental lemma of variational calculus; two are recorded in the following.

Theorem 10.1 (fundamental lemma of variational calculus) *1. Version for continuous functions: Suppose that $\Omega \subset \mathbb{R}^d$ is an open set and that a continuous multivariate function $f : \Omega \to \mathbb{R}$ satisfies the equation*

$$\iiint_\Omega f(\mathbf{x})h(\mathbf{x})\mathrm{d}\mathbf{x} = 0$$

for all compactly supported smooth functions h on Ω; then f is identically equal to zero.

2. Version for discontinuous functions: Suppose that $\Omega \subset \mathbb{R}^d$ is an open set and that a multivariate function $f \in L^2(\Omega)$ satisfies the equation

$$\iiint_\Omega f(\mathbf{x})h(\mathbf{x})\mathrm{d}\mathbf{x} = 0$$

for all compactly supported smooth functions h on Ω; then $f = 0$ in L^2, i.e., f is equal to zero almost everywhere.

In the second step, we must specify how the flux density **J** relates to the unknown concentration u. The most common and basic physical model is Fick's first law

$$\mathbf{J} := -D\nabla u,$$

where the diffusion coefficient $D : \mathbb{R}^d \to \mathbb{R}^{d\times d}$ is generally a matrix-valued function. This physical model results in the linear elliptic equation

$$-\nabla \cdot (D\nabla u) = f.$$

Many other physical models for the relationship between the flux density $\mathbf{J}$ and the unknown u are known to be useful. For example, diffusion in porous media is governed by

$$\mathbf{J} := -D\nabla u^m,$$

where $m \in \mathbb{R}$ is a constant $m > 0$ and usually $m > 1$.

10.2.1.4 Thermal Conduction

The physical model for thermal conduction is the law of heat conduction or Fourier's law, which states that the rate of heat transfer through a material is proportional to the negative gradient of the temperature and to the area orthogonal to the gradient through which the heat flows.

The derivation proceeds analogously to the modeling of diffusion processes in Sect. 10.2.1.3. Now the vector $\mathbf{J}$ is the heat flux density and it is, by the law of heat conduction, equal to

$$\mathbf{J} := -k\nabla u,$$

where the thermal conductivity $k\colon \mathbb{R}^d \to \mathbb{R}^{d\times d}$ is generally a matrix-valued function and u denotes the unknown temperature. This results in the heat equation

$$-\nabla \cdot (k\nabla u) = f.$$

However, the thermal conductivity k of a material generally varies with temperature, which gives rise to nonlinear equations in which k is a function of the unknown temperature.

It is often instructive to check that the physical units of the variables and constants in an equation and its derivation are consistent (see Problem 10.5). To check the consistency of the units in the heat equation, we note that the unknown temperature u has unit $[u] = \mathrm{K}$, the thermal conductivity k has unit $[k] = \mathrm{W}\cdot\mathrm{m}^{-1}\cdot\mathrm{K}^{-1}$, and the source term f has unit $[f] = \mathrm{W}\cdot\mathrm{m}^{-3}$. In the equation, we thus have $[\mathbf{J}] = [k\nabla u] = [k][\nabla u] = \mathrm{W}\cdot\mathrm{m}^{-2}$ and $[\nabla\cdot(k\nabla u)] = \mathrm{W}\cdot\mathrm{m}^{-3}$. The unit of $[\nabla \cdot (k\nabla u)]$ on the left-hand side of the equation is consistent with the unit of the source term f on the right-hand side.

10.2.2 Boundary Conditions

In general, equations may have any number of solutions, ranging from no solution at all, a unique solution, and a finite number of solutions to infinitely many solutions. This is true, e.g., for linear systems of equations, for polynomial equations, and for Diophantine equations, and it is also true for differential equations.

It is usually desirable that a PDE has a unique solution, because we expect the equation to be a full and unique description of the problem or system under consideration. Therefore the question naturally arises whether a given PDE has a unique solution. If we can answer this question positively, our confidence that the PDE is a useful model is much increased. This knowledge is also useful when we aim to calculate a numerical approximation of a solution; it stands to reason that the existence of a unique solution is beneficial for any numerical algorithm.

The existence and possibly uniqueness of a solution is not a property of just the equation that holds for all points in the interior of the domain, but a full problem description must be supplemented with initial and/or boundary conditions and a specification of the set of functions from which the solution is sought. Again, this should not come as a surprise; we know from algebra that, e.g., the polynomial equation $x^2 = 2$ has no solution in the rational numbers $\mathbb{Q}$ but a unique solution in the real numbers $\mathbb{R}$, and that the polynomial equation $x^2 = -1$ has no solution in the real numbers $\mathbb{R}$ but a unique solution in the complex numbers $\mathbb{C}$.

Analogously, there are different types of solutions of PDEs. A classical solution is a solution that can be substituted into the equation and that then satisfies the equation pointwise. But there are other, weaker, types of function like objects that can be interpreted as solutions of differential equations. A glimpse of the theory of elliptic equations is given in the next section, Sect. 10.2.3, and many textbooks have been written on these questions [1].

The two major types of conditions are initial conditions and boundary conditions. In transient problems, initial conditions are usually specified, and boundary conditions are usually specified in both stationary and transient problems. Initial conditions give the solution at the beginning of the time interval, and boundary conditions hold on the boundary of the spatial domain.

For elliptic equations, there are four major types of boundary conditions:

- Dirichlet boundary conditions specify the unknown u on all points of the boundary ∂U of a domain U or only on the Dirichlet part $\partial U_D \subset \partial U$ of the boundary ∂U as in the example

  $$u(\mathbf{x}) = u_D(\mathbf{x}) \qquad \forall \mathbf{x} \in \partial U_D,$$

  where u_D is a given function.
  For example, in electrostatics, this means that a contact is held at a fixed voltage; in mechanics, this means that a beam is held at a fixed position; in thermodynamics, this means that the surface of a body is held at a fixed temperature; and in fluid dynamics, such a no-slip condition means that a viscous fluid has zero velocity relative to a solid boundary.
- Neumann boundary conditions specify the directional derivative of the unknown u with respect to the outward unit normal vectors $\mathbf{n}$ of the boundary ∂U on all points of the boundary ∂U of a domain U or only on the Neumann part $\partial U_N \subset \partial U$ of the boundary ∂U as in the example

  $$\frac{\partial u}{\partial \mathbf{n}} = \mathbf{n} \cdot \nabla u(\mathbf{x}) = u_N(\mathbf{x}) \qquad \forall \mathbf{x} \in \partial U_N,$$

  where u_N is a given function.

In electrostatics, this means that a constant (often vanishing) field is applied; and in thermodynamics, this means that the heat flux from a surface is known.

- Mixed boundary conditions are different boundary conditions prescribed on disjoint parts of the boundary. The most common example is the prescription of Dirichlet boundary conditions on the Dirichlet part ∂U_D and of Neumann boundary conditions on the Neumann part ∂U_N of the boundary $\partial U = \partial U_D \cup \partial U_N$.
- Robin boundary conditions

$$au + b\frac{\partial u}{\partial \mathbf{n}} = u_R(\mathbf{x}) \qquad \forall \mathbf{x} \in \partial U,$$

  where u_R is a given function, are linear combinations of Dirichlet and Neumann boundary conditions.
  They often appear in Sturm–Liouville problems. In convection-diffusion equations, they can act as insulting boundary conditions that ensure that the sum of the convective and diffusive fluxes vanishes. In electromagnetism, they are called impedance boundary conditions.

Boundary conditions whose given function on the right-hand side vanishes are called homogeneous boundary conditions; otherwise they are called inhomogeneous.

Prescribing Dirichlet boundary conditions or mixed Dirichlet/Neumann boundary conditions to an elliptic PDE yields a unique solution.

However, prescribing Neumann boundary conditions to an elliptic PDE results in no solutions or infinitely many solutions. This can be discussed vividly using the electrostatic problem described by the elliptic boundary-value problem

$$-\nabla \cdot (A\nabla u) = f \qquad \text{in } U, \tag{10.9a}$$

$$\mathbf{n} \cdot (A\nabla u) = u_N \qquad \text{on } \partial U. \tag{10.9b}$$

Here the right-hand side f are the charges and the Neumann boundary condition u_N corresponds to a known electric field on the whole boundary ∂U. Integrating the equation, using the divergence theorem, and using the Neumann boundary condition, we find

$$-\oiint_{\partial U} u_N \mathrm{d}S = -\oiint_{\partial U} \mathbf{n} \cdot (A\nabla u)\mathrm{d}S = -\iiint_U \nabla \cdot (A\nabla u)\mathrm{d}S = \iiint_U f\mathrm{d}V.$$

In other words, the Neumann boundary conditions must match the right-hand side via the equation

$$-\oiint_{\partial U} u_N \mathrm{d}S = \iiint_U f\mathrm{d}V,$$

which hence is a necessary condition for the existence of a solution.

Furthermore, it is obvious that if u is a solution, then $\mathbf{x} \mapsto u(\mathbf{x}) + C$ is also a solution for all $C \in \mathbb{R}$, as only the gradient ∇u of the solution u appears in (10.9).

Therefore this physical examples illustrates the fact how different boundary conditions render three cases possible, namely no solution (no matching Neumann boundary conditions), a unique solution (Dirichlet or mixed Dirichlet/Neumann boundary conditions), and infinitely many solutions (matching Neumann boundary conditions).

In summary, the existence and uniqueness of solutions of PDEs deserve investigations for all combinations of equations, boundary conditions, and function spaces in which the solutions are sought. In the following, a short summary of the modern theory of elliptic PDEs is given.

10.2.3 Existence, Uniqueness, and a Pointwise Estimate *

We consider the elliptic boundary-value problem

$$\begin{aligned} -\nabla \cdot (A\nabla u) + \mathbf{b} \cdot \nabla u + cu &= f && \text{in } U, && (10.10a)\\ u &= u_D && \text{on } \partial U, && (10.10b)\\ \mathbf{n} \cdot (A\nabla u) &= 0 && \text{on } \partial U_N, && (10.10c) \end{aligned}$$

where $U \subset \mathbb{R}^d$ is a domain and $d \in \mathbb{N}$. The boundary ∂U is the complementary union of its Dirichlet part ∂U_D and its Neumann part ∂U_N, and the $\mathbf{n}$ in the Neumann boundary condition are outward pointing unit normal vector on ∂U_N. The given functions are $A : U \to \mathbb{R}^{d\times d}$, $\mathbf{b} : U \to \mathbb{R}^d$, $c : U \to \mathbb{R}$, $f : U \to \mathbb{R}$, and $u_D : U \to \mathbb{R}$.

The so-called weak formulation is generally obtained by multiplying the strong, pointwise, or classical formulation by so-called test functions, integrating over the whole domain, using partial integration to distribute the derivatives, and requiring that the resulting equation, i.e., the weak formulation, is satisfied for all test functions. Then the fundamental lemma of variational calculus, Theorem 10.1, ensures that the integrands in the weak formulation are equal, although it may not be possible to substitute the solution of the weak formulation into the original formulation because it may not be smooth enough. Hence the names weak solution and weak formulation.

Here, the appropriate function space for the elliptic problem (10.10) is the Hilbert space $H^1(U)$, as the calculations below will show. In order to take care of the inhomogeneous Dirichlet boundary conditions, we extend u_D from the boundary ∂U_D to the whole domain U by using the following theorem with $s = 1$.

Theorem 10.2 (inverse trace theorem) *Suppose that s is a positive integer, that the domain U is of class C^s, and that ∂U is bounded. Then there is a bounded trace operator $T: H^s(U) \to H^{s-1/2}(\partial U)$. Moreover, T has a bounded right inverse $E: H^{s-1/2}(\partial U) \to H^s(U)$, i.e., there exists a positive constant C such that*

$$\|Eu\|_{H^s(U)} \le C\|u\|_{H^{s-1/2}(\partial U)} \qquad \forall u \in H^{s-1/2}(\partial U).$$

For a proof, see [7, Section 7.2.5].

We denote the extension of u_D from the boundary ∂U_D to U by $\bar{u}_D$ and define

$$w := u - \bar{u}_D.$$

Then the problem (10.10) has homogeneous Dirichlet boundary conditions $w = 0$ on ∂U_D, and solutions w are sought in the function space $H^1_0(U)$.

To find the weak formulation, we first multiply (10.10a) by test functions $v \in H^1_0(U)$. The test functions are chosen from the same function space as the solution. Then partial integration of the first term yields

$$-\iiint_U \nabla \cdot (A\nabla u) v \mathrm{d}V = -\oiint_{\partial U} (A\nabla u) \cdot (v\mathbf{n}) \mathrm{d}S + \iiint_U (A\nabla u) \cdot (\nabla v) \mathrm{d}V.$$

The integral $\oint_{\partial U}(A\nabla u) \cdot (v\mathbf{n})\mathrm{d}V$ vanishes, because the test function $v \in H^1_0(U)$ vanishes on the boundary ∂U_D and because $\mathbf{n} \cdot (A\nabla u) = 0$ holds on ∂U_N. Therefore the weak formulation is to find a function u with $u - \bar{u}_D \in H^1_0(U)$ such that

$$\iiint_U (A\nabla u) \cdot \nabla v + (v\mathbf{b}) \cdot \nabla u + cuv \mathrm{d}V = \iiint_U fv \mathrm{d}V \qquad \forall v \in H^1_0(U)$$

holds, or equivalently to find a function $w \in H^1_0(U)$ such that

$$\iiint_U (A\nabla w) \cdot \nabla v + (v\mathbf{b}) \cdot \nabla w + cwv \mathrm{d}V$$
$$= \iiint_U fv - A\nabla \bar{u}_D \cdot \nabla v - (v\mathbf{b}) \cdot \nabla \bar{u}_D - c\bar{u}_D v \mathrm{d}V \qquad \forall v \in H^1_0(U)$$

holds. After defining the bilinear form

$$a(u, v) := \iiint_U (A\nabla u) \cdot \nabla v + (v\mathbf{b}) \cdot \nabla u + cuv \mathrm{d}V \tag{10.11}$$

(the left-hand side of the last equation) and the functional

$$F(v) := \iiint_U fv - A\nabla \bar{u}_D \cdot \nabla v - (v\mathbf{b}) \cdot \nabla \bar{u}_D - c\bar{u}_D v \mathrm{d}V, \tag{10.12}$$

(the right-hand side), the weak formulation of (10.10) is to find a function $u \in H_0^1(U)$ such that

$$a(u, v) = F(v) \qquad \forall v \in H_0^1(U)$$

holds.

Does such a weak solution u exist and is it unique? The existence and uniqueness of the weak solution u can be shown using the Lax–Milgram theorem. Before we can state it, we need a few definitions.

Definition 10.3 (bounded) Suppose H is a Hilbert space. A bilinear form $a: H \times H \to \mathbb{R}$ is called *bounded,* if there exists a positive constant α such that

$$|a(u, v)| \le \alpha \|u\|_H \|v\|_H \qquad \forall\forall u, v \in H$$

holds.

Definition 10.4 (coercive) Suppose H is a Hilbert space. A bilinear form $a: H \times H \to \mathbb{R}$ is called *coercive,* if there exists a positive constant β such that

$$\beta \|u\|_H^2 \le a(u, u) \qquad \forall u \in H$$

holds.

The main assumptions in the theorem are that the bilinear form a is coercive and bounded.

Theorem 10.5 (Lax–Milgram theorem [3]) *Suppose that H is a Hilbert space, that $F \in H'$, and that a is a bilinear form on H that is bounded (with constant α) and coercive (with constant β). Then there exists a unique solution $u \in H$ of the equation*

$$a(u, v) = F(v) \qquad \forall v \in H.$$

Furthermore, the estimate

$$\|u\|_H \le \frac{1}{\beta} \|F\|_{H'} \tag{10.13}$$

holds.

The proof, which mostly follows [1, Section 6.2.1], uses basic concepts such as the Riesz representation theorem from functional analysis, whose full explanation can be found in any text book on functional analysis. However, apart from an introduction to some basic concepts from functional analysis, the proof is complete.

Theorem 10.6 (Riesz representation theorem) *Suppose H is a Hilbert space. Then the dual space H' of H can be canonically identified with H. More precisely, for each $u' \in H'$ there exists a unique element $u \in H$ such that*

$$u'(v) = \langle u, v \rangle \qquad \forall v \in H$$

and the mapping $u' \mapsto u$ is a linear isomorphism of H' into H.

Before giving the proof, we note that the special case of a symmetric bilinear form a leads the way to the general case. In the symmetric case, $\langle u, v\rangle := a(u, v)$ is an inner product on the Hilbert space H, and the Riesz representation theorem, Theorem 10.6, can be directly applied, showing the existence of a unique solution $u \in H$. A proof for the general case is the following.

Proof First, we consider any $u \in H$ and note that $v \mapsto a(u, v)$ is a bounded linear functional on H by assumption. Then the Riesz representation theorem, Theorem 10.6, implies that a unique element $w \in H$ that satisfies

$$a(u, v) = \langle w, v\rangle \qquad \forall v \in H$$

exists. Since $u \in H$ was arbitrary, we can thus define an operator $A: H \to H$ satisfying

$$a(u, v) = \langle Au, v\rangle \qquad \forall\forall u, v \in H.$$

We claim that the operator A is linear and bounded. To show that it is linear, the bilinearity of a and the linearity of the inner product yield

$$\begin{aligned}\langle A(\lambda_1 u_1 + \lambda_2 u_2), v\rangle &= a(\lambda_1 u_1 + \lambda_2 u_2, v)\\ &= \lambda_1 a(u_1, v) + \lambda_2 a(u_2, v)\\ &= \lambda_1\langle Au_1, v\rangle + \lambda_2\langle Au_2, v\rangle\\ &= \langle \lambda_1 Au_1 + \lambda_2 Au_2, v\rangle\end{aligned}$$

for all λ_1 and $\lambda_2 \in \mathbb{R}$, for all u_1 and $u_2 \in \mathbb{R}$, and for all $v \in H$. Since the equality holds for all $v \in H$, the operator A is linear. To show that A is bounded, we use the assumption that the bilinear form a is bounded to calculate

$$||Au||_H^2 = \langle Au, Au\rangle = a(u, Au) \le \alpha ||u||_H ||Au||_H \qquad \forall u \in H,$$

which yields $||Au|| \le \alpha ||u||$ for all $u \in H$, i.e., A is bounded.

Having shown that the linear operator A is linear and bounded, we next show that it is injective and that it has closed range. Since the bilinear form a is coercive, we find

$$\beta ||u||_H^2 \le a(u, u) = \langle Au, u\rangle \le ||Au||_H ||u||_H \qquad \forall u \in H,$$

which implies

$$\beta ||u||_H \le ||Au||_H \qquad \forall u \in H, \tag{10.14}$$

i.e., the operator A is bounded below. It is straightforward to see that it is therefore injective. Furthermore, its range is closed: if $Au_n \to y$, then the inequality

$$\beta ||u_n - u_m||_H \le ||A(u_n - u_m)||_H = ||Au_n - Au_m||_H$$

shows that $\{u_n\}$ is a Cauchy sequence; if $u_n \to u$, then $Au_n \to Au$ by the continuity of A and hence $Au = y$, which means that A has closed range.

Next, we show that the linear operator A is surjective. Suppose it is not. Since its range $R(A)$ is closed, there would exist a nonzero element $y \in H$ with $0 \neq y \in R(A)^\perp$. This would imply $\beta\|y\|_H^2 \le a(y,y) = \langle Ay, y\rangle = 0$ and therefore $y = 0$, a contradiction. Therefore the operator A is surjective.

Since the operator $A\colon H \to H$ is both injective and surjective, it is bijective.

Next, the Riesz representation theorem, Theorem 10.6, implies that a unique $z \in H$ exists such that

$$F(v) = \langle z, v\rangle \qquad \forall v \in H$$

and that the norms $\|F\|_{H'}$ and $\|z\|_H$ agree. Since A is a bijection, there exists a unique element $u \in H$ such that $Au = z$.

In summary, we have shown that there exists a unique element $u \in H$ such that

$$a(u,v) = \langle Au, v\rangle = \langle z, v\rangle = F(v) \qquad \forall v \in H.$$

Inequality (10.13) follows from inequality (10.14) and recalling that $Au = z$ and $\|F\|_{H'} = \|z\|_H$. □

We can now state and prove the existence and uniqueness of the solution of the elliptic boundary-value problem by applying the Lax–Milgram theorem. To do so, it is customary to call coefficient matrices A that give rise to coercive bilinear forms uniformly elliptic.

Definition 10.7 (uniformly elliptic) A matrix-valued function $A\colon U \to \mathbb{R}^{d\times d}$ is called *uniformly elliptic,* if there exists a positive constant β such that

$$\beta|\mathbf{z}|^2 \le \mathbf{z}^\top A(\mathbf{x})\mathbf{z} \qquad \forall \mathbf{z} \in \mathbb{R}^d\backslash\{0\} \quad \forall \mathbf{x} \in U$$

holds.

Theorem 10.8 (existence and uniqueness of weak solutions of elliptic equations) *Suppose that the domain $U \subset \mathbb{R}^d$, $d \in \mathbb{N}$, is of class C^1 and that its boundary ∂U is bounded. Suppose that the matrix-valued function $A \in L^\infty(U, \mathbb{R}^{d\times d})$ is uniformly elliptic with constant K, that $\mathbf{b} \in L^\infty(U, \mathbb{R}^d)$, $c \in L^\infty(U, \mathbb{R})$, $f \in H^{-1}(U)$, and $u_D \in H^{1/2}(\partial U)$. Suppose further that $\mathbf{b} = 0$ and $c \ge 0$ for all $x \in U$ or that $\|\mathbf{b}\|_{L^\infty}^2 < 4K \inf_U c$ holds.*

Then the boundary-value problem (10.10) has a unique solution $u \in H^1(U)$. Furthermore, there is a positive constant C such that the estimate

$$\|u\|_{H^1(U)} \le C\|F\|_{H^{-1}(U)}$$

holds.

Proof We will show that the bilinear form a in (10.11) is coercive and bounded and that the functional F in (10.12) is in $H^{-1}(U)$ so that Theorem 10.5 can be applied.

Using the Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1, it is straightforward to see that the bilinear form a is bounded.

To see that the bilinear form a is coercive, we first find a bound from above for the second term $\iiint_U (u\mathbf{b}) \cdot \nabla u \mathrm{d}V$ in $a(u,u)$. The Cauchy–Bunyakovsky–Schwarz inequality yields

$$\iiint_U (v\mathbf{b}) \cdot \nabla u \mathrm{d}V \leq \|\mathbf{b}\|_{L^\infty(U)} \|u\|_{L^2(U)} \|\nabla u\|_{L^2(U)}.$$

For the factor $\|u\|_{L^2(U)} \|\nabla u\|_{L^2(U)}$, we use the inequality

$$xy \leq \alpha x^2 + \beta y^2 \qquad \forall\forall x, y \in \mathbb{R},$$

where α and $\beta \in \mathbb{R}^+$ with $\alpha\beta = 1/4$ (see Problem 10.8). These considerations yield the bound from below

$$\begin{aligned} a(u,u) &\geq K\|\nabla u\|^2_{L^2(U)} - \|\mathbf{b}\|_{L^\infty(U)}\big(\alpha\|u\|^2_{L^2(U)} + \beta\|\nabla u\|^2_{L^2(U)}\big) + (\inf_U c)\|u\|^2_{L^2(U)} \\ &= (K - \beta\|\mathbf{b}\|_{L^\infty(U)})\|\nabla u\|^2_{L^2(U)} + (\inf_U c - \alpha\|\mathbf{b}\|_{L^\infty(U)})\|u\|^2_{L^2(U)} \end{aligned}$$

for the bilinear form a, where $\alpha\beta = 1/4$.

In the first case, i.e., if $\mathbf{b} = 0$ and $c \geq 0$ everywhere, the Poincaré inequality

$$\exists C \in \mathbb{R}^+ : \quad \forall u \in H^1_0(U) : \quad \|u\|_{L^2(U)} \leq C\|\nabla u\|_{L^2(U)}$$

(see, e.g., [7, Theorem 7.32]) establishes that a is coercive.

In the second case, i.e., if $\|\mathbf{b}\|^2_{L^\infty} < 4K \inf_U c$, we set

$$\alpha := \frac{\inf_U c}{\|\mathbf{b}\|_{L^\infty(U)}} > 0$$

so that the coefficient of $\|u\|^2_{L^2(U)}$ vanishes. Then $\beta = \|\mathbf{b}\|_{L^\infty(U)}/(4\inf_U c)$ and the coefficient of $\|\nabla u\|^2_{L^2(U)}$ is positive due to the assumption. Therefore the bilinear form a is again coercive.

Finally, F belongs to $H^{-1}(U) = H^1_0(U)'$ provided it is a bounded linear functional on $H^1_0(U)$. F is clearly linear and it is bounded due to the assumptions on the data. $\square$

The following theorem is a pointwise estimate for solutions of the linear Poisson equation. It is quite well-known and can be found, e.g., as [2, Theorem 3.7]. Many regularity results for elliptic equations are available to ensure the smoothness of the solution based on assumptions on the coefficients and the domain. If sufficient smoothness is assumed, the operator $\nabla \cdot (A\nabla)$ in divergence form can always be rewritten in the non-divergence form that occurs in the theorem.

Theorem 10.9 (pointwise estimate for elliptic equations) *Suppose $U \subset \mathbb{R}^d$ is a bounded domain. Define the linear operator*

$$Lu := \sum_{i,j=1}^{d} a_{ij}(x)\partial_{x_i x_j} u(x) + b(x) \cdot \nabla u(x) + c(x)u(x) = f(x),$$

where $u \in C^0(\overline{U}) \cap C^2(U)$, the coefficient matrix A is symmetric, and the inequality $c(x) \le 0$ holds for all $x \in U$. Furthermore, suppose that the operator L is elliptic, i.e., $0 < \lambda(x)|\xi|^2 \le \xi^\top A(x)\xi \le \Lambda(x)|\xi|^2$ holds for all $\xi \in \mathbb{R}^d \backslash \{0\}$ and for all $x \in U$, where $\lambda(x)$ and $\Lambda(x)$ are the minimum and maximum eigenvalues, respectively.

Then the estimate

$$\sup_U |u| \le \sup_{\partial U} |u| + C \sup_U \frac{|f|}{\lambda}$$

holds, where C is a constant depending only on $\operatorname{diam}(U)$ *and $\beta := \sup |b|/\lambda < \infty$. In particular, if U lies between two parallel planes a distance d apart, then the estimate holds with $C = \mathrm{e}^{(\beta+1)d} - 1$.*

10.3 Parabolic Equations

Parabolic equations are frequently the transient versions of stationary, elliptic equations. In the case of two independent variables, they are usually called t and x as in the example of the transient one-dimensional heat equation

$$u_t = Du_{xx} + q,$$

where the positive coefficient function D is the thermal conductivity. The same equation can be interpreted as a transient one-dimensional diffusion equation; then the positive coefficient D is the diffusion constant. The function q on the right-hand side is a source of heat (in the case of the heat equation) or mass (in the case of the diffusion equation).

In higher dimensions, heat or diffusion equations have the form

$$u_t = D\Delta u + q$$

or, more generally,

$$u_t = \nabla \cdot (D\nabla u) + q$$

with space dependent coefficient functions D. Here the Laplace operator Δ and the nabla operators ∇ are applied to the unknown $u(t, \mathbf{x})$ with respect to the spatial variables $\mathbf{x} = (x_1, \dots, x_d)$.

In the case of transient parabolic equations, one ensures that a unique solution exists by specifying initial conditions $u(t = 0, \mathbf{x}) = f(\mathbf{x})$ as well as boundary conditions $u(t, \mathbf{x}) = g(t)$ for all points $\mathbf{x}$ on the boundary.

It is obvious that stationary solutions, i.e., when $u_t = 0$, satisfy the corresponding elliptic equation. However, it is not generally true that stationary and transient descriptions of physical phenomena are corresponding pairs of elliptic and parabolic equations. Counterexamples are the Poisson equation and the Maxwell equations: the Poisson equation is stationary, elliptic, and governs electrostatic problems, while the Maxwell equations are transient, hyperbolic, and govern electromagnetism. This takes us to hyperbolic equations.

10.4 Hyperbolic Equations

The solutions of hyperbolic equations behave like waves. More precisely, if the initial condition for the solution at time $t = 0$ is disturbed, it takes a finite amount of time to observe this disturbance at other points of space, meaning that the disturbance has a finite propagation speed in contrast to elliptic and parabolic equations, where a disturbance is observed everywhere immediately and the hence the propagation speed is infinite.

The simplest example of a hyperbolic equation is the one-dimensional (in space) wave equation

$$u_{tt} = c^2 u_{xx}$$

equipped with an initial condition $u(t = 0, x) = f(x)$ and boundary conditions such as $u(t, x = x_1) = g_1(t)$ and $u(t, x = x_2) = g_2(t)$. In the case of hyperbolic equations, the choice of boundary conditions is often a delicate matter. As the waves travel long enough, the waves may leave a finite boundary, and therefore one often tries to make provisions to enable the wave to leave the boundary without any disturbance or reflection. One often also generates waves on the boundary to enter the domain.

It is easy to find an exact solution of the wave equation 10.4. Any function $u(t, x) := f(x - ct)$ is a solution – as long as the boundary conditions match – as can be checked in a straightforward manner using the chain rule. Here the function f is the initial condition. It is clear that the constant $c \in \mathbb{R}$ is the speed of the wave. This exact solution is useful for testing numerical methods.

Another important class of hyperbolic equations are conservation laws. To derive a conservation law, we start from the equation

$$\frac{\mathrm{d}}{\mathrm{d}t} \iiint_{\Omega} u(\mathbf{x}) \mathrm{d}\mathbf{x} + \oiint_{\partial\Omega} \mathbf{n} \cdot \mathbf{f}(u) \mathrm{d}S = 0.$$

The first term is the time rate of change of u in the subdomain $\Omega \subset D$, which is arbitrary with a sufficiently smooth boundary in this equation. The second integral is a surface integral and gives the flux $\mathbf{f}$ of u through the boundary $\partial\Omega$ of Ω, where the $\mathbf{n}$ are outward unit normal vectors. The equation just means that the change of u contained in Ω is equal to the negative total outflow of u from Ω.

If u and $\mathbf{f}$ are sufficiently smooth functions, we can change the order of differentation and integration in the first term and use the divergence theorem in the second term to find

$$\iiint_\Omega u_t(\mathbf{x})\mathrm{d}\mathbf{x} + \iiint_\Omega \nabla \cdot \mathbf{f}(u(\mathbf{x}))\mathrm{d}\mathbf{x} = \iiint_\Omega \big(u_t(\mathbf{x}) + \nabla \cdot \mathbf{f}(u(\mathbf{x}))\big)\mathrm{d}\mathbf{x} = 0.$$

Since the subdomain Ω is arbitrary, the fundamental lemma of variational calculus, Theorem 10.1, applied to the second equation yields that the integrand vanishes everywhere, i.e.,

$$u_t + \nabla \cdot \mathbf{f}(u) = 0 \qquad \forall \mathbf{x} \in D.$$

In higher dimensions, an equation for $u\colon \mathbb{R}^d \to \mathbb{R}$ of this form is called hyperbolic if the Jacobian matrix

$$\begin{pmatrix} \frac{\partial f_1}{\partial u_1} & \cdots & \frac{\partial f_1}{\partial u_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_d}{\partial u_1} & \cdots & \frac{\partial f_d}{\partial u_d} \end{pmatrix}$$

of the flux function $\mathbf{f}\colon \mathbb{R}^d \to \mathbb{R}^d$ has only real eigenvalues and is diagonalizable. If the Jacobian matrix even has distinct real eigenvalues, it is certainly diagonalizable, and then the equation is called strictly hyperbolic.

10.5 Finite Differences

The finite-difference method is one of the most important numerical methods for PDEs. Solving a PDE using the finite-difference method requires these steps.

1. Define grid points on the domain. The solution will be calculated on these grid points.
2. Approximate the derivatives in the equation on the grid points using Taylor's theorem.
3. Substitute the approximations of the derivatives into the equation, which yields a system of algebraic equations for the values of the solution at the grid points.
4. Solve the system of algebraic equations.
5. Test the finite-difference discretization and its implementation by comparing exact and numerical solutions for different grid sizes.

Before implementing finite differences for a one-dimensional equation first, we discuss these steps in more detail. It goes without saying that there are many variants how the first, second, and fourth steps can be implemented.

Regarding the choice of grid points in the first step, it is customary to use equidistant grid points. In one dimension, this means that the solution u is calculated at the points

$$u_i := u(a + ih),$$

where the domain is the interval $U := (a, b)$, $h \in \mathbb{R}^+$ is the grid spacing, and the index $i \in \{1, \dots, N\}$ is related to the grid spacing h by

$$h = \frac{b - a}{N}.$$

In two dimensions, we have

$$u_{i,j} = u(a_1 + ih, a_2 + jh),$$

where the domain is $U := (a_1, b_1) \times (a_2, b_2)$ and the indices are $i \in \{1, \dots, N_1\}$ and $j \in \{1, \dots, N_2\}$, and so forth in higher dimensions.

Finite differences are especially well suited for domains with simple boundaries, while the finite-element method (see Sect. 10.7 below) is especially well suited for domains with complex boundaries that are to be resolved precisely.

Taylor's theorem is used to approximate the derivatives in the second step.

Theorem 10.10 (Taylor's theorem) *Suppose that $k \in \mathbb{N}$ and that the function $f: \mathbb{R} \to \mathbb{R}$ is n times differentiable at the point $a \in \mathbb{R}$. Then there exists a function $h: \mathbb{R} \to \mathbb{R}$ such that*

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x - a)^k + h(x)(x - a)^n$$

and

$$\lim_{x \to a} h(x) = 0.$$

The Taylor expansion must be truncated at one point. This gives rise to the local truncation error. Using more terms in the Taylor expansion generally results in smaller local truncation errors in the third step, and we will implement an example below. However, more terms generally complicate the system of algebraic equations, making it more time consuming to assemble and to solve. It is therefore not obvious a priori if the accuracy of the solutions is increased by increasing the number of terms in the Taylor expansion when the total computation time is the same.

Regarding the solution of the resulting system of algebraic equations, it is obvious that a linear PDE will result in a linear system of equations (see Sect. 8.4.8). As the number of dimensions of the domain increases, the system matrices become sparser, and it is imperative to use sparse matrices (see Sect. 8.2). We will discuss the implementation in more details below.

If the PDE and thus the system of algebraic equations are nonlinear, Newton methods (see Sect. 12.7) and fixed-point methods are the methods of choice.

10.5.1 One-Dimensional Second-Order Discretization

We now apply these ideas to the prototypical one-dimensional elliptic boundary-value problem

$$\begin{aligned} -u_{xx}(x) &= f(x) \qquad \forall x \in (a,b) \subset \mathbb{R}, \\ u(a) &= g_1, \\ u(b) &= g_2 \end{aligned}$$

in the interval (a, b), where the function $f\colon \mathbb{R} \to \mathbb{R}$ and the constants $g_1 \in \mathbb{R}$ and $g_2 \in \mathbb{R}$ are given.

In the first step, we use the equidistant grid $a + ih$ defined above. The two boundary conditions result in $u_0 = u(a) = g_1$ and $u_N = u(a+Nh) = u(b) = g_2$.

In the second step, to approximate the derivative u_{xx} in the equation, we apply Taylor's theorem, Theorem 10.10, to find the two expansions

$$\begin{aligned} u_{i+1} &= u_i + hu_x(a+ih) + \frac{h^2}{2}u_{xx}(a+ih) + \frac{h^3}{6}u_{xxx}(a+ih) + O(h^4), \\ u_{i-1} &= u_i - hu_x(a+ih) + \frac{h^2}{2}u_{xx}(a+ih) - \frac{h^3}{6}u_{xxx}(a+ih) + O(h^4), \end{aligned}$$

for $u_{i+1} = u(a + (i+1)h)$ and $u_{i-1} = u(a + (i-1)h)$ around the point $a + ih$. Here $O(h^4)$ includes all terms of fourth order and higher in h. More precisely, we write $f(x) = O(g(x))$ as $x \to x_0$ if and only if $\limsup_{x\to a} |f(x)/g(x)| < \infty$.

Adding these two expansions yields

$$u_{i+1} + u_{i-1} = 2u_i + h^2 u_{xx}(a+ih) + O(h^4),$$

and then dividing by h^2 and reordering yields

$$u_{xx}(a+ih) = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} + O(h^2).$$

In the third step, we write down the system of algebraic equations, which is linear for this linear equation, and solve it using JULIA. There are $N-1$ unknowns, namely the values u_i for all $i \in \{1,\dots,N-1\}$. For each unknown value u_i, we have the algebraic equation

$$-\frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} = f(a+ih) + O(h^2) \qquad \forall i \in \{1,\dots,N-1\}$$

or, equivalently,

$$-(u_{i+1} - 2u_i + u_{i-1}) = h^2 f(a+ih) + O(h^4) \qquad \forall i \in \{1,\dots,N-1\} \qquad (10.15)$$

found by substituting the equation for $u_{xx}(a+ih)$ into the boundary-value problem. We observe that the local truncation error is a term $O(h^2)$ of second order in h, rendering this finite-difference discretization a second-order one.

In order to solve this linear system of equations, we write a function that records each equation in a row of a sparse matrix and then calls the standard solver in JULIA for this type of linear equation. The vector `fs` that contains the right side is a dense vector. The system matrix `M` is initialized as an empty, sparse $(N-1)\times(N-1)$ matrix. In the loop, the coefficients of u_{i+1}, u_i, and u_{i-1} are written into the i-th row of the matrix. The two equations that contain the boundary conditions, i.e., the ones for $i=0$ and $i=N$, require special treatment, and the constant terms $u_0=g_1$ and $u_N=g_2$ go on the right side. For solving the linear system of equations, we use the built-in function.

```julia
import LinearAlgebra
import SparseArrays

function elliptic_FD_1D(a::Float64, b::Float64, f::Function,
                        g1::Float64, g2::Float64,
                        N::Int)::Vector{Float64}
    local h = (b-a) / N
    local fs = Float64[h^2 * f(a + i*h) for i in 1:N-1]
    local M = SparseArrays.spzeros(N-1, N-1)

    ## interior
    for i in 2:N-2
        M[i, i+1] = -1.0
        M[i, i]   =  2.0
        M[i, i-1] = -1.0
    end

    ## left boundary
    M[1, 1] =  2.0
    M[1, 2] = -1.0
    fs[1] += g1

    ## right boundary
    M[N-1, N-1] =  2.0
    M[N-1, N-2] = -1.0
    fs[N-1] += g2

    ## solve
    M \ fs
end
```

It is an unfortunate fact of life that the implementation of each boundary condition – which seems so unimportant, because it concerns only few points compared to the number of interior points – requires as much care as the implementation of the equation for the interior points. If the finite-difference discretization (10.15) would also contain u_{i+2} or u_{i-2}, then we would have to implement more special cases close to the boundary.

Calculating a single solution of a PDE is not only useless, but may be outright dangerous depending on what the solution is used for. In the fifth and last step, we hence write a function to test our discretization and its implementation. We can easily find the exact solution in our test cases by putting the cart before the horse: we define the solution first, in our example $u := \sin$, and only then obtain the right-hand side f and the boundary conditions after substituting the solution into the equation. After calculating the numerical approximation of the solution and evaluating the exact solution at the grid points, we calculate the error as the maximum norm of the difference between the approximation and the exact solution. This procedure is implemented in the following function.

```julia
function test_elliptic_FD_1D(u_exact::Function, f::Function,
                             a::Float64, b::Float64, N::Int)::Float64
    local h = (b-a) / N
    local u_ex = Float64[u_exact(a + i*h) for i in 1:N-1]
    local u_num = elliptic_FD_1D(a, b, f, u_exact(a), u_exact(b), N)

    LinearAlgebra.norm(u_num - u_ex, Inf)
end
```

Now we can easily test our implementation. In the following test case, the domain is the interval $(0, 2\pi)$ and $u := \sin$ yields $f = \sin$. The right-hand side could also be obtained using symbolic computations, automating testing even further. We calculate the error on four grids, namely with 10^1, 10^2, 10^3, and 10^4 points.

```julia
import Printf
for i in 1:4
    local error = test_elliptic_FD_1D(sin, sin, 0.0, 2*pi, 10^i)
    Printf.@printf("N = 10^%1d: error = %.5e\n", i, error)
end
```

```
N = 10^1: error = 3.19159e-02
N = 10^2: error = 3.29052e-04
N = 10^3: error = 3.28988e-06
N = 10^4: error = 3.29044e-08
```

We observe that dividing h by 10 (or equivalently multiplying N by 10) divides the error by factor of 100. This is exactly what is expected, since the local truncation error in (10.15) is $O(h^2)$ and the discretization is thus a second-order one.

Not much imagination is required to see that many variations in the steps above are possible. Most importantly, to affect the convergence speed of a finite-difference scheme, many variations how to apply Taylor's theorem are possible. This is the question we investigate next.

10.5.2 Compact Fourth-Order Finite-Difference Discretizations

In this section, fourth-order finite-difference discretizations of the Laplace operator

$$\Delta u = \nabla \cdot (\nabla u) = \sum_{i=1}^{d} u_{x_i x_i}$$

in two and three dimensions d are derived. In addition to being of fourth order, the schemes still have the desirable property that only neighboring grid points are used. This fact considerably simplifies the implementation at the grid points near the boundary. Another advantage is the small bandwidth in the resulting linear system of equations, meaning that it can be solved faster. Therefore these two- and three-dimensional finite-difference discretizations combine fast convergence as $h \to 0$ with ease of implementation, providing a good example of how Taylor's theorem can be applied to good effect.

10.5.2.1 For Two-Dimensional Elliptic Equations

We consider the two-dimensional elliptic equation

$$\Delta u = u_{xx} + u_{yy} = f \qquad \text{in } U := (a_1, b_1) \times (a_2, b_2)$$

with twice differentiable right-hand side f and discretize it on an equidistant grid with spacing h. The discretization

$$\begin{aligned}
&\frac{1}{6}(u_{i+1,j+1} + u_{i+1,j-1} + u_{i-1,j+1} + u_{i-1,j-1}) \\
&\qquad + \frac{2}{3}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) - \frac{10}{3} u_{i,j} \\
&\qquad = h^2\Big(\frac{2}{3} f_{i,j} + \frac{1}{12}(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1})\Big) + O(h^6)
\end{aligned}$$

can be written in symbolic form neatly as

$$\begin{bmatrix} \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \frac{2}{3} & -\frac{10}{3} & \frac{2}{3} \\ \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{bmatrix}_{i,j} u_{i,j} = h^2 \begin{bmatrix} 0 & \frac{1}{12} & 0 \\ \frac{1}{12} & \frac{2}{3} & \frac{1}{12} \\ 0 & \frac{1}{12} & 0 \end{bmatrix}_{i,j} f_{i,j} + O(h^6), \tag{10.16}$$

where the matrix elements are the coefficients of the unknown $u_{i,j}$ at the grid point (i, j) and its neighbors.

This discretization is a fourth-order compact finite-difference one and derived in the proof of the following theorem. Such a discretization is often called compact, since it only involves the eight neighboring grid points (i, j); a general fourth-order discretization involves more grid points.

Theorem 10.11 (fourth-order compact finite-difference discretization of two-dimensional elliptic equations) *The local truncation error of the finite-difference discretization (10.16) of the two-dimensional elliptic equation*

$$\Delta u = f \qquad \text{in } D \subset \mathbb{R}^2 \tag{10.17}$$

with twice differentiable f is of fourth order.

Proof We use Taylor's theorem, Theorem 10.10, to find the two expansions

$$u_{i+1,j} = u_i + hu_x + \frac{h^2}{2}u_{xx} + \frac{h^3}{6}u_{xxx} + \frac{h^4}{24}u_{xxxx} + \frac{h^5}{120}u_{xxxxx} + O(h^6),$$
$$u_{i-1,j} = u_i - hu_x + \frac{h^2}{2}u_{xx} - \frac{h^3}{6}u_{xxx} + \frac{h^4}{24}u_{xxxx} - \frac{h^5}{120}u_{xxxxx} + O(h^6)$$

for $u_{i+1,j} = u(a_1 + (i+1)h, a_2 + jh)$ and $u_{i-1} = u(a_1 + (i-1)h, a - 2 + jh)$ with respect to x around the point $a + ih$. For convenience, the arguments $(a_1 + ih, a_2 + jh)$ of the derivatives are dropped. Adding these two expansions and rearranging terms shows that the equation

$$D_x^2 u_{i,j} := \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} = u_{xx} + \frac{h^2}{12}u_{xxxx} + O(h^4) \tag{10.18}$$

holds for the central-difference operator D_x^2 that acts with respect to x.

Therefore the local truncation error $\tau_{i,j}$ of the initial (second-order) discretization

$$D_x^2 u_{i,j} + D_y^2 u_{i,j} = f_{i,j} + \tau_{i,j} \tag{10.19}$$

of (10.17) equals

$$\tau_{i,j} = \frac{h^2}{12}(u_{xxxx} + u_{yyyy}) + O(h^4).$$

In order to obtain more information about the coefficient of h^2, we differentiate (10.17) twice with respect to x and twice with respect to y to find

$$u_{xxxx} = f_{xx} - u_{xxyy},$$
$$u_{yyyy} = f_{yy} - u_{xxyy}.$$

Hence we can rewrite the truncation error as

$$\tau_{i,j} = \frac{h^2}{12}(f_{xx} + f_{yy}) - \frac{h^2}{6}u_{xxyy} + O(h^4). \tag{10.20}$$

Next, we approximate the terms in the coefficient of h^2 by

$$\begin{aligned} f_{xx} &= D_x^2 f_{i,j} + O(h^2), \\ f_{yy} &= D_y^2 f_{i,j} + O(h^2), \\ u_{xxyy} &= D_x^2 D_y^2 u_{i,j} + O(h^2), \end{aligned}$$

which are second-order discretizations as we already know, which yields

$$\tau_{i,j} = \frac{h^2}{12}(D_x^2 f_{i,j} + D_y^2 f_{i,j}) - \frac{h^2}{6}D_x^2 D_y^2 u_{i,j} + O(h^4).$$

We substitute this form of $\tau_{i,j}$ into the initial discretization (10.19).

In summary, the sought discretization is

$$\underbrace{D_x^2 u_{i,j}}_{h^{-2}} + \underbrace{D_y^2 u_{i,j}}_{h^{-2}} + \underbrace{\frac{h^2}{6}D_x^2 D_y^2 u_{i,j}}_{h^{-2}} = \underbrace{f_{i,j}}_{h^0} + \underbrace{\frac{h^2}{12}(D_x^2 + D_y^2) f_{i,j}}_{h^0} + \underbrace{O(h^4)}_{h^4}, \tag{10.21}$$

which yields (10.16) after expanding the central-difference operators D_x^2 and D_y^2 and multiplying by h^2. The local truncation error $O(h^4)$ of this discretization is a factor h^4 apart from the other terms. □

10.5.2.2 For Three-Dimensional Elliptic Equations

Analogously to Theorem 10.11, the following result holds for three-dimensional elliptic equations with sufficiently smooth right-hand side.

Theorem 10.12 (fourth-order compact finite-difference discretization of three-dimensional elliptic equations) *The local truncation error of the finite-difference discretization*

$$\begin{aligned} &-4u_{i,j,k} + \tfrac{1}{3}(u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1}) \\ &\qquad + \tfrac{1}{6}(u_{i,j+1,k+1} + u_{i,j+1,k-1} + u_{i,j-1,k+1} + u_{i,j-1,k-1} \\ &\qquad + u_{i+1,j,k+1} + u_{i+1,j,k-1} + u_{i-1,j,k+1} + u_{i-1,j,k-1} \\ &\qquad + u_{i+1,j+1,k} + u_{i+1,j-1,k} + u_{i-1,j+1,k} + u_{i-1,j-1,k}) \\ &= h^2\big(\tfrac{1}{2}f_{i,j,k} + \tfrac{1}{12}(f_{i+1,j,k} + f_{i-1,j,k} + f_{i,j+1,k} + f_{i,j-1,k} + f_{i,j,k+1} + f_{i,j,k-1})\big) \\ &\qquad + O(h^6) \end{aligned} \tag{10.22}$$

of the three-dimensional elliptic equation

$$\Delta u = f \qquad \text{in } D \subset \mathbb{R}^3 \tag{10.23}$$

with twice differentiable f is of fourth order.

Proof Analogously to the proof of Theorem 10.11, the local truncation error $\tau_{i,j,k}$ of the initial (second-order) discretization

$$D_x^2 u_{i,j,k} + D_y^2 u_{i,j,k} + D_z^2 u_{i,j,k} = f_{i,j,k} + \tau_{i,j,k}$$

equals

$$\tau_{i,j,k} = \frac{h^2}{12}(u_{xxxx} + u_{yyyy} + u_{zzzz}) + O(h^4).$$

Differentiating (10.23) yields

$$\begin{aligned} u_{xxxx} &= f_{xx} - u_{xxyy} - u_{xxzz}, \\ u_{yyyy} &= f_{yy} - u_{xxyy} - u_{yyzz}, \\ u_{zzzz} &= f_{zz} - u_{xxzz} - u_{yyzz}. \end{aligned}$$

Substituting into $\tau_{i,j,k}$ results in

$$\tau_{i,j,k} = \frac{h^2}{12}(f_{xx} + f_{yy} + f_{zz}) - \frac{h^2}{6}(u_{xxyy} + u_{xxzz} + u_{yyzz}) + O(h^4).$$

Using this expression for $\tau_{i,j,k}$ in the initial discretization yields

$$\begin{aligned} D_x^2 u_{i,j,k} + D_y^2 u_{i,j,k} + D_z^2 u_{i,j,k} + \frac{h^2}{6}(D_x^2 D_y^2 + D_x^2 D_z^2 + D_y^2 D_z^2) u_{i,j,k} \\ = f_{i,j,k} + \frac{h^2}{12}(D_x^2 + D_y^2 + D_z^2) f_{i,j,k} + O(h^4), \end{aligned} \tag{10.24}$$

which is of fourth order. Finally, expanding the central-difference operators and multiplying by h^2 yields (10.22). □

10.6 Finite Volumes

Another approach to the numerical approximation of solutions of PDEs is the finite-volume method. It is suited for PDEs in divergence form, i.e., when the operator is the divergence of an expression involving the unknown, as is often the case with elliptic and parabolic equations.

The general idea of the finite-volume method is to integrate the PDE over finite volumes or control volumes surrounding each grid point and to employ the divergence theorem in order to convert the divergence term in the equation into a surface integral. These surface integrals are fluxes through the surfaces of all finite volumes or control volumes. The fluxes are conserved by construction, as the flux through any surface of a finite volume must equal the flux out of an adjacent volume through the same surface.

The conservation of fluxes is an important feature of the finite-volume method. The theoretical treatment of the finite-volume method such as convergence proofs is more complicated compared to the finite-difference method, where Taylor expansions are available, and to the finite-element method.

Here, we derive a finite-volume discretization of the prototypical elliptic PDE in divergence form, namely the two-dimensional Poisson equation

$$-\nabla \cdot (A\nabla u) = f \qquad \text{in } U \subset \mathbb{R}^2, \tag{10.25}$$

for which Theorem 10.8 holds. We first choose a grid spacing h and grid points $(ih, jh) \in U$, where the integers i and j are chosen such that the grid points lie in the domain U. Next, we define the control volumes

$$V_{i,j} := \big((i-1/2)h, (i+1/2)h\big) \times \big((j-1/2)h, (j+1/2)h\big)$$

surrounding the grid points (see Fig. 10.1). Since the grid points we have defined lie on an equidistant grid, the control volumes are rectangles. The sought values are the values

$$u_{i,j} := u(ih, jh)$$

of the solution at the grid points. We analogously write $A_{i,j} := A(ih, jh)$ and $f_{i,j} := f(ih, jh)$.

By applying the divergence theorem

$$\iint_V \nabla \cdot \mathbf{J} \mathrm{d}V = \oint_{\partial V} \mathbf{n} \cdot \mathbf{J} \mathrm{d}S,$$

where $\mathbf{n}$ is the outward unit normal vector, to the control volume $V_{i,j}$, equation (10.25) becomes

$$-\oint_{\partial V_{i,j}} \mathbf{n} \cdot (A\nabla u) \mathrm{d}S = \iint_{V_{i,j}} f \mathrm{d}V.$$

Next, we approximate u_x at $u_{i+1/2,j}$ by

$$u_x\big((i+1/2)h, jh\big) = \frac{u_{i+1,j} - u_{i,j}}{h} + O(h^2)$$

and analogously on the other edges. This equation follows from applying Taylor's theorem to u around $u_{i+1/2,j}$ with steps $h/2$ and $-h/2$ to find

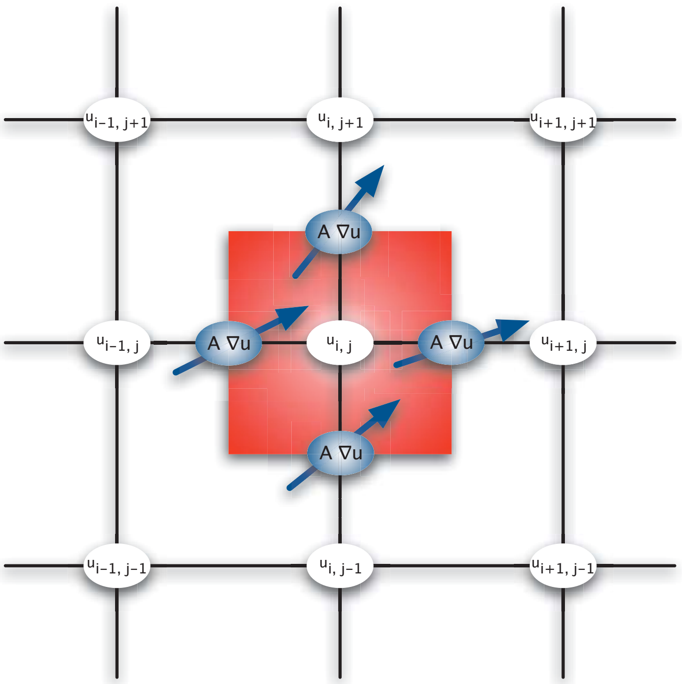

Fig. 10.1 The control volume $V_{i,j}$ of a finite-volume discretization is shown in red in the center. The fluxes $F = A\nabla u$ are shown as well and are assumed to be constant on the edges of the control volume.

$$u_{i+1,j} = u_{i+1/2,j} + \frac{h}{2}u_x + \frac{h^2}{2\cdot 4}u_{xx} + \frac{h^3}{6\cdot 8}u_{xxx} + O(h^4)$$
$$u_{i,j} = u_{i+1/2,j} - \frac{h}{2}u_x + \frac{h^2}{2\cdot 4}u_{xx} - \frac{h^3}{6\cdot 8}u_{xxx} + O(h^4)$$

and then subtracting.

Assuming that the matrix-valued function A is the diagonal matrix

$$\begin{pmatrix} a^{11}(x,y) & 0 \\ 0 & a^{22}(x,y) \end{pmatrix}$$

everywhere and that A is constant on the edges of $\partial V_{i,j}$, we hence obtain the discretization

$$\begin{aligned}-\Big(a^{11}_{i+1/2,j}\frac{u_{i+1,j}-u_{i,j}}{h}h+a^{11}_{i-1/2,j}\frac{u_{i-1,j}-u_{i,j}}{h}h\\+a^{22}_{i,j+1/2}\frac{u_{i,j+1}-u_{i,j}}{h}h+a^{22}_{i,j-1/2}\frac{u_{i,j-1}-u_{i,j}}{h}h\Big)\\=h^2f_{i,j}+O(h^3),\quad(10.26)\end{aligned}$$

since the length of all edges is h. If f is not constant on the control volume, integration formulas such as Simpson's rule are useful.

The discretization simplifies to

$$\begin{aligned}-\,(a^{11}_{i+1/2,j}u_{i+1,j}+a^{11}_{i-1/2,j}u_{i-1,j}+a^{22}_{i,j+1/2}u_{i,j+1}+a^{22}_{i,j-1/2}u_{i,j-1}\\-\,(a^{11}_{i+1/2,j}+a^{11}_{i-1/2,j}+a^{22}_{i,j+1/2}+a^{22}_{i,j-1/2})u_{i,j})\\=h^2f_{i,j}+O(h^3),\quad(10.27)\end{aligned}$$

which is of first order.

It is interesting to compare the finite-volume discretization to its finite-difference cousin in Sect. 10.5.1. The finite-difference discretization (when generalized to two dimensions) is of second order, while the finite-volume discretization here is of first order. The difference is apparently due to the fact that the second derivative was approximated in the finite-difference method, but the first derivative was approximated here in the finite-volume method. But the underlying reason is that the equations are different; in the present section, we consider (10.25), which contains the matrix-valued coefficient function A, about whose smoothness we have only supposed that it is constant on the edges of the control volumes. In particular, the coefficient function A may be discontinuous.

What happens if the coefficient function is smoother? If the coefficient function A is a constant $a \in \mathbb{R}^+$, then (10.27) simplifies to the discretization

$$-a(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j})=h^2f_{i,j}+O(h^3),$$

which is of first order, while it can be shown using the approach in Sect. 10.5 that the straightforward finite-difference discretization of $a\Delta u = f$ is

$$-a(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j})=h^2f_{i,j}+O(h^4),\qquad(10.28)$$

which is of second order. Thus, if A is constant, the finite-difference and finite-volume discretizations are identical, but the finite-difference method yields better theoretical convergence behavior.

However, the better theoretical convergence prediction of the finite-difference method is contingent upon sufficient smoothness of the solution u (and the coefficient A if applicable) to allow the use of Taylor's theorem. In the derivation of the finite-volume discretization, the only assumption on the smoothness of the solution u was that its derivatives u_x and u_y can be approximated by

$(u_{i+1,j} - u_{i,j})/h$ etc. between grid points; the rest of the calculations involves surface and volume integrals.

In summary, the main appeal of finite volumes is that they conserve the fluxes by construction, which is especially important in physical problems such as diffusion and electrostatic problems, where flux conservation is a physical principle, i.e., mass conservation and Gauss' law, respectively. Finite volumes can also easily deal with coefficient functions A that are not constant and with solutions u that are less smooth. Another difference between finite differences and finite volumes is that finite volumes are amenable to better approximations of the right-hand side via

$$\iint_{V_{i,j}} f \mathrm{d}V,$$

making it possible to conserve $\iint_U f\mathrm{d}V$ when implemented carefully.

It is instructive to implement the finite-volume discretization (10.27) and to discuss some examples. We represent the Dirichlet boundary conditions $u = g$ on ∂U by a function `g` that is evaluated only at the boundary. Again, each row in the system matrix `M` corresponds to one equation (10.27) for one unknown value $u_{i,j}$. The helper function `ind` takes a two-dimensional index (i, j) with $i \in \{0, \dots, N\}$ and $j \in \{0, \dots, N\}$ and turns it into a linear, one-dimensional index running from 1 to $(N+1)^2$, which is useful for indexing the matrix and the vector of the discretized system of equations. Points with $i = 0$, $i = N$, $j = 0$, or $j = N$ lie on the boundary of the square domain $U := (a_1, b_1) \times (a_2, b_1 + a_2 - a_1)$.

```
import LinearAlgebra
import SparseArrays

function elliptic_FV_2D(a1::Float64, b1::Float64, a2::Float64,
                        a11::Function, a22::Function, f::Function,
                        g::Function, N::Int)::Array{Float64, 2}
    local h = (b1-a1) / N
    local fs = Vector{Float64}(undef, (N+1)^2)
    local M = SparseArrays.spzeros((N+1)^2, (N+1)^2)

    function ind(i::Int, j::Int)::Int
        1 + i + j*(N+1)
    end

    for i in 0:N
        for j in 0:N
            if i == 0 || i == N || j == 0 || j == N
                M[ind(i, j), ind(i, j)] = 1.0
                fs[ind(i, j)] = g(a1 + i*h, a2 + j*h)
            else
                M[ind(i, j), ind(i+1, j)] = - a11(a1 + (i+1/2)*h, a2 +  j*h)
                M[ind(i, j), ind(i-1, j)] = - a11(a1 + (i-1/2)*h, a2 +  j*h)
                M[ind(i, j), ind(i, j+1)] = - a22(a1 + i*h, a2 + (j+1/2)*h)
                M[ind(i, j), ind(i, j-1)] = - a22(a1 + i*h, a2 + (j-1/2)*h)

                M[ind(i, j), ind(i, j)] += a11(a1 + (i+1/2)*h, a2 +  j     *h)
                M[ind(i, j), ind(i, j)] += a11(a1 + (i-1/2)*h, a2 +  j     *h)
                M[ind(i, j), ind(i, j)] += a22(a1 +  i     *h, a2 + (j+1/2)*h)
                M[ind(i, j), ind(i, j)] += a22(a1 +  i     *h, a2 + (j-1/2)*h)

                fs[ind(i, j)] = h^2 * f(h, a1 + i*h, a2 + j*h)
            end

            local s = sum(M[ind(i, j), :])
            @assert isapprox(s, 0.0, atol = 5e-15) ||
                    isapprox(s, 1.0, atol = 5e-15)
        end
    end

    reshape(M \ fs, N+1, N+1)
end
```

Here, the strategy for implementing the boundary conditions is to explicitly include the equations $u_{i,j} = g_{i,j}$ for the unknown on the boundary, i.e., for $i = 0$, $i = N$, $j = 0$, or $j = N$. This strategy leads to shorter code than substituting the values on the boundary into the system while considering all cases.

The assertion follows from (10.27) and is useful to check that the implementation is correct.

The next function is useful to assess the accuracy of the solution in examples where the exact solution is known.

```
function test_elliptic_FV_2D(u_exact::Function,
                             a1::Float64, b1::Float64, a2::Float64,
                             a11::Function, a22::Function,
                             f::Function, N::Int)::Float64
   local h = (b1-a1) / N
   local u_num = elliptic_FV_2D(a1, b1, a2, a11, a22, f, u_exact, N)
   local u_ex = [u_exact(a1 + i*h, a2 + j*h) for i in 0:N, j in 0:N]

   LinearAlgebra.norm(u_num - u_ex, Inf)
end
```

In the first example, we set $U := (-\pi, \pi)^2$, $u(x, y) := \cos x \cos y$, and

$$A(x, y) := \begin{pmatrix} 2 + \cos y & 0 \\ 0 & 2 + \cos x \end{pmatrix},$$

which results in $f(x, y) = (4 + \cos x + \cos y) \cos x \cos y$. We calculate some solutions, each time multiplying N by 2 or h by 1/2.

```
for i in 0:5
    local N = 10 * 2^i
    local error =
        test_elliptic_FV_2D((x, y) -> cos(x) * cos(y),
            -pi, Float64(pi), -pi,
            (x, y) -> 2 + cos(y),
            (x, y) -> 2 + cos(x),
            (h, x, y) -> (4 + cos(x) + cos(y)) * cos(x) * cos(y),
            N)
    Printf.@printf("N = %3d: error = %.5e\n", N, error)
end
```

```
N =  10: error = 5.13949e-02
N =  20: error = 1.25842e-02
N =  40: error = 3.12985e-03
N =  80: error = 7.81456e-04
N = 160: error = 1.95301e-04
N = 320: error = 4.88214e-05
```

We observe that the error is multiplied by a factor approximately equal to $1/4$, which implies a second-order convergence rate, which is consistent with the discussion above for smooth coefficient functions A.

In the second example, we consider a solution u that is not differentiable. This example is one-dimensional, but we still use our two-dimensional implementation. We set $U := (-1, 1)^2$,

$$u(x, y) := \begin{cases} x, & x < 0, \\ 2x, & x \geq 1, \end{cases}$$

and

$$A(x, y) := \begin{cases} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, & x < 0, \\ \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}, & x \geq 1, \end{cases}$$

which results in constant $A\nabla u$ and $f(x, y) = 0$. This example is covered by the theory in Sect. 10.2.3. In electrostatics, this example corresponds to material with a jump discontinuity in the permittivity, which results in a jump in the derivative of the solution. We again calculate some solutions, each time multiplying h by $1/2$.

```
for i in 0:5
    local N = 10 * 2^i
    local error =
        test_elliptic_FV_2D((x, y) -> if x <= 0; x else 2*x end,
                            -1.0, 1.0, -1.0,
                            (x, y) -> if x <= 0; 1 else 1/2 end,
                            (x, y) -> if x <= 0; 1 else 1/2 end,
                            (h, x, y) -> 0.0,
                            N)
    Printf.@printf("N = %3d: error = %.5e\n", N, error)
end
```

```
N =  10: error = 5.55112e-16
N =  20: error = 4.44089e-16
N =  40: error = 1.33227e-15
N =  80: error = 7.77156e-16
N = 160: error = 3.55271e-15
N = 320: error = 5.77316e-15
```

These surprisingly accurate results are explained by the simple structure of the solution u, which is linear except at the line $x = 0$ and by the fact that the control volumes are perfectly aligned with the line $x = 0$, implying that the discretization error indeed vanishes. The only sources of error are solving the linear system and floating-point errors.

Next, we redefine the domain to $U := (-1, 2)^2$.

```
for i in 0:5
    local N = 10 * 2^i
    local error =
        test_elliptic_FV_2D((x, y) -> if x <= 0; x else 2*x end,
                            -1.0, 2.0, -1.0,
                            (x, y) -> if x <= 0; 1 else 1/2 end,
                            (x, y) -> if x <= 0; 1 else 1/2 end,
                            (h, x, y) -> 0.0,
                            N)
    Printf.@printf("N = %3d: error = %.5e\n", N, error)
end
```

```
N =  10: error = 6.09231e-02
N =  20: error = 3.51955e-02
N =  40: error = 1.74043e-02
N =  80: error = 9.00100e-03
N = 160: error = 4.48787e-03
N = 320: error = 2.26274e-03
```

In each step, the error is multiplied by approximately $1/2$, and we finally observe the first-order convergence we predicted in (10.27). This is the general case.

In the third example, we consider a solution u that is not differentiable, but now the jump in the derivative of u is not offset by the coefficient A, but corresponds to a Dirac delta distribution on the right-hand side. We set $U := (-1, 1)^2$,

$$u(x, y) := \begin{cases} -x, & x < 0, \\ x, & x \geq 0, \end{cases}$$

and

$$A(x, y) := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

which results in $f(x, y) = -2\delta(x)$, where δ is the Dirac delta distribution. Its defining characteristic is the equality

$$\int_{\mathbb{R}} \delta(x)\mathrm{d}x = 1, \tag{10.29}$$

meaning that it can concentrate an integral of value 1 at a single point $x = 0$. In electrostatics, this example corresponds to two unit charges placed at $x = 0$, which results in a jump in the derivative of the solution. We again calculate some solutions, each time multiplying h by $1/2$.

In order to implement the Dirac delta distribution, we note that if it takes the value $1/h$ on the control volume (better called control interval in this one-dimensional case) of length h that contains 0, then (10.29) is satisfied. We can also arrive at the same conclusion from (10.26): in our one-dimensional example, we have $-2h = h^2 f_{i,j} + O(h^3)$ and hence $f_{i,j} := -2/h$ because of $(u_{i+1,j} - u_{i,j})/h = 1$ and $(u_{i-1,j} - u_{i,j})/h = 1$.

```
for i in 0:5
    local N = 10 * 2^i
    local error =
        test_elliptic_FV_2D((x, y) -> if x <= 0; -x else x end,
            -1.0, 1.0, -1.0,
            (x, y) -> 1,
            (x, y) -> 1,
            (h, x, y) -> if abs(x) < h/2 - sqrt(eps(Float64))
                             -2/h
                         else
                             0
                         end,
                N)
    Printf.@printf("N = %3d: error = %.5e\n", N, error)
end
```

```
N =  10: error = 4.44089e-16
N =  20: error = 3.33067e-16
N =  40: error = 6.66134e-16
N =  80: error = 5.55112e-16
N = 160: error = 1.33227e-15
N = 320: error = 4.21885e-15
```

As in the second example, the surprising accuracy is due to the linearity of the solution and the perfect alignment of the control volumes with the line $x = 0$, where the derivative of the solution jumps. This numerical result validates our implementation of the Dirac delta distribution.

Finally, we redefine the domain to $U := (-1, 2)^2$.

```
for i in 0:5
    local N = 10 * 2^i
    local error =
        test_elliptic_FV_2D((x, y) -> if x <= 0; -x else x end,
            -1.0, 2.0, -1.0,
            (x, y) -> 1,
            (x, y) -> 1,
            (h, x, y) -> if abs(x) < h/2 - sqrt(eps(Float64))
                                 -2/h
                         else
                                  0
                         end,
                N)
    Printf.@printf("N = %3d: error = %.5e\n", N, error)
end
```

```
N =  10: error = 9.91258e-02
N =  20: error = 5.40759e-02
N =  40: error = 2.75139e-02
N =  80: error = 1.40401e-02
N = 160: error = 7.04891e-03
N = 320: error = 3.54226e-03
```

In each step, the error is again multiplied by approximately $1/2$, which is again consistent with the general case of first-order convergence as predicted in (10.27).

Finally, we note that the implementation of the boundary conditions may require thought and effort. While Dirichlet boundary conditions are relatively straightforward to implement, Neumann boundary conditions generally require an approximation of the directional derivative. If this approximation is not good enough, it may reduce the convergence order, although the discretization in the interior would support a higher convergence order.

10.7 Finite Elements

We have seen in the previous section that the use of integration in the finite-volume method was advantageous compared to finite differences, as it made it possible to relax the assumptions on the smoothness of the solution. Finite elements take these considerations further. We have already laid the foundation for finite elements above in Sect. 10.2.3; finite elements are just the numerical implementation of weak solutions. (If you skipped Sect. 10.2.3, the main points are explained in the following again for the purposes of finite elements.)

We again use the elliptic boundary-value problem

$$-\nabla \cdot (A\nabla u) = f \qquad \text{in } U \subset \mathbb{R}^d, \tag{10.30a}$$

$$u = 0 \qquad \text{on } \partial U \tag{10.30b}$$

in divergence form as the leading equation to show how finite-element discretizations follow from weak solutions. As will become apparent after the integration-by-parts step, we seek solutions u in $H_0^1(U)$, which is – simply put – the function space of all locally summable functions whose weak first (indicated by the index 1) partial derivatives are square integrable and that vanish on the boundary ∂U (indicated by the index 0); the reader is referred to [1, Section 5.2] for details. We first multiply (10.30) by an arbitrary test function v chosen from $H_0^1(U)$. The reason why we use the same function space for the test function will also become apparent after the step in which we integrate by parts.

Hence we seek solutions $u \in H_0^1(U)$ such that

$$-\int_U \nabla \cdot (A\nabla u) v \mathrm{d}V = \int_U f v \mathrm{d}V \qquad \forall v \in H_0^1(U).$$

Can we infer (10.30) from this equality? The answer is positive and is provided by the fundamental lemma of variational calculus, Theorem 10.1, which states that since this equality holds for sufficiently many test functions v (namely all elements of $H_0^1(U)$), the other factors $-\nabla \cdot (A\nabla u)$ and f in the integrands must agree. The intuitive explanation is that if $\int_U wv\mathrm{d}V = 0$ holds for sufficiently many functions v, especially those with tiny support that allow to zoom into smaller and smaller intervals, then the function w must vanish.

Next, we use integration by parts – called Green's first identity in this case (see Problem 10.21) – to see that the problem is equivalent to finding solutions $u \in H_0^1(U)$ such that

$$\int_U (A\nabla u) \cdot \nabla v \mathrm{d}V = \int_U f v \mathrm{d}V \qquad \forall v \in H_0^1(U) \tag{10.31}$$

holds, since the boundary term vanishes because of $H_0^1(U) \ni v = 0$ on the boundary ∂U. This formulation relaxes the requirements on the smoothness of u and explains the choice of function spaces for both u and v. The first derivatives of both u and v are required to exist only in the weak sense (since they appear in the integrand), whereas a classical solution u of (10.30) must be twice differentiable in the whole domain. Due to the symmetry of ∇u and ∇v appearing in the integrand, we can choose $H^1(U)$ as the function space for both the solution u and the test function v. Moreover, the zero Dirichlet boundary conditions are incorporated by choosing $H_0^1(U)$.

Such solutions u that satisfy (10.31) are called weak solutions. In its most general form, weak formulations are to find a function $u \in W$, the weak solution, such that

$$a(u, v) = F(v) \qquad \forall v \in V \tag{10.32}$$

holds. Here the Hilbert spaces V and W are the space of test functions and the solution space, respectively.

Up to now we have only explained the concept of a weak solution, but finite elements are just around the corner. Since we cannot perform numerical calculations using elements of the infinite-dimensional function spaces V and W, we restrict both the test space V and the solution space W to much simpler, finite-dimensional function spaces V_h and W_h that are both amenable to calculations and that will result in an algebraic system of equations with a unique solution at the same time. The index h indicates the fineness of these discretized function spaces and will become concrete in the example below. For now, it is important that a smaller h implies more elements in V_h and W_h, i.e., a higher-dimensional function space that can approximate functions in V and W better. The parameter h is inversely proportional to the dimension of V_h and W_h.

We can make this notion more precise by considering the family

$$\{V_h \subset V \mid h \in \mathbb{R}^+\}$$

of finite-dimensional subspaces V_h of V. We will choose the subspaces V_h such that

$$\forall v \in V: \qquad \lim_{h\to 0} \inf_{v_h \in V_h} \|v_h - v\|_V = 0. \tag{10.33}$$

Since $V_h \subset V$ for all $h \in \mathbb{R}^+$, an approximation using the subspaces V_h is called an internal approximation.

After choosing the function spaces V_h and W_h, the weak formulation (10.32) becomes the task to find a function $u_h \in W_h$ such that

$$a(u_h, v_h) = F(v_h) \qquad \forall v_h \in V_h. \tag{10.34}$$

The solution $u_h \in W_h$ of this problem is called the finite-element or Galerkin approximation of the solution u of (10.32).

In general, many different choices for such discretized function spaces V_h and W_h are possible, and we use the choice that is prototypical for finite elements here. We consider the one-dimensional case, denote the domain by $U := (a, b)$, and divide it into N intervals which form a partition of the whole domain and are called the finite elements. We denote the length of the largest interval by h. A set of all such intervals or finite elements with a certain value h is called a triangulation T_h, and h is called the fineness of the triangulation. (The name triangulation stems of course from the two-dimensional case.)

The intervals can be chosen to be of the same length

$$h := (b - a)/N$$

after choosing N in order to simplify the implementation. We can then define equidistant points

$$x_i := a + ih$$

for $i \in \{0, \dots, N\}$ such that there are N intervals of length h and $N + 1$ points x_i, but in general the points x_i are not necessarily equidistant.

The so-called hat functions ϕ_j^h, $j \in \{0, \dots, N\}$, are piecewise linear functions that have the value 1 at exactly one point x_i and that vanish at all other points x_i, i.e.,

$$\phi_j^h(x_i) := \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Make sure to visualize the triangular shape of these $N + 1$ functions.

Next, we define the function spaces

$$P_h(U) := \left\{ \sum_{j=0}^{N} \alpha_j \phi_j^h \;\middle|\; \alpha_j \in \mathbb{R} \quad \forall j \in \{0, \dots, N\} \right\},$$

$$P_h^0(U) := \left\{ \sum_{j=1}^{N-1} \alpha_j \phi_j^h \;\middle|\; \alpha_j \in \mathbb{R} \quad \forall j \in \{1, \dots, N-1\} \right\}$$

for any $h \in \mathbb{R}^+$, which are both sets of all linear combinations of certain hat functions. The difference between the two function spaces is that all functions in $P_h^0(U)$ vanish on both endpoints a and b of the interval domain $U = (a, b)$.

The function space $P_h(U)$ satisfies the required approximation property (10.33) for $H^1(U)$, and $P_h^0(U)$ satisfies it for $H_0^1(U)$. Therefore, we can choose

$$V_h := P_h^0(U)$$

as the discretized, finite-dimensional space of test functions and

$$W_h := P_h^0(U) = V_h$$

as the discretized, finite-dimensional solution space for solving (10.30), which requires that the solutions vanish on the boundary.

Now how can we calculate the finite-element approximation u_h in (10.34)? We denote the basis functions of the M-dimensional space V_h by ψ_j and can hence write the solution as the linear combination

$$u_h = \sum_{j=0}^{N} u_j \psi_j, \tag{10.35}$$

where the $u_j \in \mathbb{R}$ are unknown coefficients for $j \in \{0, \dots, N\}$. In our example, the basis functions are hat functions. Satisfying the weak formulation (10.34) (for all test functions $v_h \in V_h$) is equivalent to satisfying the equation for all M basis functions ψ_i, i.e., we seek solutions $u_h \in W_h$ such that

$$a(u_h, \psi_i) = a\left(\sum_{j=0}^{N} u_j \psi_j, \psi_i \right) = \sum_{j=0}^{N} u_j a(\psi_j, \psi_i) = F(\psi_i) \qquad \forall i \in \{0, \dots, N\}.$$

After defining an $(N + 1) \times (N + 1)$ matrix M by setting

$$m_{ij} := a(\psi_j, \psi_i)$$

and a vector $\mathbf{z}$ by setting $z_i := F(\psi_i)$, this condition becomes the linear system of equations

$$M\mathbf{u} = \mathbf{z} \tag{10.36}$$

for the unknown vector $\mathbf{u} = (u_0, \dots, u_{N+1})$. This linear system of equations is the finite-element discretization we will implement below.

Before we do so, we however pose the question whether the linear system (10.36) has a unique solution. If we can answer this question positively, then our confidence in the whole finite-element procedure will be much increased. It is clear that we must ensure that our original equation (10.30) has a unique solution u to begin with. To do so, we assume that the assumptions of the Lax–Milgram theorem, Theorem 10.5, are satisfied.

The first argument that a unique solution $\mathbf{u}$ of (10.36) exists is an algebraic one.

Theorem 10.13 *Suppose that the assumptions of Theorem 10.5 hold for the boundary-value problem (10.30). Then the matrix M in (10.36) is positive definite.*

Proof The matrix S being positive definite is equivalent to

$$\mathbf{v}^\top M\mathbf{v} > 0 \qquad \forall \mathbf{v} \in \mathbb{R}^{N+1}\backslash\{\mathbf{0}\}$$

(see Definition 8.9). For any vector $\mathbf{v} \in \mathbb{R}^{N+1}$, we define the function

$$v := \sum_{i=0}^{N} v_i \psi_i \in V_h$$

and calculate

$$\mathbf{v}^\top M\mathbf{v} = \sum_{i=0}^{N}\sum_{j=0}^{N} v_i v_j a(\psi_j, \psi_i) = a(v, v) \geq \beta \|v\|_V^2 > 0$$

unless $\mathbf{v} = \mathbf{0}$. The first inequality holds since the bilinear form a is coercive (with constant β) by assumption. □

Since every positive definite matrix is invertible, the linear system (10.36) has a unique solution.

The second argument that a unique solution $\mathbf{u}$ of (10.36) exists is the following. For the elliptic equation (10.30), the solution space and the space of the test functions are identical.

Theorem 10.14 (Céa's lemma) *Define $V := H_0^1(U)$ and suppose that the assumptions of Theorem 10.5 hold for the boundary-value problem (10.30): the bilinear form a is bounded with constant α and coercive with constant β. Then there exists a unique solution $u \in V$ of (10.30) and a unique solution $u_h \in V_h$ of (10.34). Furthermore, the inequalities*

$$\|u_h\|_V \le \frac{1}{\beta}\|F\|_{V'}$$

(stability) and

$$\|u_h - u\|_V \le \frac{\alpha}{\beta} \inf_{v_h \in V_h} \|v_h - u\|_V \tag{10.37}$$

(convergence; Céa's lemma) hold.

The first inequality means that the solutions u_h are stable for varying h, as the norm of every solution u_h is bounded by a constant that does not depend on h. The second inequality is called Céa's lemma and will be important for the convergence of the approximations u_h to the exact solution u in Theorem 10.15.

Proof Since $V_h \subset V$ is a subspace of V, Theorem 10.5 implies that there exists a unique solution $u_h \in V_h$ of (10.34).

Since the bilinear form a stemming from (10.30) is coercive with constant β, we find

$$\beta\|u_h\|_V^2 \le a(u_h, u_h) = F(u_h) \le \|F\|_{V'}\|u_h\|_V,$$

which immediately implies the first inequality.

To show the second inequality, we first subtract (10.32) (for all $v_h \in V_h \subset V$) from (10.34) to find

$$a(u_h - u, v_h) = 0 \qquad \forall v_h \in V_h.$$

Since the bilinear form a is coercive, we can calculate

$$\beta\|u_h - u\|_V^2 \le a(u_h - u, u_h - u) = a(u_h - u, u_h + v_h - u) \qquad \forall v_h \in V_h$$

using this equality. Next, we define $w_h := u_h + v_h$ and note that statements that hold for all $v_h \in V_h$ are equivalent to statements that hold for all $w_h \in V_h$ because $u_h + V_h = V_h$ (and u_h is fixed). Therefore we have

$$\beta\|u_h - u\|_V^2 \le a(u_h - u, w_h - u) \qquad \forall w_h \in V_h.$$

Since the bilinear form a is bounded, we furthermore find

$$\beta\|u_h - u\|_V^2 \le \alpha\|u_h - u\|_V\|w_h - u\|_V \qquad \forall w_h \in V_h,$$

which yields the second inequality. □

Under the reasonable assumption (10.33) that the subspaces V_h become better and better approximations of the function space V, Theorem 10.14 immediately shows that the approximations $u_h \in V_h$ converge to the exact solution $u \in V$ as $h \to 0$. This is expressed by the following theorem.

Theorem 10.15 (convergence) *Suppose $V_h \subset V$ are subspaces of V such that (10.33) holds and suppose that the assumptions of Theorem 10.14 hold. Then*

$$\lim_{h\to 0} \|u_h - u\|_V = 0$$

holds.

Proof Inequality (10.37) and equation (10.33) yield

$$\lim_{h\to 0} \|u_h - u\|_V \le \frac{\alpha}{\beta} \lim_{h\to 0} \inf_{v_h\in V_h} \|v_h - u\|_V = 0$$

showing convergence. □

After these theoretic considerations, we now implement the finite-element discretization (10.36) in a one-dimensional example. We use the domain $U := (a, b)$, the equidistant grid points defined above, and the linear hat functions ϕ_i^h defined above on these finite elements. We assume that the coefficient matrix A in (10.30) is piecewise constant on the N finite elements or intervals with the value $a_i \in \mathbb{R}$, $i \in \{1, \dots, N\}$, on the interval $(a + (i-1)h, a + ih)$.

Sketching the hat functions and calculating the integral in the bilinear form a yields

$$m_{ij} = a(\phi_j^h, \phi_i^h) = \begin{cases} \frac{a_i+a_{i+1}}{h}, & i = j, \\ -\frac{a_{\max(i,j)}}{h}, & |i-j| = 1, \\ 0, & |i-j| > 2 \end{cases} \tag{10.38}$$

whenever $0 < i < N$ and $0 < j < N$. Obviously, the system matrix S is sparse, and thus it is instrumental to use linear solvers that take advantage of its sparsity in realistic applications.

The vector $\mathbf{z}$ on the right side of (10.36) has the elements

$$z_i = F(\phi_i) = \int_a^b f\phi_i \mathrm{d}V.$$

Again assuming that the right-hand side f is constant on the finite elements, we set

$$z_i := \frac{h}{2}(f_i + f_{i+1}).$$

In general, good approximations of this integral are important.

Dirichlet boundary conditions can be implemented by noting that the form (10.35) of the approximation u_h as a linear combination of hat functions yields

$$u_h(a) = u_0,$$
$$u_h(b) = u_N.$$

Regarding the implementation of Neumann boundary conditions (see Problem 10.24), we note that the outward unit normal derivatives of u_h are given by the formulas

$$\mathbf{n} \cdot \nabla u_h(a) = \frac{1}{h}(u_0 - u_1),$$
$$\mathbf{n} \cdot \nabla u_h(b) = \frac{1}{h}(u_N - u_{N-1}),$$

which follow immediately by differentiating (10.35).

The function `elliptic_FE_1D` implements the discretization (10.36). The purpose of the function `ind` is to translate the index i in the discussion above to a linear index of JULIA vectors and arrays, which always start at index one. (Another option is to use the package `OffsetArrays`.)

```
function elliptic_FE_1D(a::Float64, b::Float64, A::Function,
                        f::Function, g::Function,
                        N::Int)::Vector{Float64}
    local h = (b-a) / N
    local z = Vector{Float64}(undef, N+1)
    local M = SparseArrays.spzeros(N+1, N+1)

    function ind(i::Int)::Int
        1 + i
    end

    for i in 0:N
        if i == 0 || i == N
            M[ind(i), ind(i)] = 1.0
            z[ind(i)] = g(a + i*h)
        else
            M[ind(i), ind(i-1)] = - A(a + (i-1/2)*h)
            M[ind(i), ind(i)] = A(a + (i-1/2)*h) + A(a + (i+1/2)*h)
            M[ind(i), ind(i+1)] = - A(a + (i+1/2)*h)

            z[ind(i)] = (h^2/2) * (f(a + (i-1/2)*h)
                                   + f(a + (i+1/2)*h))
        end
    end

    M \ z
end
```

The next function `test_elliptic_FE_1D` calculates the maximum norm of the difference between a known exact solution and its numerical approximation.

```
function test_elliptic_FE_1D(u_exact::Function,
                             a::Float64, b::Float64,
                             A::Function, f::Function,
                             N::Int)::Float64
    local h = (b-a) / N
    local u_ex = Float64[u_exact(a + i*h) for i in 0:N]
    local u_num = elliptic_FE_1D(a, b, A, f, u_exact, N)

    LinearAlgebra.norm(u_num - u_ex, Inf)
end
```

In the following numerical example, we set $U := (-\pi, \pi)$, $u := \cos$, and $A(x) := 2 + \cos(x)$, which results in $f(x) := 2\cos^2(x) + 2\cos(x) - 1$.

```
for i in 0:5
    local N = 10 * 2^i
    local error =
        test_elliptic_FE_1D(cos,
                            -pi, Float64(pi),
                            x -> 2 + cos(x),
                            x -> 2*cos(x)^2 + 2*cos(x) - 1,
                            N)
    Printf.@printf("N = %3d: error = %.5e\n", N, error)
end
```

```
N =  10: error = 2.78004e-02
N =  20: error = 6.24532e-03
N =  40: error = 1.52113e-03
N =  80: error = 3.77824e-04
N = 160: error = 9.43483e-05
N = 320: error = 2.35835e-05
```

Since the error is multiplied by a factor of approximately $1/4$ when h is multiplied by $1/2$, we observe second-order convergence in this example, where the functions A, f, and u are very smooth. Comparing the resulting discretization (10.38) with the finite-difference and finite-volume discretizations immediately shows that this finite-element discretization is of second order.

The finite-element method is deeply rooted in the theory of weak solutions of PDEs. In fact, equations (10.32) and (10.34) make it clear that the numerical approximation stems from posing the problem on discretized, finite-dimensional function spaces instead of on the Sobolev spaces generally used in the modern theory of PDEs, which are unsuitable for numerical approximations. A great advantage of finite elements especially in higher spatial dimensions is that geometrically complex domains as they may occur in realistic problems can be approximated by finite elements very well. Lots of mesh generation software to partition a given geometry into finite elements has been developed.

10.8 Julia Packages

Regarding the basic operations, the package `ApproxFun` can be used for approximating functions. Regarding finite differences, the package `DiffEqOperators` constructs finite-difference operators to discretize PDEs, reducing the equations to systems of ODEs which can be solved using the package `DifferentialEquations`. Regarding finite volumes, the package `VoronoiFVM` can solve coupled nonlinear PDEs. Regarding finite elements, the `JuliaFEM` project contains software and documentation for (nonlinear) equations and distributed calculations. The package `Gridap` provides software for various problem types, including linear, nonlinear, and multi-physics problems and is written in Julia. The package `FEniCS` is a wrapper for the FENICS library for finite-element discretizations.

10.9 Bibliographical Remarks

A standard textbook on the theory of PDEs is [1]. A few books [4, 5, 6, 8, 9] are mentioned here among the multitude of textbooks on their numerical methods.

Problems

10.1 Prove (10.2).

10.2 Prove (10.4).

10.3 Prove that the potential defined in (10.6) satisfies (10.5).

10.4 ∗ Prove (10.8) for the fundamental solution G defined in (10.7).

10.5 Choose a PDE and its derivation and write down the units of each variable and constant in the equation and its derivation. Also check that all equations in the derivation and the PDE itself have consistent units.

10.6 ∗ Prove Theorem 10.1.

10.7 ∗ Prove Theorem 10.6.

10.8 (Mean inequalities)

1. Prove the inequality

$$\sqrt{xy} \le \frac{x+y}{2} \qquad \forall\forall x, y \in \mathbb{R}^+.$$

2. Suppose $\alpha \in \mathbb{R}^+$, $\beta \in \mathbb{R}^+$, and $\alpha\beta = 1/4$. Prove the inequality

$$xy \le \alpha x^2 + \beta y^2 \qquad \forall\forall x, y \in \mathbb{R}.$$

3. $*$ Suppose $n \in \mathbb{N}$ and $x_i \in \mathbb{R}^+$ for all $i \in \{1, \dots, n\}$. Prove the inequality

$$\left(\prod_{i=1}^{n} x_i\right)^{1/n} \le \frac{1}{n}\sum_{i=1}^{n} x_i.$$

This inequality is known as the inequality of the arithmetic and geometric means.

10.9 $*$ Prove Theorem 10.9.

10.10 $*$ Prove Theorem 10.10.

10.11 Write a function to plot the solution in Sect. 10.5.1.

10.12 Change the function `elliptic_FD_1D` in Sect. 10.5.1 to assemble the system matrix using `sparse` (see Sect. 8.2). Is the new version faster? By how much? Why?

10.13 Show that (10.16) follows from (10.21) by expanding the central-difference operators.

10.14 Implement the fourth-order compact finite-difference discretization in Theorem 10.11 and use an example to validate that it has order four.

10.15 Show that (10.22) follows from (10.24) by expanding the central-difference operators.

10.16 Implement the fourth-order compact finite-difference discretization in Theorem 10.12 and use an example to validate that it has order four.

10.17 Prove (10.28).

10.18 Implement Simpson's rule to approximate $\iint_{V_{i,j}} f \mathrm{d}V$ (instead of $h^2 f_{i,j}$). Compare the error using $h^2 f_{i,j}$ and Simpson's rule in a (well-chosen) example.

10.19 Implement zero and general Neumann boundary conditions for the finite-volume discretization in Sect. 10.6 on one edge of a square domain.

10.20 Derive, implement, and test a three-dimensional version of the two-dimensional finite-volume discretization in Sect. 10.6.

10.21 (Green's first identity) $*$ Suppose $U \subset \mathbb{R}^d$ is a domain, $u : U \to \mathbb{R}$ a twice continuously differentiable function, $v : U \to \mathbb{R}$ a once continuously differentiable function, and $A : U \to \mathbb{R}^{d\times d}$ a once continuously differentiable matrix-valued function. Prove that the identity

$$\iiint_U \nabla\cdot(A\nabla u)v\mathrm{d}V = \oiint_{\partial U} v\mathbf{n}\cdot(A\nabla u)\mathrm{d}S - \iiint_U (A\nabla u)\cdot\nabla v\mathrm{d}V$$

holds by applying the divergence theorem

$$\iiint_U \nabla\cdot\mathbf{F}\mathrm{d}V = \oiint_{\partial U}\mathbf{n}\cdot\mathbf{F}\mathrm{d}S$$

to the vector field $\mathbf{F} := vA\nabla u$.

10.22 ∗ Prove Theorem 10.14.

10.23 Prove (10.38).

10.24 Generalize the function `elliptic_FE_1D` by implement Neumann boundary conditions.

10.25 Implement a problem with mixed Dirichlet/Neumann boundary conditions and validate the implementation using a test problem.

10.26 (Finite elements in two dimensions) The one-dimensional finite-element discretization is generalized to two dimensions in the following steps.

1. Partition a general rectangular domain into rectangles and subdivide the rectangles into two right-angled triangles. Use these triangles as the finite elements and derive a finite-element discretization of the elliptic problem (10.30) analogously to the one-dimensional case.
2. Implement the discretization and validate it using a test problem.
3. Implement mixed Dirichlet/Neumann boundary conditions and validate the implementation using a test problem.

References

1. Evans, L.: *Partial Differential Equations*, 1st edn. American Mathematical Society (1998)
2. Gilbarg, D., Trudinger, N.: Elliptic Partial Differential Equations of Second Order. Springer-Verlag (2001)
3. Lax, P., Milgram, A.: Parabolic equations. *Ann. Math. Stud.* **33**, 167–190 (1954)
4. LeVeque, R.: *Numerical Methods for Conservation Laws*, 2nd edn. Birkhäuser, Basel (1992)
5. LeVeque, R.: *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press (2002)
6. LeVeque, R.: *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-dependent Problems.* Society for Industrial and Applied Mathematics (SIAM) (2007)
7. Renardy, M., Rogers, R.: *An introduction to partial differential equations*, 2nd edn. Springer-Verlag, New York, NY (2004)
8. Strang, G., Fix, G.: *An Analysis of the Finite Element Method*, 2nd edn. Wellesley-Cambridge Press (2008)
9. Toro, E.: *Riemann Solvers and Numerical Methods for Fluid Dynamics*, 3rd edn. Springer (2009)

Part III
Algorithms for Optimization

Chapter 11
Global Optimization

But… TANSTAAFL.
"There ain't no such thing as a free lunch," in Bombay or in Luna.

—Robert A. Heinlein, *The Moon is a Harsh Mistress* (1966)

Abstract The optimization of functions is a topic of both great theoretical and practical importance. Optimization problems occur in many different contexts, where a common setting is that a model of the quantity of interest is to be optimized with respect to its parameters. In this chapter, we present important classes of algorithms for global optimization, i.e., for finding all global minima or maxima of a given function on a given domain disregarding any local optima. The methods described here are simulated annealing, particle-swarm optimization, and genetic algorithms. A list of benchmark problems of varying difficulty is also included, inviting the reader to experiment with the optimization algorithms and their parameters. Local optimization methods, usually based on the gradient of the function, are discussed in the next chapter.

11.1 Introduction

Global optimization is a large field with many applications, and many deterministic and stochastic optimization methods have been developed. Deterministic methods include branch-and-bound methods, cutting-plane methods, inner-and-outer approximation, and interval methods. Stochastic methods, on the other hand, are methods whose results depend on random numbers drawn while running the algorithm; the function to be optimized is still deterministic. Stochastic methods include direct Monte Carlo sampling, stochastic tunneling, and parallel tempering.

Heuristic methods are strategies for searching the domain in a – hopefully – intelligent manner. They include differential evolution, evolutionary computa-

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_11

tion (including, e.g., evolution strategies, genetic algorithms, and genetic programming), graduated optimization, simulated annealing, swarm based optimization (including, e.g., particle-swarm optimization and ant-colony optimization), and taboo search.

Other methods are Bayesian optimization and memetic algorithms. In the next chapter, Chap. 12, local optimization methods are discussed. They are usually based on the gradient of the function to be optimized or on its second derivatives. Global and local optimization methods can be combined by performing global optimization before or while improving promising candidate points by local optimization. This synergy between evolutionary and local optimization is called memetic algorithms.

Two branches of optimization are continuous and discrete optimization. In discrete optimization, some or all of the variables of the objective function are discrete, i.e., they assume only values from a finite set of values. Two important fields of discrete optimization are combinatorial optimization and integer programming.

Furthermore, optimization problems can be categorized with respect to the presence of constraints on the independent variables of the objective function. The constraints can be hard constraints, which must be satisfied by the independent variables, or soft constraints, where some values are penalized if the constraints are not satisfied and based on the extent that they do not.

The choice of optimization method is also influenced by the computational cost of evaluating the objective function. If the objective function is given as an expression (see, e.g., the benchmark problems in Sect. 11.8), it is usually possible to perform many function evaluations thus rendering the optimization problem more tractable. On the other hand, if the objective function is computationally expensive such as a function of the solution of a partial differential equation, the choice of optimization method is more limited and also more important. A closely related question is whether first- or second-order derivatives of the objective function are available and what their computational cost is. The case when no derivatives are available is called derivative free optimization.

The long, but not exhaustive, lists of optimization methods above pose the question which one to use when presented with an optimization problem. Unfortunately, there is no general answer to this question, as there is no best optimization algorithm. It will always be the case that for any given class of optimization problems, there is a best class of algorithms, and – vice versa – for any given class of algorithms, there is a class of optimization problems for which the algorithms work best. This notion is formalized in no-free-lunch theorems. Therefore, we start by examining in the next section the question what can generally be said about the relationship between classes of optimization problems and classes of optimization algorithms.

11.2 No Free Lunch

No-free-lunch (NFL) theorems are statements in optimization and computational complexity that imply that, for certain types of problems, the computational cost of finding a solution when averaged over all problems in a class is the same for any solution method.

In other words, there is no optimization method that outperforms all other methods on all problems. Any improved performance over one class of problems is always offset by decreased performance over another class of problems. Specialized optimization methods have the best performance for solving a certain class of problems. There may still be algorithms yielding good results on a variety of classes of problems, but they are still outperformed by specialized methods in each of them.

These notions have been formalized in the NFL theorems [17] for search and optimization problems. More precisely, the connection between algorithms and their cost functions is analyzed. The finite search set is denoted by X and the finite set of all possible values of the objective function is denoted by Y. (The assumption that both X and Y are finite is met when the elements are represented by floating-point numbers.) An optimization problem is represented by its objective function $f : X \to Y$, and the set of all optimization problems or objective functions is denoted by $F := Y^X$.

The points in $X \times Y$ visited or evaluated while running an optimization algorithm form a sample of size m, i.e., they are a (time ordered) set of m distinct visited points and are denoted by

$$d_m := \{(d_m^x(1), d_m^y(1)), \dots, (d_m^x(m), d_m^y(m))\}.$$

The element $d_m^x(i) \in X$ denotes the i-th element in a sample of size m and $d_m^y(i) := f(d_m^x(i)) \in Y$ is the corresponding value of the objective function. The ordered set of values of the objective function is denoted by $d_m^y := \{d_m^y(1), \dots, d_m^y(m)\}$.

The set of all samples of size m is denoted by

$$D_m := (X \times Y)^m$$

and the set of all samples (of arbitrary size) by

$$D := \bigcup_{m \ge 0} D_m.$$

An optimization algorithm a is represented by a function mapping a previously visited set of points to a new, previously unseen or unevaluated point in X, i.e.,

$$a : D \to X, \quad d \mapsto x \notin d^x.$$

These optimization algorithms are deterministic, as every sample is mapped to a new point deterministically. The results below for deterministic algorithms can be extended to stochastic algorithms [17].

We also define performance measures $\Phi(d_m^y)$ of algorithms after m iterations. The performance measures are functions of the sample d_m^y. For example, when minimizing the objective function f, a self-evident performance measure is $\Phi(d_m^y) := \min_{i\in\{1,\dots,m\}} d_m^y(i)$.

NFL theorems are formulated within the framework of probability theory. The conditional probability $P(d_m^y|f,m,a)$ is the probability of obtaining the sample d_m^y after iterating an algorithm a m times on an objective function f. Knowing this conditional probability, performance measures $\Phi(d_m^y)$ can be calculated easily.

The first NFL theorem discussed here addresses the question how the set $F_1 \subset F$ of problems for which an algorithm a_1 performs better than an algorithm a_2 compares to the set $F_2 \subset F$ of problems for which the converse is true. This is done by summing the conditional probabilities $P(d_m^y|f,m,a)$ over all objective functions f and comparing the sums obtained for $a = a_1$ and $a = a_2$. A major result of [17] is that the sum

$$\sum_{f\in F} P(d_m^y|f,m,a),$$

i.e., the conditional probability $P(d_m^y|f,m,a)$ summed over all objective functions f, is independent of the algorithm a.

Theorem 11.1 (no free lunch) *For any pair of algorithms a_1 and a_2, the equation*

$$\sum_{f\in F} P(d_m^y|f,m,a_1) = \sum_{f\in F} P(d_m^y|f,m,a_2).$$

holds under the above assumptions.

A proof can be found in [17, Appendix A]. This theorem means that the sum of such probabilities over all possible optimization problems f is identical for all optimization algorithms. In other words, the average of $P(d_m^y|f,m,a_1)$ over all problems f is independent of the algorithm a. This implies that any performance gain that a certain algorithm provides on one class of problems must be offset by a performance loss on the remaining problems.

These deliberations can be extended to time-dependent objective functions. The initial cost function is called f_1 and is used to sample the first X value. Before the next iteration i of the optimization algorithm, the cost function is transformed to a new cost function by $T: F\times\mathbb{N} \to F$, i.e., $f_{i+1} := T(f_i,i)$. It is assumed that all the transformations $T(.,i)$ are bijections on F. This assumption is important, because otherwise a bias in a region of cost functions could be introduced and exploited by some optimization algorithms.

Analogously to Theorem 11.1, the following theorem shows that the average performance of any two algorithms is the same also in the case of time-dependent objective functions. But now we average over all possible time dependencies of cost functions, meaning the average is calculated over all transformations T rather than over all objective functions f. These averages are given by

$$\sum_T P(d_m^y | f_1, T, m, a),$$

where f_1 is the initial objective function. The samples are redefined to drop their first elements such that the transformations T can take full effect, although the initial object function f_1 is fixed. Then the following result can be shown.

Theorem 11.2 (no free lunch for time-dependent problems) *For all d_m^y, D_m^y, $m > 1$, algorithms a_1 and a_2, and initial cost functions f_1, the equation*

$$\sum_T P(d_m^y | f_1, T, m, a_1) = \sum_T P(d_m^y | f_1, T, m, a_2)$$

holds.

A proof can be found in [17, Appendix B]. The interpretation is analogous to the one of Theorem 11.1. The average performances of any two algorithms a_1 and a_2 over all cost-function dynamics are identical, meaning that any performance gain over a class of problems must be offset by a performance loss over the rest of the problems.

These NFL results tell us that unfortunately there is no ingenious optimization algorithm that works better than all other algorithms on all problems. On the other hand, the NFL theorems motivate specialized considerations and work focused on well-defined classes of problems, for which specialized algorithms that outperform generic ones exist.

Having discussed the theoretic limitations of optimization algorithms, modern methods for global optimization will be presented in the rest of this chapter.

11.3 Simulated Annealing

Simulated annealing is a very practical optimization method. We start by discussing its roots.

11.3.1 The Metropolis Monte Carlo Algorithm

In the middle of the last century, Monte Carlo algorithms for calculating properties of materials consisting of interacting individual particles were developed [11]. These Monte Carlo integrations over the configuration spaces of the particles turned out to be highly useful in statistical mechanics.

In simple terms, the canonical ensemble is used to calculate the equilibrium value $\bar{F}$ of any quantity F of interest as

$$\bar{F} = \frac{\int F \mathrm{e}^{-E/kT} \mathrm{d}^{3N}\mathbf{p}\mathrm{d}^{3N}\mathbf{q}}{\int \mathrm{e}^{-E/kT} \mathrm{d}^{3N}\mathbf{p}\mathrm{d}^{3N}\mathbf{q}}. \tag{11.1}$$

There are N particles so that the phase space is $6N$-dimensional. The factor

$$\mathrm{e}^{-E/kT}$$

stems from the Boltzmann probability distribution of the particles, where E is the energy of the state and the constant kT is the product of the Boltzmann constant k and the thermodynamic temperature T. The probability of a state with energy E is proportional to $\mathrm{e}^{-E/kT}$.

Since the forces between the particles are independent of velocity, the momentum integrals ($\mathrm{d}^{3N}\mathbf{p}$) cancel and only the integrals ($\mathrm{d}^{3N}\mathbf{q}$) over the $3N$-dimensional configuration space must be computed. The method of choice to do so is the Monte Carlo method, meaning that random samples of particles are averaged.

The naive approach to perform these computations is to generate configurations with N particles each at random positions, each corresponding to a random point in the $3N$-dimensional configuration space, then to calculate the energy E of each configuration (depending on the forces considered), and finally to use the weight $\mathrm{e}^{-E/kT}$ of each configuration in (11.1).

However, this naive approach is not practical when the particles are close-packed, since the probability to choose a configuration with very small $\mathrm{e}^{-E/kT}$ (and large E) is high. Therefore, in [11], a modification of the Monte Carlo method was introduced. Instead of choosing random configurations and weighing them with the factor $\mathrm{e}^{-E/kT}$, configurations are chosen with probability $\mathrm{e}^{-E/kT}$ and weighed evenly.

This alternative sampling can be achieved by starting from any configuration and then moving each particle in succession by adding random displacements chosen according to a uniform distribution on an interval $[-\alpha, \alpha]$. Next, the energy change ΔE before and after the move is calculated. If $\Delta E \leq 0$, i.e., the new configuration has lower energy, the move is allowed and the particle assumes its new position. On the other hand, if $\Delta E > 0$, the move is allowed and performed only with probability $\mathrm{e}^{-\Delta E/kT}$. This can be achieved by drawing a random number ρ uniformly from the interval $[0, 1]$ and moving the particle to its new position if $\rho \leq \mathrm{e}^{-\Delta E/kT}$. Otherwise, it remains at its old position.

The new configuration is considered different from the old one for the purpose of calculating the average in any case, irrespective of whether the move was allowed (and hence performed) or not. In each iteration, a new value F_i of the quantity of interest is obtained. Finally, after M iterations, the equilibrium value of the quantity of interest is calculated as

$$\bar{F} = \frac{1}{M}\sum_{i=1}^{M} F_i.$$

It can be shown that this algorithm does indeed choose configurations with probabilities $e^{-E/kT}$ [11] and therefore indeed calculates the equilibrium value of the quantity of interest.

Another question concerns the convergence speed of this algorithm. Here we can already observe a major effect that is common to many Monte Carlo procedures that construct sequences of states. The maximum displacement α is of great importance and should be chosen with care or – even better – should be adjusted automatically. If it is too small, the configuration changes only little and sampling the whole space requires many iterations. On the other hand, if it is too large, most moves are forbidden. In both cases, it takes longer to arrive at the equilibrium than with a suitable maximum displacement.

The reason why we have discussed the classical Metropolis algorithm in detail is that it is not only foundational in the Monte Carlo, but also because it motivates the optimization algorithm that is the subject of this section.

11.3.2 The Simulated-Annealing Algorithm

Simulated annealing was introduced in [9] and copies the way of sampling the whole space in the Metropolis algorithm and applies it to global optimization. Simulated annealing can be applied to both continuous and discrete optimization and search problems. It only requires two simple operations, namely choosing a random starting point and an unary operation that maps points to neighboring points.

In order to formulate the simulated-annealing algorithm, the sampling method of the Metropolis algorithm is applied to a single point or particle. The energy E of a configuration in the Metropolis algorithm now corresponds to the objective function in the minimization problem, and therefore we denote it by f in the algorithm. This analogy yields the simulated-annealing algorithm.

Algorithm 11.3 (simulated annealing for minimization)

1. Choose a random starting point x_1 and let $t := 1$.
2. Denote the current point by x_t and generate a candidate point $\tilde{x}_{t+1}$ by adding a random displacement, e.g., a normally distributed random variable with zero mean such that
$$\tilde{x}_{t+1} := x_t + N(0, \sigma).$$
3. Calculate the difference in the objective function by letting
$$\Delta f := f(\tilde{x}_{t+1}) - f(x_t).$$

4. Calculate the acceptance probability
$$p := \begin{cases} e^{-\Delta f/kT}, & \Delta f > 0, \\ 1, & \Delta f \leq 0. \end{cases}$$
(Note that for implementation purposes simply setting $p := e^{-\Delta f/kT}$ suffices and yields the same x_{t+1} in Step 6.)
5. Draw a uniformly distributed random number $\rho \sim U(0,1)$ (such that $\rho \in [0,1]$).
6. If $\rho \leq p$, accept the candidate point $\tilde{x}_{t+1}$, otherwise the point remains unchanged, i.e., the next point is
$$x_{t+1} := \begin{cases} \tilde{x}_{t+1}, & \rho \leq p, \\ x_t, & \rho > p. \end{cases}$$
7. Set $t := t+1$ and go to Step 2 until a termination criterion has been reached.
8. Return the best of the points x_t found.

In summary, candidate points that yield a better solution of the minimization problem are always accepted (the case $\Delta f \leq 0$), while uphill steps are still performed with probability $e^{-\Delta f/kT}$ (the case $\Delta f > 0$). This probability depends on the difference Δf in the objective function as well as the factor kT. If the temperature T is high, the acceptance probability $e^{-\Delta f/kT}$ is close to one, and therefore uphill steps are likely to be accepted. Conversely, if the temperature is low, the acceptance probability is close to zero, and uphill steps are unlikely to be accepted.

11.3.3 Cooling Strategies

Therefore the temperature T provides a way to adjust the behavior of the algorithm. It is customary to start with a higher temperature and reduce it during the iterations. In the beginning, while the temperature is high, simulated annealing resembles a global random search. As the temperature falls, the probability of accepting an uphill step decreases. Then points become more and more unlikely to escape regions with local minima and hopefully converge to a global minimum as the temperatures goes to zero. Good cooling strategies can thus combine global search with local refinement.

This consideration shows that reasonable cooling strategies have the following three properties. From now on, we denote the temperature in iteration t by T_t.

1. The initial temperature T_1 is greater than zero.
2. The temperature decreases, i.e., $T_{t+1} \leq T_t$.

3. The temperature approaches zero as the number of iterations increases, i.e., $\lim_{t\to\infty} T_t = 0$. If the search space is finite, the temperature is equal to zero after a certain (large) number of iterations.

Many variants of cooling strategies are possible and have been investigated. Here three cooling strategies are presented and invite the reader to experiment with cooling strategies and the benchmark problems in Sect. 11.8 (see Problem 11.7).

1. Multiply the temperature T by a factor $1 - \epsilon$ every s iterations, where $0 < \epsilon \ll 1$. Suitable values for ϵ and s are found by experimentation and depend on the minimization problem.
2. Using an iteration budget of N iterations in total, reduce the temperature to the new value
$$T_t := \left(1 - \frac{t}{N}\right)^{\beta} T_1$$
every s iterations. Again, good values for the parameters N, β, and s are found by experimentation.
3. Every s iterations, set the temperature to the new value
$$T_t := \begin{cases} \gamma(f(x_t) - f(x_t^*)), & T_t \neq 0, \\ \delta T_{t-1}, & T_t = 0, \end{cases}$$
where x_t^* is the best point found up to iteration t and $\delta \in (0,1)$. You have guessed it, there is no general theory to find the parameters s, γ, and δ. In this cooling strategy, our rule that the cooling strategy should be decreasing may be violated.

It can be shown that simulated annealing algorithms with appropriate cooling strategies converge to the global optimum as the number of iterations goes to infinity [10, 4, 13]. The domain can be searched faster if cooling is sped up, but then convergence is not guaranteed anymore.

11.4 Particle-Swarm Optimization

Particle-swarm optimization [2, 8] can be viewed as an extension of simulated annealing. Instead of using a single particle, a swarm of particles is moved from iteration to iteration. The movement of the members of the swarm may be considered to mimic the "swarm intelligence" of social systems such as flocks of birds or schools of fish. The members of the swarm spread out and move randomly hoping to find a local optimum. As the swarm is social, the members announce their success to their neighbors in order to attract them to promising regions and to help in the search for an optimum.

Each particle is an element of an n-dimensional search space $X \subset \mathbb{R}^n$. Each particle k has a position $\mathbf{x}_{k,t} \in X$ and a velocity $\mathbf{v}_{k,t} \in \mathbb{R}^n$ in each iteration t. If the speed of a particle is high, its search is explorative, and if it is low, the search is exploitive. We briefly mention that it is possible to use genotype-phenotype mapping, which will be discussed in the next section.

The positions and velocities of all particles are initialized randomly. In each iteration, the velocity is updated first and then the position. Each particle k also keeps track of its history by remembering the best position $\mathbf{b}_{k,t}$ it is has seen up to iteration t.

The social component of the swarm is realized by communication between the particles. To that end, each particle updates in each iteration t its set $N_{k,t}$ of neighbors. The set $N_{k,t}$ of neighbors is usually defined as the set of all particles within a certain distance from $\mathbf{x}_k$ measured by a certain metric such as the Euclidean distance in the simplest case. Knowing its neighbors, each particle then communicates its best position $\mathbf{b}_{k,t}$ found so far to its neighbors in each iteration. Therefore each particle knows the best point $\mathbf{n}_{k,t}$ found in its neighborhood so far. Furthermore, the swarm records the best position $\mathbf{b}_t$ found by all particles in the swarm up to iteration t.

These best points that the particles communicate among themselves constitute the social component of particle-swarm optimization and enter the calculations in the updates of the velocities of the particles. When updating the velocity of a particle, we can use the best point $\mathbf{n}_{k,t}$ found in its neighborhood so far or the best point $\mathbf{b}_t$ found by all particles so far.

These two possibilities lead to local and global updates, respectively. In a local update, the velocity $\mathbf{v}_{k,t}$ of particle k is updated by

$$\mathbf{v}_{k,t+1} := \mathbf{v}_{k,t} + (\mathbf{b}_{k,t} - \mathbf{x}_{k,t})U(0,\mathbf{c}) + (\underbrace{\mathbf{n}_{k,t}}_{\text{best neighbor}} - \mathbf{x}_{k,t})U(0,\mathbf{d}). \tag{11.2}$$

In a global update, the velocity becomes

$$\mathbf{v}_{k,t+1} := \mathbf{v}_{k,t} + (\mathbf{b}_{k,t} - \mathbf{x}_{k,t})U(0,\mathbf{c}) + (\underbrace{\mathbf{b}_{t}}_{\text{best in population}} - \mathbf{x}_{k,t})U(0,\mathbf{e}). \tag{11.3}$$

For each particle, either a local or a global velocity update is chosen randomly. Here $U(0,\mathbf{c})$, $U(0,\mathbf{d})$, and $U(0,\mathbf{e})$ are random vectors whose entries are uniformly distributed random variables in the intervals $[0,c_i]$, $[0,d_i]$, and $[0,e_i]$, respectively. In other words, the vectors $\mathbf{c}$, $\mathbf{d}$, and $\mathbf{e}$ are parameters of the algorithm.

Having updated the velocities of all particles, their positions $\mathbf{x}_{k,t}$ are updated by

$$\mathbf{x}_{k,t+1} := \mathbf{x}_{k,t} + \mathbf{v}_{k,t+1} \tag{11.4}$$

using the new velocities $\mathbf{v}_{k,t+1}$. Usually, the size of the search space X is finite, and then it is necessary to ensure that particles do not move out of X due to this update.

The updates mean that two steps are added to the previous velocity. In both the local and the global updates, the term $(\mathbf{b}_{k,t} - \mathbf{x}_{k,t})U(0,\mathbf{c})$ points the particle towards its best position so far. In the local update, the term $(\mathbf{n}_{k,t} - \mathbf{x}_{k,t})U(0, \mathbf{d})$ points the particle towards its best neighbor found so far, and in the global update, the term $(\mathbf{b}_t - \mathbf{x}_{k,t})U(0, \mathbf{e})$ ensures that it points towards the best point found by the whole swarm so far.

The learning-rate vectors **c**, **d**, and **e** strongly influence convergence speed. The components of these three vectors determine how vigorously the particles move towards their best positions, their best neighbors, and the best position in the whole swarm so far. The components can of course be different for all directions in the search space.

If the components of **e** are large and the update hence relies much on the best position $\mathbf{b}_t$ in the whole swarm, the algorithm converges faster, but is less likely to find a global minimum. If the components of **d** are large and the update hence relies much on the best neighbor $\mathbf{b}_{k,t}$, convergence is slower, but a global minimum is more likely to be found.

Having discussed how the particles are updated, we can summarize particle-swarm optimization as follows.

Algorithm 11.4 (particle-swarm optimization)

1. Choose a swarm size and initialize the particles with random positions and velocities.
2. Set the iteration counter $i := 1$.
3. Loop over all particles, update their velocities choosing either the local update (11.2) or the global update (11.3) and then update their positions using (11.4).
4. While the termination criterion has not been met, increase i by 1 and go to Step 2.
5. Finally, return the best position found.

In practice, it is useful to return the history of the best points $\mathbf{b}_t$ so that the progress of the algorithm can be plotted. It is also useful to return the whole last swarm as well so that the algorithm can be restarted (without random initialization) whenever the results are not satisfactory and there is still progress. Since the global optimum or optima are unknown unless a test problem is considered, practical termination criteria judge the progress in the past iterations. When it has become very slow, the algorithm is stopped; but of course, long periods of stagnation may be deceptive.

11.5 Genetic Algorithms

Evolutionary computation is an umbrella term for genetic algorithms, genetic programming, and evolution strategies. In this section, genetic algorithms are discussed. Genetic algorithms share with particle-swarm optimization the idea that there is a population of points, particles, or individuals that is updated from iteration to iteration such that the individuals perform a global optimization.

Genetic algorithms follow biological evolution [1] by implementing natural selection. We start by discussing the basic algorithm, of which many variations have been developed. The concepts, the data structures, and the operations that appear in the algorithm are discussed afterwards. A more recent development closes this section.

11.5.1 The Algorithm

All algorithms in evolutionary computation and all genetic algorithms can be summarized as follows.

Algorithm 11.5 (genetic algorithm)

1. Generate a population of individuals with random genomes.
2. Map the genome of each individual to its phenotype, and evaluate the objective function for the phenotype of each individual. (Evaluating the objective function may be computationally expensive, but the evaluations can be parallelized easily.)
3. Map the value of the objective function for each individual to the fitness of each individual.
4. Select individuals for reproduction such that individuals with higher fitness have a higher probability of reproduction.
5. In reproduction, create offspring by varying or combining the genotypes of the individuals to create new individuals, which are then added to the population.
6. Go to Step 2 unless the termination criterion has been met.

While formulating the algorithm, we have introduced a number of concepts that must be made more precise. What are genotypes and phenotypes in the context of optimization problems? What is fitness and how are values of the objective function translated to fitness values? How are individuals selected for reproduction, and how do the reproduction operations work? These building blocks of the algorithm are discussed next. Many variants of these operations are obviously possible, which brings us back to Sect. 11.2.

11.5.2 Genotypes and Phenotypes

The search space on which the reproduction operations act is the genome, and the elements of the genome are the genotypes. A genotype is the collection of the genes of an individual. In biology, the concept of a gene has changed as new discoveries, for example about gene regulation, have been made. In genetic algorithms, we are free to choose the genes and genotypes. Beneficial choices of course help in solving the optimization problem.

Genotypes are often vectors of genes in contrast to other, nonlinear data structures. In this leading case of vectors, the genotypes are usually referred to as chromosomes. Chromosomes can either be vectors of fixed length or of variable length.

In the case of chromosomes with a fixed-length vector $\mathbf{a}$, the locus of each gene is always the same element a_i of the vector, implying that the competing alleles of a gene are always positioned at the same element number i of the vector. The genes may have different data types so that the elements of the vector may have different types.

In the case of chromosomes with a variable-length vector, the positions of the genes may have been shifted after reproduction operations are applied. In this case, the genes often have the same data type.

Leading examples of chromosomes are vectors that consist of bits, of integers, or of real numbers.

The phenotype of an individual in the context of an optimization problem is an element $x \in X$ in the preimage X of the objective function

$$f : X \to \mathbb{R}.$$

The relationship between genotypes and phenotypes is given by the genotype-phenotype mapping

$$g : G \to X$$

from the set G of all genotypes to the preimage X of the objective function f. This means that from the viewpoint of a genetic algorithm the composition

$$g \circ f : G \to \mathbb{R}$$

is optimized.

Therefore the significance of the genotype-phenotype mapping is that its choice should facilitate the reproduction operations and optimization by supporting advantageous genotypes. A good choice of genotype is, of course, highly problem dependent.

To clarify these concepts, we mention the canonical choice of genotype for objective functions $f : \mathbb{R}^d \to \mathbb{R}$. The canonical genome is $G := \mathbb{R}^d$ such that any vector $\mathbf{g} \in \mathbb{R}^d$ is a genotype. Each element g_i of $\mathbf{g}$ is a gene and has locus i. Because of this linear arrangement of the genes, the vector $\mathbf{g}$ is a fixed-length chromosome. The canonical choice for the genotype-phenotype mapping g is the identity $g := \text{id}$, and therefore $g \circ f : \mathbb{R}^d \to \mathbb{R}$.

11.5.3 Fitness

In the next step of the algorithm, the objective function is evaluated for the phenotypes of all individuals. Based on these values, each individual is assigned a fitness value. Then individuals with higher fitness are more likely to be selected for reproduction.

It is obvious that the fitness of an individual should reflect how well it solves the optimization problem. But it should also reflect the variety of the population and incorporate information on population density and niches. By doing so, the probability of finding the global optima can be increased significantly. Fitness assignment is therefore in general a function that acts on the whole population.

The simplest way of assigning fitness is, however, to just use the value $f(x)$ of the objective function or – in multiobjective optimization – the weighted sum

$$\sum_i w_i f_i(x),$$

where the functions f_i are the objectives and the constants w_i are weights.

Another option is Pareto ranking, which also works for multiobjective optimization. To explain it, we first consider the following fitness assignment. If an individual *prevails m* other individuals, it is assigned the fitness value $1/(1+m)$. Its disadvantage especially in multiobjective optimization is, however, that individuals in a crowded part of the search space have the chance to prevail many others, while individuals in a less explored part of the space are assigned a much worse fitness value only because there are fewer neighbors, although they score best with respect to the objective function.

An alternative is to assign to each individual the fitness value n, where n is the number of other individuals it is *prevailed by.* In multiobjective optimization, this choice recognizes individuals on the Pareto frontier and avoids the disadvantages of the fitness choice $1/(1+m)$ from the previous paragraph. The Pareto fitness value or Pareto rank of an individual i is found by looping over all other individuals j. If individual i is prevailed by individual j, the Pareto rank of i is increased by one.

Still, Pareto ranking does not exploit any information about population density or variety. In global optimization, crowding of the individuals is undesirable and variety is beneficial to explore the whole space. Sharing is a method to include variety information into fitness assignment, and variety preserving ranking is another method worth mentioning here.

11.5.4 Selection

After having assigned fitness values, a sufficient number of individuals is chosen from the whole population in the selection step according to their fitness values and placed in the mating pool. In the following step, reproduction operations are applied to the mating pool. Selection may be deterministic or stochastic. Furthermore, it may proceed with replacement or without replacement depending whether an individual may be placed in the mating pool multiple times (with replacement) or at most once (without replacement).

Elitism means that the best n individuals, where $n \geq 1$, get a free pass and are placed in the mating pool in each generation.

The simplest selection method is truncation selection. The individuals are ordered by their fitness and the desired number of the best individuals is placed in the mating pool. When truncation selection is used, care should be taken to combine it with an fitness assignment that ensures variety in order to prevent premature convergence.

Another classical selection method is fitness proportionate selection. The probability of the individual i being placed in the mating pool is proportional to its fitness, i.e., the probability is given by

$$\frac{v_i}{\sum_j v_j},$$

where v_k is the fitness of individual k.

Tournament selection is one of the most popular and effective selection methods. In tournament selection, k individuals are selected from the population at random and compared with each other in a tournament. The winner of the tournament is placed in the mating pool. Tournament selection can be used with and without replacement.

We can estimate the probability of an individual being chosen for the mating pool in deterministic tournament selection with replacement and with tournament size two. In each tournament, the individuals are chosen randomly according to a uniform distribution. If the size of the mating pool is about the size of the population, each individual will participate in about two tournaments on average. The individual with the highest fitness will win all tournaments and will be placed twice in the mating pool; the individual with the median fitness will be placed in the mating pool once; and the individual with the worst fitness can only win against itself, but it is unlikely that it is chosen twice for a tournament. This means that the number of copies of an individual in the mating pool is a decreasing function of fitness, being two for the best fitness, one for the median fitness, and close to zero for the worst fitness.

In stochastic tournament selection, the best individual in the tournament is selected with a probability p and the i-th best individual with probability $p(1-p)^i$.

11.5.5 Reproduction

In the final step of the algorithm, the individuals in the mating pool are reproduced and their offspring is placed in the next generation of the population. We present four reproduction operations here.

First, in creation, a new genotype is created with random genes. Creation is usually applied only when generating the initial population. Creation is a nullary operation.

Second, in duplication, an exact copy of a genotype is created. It is mostly useful to increase the number of individuals of a certain type in a population.

Third, mutation introduces small random changes and is important for preserving variety. In fixed-length chromosomes, one or more of the genes are randomly changed to another allele. If the type of the gene is a bit, it is toggled. If the gene is a real number, a random value from a normal distribution can be added. Duplication and mutation are unary operations.

Mutation in variable-length chromosomes can mean two more operations, namely insertion and deletion, in addition to changing a gene. Insertion means that random genes are inserted, and deletion means that some of the genes are deleted.

Fourth, crossover is a binary operation. In single-point crossover, the two parental chromosomes **a** and **b** are split at a random crossover point, called k here, and the offspring

$$(a_1, a_2, \ldots, a_{k-1}, a_k, b_{k+1}, b_{k+2}, \ldots, b_{d-1}, b_d)$$

consists of the first part of the first parent and second part of the second parent. More generally, in multipoint crossover, several crossover points are chosen randomly and the offspring consists of the subsequences taken alternately from the two parental chromosomes.

Crossover in variable-length chromosomes is analogous, but the loci where the chromosomes are split are not necessarily the same anymore, and the offspring generally has a different length than the parents.

This discussion of reproduction completes the discussion of the operations that occur in a genetic algorithm.

Many variations of the operations in a genetic algorithm have been devised. If the genotypes or the phenotypes are elements of unusual spaces, it may be a challenge to adapt these ideas such that the algorithm works well, i.e., that it converges while still performing a global search.

11.6 Ablation Studies

A profound way to assess the performance of variants of an algorithm are ablation studies. In an ablation study, we define a basic algorithm and a certain number of variations of the basic algorithm, usually more complicated versions of parts of the basic algorithm expected to yield improvements. The basic algorithm is then tested on a set of benchmark problems, as well as the most sophisticated algorithm with all variants enabled, which would be expected to perform best. Furthermore, the performance of the basic algorithm with single variations enabled and the performance of the most sophisticated algorithm with single variations disabled are assessed. Of course, more algorithms with some variations enabled and some variations disabled can be assessed additionally. Finally, the performances on the benchmark problems are compared, which usually gives a good overview over the variations that are indeed beneficial.

11.7 Random Restarting and Hybrid Algorithms

In global optimization, we usually face the problem that there is no guarantee that all global optima or even at least one has indeed been found. A simple, but effective idea is to restart the optimization algorithm many times hoping that many runs with different, random initializations increase the probability of finding global optima.

The global-optimization algorithms in this chapter can be highly effective in searching globally and they do not presuppose much on the smoothness of the objective function. If approximations of global optima have been found and the objective function is so smooth that the first derivative or even more exist, these approximations can be used as starting points in algorithms for local optimizations. Local-optimization algorithms are discussed in the next chapter. Algorithms combining global and local optimization are called hybrid algorithms.

But before we continue with optimization algorithms, benchmark problems are presented in the next section to guide experimentation with algorithms and objective functions.

11.8 Benchmark Problems

Benchmark problems for multidimensional global optimization are presented in the following. The independent variable is usually $\mathbf{x} \in \mathbb{R}^d$. These functions are useful for evaluating the performance of the optimization algorithms in this chapter and the next. A few are simple test cases for quickly checking an optimization algorithm, but most have properties that makes them difficult to minimize.

1. The Ackley function

$$\begin{aligned} f_{\mathrm{AC}}\colon \quad & [-32.768, 32.768]^d \to \mathbb{R}, \\ & \mathbf{x} \mapsto -\alpha \exp\left(-\beta\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(\gamma x_i)\right) + \alpha + \exp(1), \end{aligned}$$

has many local minima in the nearly flat outer region and a large hole at the center. Usually, the values $\alpha := 20$, $\beta := 0.2$, and $\gamma := 2\pi$ are used. Its global minimum is $f_{\mathrm{AC}}(\mathbf{0}) = 0$.

2. The Bukin function no. 6

$$f_{\mathrm{BU6}}: \quad [-15,-5]\times[-3,3] \to \mathbb{R}, \quad \mathbf{x} \mapsto 100\sqrt{|x_2 - 0.01x_1^2|} + 0.01|x_1 + 10|$$

has many local minima, all of which lie in a narrow valley. Its global minimum is $f_{\mathrm{BU6}}(-10,1) = 0$.

3. The drop-wave function

$$f_{\mathrm{DW}}: \quad [-5.12, 5.12]^2 \to \mathbb{R}, \quad \mathbf{x} \mapsto -\frac{1 + \cos(12\sqrt{x_1^2 + x_2^2})}{(x_1^2 + x_2^2)/2 + 2}$$

has a very complicated structure. Its global minimum is $f_{\mathrm{DW}}(0,0) = -1$.

4. The Easom function

$$f_{\mathrm{EA}}: \quad [-100,100]^2 \to \mathbb{R}, \\ \mathbf{x} \mapsto -\cos(x_1)\cos(x_2)\exp(-(x_1 - \pi)^2 - (x_2 - \pi)^2)$$

has several local minima, while the area near the global minimum is small relative to the search space. Its global minimum is $f_{\mathrm{EA}}\big((\pi,\pi)\big) = -1$.

5. The Gramacy–Lee function

$$f_{\mathrm{GL}}: \quad [0.5, 2.5] \to \mathbb{R}, \quad x \mapsto \frac{\sin(10\pi x)}{2x} + (x-1)^4$$

is a one-dimensional function simple to minimize. Its global minimum is near 0.549.

6. The Griewank function

$$f_{\mathrm{GR}}: \quad [-600,600]^d \to \mathbb{R}, \quad \mathbf{x} \mapsto \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

has many regularly distributed, widespread local minima. Its global minimum is $f_{\mathrm{GR}}(\mathbf{0}) = 0$.

7. The Hölder table function

$$f_{\mathrm{HT}}: \quad [-10,10]^2 \to \mathbb{R}, \quad \mathbf{x} \mapsto -\left|\sin x_1 \cos x_2 \exp\left(\left|1 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi}\right|\right)\right|$$

has many local minima and the four global minima

$$f_{\mathrm{HT}}\big((\pm 8.05502, \pm 9.66459)\big) \approx -19.2085.$$

8. The Levy function

$$\begin{aligned} f_{\mathrm{LE}}:\quad & [-10,10]^d \to \mathbb{R},\\ \mathbf{x} \mapsto \sin(\pi y_1)^2 &+ \sum_{i=1}^{d-1} (y_i - 1)^2\big(1 + 10\sin(\pi y_i + 1)^2\big)\\ &+ (y_d - 1)^2\big(1 + \sin(2\pi y_d)^2\big),\\ y_i &:= 1 + \frac{x_i - 1}{4}, \end{aligned}$$

has many local minima. Its global minimum is $f_{\mathrm{LE}}\big((1,\dots,1)\big) = 0$.

9. The Michalewicz function

$$f_{\mathrm{MI}}:\quad [0,\pi]^d \to \mathbb{R}, \quad \mathbf{x} \mapsto -\sum_{i=1}^{d} \sin(x_i)\sin(ix_i^2/\pi)^{2m}$$

has $d!$ local minima. Its parameter m determines the steepness of the slopes, where larger m makes the search more difficult. Usually, the value $m := 10$ is used.

10. The Rastrigin function

$$f_{\mathrm{RA}}:\quad [-5.12, 5.12]^d \to \mathbb{R}, \quad \mathbf{x} \mapsto \alpha n + \sum_{i=1}^{d}\big(x_i^2 - \alpha\cos(2\pi x_i)\big), \quad \alpha := 10,$$

has many regularly distributed local minima. Its global minimum is $f_{\mathrm{RA}}(\mathbf{0}) = 0$.

11. The two-dimensional Rosenbrock function

$$f_{\mathrm{RO2}}:\quad [-5,10]^2 \to \mathbb{R}, \quad \mathbf{x} \mapsto (\alpha - x_1)^2 + \beta(x_2 - x_1^2)^2,$$

has the global minimum $f_{\mathrm{RO2}}\big((a, a^2)\big) = 0$. Usually the parameters $\alpha := 1$ and $\beta := 100$ are used. The global minimum lies in a narrow, parabolic valley. While finding this valley is easy, converging to the global minimum in the valley is difficult.

The d-dimensional Rosenbrock function

$$f_{\mathrm{ROD}}:\quad [-5,10]^d \to \mathbb{R}, \quad \mathbf{x} \mapsto \sum_{i=1}^{d-1}\big((1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2\big)$$

is nonconvex. In the case $d = 3$, it has one global minimum $f_{\text{ROD}}\big((1,1,1)\big) = 0$. In the case $4 \le d \le 7$, it has the global minimum $f_{\text{ROD}}\big((1,\dots,1)\big) = 0$ and a local minimum near the point $(-1,1,\dots,1)$. In all dimensions, the global minimum is $f_{\text{ROD}}\big((1,\dots,1)\big) = 0$.

12. The Schaffer function no. 2

$$f_{\text{SCH2}}: \quad [-100,100]^2 \to \mathbb{R}, \quad \mathbf{x} \mapsto 1/2 + \frac{\sin(x_1^2 - x_2^2)^2 - 1/2}{(1 + (x_1^2 + x_2^2)/1000)^2}$$

has a complicated structure. Its global minimum is $f_{\text{SCH2}}\big((0,0)\big) = 0$.

13. The Shubert function

$$\begin{aligned} f_{\text{SH}}: \quad & [-10,10]^2 \to \mathbb{R}, \\ & \mathbf{x} \mapsto \left(\sum_{i=1}^{5} i\cos\big((i+1)x_1 + i\big)\right)\left(\sum_{i=1}^{5} i\cos\big((i+1)x_2 + i\big)\right) \end{aligned}$$

has several local minima and many global minima. Its global minimum is approximately -186.731.

14. The six-hump camel function

$$f_{\text{SHC}}: \quad [-3,3]\times[-2,2], \quad \mathbf{x} \mapsto (4 - 2.1x_1^2 + x_1^4/3)x_1^2 + x_1x_2 + (-4 + x_2^2)x_2^2$$

has six local minima, two of which are global. The two global minima are $f_{\text{SHC}}\big((\pm 0.0898, \mp 0.7126)\big) \approx -1.0316$.

15. The sphere function

$$f_{\text{SPH}}: \quad \mathbb{R}^d \to \mathbb{R}, \quad \mathbf{x} \mapsto \sum_{i=1}^{d} x_i^2$$

is a simple, convex function to check the implementation of optimization algorithms. It has d local minima except for the global minimum $f_{\text{SPH}}(\mathbf{0}) = 0$.

16. The Zakharov function

$$f_{\text{ZA}}: \quad [-5,10] \to \mathbb{R}, \quad \mathbf{x} \mapsto \sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} 0.5ix_i^2\right)^2 + \left(\sum_{i=1}^{d} 0.5ix_i^2\right)^4$$

has no local minima except the global minimum $f_{\text{ZA}}(\mathbf{0}) = 0$.

11.9 JULIA Packages

Many optimization packages are available under the `JuliaOpt` umbrella. Various capabilities for global optimization are provided, e.g., by the `Alpine`, `BlackBoxOptim`, `Evolutionary`, `GeneticAlgorithms`, `StochasticSearch` packages.

11.10 Bibliographical Remarks

A classic book on evolutionary computation is [5], while a more recent one is [12]. Two text books on differential evolution are [3, 15]. Good overviews over global optimization are [7, 6, 14, 16, 18].

Problems

11.1 Implement the benchmark functions in Sect. 11.8.

11.2 Plot the benchmark functions in Sect. 11.8 and categorize them with respect to properties that make their minimization difficult.

11.3 Show that the global minimum of the Rastrigin function f_{RA} is $f(\mathbf{0}) = 0$.

11.4 Show the statement about the (local and global) minima of the d-dimensional Rosenbrock function f_{ROD} in Sect. 11.8.

11.5 Implement simulated annealing.

11.6 Apply simulated annealing to the benchmark problems.

11.7 Implement the cooling strategies in Sect. 11.3.3 for simulated annealing and apply them to the benchmark problems.

11.8 Implement particle-swarm optimization.

11.9 Apply particle-swarm optimization to the benchmark problems.

11.10 Implement a genetic algorithm.

11.11 Apply a genetic algorithm to the benchmark problems. Can you find an algorithm such that the same set of parameters works well for all benchmark problems?

11.12 Design and perform an ablation study for a genetic algorithm.

11.13 Implement random restarting. The function should take an optimization algorithm and take care of running it a given number of times and recording the results. The overall results should be presented in a plot.

11.14 Write parallel versions of the algorithms in the previous exercises.

References

1. Darwin, C.: *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London, UK (1859)
2. Eberhart, R., Kennedy, J.: A new optimizer using particle swarm theory. In: *Proc. 6th International Symposium on Micro Machine and Human Science*, p. 39–43. IEEE Press (1995)
3. Feoktistov, V.: *Differential Evolution: in Search of Solutions*. Springer (2006)
4. Hajek, B.: Cooling schedules for optimal annealing. *Mathematics of Operations Research* **13**(2), 311–329 (1988)
5. Holland, J.: *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, USA (1975)
6. Horst, R., Pardalos, P., Thoai, N.: *Introduction to Global Optimization*, 2nd edn. Kluwer Academic Publishers (2000)
7. Horst, R., Tuy, H.: *Global Optimization: Deterministic Approaches*. Springer (1996)
8. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: *Proc. IEEE International Conference on Neural Networks*, pp. 1942–1948. IEEE Press (1995)
9. Kirkpatrick, S., Gelatt Jr., C., Vecchi, M.: Optimization by simulated annealing. *Science* **220**(4598), 671–680 (1983)
10. van Laarhoven, P., Aarts, E.: *Simulated Annealing: Theory and Applications*. Mathematics and its Applications. Kluwer Academic Publishers (1987)
11. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E.: Equation of state calculations by fast computing machines. *J. Chem. Phys.* **21**, 1087–1092 (1953)
12. Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd edn. Springer (1996)
13. Nolte, A., Schrader, R.: A note on the finite time behaviour of simulated annealing. *Mathematics of Operations Research* **25**(3), 476–484 (2000)
14. Pintér, J.: *Global Optimization in Action – Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications*, reprint edn. Springer (2010)
15. Price, K., Storn, R., Lampinen, J.: *Differential Evolution: a Practical Approach to Global Optimization*. Springer (2005)
16. Strongin, R., Sergeyev, Y.: *Global Optimization with Non-Convex Constraints: Sequential and Parallel Algorithms*. Kluwer Academic Publishers (2000)
17. Wolpert, D., Macready, W.: No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* **1**(1) (1997)
18. Zhigljavsky, A.: *Theory of Global Random Search*. Kluwer Academic Publishers (1991)

Chapter 12
Local Optimization

Gradior, gradi, gressus (latin):
to walk, to step, to advance

Gradiens (present participle of *gradior*):
walking, stepping, advancing

Abstract Optimization theory and algorithms can take advantage of the smoothness of real-valued functions. The underlying assumption is that a reasonable starting point sufficiently close to a local extremum is already known, for example from performing a global optimization, and that (at least) the gradient of the objective function is available. After a discussion of the convergence rates of gradient descent, accelerated gradient descent, and the Newton method, the BFGS method is presented in detail, as it is one of the most popular quasi-Newton methods and highly effective in practice.

12.1 Introduction

The general assumption in this chapter is that the real scalar objective function

$$f : \mathbb{R}^d \to \mathbb{R}$$

is smooth enough in the sense that all the derivatives we use in certain contexts exist. The gradient ∇f provides us with valuable knowledge how the function changes locally: since the gradient is the direction in which the function changes the most, it is reasonable to follow the gradient ∇f when maximizing the function and the negative gradient $-\nabla f$ when minimizing it. But this is, a bit surprisingly, only a general rule as we will see in Sect. 12.5.

Why is the gradient the direction of the largest change? It can be shown that the directional derivative

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_12

$$\frac{\partial f}{\partial \mathbf{e}}(\mathbf{r}) := \lim_{h\to 0} \frac{f(\mathbf{r}+h\mathbf{e}) - f(\mathbf{r})}{h}$$

of f at a point $\mathbf{r}$ in direction $\mathbf{e}$ is equal to

$$\frac{\partial f}{\partial \mathbf{e}}(\mathbf{r}) = \mathbf{e} \cdot \nabla f,$$

where the vector $\mathbf{e}$ is a unit vector. The Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1, states that

$$|\mathbf{x} \cdot \mathbf{y}| \le ||\mathbf{x}||||\mathbf{y}|| \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d,$$

and equality only holds if and only if there exists a $\lambda \in \mathbb{R}$ such that $\mathbf{x} = \lambda \mathbf{y}$, i.e., if the two vectors $\mathbf{x}$ and $\mathbf{y}$ are parallel. Here $||.||$ denotes the Euclidean norm. Applying the inequality to the directional derivative yields

$$\left|\frac{\partial f}{\partial \mathbf{e}}(\mathbf{r})\right| \le ||\nabla f||,$$

since $||\mathbf{e}|| = 1$, yielding an upper bound for the absolute value of the directional derivative. The Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1, also tells us that this upper bound is achieved if and only if $\mathbf{e}$ and ∇f are parallel, i.e., if and only if the directional directive is taken in the direction $\mathbf{e} := \nabla f / ||\nabla f||$ of the gradient. This is the reason for the importance of the gradient for optimization.

In optimization we seek local and/or global extrema, whose definitions are the following.

Definition 12.1 ((strict) global extremum) A function $f : U \to V$ has a *global minimum* at a point $x^* \in U$ if $f(x^*) \le f(x)$ holds for all $x \in U$. Analogously, it has a *global maximum* at a point $x^* \in U$ if $f(x^*) \ge f(x)$ holds for all $x \in U$. *Strict global extrema* are ones where the conditions hold with $<$ and $>$ instead of $\le$ and $\ge$.

A function may have more than one global extremum; for example, consider the function $f : \mathbb{R} \to \mathbb{R}, x \mapsto (x^2 - 1)^2$.

Definition 12.2 ((strict) local extremum) A function $f : U \to V$ has a *local minimum* at a point $x^* \in U$ if there exists a neighborhood W such that $f(x^*) \le f(x)$ holds for all $x \in W$. Analogously, it has a *local maximum* at a point $x^* \in U$ if there exists a neighborhood W such that $f(x^*) \ge f(x)$ holds for all $x \in W$. *Strict local extrema* are ones where the conditions hold with $<$ and $>$ instead of $\le$ and $\ge$.

In order to fully specify an optimization problem, the domain where a global extremum is sought must be specified. However, since we are interested in local optimization here, we will refrain from doing so in the following in order not to repeat this point continuously. But it should be remembered that a global extremum may be located on the boundary of a closed domain, and the derivatives of the function cannot help us to find it there.

Any point where the gradient of the function vanishes is called a stationary or critical point. If a stationary point of a real-valued function $f : \mathbb{R} \to \mathbb{R}$ is isolated, it can be classified into four kinds depending on the signs of the first derivative to the left and to the right of the stationary point:

1. a local minimum is a stationary point x where the first derivative f' changes from negative to positive (and hence the second derivative $f''(x)$ is positive),
2. a local maximum is a stationary point x where the first derivative f' changes from positive to negative (and hence the second derivative $f''(x)$ is negative),
3. an increasing point of inflection is a stationary point x where the first derivative f' is positive on both sides of the stationary point, and
4. a decreasing point of inflection is a stationary point x where the first derivative f' is negative on both sides of the stationary point.

The first two categories are local extrema, and the other two cases are known as inflection points or saddle points. Using this naming convention, Fermat's theorem (the one easier to prove) on interior extrema states that the condition that the first derivative of a (smooth) real function defined on an open interval vanishes is a necessary condition for a local extremum.

Theorem 12.3 (Fermat's theorem, interior extremum theorem) *Suppose that $f : (a, b) \to \mathbb{R}$ is a function on the open interval $(a, b) \subset \mathbb{R}$ and that x^* is a local extremum of f. If f is differentiable at the point x^*, then $f'(x^*) = 0$.*

Proof Suppose that x^* is a local minimum. (The proof in the case of a local maximum proceeds analogously.) Then, by the definition of a local minimum, there exists a $\delta \in \mathbb{R}^+$ such that $(x^* - \delta, x^* + \delta) \subset (a, b)$ and such that $f(x^*) \le f(x)$ for all $x \in (x^* - \delta, x^* + \delta)$. Dividing by positive and negative h implies

$$\frac{f(x^* + h) - f(x^*)}{h} \ge 0 \qquad \forall h \in (0, \delta),$$
$$\frac{f(x^* + h) - f(x^*)}{h} \le 0 \qquad \forall h \in (-\delta, 0).$$

Since f is differentiable at x^* by assumption, the limits of these two quotients as $h \to 0$ exist and taking the limits implies both $f'(x^*) \ge 0$ and $f'(x^*) \le 0$. □

A counterexample showing that the condition in the theorem is not sufficient and – at the same time – the simplest example of an inflection or saddle point is the function $f(x) := x^3$ on any interval that contains zero. Then $f'(0) = 0$, but zero is not a local extremum; it is a saddle point.

In higher dimensions, a stationary or critical point is a point where the gradient vanishes. Again, a vanishing gradient is not a sufficient condition for a local extremum. The prototypical example is the function $f(x, y) := x^2 + y^3$ at the point $(0, 0)$, where it resembles a saddle.

12.2 The Hessian Matrix

The relationships between the first and second derivatives of a multivariate function and its local extrema are summarized in Theorem 12.7 below, where the Hessian matrix of the multivariate function f plays a major role. Before stating the theorem, we define the Hessian matrix of a function and see where it occurs in multivariate Taylor expansions.

Definition 12.4 (Hessian matrix) The *Hessian matrix* of a function $f : \mathbb{R}^d \to \mathbb{R}$ is the $(d \times d)$-dimensional matrix H with the entries

$$h_{ij} := \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}.$$

The significance of the Hessian matrix is that it is the coefficient of the quadratic term in multivariate Taylor expansions.

Theorem 12.5 (multivariate Taylor expansion) *Suppose that $f : \mathbb{R}^d \to \mathbb{R}$ is an $(n+1)$-times continuously differentiable function on an open, convex set S. For any two points $\mathbf{x} \in S$ and $\mathbf{x} + \mathbf{h} \in S$, the Taylor expansion*

$$f(\mathbf{x}+\mathbf{h}) = \sum_{|\alpha| \le n} \frac{\mathbf{h}^\alpha}{\alpha!} \partial_\alpha f(\mathbf{x}) + R_n(\mathbf{x}, \mathbf{h})$$

holds, where the integral form of the remainder term is given by

$$R_n(\mathbf{x}, \mathbf{h}) = (n+1) \sum_{|\alpha|=n+1} \frac{\mathbf{h}^\alpha}{\alpha!} \int_0^1 (1-t)^n \partial_\alpha f(\mathbf{x} + t\mathbf{h}) \mathrm{d}t$$

and the Lagrange form of the remainder term is given by

$$R_n(\mathbf{x}, \mathbf{h}) = \sum_{|\alpha|=n+1} \frac{\mathbf{h}^\alpha}{\alpha!} \partial_\alpha f(\mathbf{x} + t\mathbf{h}) \qquad \exists t \in (0,1).$$

In particular, the expansion

$$f(\mathbf{x}+\mathbf{h}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{h} + \frac{1}{2!} \mathbf{h}^\top H(\mathbf{x}) \mathbf{h} + O(\|\mathbf{h}\|^3)$$

holds whenever $n \ge 3$.

The classification of saddle points into degenerate and nondegenerate ones in the following definition will also be needed in Theorem 12.7.

Definition 12.6 ((non-)degenerate saddle point) A saddle point $\mathbf{x}$ is called *degenerate* if the Hessian matrix $H(\mathbf{x})$ at $\mathbf{x}$ is singular. Conversely, the saddle point is called *nondegenerate* if the matrix $H(\mathbf{x})$ is regular.

Based on these definitions, we can formulate important relationships between local extrema and properties of the Hessian matrix.

Theorem 12.7 (Hessian and local extrema) *Suppose that $D \subset \mathbb{R}^d$ is a domain, that $f : \mathbb{R}^d \to \mathbb{R}$ a function with continuous partial derivatives of first and second order, and that $\mathbf{x} \in D$ is a stationary point, i.e., $\nabla f(\mathbf{x}) = 0$. Then the following statements hold true.*

1. *If $H(\mathbf{x})$ is positive definite, then $\mathbf{x}$ is a strict local minimum of f.*
2. *If $H(\mathbf{x})$ is negative definite, then $\mathbf{x}$ is a strict local maximum of f.*
3. *If $\mathbf{x}$ is a local minimum of f, then $H(\mathbf{x})$ is positive semidefinite.*
4. *If $\mathbf{x}$ is a local maximum of f, then $H(\mathbf{x})$ is negative semidefinite.*
5. *If $H(\mathbf{x})$ is indefinite (i.e., if there exist nonzero vectors $\mathbf{y}$ and $\mathbf{z}$ such that $\mathbf{y}^\top H(\mathbf{x})\mathbf{y} < 0 < \mathbf{z}^\top H(\mathbf{x})\mathbf{z}$), then $\mathbf{x}$ is a nondegenerate saddle point.*

These considerations are often also called the first and second partial-derivative test and provide a tool how to identify local extrema of a sufficiently smooth function in the interior of a domain. But the test may be inconclusive, as the statements in Theorem 12.7 do not cover all cases that may occur.

12.3 Convexity

Another question that naturally arises is whether a local extremum that we may have found is already a (or the unique) global extremum. Are there properties of the function or its domain that always ensure that we can draw such a conclusion?

The answer is yes, such properties exist: they are the convexity of the domain and the convexity of the function. The concept of convexity substantially simplifies the search for global extrema. We start by defining convex sets and convex functions.

Definition 12.8 (convex set) A subset $C \subset \mathbb{R}^d$ is called *convex* if

$$\forall t \in [0,1] : \quad \forall\forall \mathbf{x}, \mathbf{y} \in C : \quad t\mathbf{x} + (1-t)\mathbf{y} \in C$$

holds.

Definition 12.9 ((strictly) convex function) A function $f : \mathbb{R}^d \supset C \to \mathbb{R}$ on a convex set $C \subset \mathbb{R}^d$ is called *convex* if the inequality

$$\forall t \in (0,1) : \quad \forall\forall \mathbf{x}, \mathbf{y} \in C : \quad f(t\mathbf{x} + (1-t)\mathbf{y}) \le t f(\mathbf{x}) + (1-t) f(\mathbf{y})$$

holds and it is called *strictly convex* if the inequality holds with $<$ instead of $\le$.

Definition 12.10 ((strictly) concave function) A function $f : \mathbb{R}^d \supset C \to \mathbb{R}$ on a convex set $C \subset \mathbb{R}^d$ is called *(strictly) concave* if $-f$ is (strictly) convex.

Convexity has the useful property that it ensures that a local minimum is already a global minimum (while disregarding the boundary as usual by supposing that the domain of the function is an open set). If the function is even strictly convex, we can additionally conclude that this global minimum is unique.

Theorem 12.11 (convexity and minima) *Suppose that $C \subset \mathbb{R}^d$ is a convex open set, that $f : C \to \mathbb{R}$ is a convex function, and that $\mathbf{x}^*$ is a local minimum of f. Then the local minimum $\mathbf{x}^*$ is a global minimum. Furthermore, if the function f is even strictly convex, then the local minimum $\mathbf{x}^*$ is the unique global minimum.*

Proof By assumption, $\mathbf{x}^*$ is a local minimum of f, i.e.,

$$\exists \delta \in \mathbb{R}^+ : \quad \forall \mathbf{x} \in \{\mathbf{x} \in C \mid \|\mathbf{x} - \mathbf{x}^*\| < \delta\} : \quad f(\mathbf{x}) \geq f(\mathbf{x}^*). \tag{12.1}$$

The proof is indirect. Assuming that there exists a different point $\mathbf{x}_0 \in C\backslash\{\mathbf{x}^*\}$ such that $f(\mathbf{x}_0) < f(\mathbf{x}^*)$ yields

$$\begin{aligned} f(t\mathbf{x}_0 + (1-t)\mathbf{x}^*) &\leq tf(\mathbf{x}_0) + (1-t)f(\mathbf{x}^*) \\ &< tf(\mathbf{x}^*) + (1-t)f(\mathbf{x}^*) = f(\mathbf{x}^*) \qquad \forall t \in (0,1). \end{aligned}$$

If a $t \in (0,1)$ can be found such that $\|(t\mathbf{x}_0 + (1-t)\mathbf{x}^*) - \mathbf{x}^*\| < \delta$, we have found a point $t\mathbf{x}_0 + (1-t)\mathbf{x}^*$ that contradicts the assumption (12.1) that $\mathbf{x}^*$ is a local minimum. We find

$$\|(t\mathbf{x}_0 + (1-t)\mathbf{x}^*) - \mathbf{x}^*\| = t\|\mathbf{x}_0 - \mathbf{x}^*\| = \alpha\delta < \delta$$

after defining

$$0 < t := \frac{\alpha\delta}{\|\mathbf{x}_0 - \mathbf{x}^*\|} < 1$$

and choosing any $\alpha < 1$ from the interval $(0, \|\mathbf{x}_0 - \mathbf{x}^*\|/\delta)$, which shows that such a t exists. Hence the first part of theorem follows.

To show the second part, we assume that there exists a different point $\mathbf{x}_0 \in C\backslash\{\mathbf{x}^*\}$ such that $f(\mathbf{x}_0) \leq f(\mathbf{x}^*)$. Since f is now even strictly convex, we can conclude that

$$\begin{aligned} f(t\mathbf{x}_0 + (1-t)\mathbf{x}^*) &< tf(\mathbf{x}_0) + (1-t)f(\mathbf{x}^*) \\ &\leq tf(\mathbf{x}^*) + (1-t)f(\mathbf{x}^*) = f(\mathbf{x}^*) \qquad \forall t \in [0,1]. \end{aligned}$$

Proceeding similarly, we again see that the existence of the point $t\mathbf{x}_0 + (1-t)\mathbf{x}^*$ contradicts (12.1), which concludes the indirect proof. □

In the case of maxima, the function must be (strictly) concave instead of (strictly) convex.

Corollary 12.12 (convexity and maxima) *Suppose that $C \subset \mathbb{R}^d$ is a convex open set, that $f : C \to \mathbb{R}$ is a concave function, and that $\mathbf{x}^*$ is a local maximum of f. Then the local maximum $\mathbf{x}^*$ is a global maximum. Furthermore, if the function f is even strictly concave, then the local maximum $\mathbf{x}^*$ is the unique global maximum.*

Because of Theorem 12.11 and Corollary 12.12, it is useful to try to partition the domain of a given function that is not (strictly) convex (or concave) on the whole domain such that the function is (strictly) convex (or concave) on as many subdomains as possible. Then Theorem 12.11 and Corollary 12.12 may be applied to the subdomains, splitting the problem into more manageable parts.

12.4 Gradient Descent

The blueprint of any iterative optimization algorithm is the following.

Algorithm 12.13 (iterative minimization)

```
1. Choose an initial point $\mathbf{x}_0$.
2. Repeat the iteration until the stopping criterion is satisfied.
    a. Choose the descent direction $\Delta\mathbf{x}_n$.
    b. Choose the step size $h_n \in \mathbb{R}^+$.
    c. Update by defining
        $$\mathbf{x}_{n+1} := \mathbf{x}_n + h_n \Delta\mathbf{x}_n.$$
```

A suitable starting point can be found by global optimization (see Chap. 11). Two common choices for the stopping criterion is to stop when the change in the points becomes smaller than a prescribed value, i.e., when $|\mathbf{x}_{n+1} - \mathbf{x}_n| = |h_n \Delta\mathbf{x}_n|$ becomes small, or when the change in the function values becomes small, i.e., when $|f(\mathbf{x}_{n+1} - f(\mathbf{x}_n)|$ becomes small.

We already know from Sect. 12.1 that the negative gradient is the direction of steepest descent. Therefore, for minimizing a multivariate function $f : \mathbb{R}^d \to \mathbb{R}$, setting

$$\Delta\mathbf{x}_n := -\nabla f(\mathbf{x}_n) \tag{12.2}$$

and $h_n := h \in \mathbb{R}^+$ suggests itself. These definitions yield the iteration

$$\mathbf{x}_{n+1} := \mathbf{x}_n - h\nabla f(\mathbf{x}_n),$$

which is called gradient descent. In this straightforward usage of the gradient, the step size $h_k = h \in \mathbb{R}^+$ is constant; we will soon see other choices.

When the objective function is to be maximized, the update

$$\mathbf{x}_{n+1} := \mathbf{x}_n + h\nabla f(\mathbf{x}_n)$$

is called gradient ascent.

Another way to see why the choice (12.2) is expedient, is the following argument, which is based on a basic inequality, which is shown in Problem 12.4.

Lemma 12.14 (gradient inequality) *A continuously differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ is convex if and only if the inequality*

$$f(\mathbf{y}) - f(\mathbf{x}) \ge \nabla f(\mathbf{x}) \cdot (\mathbf{y} - \mathbf{x}) \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d$$

holds.

By this inequality, we have

$$f(\mathbf{x}_{n+1}) \ge f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot (\mathbf{x}_{n+1} - \mathbf{x}_n) = f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot (h_n \Delta \mathbf{x}_n).$$

Therefore, in order to see an improvement in every step, i.e., in order to have $f(\mathbf{x}_n) > f(\mathbf{x}_{n+1})$, the descent direction must satisfy

$$0 > \nabla f(\mathbf{x}_n) \cdot \Delta \mathbf{x}_n.$$

This inequality is certainly satisfied by the choice (12.2).

The first theorem below will show how fast gradient descent converges, if the function f is convex and sufficiently smooth. To state the theorems in this section, the concept of a L-smooth function is useful.

Definition 12.15 (L-smooth function) A function $f : \mathbb{R}^d \supset D \to \mathbb{R}$ is called *L-smooth* if its gradient ∇f exists and is L-Lipschitz continuous on its domain D, i.e., if

$$\exists L \in \mathbb{R}^+ : \quad \forall\forall \mathbf{x}, \mathbf{y} \in D : \quad \|\nabla f(\mathbf{y}) - \nabla f(\mathbf{x})\| \le L\|\mathbf{y} - \mathbf{x}\|$$

holds.

The following theorem shows that the convergence rate is linear (as a function of the number of iterations) when an appropriate constant step size is used.

Theorem 12.16 (convergence rate of gradient descent) *Suppose that the function $f : \mathbb{R}^d \to \mathbb{R}$ is convex and L-smooth and that it has the unique global minimum x^*. Then the gradient-descent update*

$$\mathbf{x}_{n+1} := \mathbf{x}_n - h_n \nabla f(\mathbf{x}_n)$$

with step sizes $h_n \le 1/L$ satisfies the inequality

$$f(\mathbf{x}_n) - f(\mathbf{x}^*) \le \frac{\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{\sum_{k=0}^n h_k(1 - Lh_k/2)} \qquad \forall n \in \mathbb{N}.$$

Proof Lemma 12.17 below applied to the points $\mathbf{x}_{k+1} = \mathbf{x}_k - h_k \nabla f(\mathbf{x}_k)$ and $\mathbf{x}_k$ yields the inequality

$$\begin{aligned} f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k) &\le \nabla f(\mathbf{x}_k)\cdot(\mathbf{x}_{k+1}-\mathbf{x}_k) + \frac{L}{2}\|\mathbf{x}_{k+1}-\mathbf{x}_k\|^2 \\ &= -h_k\left(1-\frac{Lh_k}{2}\right)\|\nabla f(\mathbf{x}_k)\|^2 \qquad \forall k\in\mathbb{N}_0. \end{aligned}$$

Next, we define

$$e_k := f(\mathbf{x}_k) - f(\mathbf{x}^*) \ge 0,$$

which is the error in the k-th step and greater than or equal to zero for all k, since x^* is the global minimum. The last inequality implies

$$e_{k+1} \le e_k - h_k\left(1-\frac{Lh_k}{2}\right)\|\nabla f(\mathbf{x}_k)\|^2 \qquad \forall k\in\mathbb{N}_0. \tag{12.3}$$

Since f is convex (and continuously differentiable), we have $-e_k = f(\mathbf{x}^*) - f(\mathbf{x}_k) \ge \nabla f(\mathbf{x}_k)\cdot(\mathbf{x}^*-\mathbf{x}_k)$ and hence $e_k \le \|\nabla f(\mathbf{x}_k)\|\|\mathbf{x}_k-\mathbf{x}^*\|$. These two inequalities yield

$$e_{k+1} \le e_k - h_k\left(1-\frac{Lh_k}{2}\right)\frac{e_k^2}{\|\mathbf{x}_k-\mathbf{x}^*\|^2} \qquad \forall k\in\mathbb{N}_0.$$

As Lemma 12.18 below shows, the sequence $\|\mathbf{x}_k-\mathbf{x}^*\|$ decreases as k increases; this is where the assumption $h_k < 1/L$, and not only $h_k < 2/L$, is used. Therefore we have the estimate

$$e_{k+1} \le e_k - h_k\left(1-\frac{Lh_k}{2}\right)\frac{e_k^2}{\|\mathbf{x}_0-\mathbf{x}^*\|^2} \qquad \forall k\in\mathbb{N}_0$$

and hence

$$\frac{1}{e_{k+1}} - \frac{1}{e_k} \ge h_k\left(1-\frac{Lh_k}{2}\right)\frac{e_k}{\|\mathbf{x}_0-\mathbf{x}^*\|^2 e_{k+1}} \qquad \forall k\in\mathbb{N}_0$$

after division by $e_k e_{k+1}$ and rearranging terms. (If $e_k = 0$ for any k, then $\mathbf{x}_k = \mathbf{x}^*$ and $\nabla f(\mathbf{x}_k) = 0$, implying that all $\mathbf{x}_k$ will be equal to $\mathbf{x}^*$ from this point on and trivially satisfying the asserted inequality.)

Because of $e_{k+1} \le e_k$ due to (12.3), we find

$$\frac{1}{e_{k+1}} - \frac{1}{e_k} \ge h_k\left(1-\frac{Lh_k}{2}\right)\frac{1}{\|\mathbf{x}_0-\mathbf{x}^*\|^2} \qquad \forall k\in\mathbb{N}_0.$$

Summing all these inequalities for $k\in\{0,\ldots,n-1\}$ yields a telescopic sum and hence the estimate

$$\frac{1}{e_n} - \frac{1}{e_0} \ge \frac{1}{\|\mathbf{x}_0-\mathbf{x}^*\|^2}\sum_{k=0}^{n-1} h_k\left(1-\frac{Lh_k}{2}\right) \qquad \forall n\in\mathbb{N}.$$

Since $e_n > 0$, we find

$$e_n \le \frac{\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{\sum_{k=0}^{n-1} h_k(1 - Lh_k/2)},$$

which is the asserted inequality. □

The following two lemmata were used in the proof. Note that Lemma 12.14 provides a lower bound for $f(\mathbf{y}) - f(\mathbf{x})$ due to convexity and that Lemma 12.17 provides an upper bound for $f(\mathbf{y}) - f(\mathbf{x})$ due to L-smoothness.

Lemma 12.17 *Suppose that the function $f\colon \mathbb{R}^d \to \mathbb{R}$ is L-smooth. Then the inequality*

$$f(\mathbf{y}) - f(\mathbf{x}) \le \nabla f(\mathbf{x}) \cdot (\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2 \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d$$

holds.

Proof We calculate

$$\begin{aligned} f(\mathbf{y}) - f(\mathbf{x}) - \nabla f(\mathbf{x}) \cdot (\mathbf{y} - \mathbf{x}) &= \int_0^1 \big(\nabla f(\mathbf{x} + t(\mathbf{y} - \mathbf{x})) - \nabla f(\mathbf{x})\big) \cdot (\mathbf{y} - \mathbf{x})\mathrm{d}t \\ &\le \int_0^1 Lt\|\mathbf{y} - \mathbf{x}\|^2 \mathrm{d}t \\ &= \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2, \end{aligned}$$

where the inequality follows using the Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1, and the L-smoothness of f. □

Lemma 12.18 *Suppose the function f and the sequences $\langle h_n\rangle$ and $\langle \mathbf{x}_n\rangle$ are as in Theorem 12.16. Then the sequence $\langle \|\mathbf{x}_n - \mathbf{x}^*\|\rangle$ decreases as n increases.*

Proof We start by calculating

$$\begin{aligned} \|\mathbf{x}_{n+1} - \mathbf{x}^*\|^2 &= \|\mathbf{x}_n - h_n\nabla f(\mathbf{x}_n) - \mathbf{x}^*\|^2 \\ &= \|\mathbf{x}_n - \mathbf{x}^*\|^2 - 2h_n\nabla f(\mathbf{x}_n) \cdot (\mathbf{x}_n - \mathbf{x}^*) + h_n^2\|\nabla f(\mathbf{x}_n)\|^2. \end{aligned} \quad (12.4)$$

In the second step, we show that

$$f(\mathbf{x}) - f(\mathbf{y}) \le \nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{y}) - \frac{1}{2L}\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\|^2 \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d. \quad (12.5)$$

In the estimate

$$\begin{aligned} f(\mathbf{x}) - f(\mathbf{y}) &= \big(f(\mathbf{x}) - f(\mathbf{z})\big) + \big(f(\mathbf{z}) - f(\mathbf{y})\big) \\ &\leq \nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{z}) + \nabla f(\mathbf{y}) \cdot (\mathbf{z} - \mathbf{y}) + \frac{L}{2}\|\mathbf{z} - \mathbf{y}\|^2 \\ &= \nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{y}) + \big(\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\big) \cdot (\mathbf{y} - \mathbf{z}) + \frac{L}{2}\|\mathbf{z} - \mathbf{y}\|^2, \end{aligned}$$

the first term $f(\mathbf{x}) - f(\mathbf{z})$ is estimated using Lemma 12.14 and the second term $f(\mathbf{z}) - f(\mathbf{y})$ is estimated using Lemma 12.17. Then substituting

$$\mathbf{z} := \mathbf{y} - \frac{1}{L}\big(\nabla f(\mathbf{y}) - \nabla f(\mathbf{x})\big)$$

yields (12.5).

Inequality (12.5) applied to the situation in this lemma implies that

$$0 \leq f(\mathbf{x}_n) - f(\mathbf{x}^*) \leq \nabla f(\mathbf{x}_n) \cdot (\mathbf{x}_n - \mathbf{x}^*) - \frac{1}{2L}\|\nabla f(\mathbf{x}_n)\|^2. \tag{12.6}$$

In the last step, we combine (12.4) and (12.6) to find

$$\begin{aligned} \|\mathbf{x}_{n+1} - \mathbf{x}^*\|^2 &\leq \|\mathbf{x}_n - \mathbf{x}^*\|^2 - \frac{h_n}{L}\|\nabla f(\mathbf{x}_n)\|^2 + h_n^2\|\nabla f(\mathbf{x}_n)\|^2 \\ &= \|\mathbf{x}_n - \mathbf{x}^*\|^2 - h_n\left(\frac{1}{L} - h_n\right)\|\nabla f(\mathbf{x}_n)\|^2 \\ &\leq \|\mathbf{x}_n - \mathbf{x}^*\|^2, \end{aligned}$$

which concludes the proof. □

Theorem 12.16 provides two interesting corollaries. The first is the answer to the question how to choose the step sizes h_n such that convergence is fastest based on the estimate in the theorem.

Corollary 12.19 (optimal step sizes) *The optimal step sizes h_n based on the estimate in Theorem 12.16 are*

$$h_n := \frac{1}{L},$$

which result in the estimate

$$f(\mathbf{x}_n) - f(\mathbf{x}^*) \leq \frac{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{n+1} \qquad \forall n \in \mathbb{N}.$$

Proof We maximize the denominator in the inequality in Theorem 12.16. Since all the summands in $\sum_{k=0}^{n} h_k(1 - Lh_k/2)$ are independent, the optimal h_n are $h_n := \arg\max_{h\in(0,1/L)} h(1 - Lh/2) = 1/L$. □

The second corollary concerns diminishing step sizes, which very commonly occur in stochastic (gradient-descent) optimization (see Sect. 13.6). The denominator in the estimate in Theorem 12.16 can be written as

$$\sum_{k=0}^{n} h_k \left(1 - \frac{Lh_k}{2}\right) = \sum_{k=0}^{n} h_k - \frac{L}{2} \sum_{k=0}^{n} h_k^2.$$

In order to ensure convergence, we hence require that

$$\sum_{k=0}^{n} h_k = \infty \qquad \text{and} \qquad \sum_{k=0}^{n} h_k^2 < \infty.$$

This is indeed the usual requirement in stochastic optimization. The step sizes $h_n := a/(n+b)$ in the second corollary are very common in stochastic optimization and satisfy these two requirements.

Corollary 12.20 (diminishing step sizes) *Suppose that the assumptions in Theorem 12.16 hold and that the step sizes are defined such that*

$$h_n := \frac{a}{n+b} \le \frac{1}{L}, \qquad a, b \in \mathbb{R}^+.$$

Then the estimate

$$f(\mathbf{x}_n) - f(\mathbf{x}^*) \le \frac{\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{a \ln\left(\frac{n+1+b}{b}\right) - \frac{La^2(n+1)}{2b(n+1+b)}} \qquad \forall n \in \mathbb{N}$$

holds.

Proof In general, if $g\colon \mathbb{R} \to \mathbb{R}$ is a monotonically decreasing Riemann-integrable function, the estimates

$$\int_{n_0}^{n_1+1} g(x)\mathrm{d}x \le \sum_{k=n_0}^{n_1} g(k) \le \int_{n_0-1}^{n_1} g(x)\mathrm{d}x$$

hold. (To see this, the sum is interpreted as the Riemann sum of an integral; it is useful to draw a sketch of a monotonically decreasing function and the rectangles in the Riemann sum.)

In our case, we have

$$\begin{aligned}\sum_{k=0}^{n} h_k \left(1 - \frac{Lh_k}{2}\right) &\ge \int_0^{n+1} \frac{a}{x+b}\left(1 - \frac{L}{2}\frac{a}{x+b}\right)\mathrm{d}x \\ &= a \ln\left(\frac{n+1+b}{b}\right) - \frac{La^2(n+1)}{2b(n+1+b)}, \qquad (12.7)\end{aligned}$$

which concludes the proof. □

We can also interpret the estimates in the theorem and its corollaries differently by asking the question how many iterations are necessary to ensure that the error is smaller than a prescribed tolerance ϵ, i.e., that

$$f(\mathbf{x}_n) - f(\mathbf{x}^*) < \epsilon$$

holds. Based on the estimate in Corollary 12.19, we have

$$n + 1 > \frac{2L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{\epsilon}.$$

Therefore $O(1/\epsilon)$ iterations are required in order to achieve an error $f(\mathbf{x}_k) - f(\mathbf{x}^*) < \epsilon$. Furthermore, convergence is proportional to the Lipschitz constant L and the distance of the starting point $\mathbf{x}_0$ from the minimum $\mathbf{x}^*$ and it is inversely proportional to the prescribed tolerance.

There is also a connection with regularization (see Sect. 13.9.1) in the context of neural networks (see Chap. 13). Regularization diminishes the Lipschitz constant of the neural network, thus having the additional benefit that it speeds up convergence.

In practice, the Lipschitz constant L can be calculated using the Hessian matrix if the function f is given in a sufficiently explicit and differentiable form. But if it is not represented in a straightforward manner, it may be hard or impossible to determine the Lipschitz constant L. Also, intuition suggests that the step size should become smaller as the minimum is approached. These considerations motivate deliberations on the step size in the next sections.

Another practical consideration is that if the function f is so smooth that the Hessian matrix exists, then we can take advantage of the second derivatives directly by using the BFGS method (see Sect. 12.8).

An question that suggests itself having shown Theorem 12.16 is: what is the best convergence rate that an optimization algorithm that only uses the gradient of a function can achieve? In order to be able to answer this question, we have to restrict the class of functions that the optimization is supposed to work on. The reason is simply that if we demand the optimization algorithm to work on functions that are not smooth at all, it is always possible to construct counterexamples because the function values may jump without restriction. Therefore we require the functions to have the same smoothness as in the (only) convergence results we already know, namely Theorem 12.16.

The following theorem states that any optimization algorithm that uses only the gradient can achieve at most quadratic convergence on this class of functions. Such iterative optimization algorithms are called first-order methods.

Theorem 12.21 (lower bound for convergence rate of gradient descent) *Suppose that the sequence $\langle \mathbf{x}_k \rangle \subset \mathbb{R}^d$ is generated by a first-order optimization algorithm, i.e., the points in the sequence satisfy the inclusion*

$$\mathbf{x}_{n+1} \in \mathbf{x}_0 + \operatorname{span}\{\nabla f(\mathbf{x}_1), \ldots, \nabla f(\mathbf{x}_n)\} \qquad \forall n \in \mathbb{N}_0.$$

Then for any $\mathbf{x}_0 \in \mathbb{R}^d$ and any $k \in \mathbb{N}$ with $1 \le k \le (d-1)/2$ there exists a continuously differentiable, convex, and L-smooth function $f : \mathbb{R}^d \to \mathbb{R}$ that has the global minimum $x^* \in \mathbb{R}^d$ such that the inequalities

$$f(\mathbf{x}_n) - f(\mathbf{x}^*) \ge \frac{3L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{32(n+1)^2} \qquad \forall n \in \mathbb{N},$$
$$\|\mathbf{x}_n - \mathbf{x}^*\|^2 \ge \frac{1}{8}\|\mathbf{x}_0 - \mathbf{x}^*\|^2 \qquad \forall n \in \mathbb{N}$$

hold.

For proof, see [14, Theorem 2.1.7].

In summary, we now know that the gradient-descent algorithm in Theorem 12.16 achieves linear convergence, while Theorem 12.21 means that the best algorithm on this class of functions can achieve at most quadratic convergence.

Hence the next questions pose themselves. Does an algorithm that achieves quadratic convergence exists? What have we missed?

12.5 Accelerated Gradient Descent *

The answer to this question was found in 1983 [13] and is discussed in [14, Section 2.2]. It turns out that the requirement $f(\mathbf{x}_{n+1}) < f(\mathbf{x}_n)$ discussed at the beginning of Sect. 12.4 is not conducive for the minimization of convex functions; the requirement is a local statement, but convexity is a global property.

As the next theorem shows, quadratic convergence, i.e., the optimal convergence rate, can indeed be achieved in the minimization of convex, smooth functions. The iteration in accelerated gradient descent combines the gradient with the difference $\mathbf{x}_n - \mathbf{x}_{n-1}$, which is the momentum of the trajectory of the sequence $\langle \mathbf{x}_n \rangle$.

Algorithm 12.22 (accelerated gradient descent) Algorithm 12.13 with the update

$$\begin{aligned}
\lambda_0 &:= 1,\\
\lambda_n &:= \frac{1 + \sqrt{4\lambda_{n-1}^2 + 1}}{2},\\
\gamma_n &:= \frac{\lambda_{n-1} - 1}{\lambda_n},\\
\mathbf{d}_n &:= \gamma_n(\mathbf{x}_n - \mathbf{x}_{n-1}),\\
\mathbf{y}_n &:= \mathbf{x}_n + \mathbf{d}_n,\\
\mathbf{g}_n &:= -\frac{1}{L}\nabla f(\mathbf{y}_n) = -\frac{1}{L}\nabla f(\mathbf{x}_n + \mathbf{d}_n),\\
\mathbf{x}_{n+1} &:= \mathbf{y}_n + \mathbf{g}_n = \mathbf{x}_n + \mathbf{d}_n + \mathbf{g}_n
\end{aligned}$$

is called accelerated gradient descent.

Theorem 12.23 (accelerated gradient descent) *Suppose that the function $f\colon \mathbb{R}^d \to \mathbb{R}$ is convex and L-smooth and has the unique global minimum x^*. Then the accelerated gradient-descent algorithm, Algorithm 12.22, satisfies the inequality*

$$f(\mathbf{x}_{n+1}) - f(\mathbf{x}^*) \le \frac{f(\mathbf{x}_1) - f(\mathbf{x}^*) + \frac{L}{2}\|\mathbf{x}_1 - \mathbf{x}^*\|^2}{(n/2+1)^2} \qquad \forall n \in \mathbb{N}.$$

The following proof partly follows [1].

Proof Lemma 12.14 implies the inequality

$$\begin{aligned} &f\big(\mathbf{x} - h\nabla f(\mathbf{x})\big) - f(\mathbf{y}) \\ &\quad \le f\big(\mathbf{x} - h\nabla f(\mathbf{x})\big) - f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{y}) \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d \quad \forall h \in \mathbb{R}. \end{aligned}$$

Furthermore estimating the first two terms on the right-hand side using Lemma 12.17 yields

$$\begin{aligned} &f\big(\mathbf{x} - h\nabla f(\mathbf{x})\big) - f(\mathbf{y}) \\ &\quad \le \left(-h + \frac{Lh^2}{2}\right)\|\nabla f(\mathbf{x})\|^2 + \nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{y}) \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d \quad \forall h \in \mathbb{R}. \end{aligned}$$

In order to find the strongest estimate, we define $h := \arg\min_{h\in\mathbb{R}}(Lh^2/2 - h) = 1/L$, which results in the inequality

$$f\big(\mathbf{x} - \frac{1}{L}\nabla f(\mathbf{x})\big) - f(\mathbf{y}) \le -\frac{1}{2L}\|\nabla f(\mathbf{x})\|^2 + \nabla f(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{y}) \qquad \forall\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d.$$

We define the error in the n-th iteration as

$$e_n := f(\mathbf{x}_n) - f(\mathbf{x}^*).$$

Substituting $\mathbf{x} := \mathbf{x}_k + \mathbf{d}_k$ and $\mathbf{y} := \mathbf{x}_k$ into the last inequality yields

$$e_{n+1} - e_n = f(\mathbf{x}_{n+1}) - f(\mathbf{x}_n) \le -\frac{1}{2L}\|L\mathbf{g}_n\|^2 + \mathbf{g}_n \cdot \mathbf{d}_n \qquad \forall n \in \mathbb{N}.$$

Similarly substituting $\mathbf{x} := \mathbf{x}_k + \mathbf{d}_k$ and $\mathbf{y} := \mathbf{x}^*$ into the same inequality yields

$$e_{n+1} = f(\mathbf{x}_{n+1}) - f(\mathbf{x}^*) \le -\frac{1}{2L}\|L\mathbf{g}_n\|^2 + \mathbf{g}_n \cdot (\mathbf{x}_n + \mathbf{d}_n - \mathbf{x}^*) \qquad \forall n \in \mathbb{N}.$$

The last two inequalities can be rewritten as

$$e_{n+1} - e_n \le -\frac{L}{2}\big(\|\mathbf{g}_n\|^2 + 2\mathbf{g}_n \cdot \mathbf{d}_n\big) \qquad \forall n \in \mathbb{N},$$
$$e_{n+1} \le -\frac{L}{2}\big(\|\mathbf{g}_n\|^2 + 2\mathbf{g}_n \cdot (\mathbf{x}_n + \mathbf{d}_n - \mathbf{x}^*)\big) \qquad \forall n \in \mathbb{N}.$$

We would like to use in the identity

$$\|\mathbf{a}\|^2 + 2\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a} + \mathbf{b}\|^2 - \|\mathbf{b}\|^2$$

on the right-hand side in order to arrive at a telescoping structure. Therefore we add $(\lambda_n - 1)$ times the first inequality to the second inequality, yielding

$$\lambda_n(e_{n+1} - e_n) + e_n \le -\frac{L}{2}\big(\lambda_n\|\mathbf{g}_n\|^2 + 2\mathbf{g}_n \cdot (\lambda_n\mathbf{d}_n + \mathbf{x}_n - \mathbf{x}^*)\big) \qquad \forall n \in \mathbb{N}.$$

where the

$$\lambda_n \ge 1 \tag{12.8}$$

will be specified later. We multiply the inequality by λ_n, noting that the λ_n are nonnegative, and apply the identity above to find

$$\lambda_n^2(e_{n+1} - e_n) + \lambda_n e_n \le -\frac{L}{2}\big(\|\lambda_n\mathbf{g}_n\|^2 + 2\lambda_n\mathbf{g}_n \cdot (\lambda_n\mathbf{d}_n + \mathbf{x}_n - \mathbf{x}^*)\big)$$
$$= -\frac{L}{2}\big(\|\lambda_n\mathbf{g}_n + \lambda_n\mathbf{d}_n + \mathbf{x}_n - \mathbf{x}^*\|^2 - \|\lambda_n\mathbf{d}_n + \mathbf{x}_n - \mathbf{x}^*\|^2\big) \qquad \forall n \in \mathbb{N}.$$

We call the arguments of the norms

$$\mathbf{r}_n := \lambda_n\mathbf{g}_n + \lambda_n\mathbf{d}_n + \mathbf{x}_n - \mathbf{x}^*,$$
$$\mathbf{s}_n := \lambda_n\mathbf{d}_n + \mathbf{x}_n - \mathbf{x}^*.$$

In order to create a telescoping structure, we would like to determine the λ_n and γ_n such that $\mathbf{r}_n = \mathbf{s}_{n+1}$, which is equivalent to

$$\lambda_n(\mathbf{g}_n + \mathbf{d}_n) + \mathbf{x}_n - \mathbf{x}^* = \lambda_{n+1}\mathbf{d}_{n+1} + \mathbf{x}_{n+1} - \mathbf{x}^* \qquad \forall n \in \mathbb{N}.$$

Using the definition of $\mathbf{d}_{n+1}$ and $\mathbf{x}_{n+1}$ on the right-hand side, this condition is furthermore equivalent to

$$\lambda_n(\mathbf{g}_n + \mathbf{d}_n) + \mathbf{x}_n - \mathbf{x}^* = \lambda_{n+1}\gamma_{n+1}(\mathbf{d}_n + \mathbf{g}_n) + \mathbf{x}_n + \mathbf{d}_n + \mathbf{g}_n - \mathbf{x}^* \qquad \forall n \in \mathbb{N}.$$

This is always true if

$$\lambda_n = \lambda_{n+1}\gamma_{n+1} + 1 \qquad \forall n \in \mathbb{N} \tag{12.9}$$

holds.

Therefore the last inequality becomes

$$\lambda_n^2 e_{n+1} - (\lambda_n^2 - \lambda_n)e_n \le -\frac{L}{2}\big(\|\mathbf{s}_{n+1}\|^2 - \|\mathbf{s}_n\|^2\big) \qquad \forall n \in \mathbb{N}.$$

To achieve a suitable telescoping structure on the left side as well, we would like to meet the condition $u_n = v_{n+1}$ for all n, where

$$\begin{aligned} u_n &:= \lambda_n^2 e_{n+1}, \\ v_n &:= (\lambda_n^2 - \lambda_n) e_n. \end{aligned}$$

This condition is met if

$$\lambda_n^2 = \lambda_{n+1}^2 - \lambda_{n+1} \qquad \forall n \in \mathbb{N} \tag{12.10}$$

holds.

Thus the last inequality becomes

$$v_{n+1} - v_n \le -\frac{L}{2}\left(\|\mathbf{s}_{n+1}\|^2 - \|\mathbf{s}_n\|^2\right) \qquad \forall n \in \mathbb{N}. \tag{12.11}$$

The last step now involves inequalities with this telescoping structure. Suppose there are two real sequences $\langle a_n \rangle$ and $\langle b_n \rangle$ such that $a_{n+1} - a_n \le b_n - b_{n+1}$ holds for all n. Then the chain

$$a_{n+1} + b_{n+1} \le a_n + b_n \le \cdots \le a_1 + b_1$$

of inequalities clearly holds true. Applying this idea via $a_n := v_n$ and $b_n := (L/2)\|\mathbf{s}_n\|^2$ to inequality (12.11), we find

$$\begin{aligned} \lambda_n^2 e_{n+1} = v_{n+1} \le v_{n+1} + \frac{L}{2}\|\mathbf{s}_{n+1}\|^2 &\le v_1 + \frac{L}{2}\|\mathbf{s}_1\|^2 \\ &= \lambda_0^2 e_1 + \frac{L}{2}\|(\lambda_0 - 1)(\mathbf{x}_1 - \mathbf{x}_0) + \mathbf{x}_1 - \mathbf{x}^*\|^2 \qquad \forall n \in \mathbb{N}. \end{aligned}$$

After setting $\lambda_0 := 1$, we have

$$f(\mathbf{x}_{n+1}) - f(\mathbf{x}^*) \le \frac{f(\mathbf{x}_1) - f(\mathbf{x}^*) + \frac{L}{2}\|\mathbf{x}_1 - \mathbf{x}^*\|^2}{\lambda_n^2} \qquad \forall n \in \mathbb{N}.$$

Finally, we check that all three conditions for the λ_n and γ_n can be met and that the λ_n grow (at least) linearly. The three conditions are (12.8), (12.9), and (12.10), which yield

$$\begin{aligned} \lambda_n &:= \frac{1 + \sqrt{4\lambda_{n-1}^2 + 1}}{2} \ge \frac{1}{2} + \lambda_{n-1}, \\ \gamma_n &:= \frac{\lambda_{n-1} - 1}{\lambda_n}. \end{aligned}$$

It is now straightforward to see by induction (and using $\lambda_0 = 1$) that

$$\lambda_n \geq \frac{n}{2} + 1 \qquad \forall n \in \mathbb{N}_0, \tag{12.12}$$

which immediately implies $\lim_{n\to\infty} \gamma_n = 1$; furthermore, the γ_n are bounded by

$$0 < \gamma_n < 1. \tag{12.13}$$

In summary, the linear growth of the λ_n implies quadratic convergence. □

Accelerated gradient descent is implemented in Problem 12.8 and tested in Problem 12.9.

12.6 Line Search and the Wolfe Conditions

It is important to note that up to now we have only used fixed step sizes, i.e., the factors of the gradients in the update formulas have been constants determined by the Lipschitz constant of the objective function. We now lift this restriction (although it has made the analysis easier), because the Lipschitz constant may not be known or the function may not be Lipschitz continuous at all, but we still wish to optimize such functions.

Since we search along the direction given by the gradient, such optimization algorithms are called line-search algorithms. In general, they have the following form.

Algorithm 12.24 (line-search algorithm)

1. Compute the search direction $\mathbf{p}_n := -\nabla f$.
2. Determine the step size $\alpha_n > 0$ such that a sufficient-decrease condition condition is satisfied.
3. Set $\mathbf{x}_{n+1} := \mathbf{x}_n + \alpha_n \mathbf{p}_n$.

A simple-minded decrease condition such as $f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \alpha_n \mathbf{p}_n) \leq f(\mathbf{x}_n)$, meaning that the function value at the new approximation point $\mathbf{x}_{n+1}$ is smaller than before, is *not* a useful requirement, since the function value, a real number, may decrease forever without getting close to a minimum.

Instead, we would ideally like to use a global minimizer

$$\alpha_n := \arg\min_{\alpha\in\mathbb{R}^+} h(\alpha)$$

of the function

$$h\colon \quad \mathbb{R}^+ \to \mathbb{R}, \quad \alpha \mapsto f(\mathbf{x}_n + \alpha \mathbf{p}_n)$$

as the step size. Finding an approximation of the minimizer may be quite an elaborate task (but at least it is always a one-dimensional problem) and is usually performed in two steps. In the first step, an interval of acceptable step sizes is identified. In the second step, the function is interpolated and the interval is bisected to find a good approximation of the best step size.

However we face a trade-off here: while we want to find a good approximation of the ideal step size, spending too much computational effort on the search for the step size is not conducive for the performance of the whole algorithm.

Useful and popular conditions that ensure a sufficient decrease of the function value are the (two) Wolfe conditions and the (two) strong Wolfe conditions [19, 20] [16, Section 3.1], which we discuss in detail in the rest of this section.

The first Wolfe condition (or Armijo condition) is the inequality

$$f(\mathbf{x}_n + \alpha_n \mathbf{p}_n) \le \underbrace{f(\mathbf{x}_n) + c_1 \alpha \nabla f(\mathbf{x}_n) \cdot \mathbf{p}_n}_{l(\alpha):=}, \tag{12.14}$$

where $c_1 \in (0, 1)$ is a constant, which is in practice chosen to be quite small, e.g., $c_1 := 10^{-4}$. The inequality means that the reduction is required to be (at least) proportional to the step size

$$\alpha_n := c_1 \alpha$$

and the directional derivative $\nabla f(\mathbf{x}_n) \cdot \mathbf{p}_n$.

The right-hand side of the inequality is a linear function of α, and we denote it by $l(\alpha)$. Using this notation, the condition becomes $h(\alpha) \le l(\alpha)$. The function l has the negative derivative $c_1 \nabla f(\mathbf{x}_n) \cdot \mathbf{p}_n$ with respect to α, and its tangent lies above the tangent of h because $c_1 \in (0, 1)$. The larger c_1 is, the more reduction in the function value is required.

But this first condition alone does not guarantee good progress of the algorithm, since it can always be satisfied for sufficiently small values of α. To exclude this possibility, a second condition is needed.

The second Wolfe condition is called the curvature condition and it is the requirement that

$$\nabla f(\mathbf{x}_n + \alpha_n \mathbf{p}_n) \cdot \mathbf{p}_n \ge c_2 \nabla f(\mathbf{x}_n) \cdot \mathbf{p}_n \tag{12.15}$$

on the step size α_n, where $c_2 \in (c_1, 1)$ is a constant. The left-hand side is equal to $h'(\alpha_n)$, implying the interpretation that the slope $h'(\alpha_n)$ of h at α_n must be greater than or equal to the constant c_2 times the slope $h'(0)$.

If the slope $h'(\alpha)$ is strongly negative for small α, then this second condition requires that the step size α cannot be chosen too small, which is reasonable, since a strongly negative slope indicates that the function f can be reduced significantly by moving further along.

If line search is used in conjunction with a Newton or quasi-Newton method (see the next sections), a typical value for c_2 is 0.9.

Having defined the Wolfe conditions, the question whether they can be satisfied arises naturally. The answer to his question is always yes under reasonable assumptions on the objective function f, as the following theorem shows.

Theorem 12.25 (Wolfe conditions) *Suppose that $f : \mathbb{R}^d \to \mathbb{R}$ is a continuously differentiable function, that $B \in \mathbb{R}^{d\times d}$ is a positive definite matrix, that $\mathbf{p} := -B\nabla f(\mathbf{x})$, that $0 < c_1 < c_2 < 1$, and that the function $h : \mathbb{R}^+ \to \mathbb{R}$, $\alpha \mapsto f(\mathbf{x}_n + \alpha\mathbf{p}_n)$ is bounded below. Then there exists an interval of step sizes α satisfying the strict inequalities in both Wolfe conditions (12.14) and (12.15).*

Proof Since the matrix B is positive definite by assumption and since $\alpha > 0$ and $c_1 > 0$, the function $l : \mathbb{R}^+ \to \mathbb{R}$, $\alpha \mapsto f(\mathbf{x}) + \alpha c_1 \nabla f(\mathbf{x}) \cdot \mathbf{p}$ is unbounded below and thus intersects the function h, which is bounded below, at least once. We denote the smallest such value by α_1 such that $\alpha_1 > 0$ and

$$f(\mathbf{x} + \alpha_1\mathbf{p}) = f(\mathbf{x}) + \alpha_1 c_1 \nabla f(\mathbf{x}) \cdot \mathbf{p}.$$

Hence the strict inequality in the first Wolfe condition (12.14) is satisfied for all $\alpha \in (0, \alpha_1)$.

Next, the mean-value theorem implies that

$$\exists \alpha_0 \in (0, \alpha_1) : \qquad f(\mathbf{x} + \alpha_1\mathbf{p}) - f(\mathbf{x}) = \alpha_1 \nabla f(\mathbf{x} + \alpha_0\mathbf{p}) \cdot \mathbf{p}.$$

The last two equations yield

$$\nabla f(\mathbf{x} + \alpha_0\mathbf{p}) \cdot \mathbf{p} = c_1 \nabla f(\mathbf{x}) \cdot \mathbf{p} > c_2 \nabla f(\mathbf{x}) \cdot \mathbf{p}$$

after noting that $c_1 < c_2$ and $\nabla f(\mathbf{x}) \cdot \mathbf{p} = -\nabla f(\mathbf{x}) B \nabla f(\mathbf{x}) < 0$.

Finally, because f is continuously differentiable, there exists an interval around α_0 satisfying the strict inequalities in both Wolfe conditions. □

Having seen that the Wolfe conditions can always be satisfied under reasonable assumptions on the objective function, we next mention a bracketing algorithm [9, Algorithm 4.6] that calculates such a permissible step size. It can be shown [9, Theorem 4.7] that the output of the algorithm is always a step size that satisfies the Wolfe conditions (12.14) and (12.15) if the algorithm terminates.

Algorithm 12.26 (Wolfe line search)

```
1. Initialize $\alpha := 0$, $\beta := \infty$, and $t := 1$.
2. Repeat:
     a. If the first Wolfe condition (12.14) is not satisfied
          set $\beta := t$
        else if the second Wolfe condition (12.15) is not satisfied
          set $\alpha := t$
        else break the loop.
     b. If $\beta < \infty$
          set $t := (\alpha + \beta)/2$
        else
          set $t := 2\alpha$.
3. Finally return the step size $\alpha$.
```

The line-search algorithm Algorithm 12.26 is implemented in Problem 12.11 and tested in Problem 12.12.

While other conditions for line search can be used, the significance of the Wolfe conditions is that they are beneficial for the convergence of the BFGS method, which we will discuss in Sect. 12.8 below.

12.7 The Newton Method

In this section, the Newton method for approximating the roots of functions, i.e., for approximating points x such that $g(x) = 0$, is summarized in the one-dimensional case, i.e., for functions $g\colon \mathbb{R} \supset [a, b] \to \mathbb{R}$. In addition to finding roots of general, nonlinear functions, we can also use the Newton method to find stationary points, which are candidates for local extrema, by approximating the roots of the derivative of a given function. The second use is extended in the following section, Sect. 12.8, where we will discuss a generalization of the Newton method for nonlinear, multidimensional optimization.

The Newton method works iteratively and is best summarized as replacing the function by its tangent at the current approximation x_n of the root and then using the root of the tangent, which is a linear function, as the next approximation x_{n+1}.

In more detail, we start from a differentiable function $g\colon [a, b] \to \mathbb{R}$ and an initial approximation x_n of a root. First, we write the tangent $y\colon [a, b] \to \mathbb{R}$ of g at the point $(x_n, g(x_n))$ as the linear function

$$y(x) := g'(x_n)(x - x_n) + g(x_n).$$

This formula is easily checked by noting that y is a linear function, that its has the correct slope $g'(x_n)$, and that it takes the correct value $y(x_n) = g(x_n)$ of the tangent at the point $(x_n, g(x_n))$.

The next approximation x_{n+1} is the root of the tangent y and is hence found by solving the equation

$$g'(x_n)(x_{n+1} - x_n) + g(x_n) = 0 \tag{12.16}$$

for x_{n+1}. This yields the next approximation x_{n+1} as

$$x_{n+1} := x_n - \frac{g(x_n)}{g'(x_n)}. \tag{12.17}$$

Unless x_n is a stationary point, this fraction is defined. These considerations yield the following algorithm.

Algorithm 12.27 (Newton method, Newton iteration)

1. Initialize the counter $n := 0$ and choose a starting value x_0 sufficiently close to a root of the given differentiable function $g\colon [a,b] \to \mathbb{R}$ (e.g. by performing a global optimization first).
2. Repeat until the difference $|x_{n+1} - x_n|$ or the residuum $|g(x_{n+1}) - g(x_n)|$ is sufficiently small (i.e., smaller than a prescribed value) or the maximum number of iterations has been reached.

   a. Define
   $$x_{n+1} := x_n - \frac{g(x_n)}{g'(x_n)}.$$
   b. Increase the counter $n := n + 1$.

The main appeal of the Newton method is that it converges quadratically to a root provided that the function is smooth enough and that the starting point is sufficiently close to a root.

Theorem 12.28 (quadratic convergence of the Newton method) *Suppose $\xi \in \mathbb{R}$, $x_0 \in \mathbb{R}$, and $r \ge |x_0 - \xi|$, and define the interval $I := [\xi - r, \xi + r] \subset \mathbb{R}$. Suppose further that $\xi \in \mathbb{R}$ is a root of a function $g\colon I \to \mathbb{R}$, that $g'(x) \neq 0$ for all $x \in I$, and that $g''(x)$ is continuous for all $x \in I$. If the starting point $x_0 \in I$ of the Newton iteration is sufficiently close to the root $\xi \in I$, then the sequence $\langle x_n \rangle$ converges* quadratically *to the root ξ.*

Proof The first-order Taylor expansion of $g(\xi)$ around x_n exists by the assumptions on the function g and can be written as

$$g(\xi) = g(x_n) + g'(x_n)(\xi - x_n) + \frac{1}{2!}g''(\xi_n)(\xi - x_n)^2,$$

where the last term is the Lagrange form of the remainder and ξ_n lies between the root ξ and x_n.

Since ξ is a root and the first derivative g' does not vanish by assumption, dividing the Taylor expansion by $g'(x_n)$ yields

$$\frac{g(x_n)}{g'(x_n)} + \xi - x_n = -\frac{g''(\xi_n)}{2g'(x_n)}(\xi - x_n)^2.$$

Using the definition (12.17) of x_{n+1} yields

$$\xi - x_{n+1} = -\frac{g''(\xi_n)}{2g'(x_n)}(\xi - x_n)^2.$$

We denote the error in the n-th step by

$$e_n := x_n - \xi.$$

Using this definition, we have just shown that the relationship

$$|e_{k+1}| = \frac{|g''(\xi_n)|}{2|g'(x_n)|} e_n^2$$

holds between the errors e_n and e_{n+1} in steps n and $n+1$, which implies quadratic convergence due to the assumptions on g' and g'' if the starting point x_0 is sufficiently close to the root ξ. □

In the multidimensional case of vector-valued functions $\mathbf{g}\colon \mathbb{R}^d \to \mathbb{R}^d$, the condition (12.16) that the tangent vanishes becomes

$$J_{\mathbf{g}}(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) + \mathbf{g}(\mathbf{x}_n) = \mathbf{0},$$

where $J_{\mathbf{g}}(\mathbf{x}_n)$ is the Jacobi matrix of $\mathbf{g}$ at the point $\mathbf{x}_n$. Although the next approximation $\mathbf{x}_{n+1}$ can be written concisely as

$$\mathbf{x}_{n+1} := \mathbf{x}_n - J_{\mathbf{g}}^{-1}(\mathbf{x}_n)\mathbf{g}(\mathbf{x}_n),$$

it is much more computationally efficient not to calculate the inverse $J_{\mathbf{g}}^{-1}(\mathbf{x}_n)$ of the Jacobi matrix, but to solve the linear system

$$J_{\mathbf{g}}(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) = -\mathbf{g}(\mathbf{x}_n) \tag{12.18}$$

for $\mathbf{x}_{n+1} - \mathbf{x}_n$ and hence for $\mathbf{x}_{n+1}$ (see Sect. 8.4.8).

Naturally, we would like to take advantage of the quadratic convergence of the Newton method for optimizing functions as well. As mentioned above, we can use the Newton method for approximating roots of the derivative of a function to optimize. To find these stationary points, we hence need the second derivative. In higher dimensions, i.e., when optimizing functions $f\colon \mathbb{R}^d \to \mathbb{R}$, this means that we need the Hessian matrix of the function.

Obviously, calculating the $d \times d$ Hessian matrix is computationally expensive in higher dimensions. In order to circumvent this problem, quasi-Newton methods have been developed. In quasi-Newton methods, the Hessian matrix of the second derivatives is not evaluated directly, but only approximated. One of the most prominent quasi-Newton methods is BFGS, discussed in the next section.

12.8 The BFGS Method

The BFGS method is one of the most popular quasi-Newton methods and named after Charles G. Broyden, Roger Fletcher, Donald Goldfarb, and David Shanno [3, 4, 6, 8, 17, 18]. As a quasi-Newton method it is iterative and suitable for nonlinear optimization problems. Although it does not handle constraints in its original form, a version for box constraints has been developed [5].

In this section, the problem is to minimize a real differentiable scalar function $f\colon \mathbb{R}^d \to \mathbb{R}$ without constraints. We denote the starting point of the iteration by $\mathbf{x}_0$ and the approximation in iteration n by $\mathbf{x}_n$. If $H_f(\mathbf{x}_n)$ denotes the Hessian matrix of f at the point $\mathbf{x}_n$, the step

$$\mathbf{s}_n := \mathbf{x}_{n+1} - \mathbf{x}_n$$

in iteration n is found by solving the linear system

$$H_f(\mathbf{x}_n)\mathbf{s}_n = -\nabla f(\mathbf{x}_n) \tag{12.19}$$

because of the multidimensional Newton iteration (12.18), where we know calculate roots of the function

$$\mathbf{g} := \nabla f.$$

Calculating roots of the gradient $\mathbf{g} = \nabla f$ of the function f is – for the purposes of this section – equivalent to minimizing the quadratic approximation

$$f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \mathbf{s}_n) \approx f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n) \cdot \mathbf{s}_n + \frac{1}{2!}\mathbf{s}_n^\top H_f(\mathbf{x}_n)\mathbf{s}_n$$

of f, which is a truncated Taylor expansion (see Theorem 12.5). Computing the gradient of this approximation with respect to $\mathbf{s}_n$ and setting it to zero yields again (12.19).

But now we make two important changes. First, we do not calculate the Hessian matrix $H_f(\mathbf{x}_n)$ in each step, which would be very computationally expensive, but we will calculate approximations

$$H_n \approx H_f(\mathbf{x}_n)^{-1}$$

instead as discussed in detail below. The structure of the approximations H_n will make it possible to find the inverses H_n^{-1} explicitly so that the linear system resulting from (12.19) can be solved efficiently by just multiplying H_n^{-1} with a vector. Another advantage is that no second derivatives are ever used.

Second, we do not use $\mathbf{s}_n$ as the step $\mathbf{s}_n = \mathbf{x}_{n+1} - \mathbf{x}_n$, but we only consider $\mathbf{s}_n$ as the search direction, i.e.,

$$\alpha_n\mathbf{s}_n = \mathbf{x}_{n+1} - \mathbf{x}_n,$$

because we only approximate the Hessian matrix, and we use line search to make up for this approximation. Having thus determined the search direction $\mathbf{s}_n$, any line-search method (see Sect. 12.6) can be employed to find the next approximation $\mathbf{x}_{n+1}$ by minimizing the objective function $f(\mathbf{x}_{n+1}) = f(\mathbf{x}_n + \alpha_n\mathbf{s}_n)$ in the search direction $\mathbf{s}_n$ over the scalar $\alpha \in \mathbb{R}^+$, i.e.,

$$\alpha_n := \arg\min_{\alpha\in\mathbb{R}^+} f(\mathbf{x}_n + \alpha\mathbf{s}_n),$$
$$\mathbf{x}_{n+1} := \mathbf{x}_n + \alpha_n\mathbf{s}_n.$$

How are the approximations H_n of the Hessian matrices calculated? We impose the two conditions that H_n is symmetric (as all Hessian matrices are as long as the function is sufficiently smooth) and that H_n is positive definite (as all Hessian matrices are at a local minimum, see Theorem 12.7). If H_n is positive definite, this property also implies the so-called descent property

$$\mathbf{g}(\mathbf{x}_n) \cdot \mathbf{s}_n < 0 \tag{12.20}$$

(cf. Sect. 12.1). Equation (12.19) becomes $\alpha_n \mathbf{s}_n = -H_n \mathbf{g}(\mathbf{x}_n)$ for the new step $\alpha_n \mathbf{s}_n$ (instead of $\mathbf{s}_n$), implying that

$$\alpha_n \mathbf{g}(\mathbf{x}_n) \cdot \mathbf{s}_n = -\mathbf{g}(\mathbf{x}_n)^\top H_n \mathbf{g}(\mathbf{x}_n) < 0$$

whenever $\mathbf{g}(\mathbf{x}_n) \neq \mathbf{0}$, since H_n is positive definite. This inequality implies the descent property (12.20) because $\alpha_n > 0$.

A third condition we impose on the approximations H_n stems from the following consideration. Taylor expansion of $\mathbf{g}$ at $\mathbf{x}_n$ yields

$$\mathbf{g}(\mathbf{x}_{n+1}) = \mathbf{g}(\mathbf{x}_n) + H_f(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) + O(\|\mathbf{x}_{n+1} - \mathbf{x}_n\|^2).$$

The higher order terms in $O(\|\mathbf{x}_{n+1} - \mathbf{x}_n\|^2)$ vanish for quadratic functions f and we will neglect them, meaning that the following construction will be exact for quadratic functions. This yields

$$\begin{aligned} \mathbf{d}_n &:= \mathbf{x}_{n+1} - \mathbf{x}_n, \\ \mathbf{y}_n &:= \mathbf{g}(\mathbf{x}_{n+1}) - \mathbf{g}(\mathbf{x}_n) = \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n), \\ H_n \mathbf{d}_n &= \mathbf{y}_n, \end{aligned}$$

but the matrix H_n does usually not satisfy this equation, since $\mathbf{d}_n$ and hence $\mathbf{y}_n$ are only known after the line search is complete. Therefore the next approximation H_{n+1} is chosen such that it satisfies the updated equation

$$H_{n+1} \mathbf{d}_n = \mathbf{y}_n, \tag{12.21}$$

which is called the quasi-Newton condition.

Often simple initial values such as the identity matrix or multiples thereof are used for H_0, and they clearly satisfy these two conditions.

The next value H_{n+1} is calculated as an update of H_n. At this point it is clear that the updating formula, which enables H_{n+1} to be calculated from H_n, is crucial. It has to incorporate as much information on the second derivatives into H_{n+1} as possible, and its repeated application should change the arbitrary initial matrix H_0 into a close approximation of the Hessian, while only requiring few computations. It also should satisfy these three conditions.

Perhaps the simplest possible way is to define

$$H_{n+1} := H_n + a\mathbf{u}\mathbf{u}^\top, \qquad a \in \mathbb{R}, \tag{12.22}$$

where the symmetric rank-one matrix $a\mathbf{u}\mathbf{u}^\top$ is added to the previous matrix. The quasi-Newton condition (12.21) can be satisfied by setting $\mathbf{u} := \mathbf{y}_n - H_n\mathbf{d}_n$ and requiring that $a\mathbf{u}^\top\mathbf{d}_n = 1$ (see Problem 12.13), although these updates are usually applied to approximating the inverses $H_f(\mathbf{x}_N)^{-1}$ directly. More importantly, however, they have serious disadvantages [7, Section 3.2]: positive definiteness of the approximate matrices cannot be guaranteed and the denominator in the resulting update formula may become zero.

A better approach are rank-two updates, which have the form

$$H_{n+1} := H_n + a\mathbf{u}\mathbf{u}^\top + b\mathbf{v}\mathbf{v}^\top, \qquad a, b \in \mathbb{R}, \tag{12.23}$$

in general. Substituting this form of H_{n+1} into the quasi-Newton condition (12.21), setting $\mathbf{u} := \mathbf{y}_n$ and $\mathbf{v} := H_n\mathbf{d}_n$, and requiring that $a\mathbf{u}^\top\mathbf{d}_n = 1$ and $b\mathbf{v}^\top\mathbf{d}_n = 1$ yields the BFGS update

$$H_{n+1} = H_n + \frac{\mathbf{y}_n\mathbf{y}_n^\top}{\mathbf{y}_n^\top\mathbf{d}_n} - \frac{H_n\mathbf{d}_n\mathbf{d}_n^\top H_n^\top}{\mathbf{d}_n^\top H_n\mathbf{d}_n} \tag{12.24}$$

(see Problem 12.14).

Its inverse is

$$H_{n+1}^{-1} = \left(I - \frac{\mathbf{d}_n\mathbf{y}_n^\top}{\mathbf{y}_n^\top\mathbf{d}_n}\right) H_n^{-1} \left(I - \frac{\mathbf{y}_n\mathbf{d}_n^\top}{\mathbf{y}_n^\top\mathbf{d}_n}\right) + \frac{\mathbf{d}_n\mathbf{d}_n^\top}{\mathbf{y}_n^\top\mathbf{d}_n} \tag{12.25}$$

(see Problem 12.15). Expanding and noting that $\mathbf{y}_n^\top H_n^{-1}\mathbf{y}_n$ is a scalar yields the equivalent form

$$H_{n+1}^{-1} = H_n^{-1} - \frac{\mathbf{d}_n\mathbf{y}_n^\top H_n^{-1} + H_n^{-1}\mathbf{y}_n\mathbf{d}_n^\top}{\mathbf{y}_n^\top\mathbf{d}_n} + \frac{(\mathbf{y}_n^\top\mathbf{d}_n + \mathbf{y}_n^\top H_n^{-1}\mathbf{y}_n)\mathbf{d}_n\mathbf{d}_n^\top}{(\mathbf{y}_n^\top\mathbf{d}_n)^2},$$

which has the advantage that it can be evaluated more efficiently.

We imposed three conditions on the approximations H_n of the Hessian matrices: that they are symmetric and positive definite and that they satisfy the quasi-Newton condition (12.21). Now is a good time to check that these three conditions are indeed satisfied: symmetric and positive definiteness are checked in Problem 12.16 and Problem 12.17, and the quasi-Newton condition is obviously satisfied by the construction of H_{n+1}. In summary, we have derived the BFGS algorithm.

Algorithm 12.29 (BFGS)

1. Initialize the starting point $\mathbf{x}_0$ and the initial approximation $H_0^{-1} := I$ of the inverse of the Hessian matrix.

2. Repeat:
   a. Calculate the search direction
   $$\mathbf{s}_n := -H_n^{-1}\nabla f(\mathbf{x}_n).$$
   b. Perform a line search to find the step size α_n as
   $$\alpha_n := \arg\min_{\alpha\in\mathbb{R}^+} f(\mathbf{x}_n + \alpha\mathbf{s}_n),$$
   where α_n is chosen to satisfy the Wolfe conditions (12.14) and (12.15).
   c. Calculate
   $$\begin{aligned}\mathbf{x}_{n+1} &:= \mathbf{x}_n + \alpha_n\mathbf{s}_n,\\ \mathbf{d}_n &:= \mathbf{x}_{n+1} - \mathbf{x}_n,\\ \mathbf{y}_n &:= \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n).\end{aligned}$$
   d. Calculate H_{n+1}^{-1} using (12.25).
   e. Repeat until convergence, i.e., until a norm of $\nabla f(\mathbf{x}_{n+1})$ has become sufficiently small or a norm of $\mathbf{x}_{n+1} - \mathbf{x}_n$ has become sufficiently small.
3. Return the approximation $\mathbf{x}_n$ of a minimum.

The BFGS algorithm shows superlinear convergence. A convergence analysis can be found, e.g., in [16, Section 6.4].

Finally, we note that the same BFGS update formula for the inverse of the Hessian matrix can also be found in a different way. We again require that the next approximation of the matrix is symmetric and that it satisfies the quasi-Newton condition (12.21). The new main requirement is that we choose the next approximation H_{n+1}^{-1} as the (of course regular) matrix H^{-1} that is closest to the previous approximation H_n^{-1}, i.e.,

$$\begin{aligned}H_{n+1}^{-1} := &\min_{H^{-1}\in\mathbb{R}^{d\times d}} \|H^{-1} - H_n^{-1}\|_W^2\\ &\text{subject to } H^{-1}\mathbf{d}_n = \mathbf{y}_n \text{ and } H^{-1} = (H^{-1})^\top,\end{aligned} \tag{12.26}$$

where the norm is a weighted Frobenius norm [16, Section 6.1] (see Problem 12.20). More precisely, it is a weighted Frobenius norm

$$\|A\|_W := \|W^{1/2}AW^{1/2}\|_\mathrm{F}$$

whose weight matrix W is any positive definite matrix that satisfies $W\mathbf{d}_n = \mathbf{y}_n$; the Frobenius norm is defined as

$$\|A\|_{\mathrm{F}} := \sum_{i=1}^{d} \sum_{j=1}^{d} a_{ij}^2.$$

The update for H_{n+1} can be formulated as an analogous minimization problem.

12.9 The L-BFGS (Limited-Memory BFGS) Method

Quasi-Newton methods such as the BFGS algorithm in the previous section provide superlinear convergence while never calculating second derivatives explicitly. Their remaining disadvantage when applied to large optimization problems is that the whole approximate Hessian matrix H_n^{-1} is stored during the iteration, meaning that the memory requirement increases quadratically with the number of variables.

In order to make the BFGS algorithm amenable to large optimization problems, the limited-memory BFGS (L-BFGS) method has been developed [15, 10, 11]. In the L-BFGS algorithm, not the whole matrix H_n^{-1} is stored and manipulated, which would be prohibitive when the number of variables is large, but only a few vectors which suffice to calculate the recursion (12.25) only for products $H_n^{-1}\mathbf{q}$ [16, Section 7.2]. Furthermore, not the whole recursion starting from H_0^{-1} is calculated, but only a fixed number of previous steps is used. Thus the number of vectors stored is fixed and often relatively small, and hence the amount of memory required is only linear in the number of variables.

The key to understanding the L-BFGS algorithm is the efficient way of calculating only the products $H_n^{-1}\mathbf{q}$ in Algorithm 12.30. We start by summarizing the recursion (12.25) for the approximations H_n^{-1} of the inverses of the Hessian matrices as

$$\begin{aligned} H_{n+1}^{-1} &:= V_n^\top H_n^{-1} V_n + \rho_n \mathbf{d}_n \mathbf{d}_n^\top, \\ V_n &:= I - \rho_n \mathbf{y}_n \mathbf{d}_n^\top, \\ \rho_n &:= \frac{1}{\mathbf{y}_n^\top \mathbf{d}_n}. \end{aligned}$$

Expanding the iteration yields the formula

$$\begin{aligned} H_n^{-1} = &(V_{n-1}^\top \cdots V_0^\top) H_0^{-1} (V_0 \cdots V_{n-1}) \\ &+ \rho_0 (V_{n-1}^\top \cdots V_1^\top) \mathbf{d}_0 \mathbf{d}_0^\top (V_1 \cdots V_{n-1}) \\ &+ \rho_1 (V_{n-1}^\top \cdots V_2^\top) \mathbf{d}_1 \mathbf{d}_1^\top (V_2 \cdots V_{n-1}) \\ &+ \cdots \\ &+ \rho_{n-1} \mathbf{d}_{n-1} \mathbf{d}_{n-1}^\top \end{aligned} \tag{12.27}$$

(see Problem 12.21).

How is this formula used? In each iteration n, we only store the last m vectors $\mathbf{d}_i$ and $\mathbf{y}_i$ ($i \in \{n-m, \dots, n-1\}$). This leads to the linear memory requirement. In each iteration n, the initial approximation $H_{n,0}^{-1}$ of the inverse of the Hessian matrix is allowed to vary – in contrast to the BFGS algorithm – and is chosen. A formula for choosing $H_{n,0}^{-1}$ that has proved effective is to define

$$H_{n,0}^{-1} := \frac{\mathbf{d}_{n-1}^\top \mathbf{y}_{n-1}}{\mathbf{y}_{n-1}^\top \mathbf{y}_{n-1}} I. \tag{12.28}$$

The scaling factor in front of the identity matrix estimates the size of the true Hessian matrix along the most recent search direction, which helps to keep the scale of the search direction correct and hence a step length of one is accepted in most iterations. Based on the starting value and using only the last m vectors, the expanded iteration becomes

$$\begin{aligned} H_n^{-1} = {} & (V_{n-1}^\top \cdots V_{n-m}^\top) H_{n,0}^{-1} (V_{n-m} \cdots V_{n-1}) \\ & + \rho_{n-m} (V_{n-1}^\top \cdots V_{n-m+1}^\top) \mathbf{d}_{n-m} \mathbf{d}_{n-m}^\top (V_{n-m+1} \cdots V_{n-1}) \\ & + \rho_{n-m+1} (V_{n-1}^\top \cdots V_{n-m+2}^\top) \mathbf{d}_{n-m+1} \mathbf{d}_{n-m+1}^\top (V_{n-m+2} \cdots V_{n-1}) \\ & + \cdots \\ & + \rho_{n-1} \mathbf{d}_{n-1} \mathbf{d}_{n-1}^\top. \end{aligned} \tag{12.29}$$

This formula yields the following algorithm (see Problem 12.22).

Algorithm 12.30 (L-BFGS two-loop recursion) Input: $\mathbf{v} \in \mathbb{R}^d$, initial matrix approximation $H_{n,0}^{-1}$ (for example (12.28)), length $m \in \mathbb{N}$ of history, vectors $\mathbf{d}_i \in \mathbb{R}^d$ and $\mathbf{y}_i \in \mathbb{R}^d$ ($i \in \{n-m, \dots, n-1\}$). Output: $H_n^{-1}\mathbf{q}$.

```
1. Define $\mathbf{q} := \mathbf{v}$.
2. For $i := n-1, n-2, \dots, n-m$:
    a. Set $\alpha_i := \rho_i \mathbf{d}_i^\top \mathbf{q}$.
    b. Set $\mathbf{q} := \mathbf{q} - \alpha_i \mathbf{y}_i$.
3. Set $\mathbf{r} := H_{n,0}^{-1}\mathbf{q}$.
4. For $i := n-m, n-m+1, \dots, n-1$:
    a. Set $\beta := \rho_i \mathbf{y}_i^\top \mathbf{r}$.
    b. Set $\mathbf{r} := \mathbf{r} + (\alpha_i - \beta)\mathbf{d}_i$.
5. Return $\mathbf{r}$, which is equal to $H_n^{-1}\mathbf{q}$.
```

In the first loop, the factors to the right of $H_{n,0}^{-1}$ are calculated. In the second loop, the multiplications on the left are performed, and the terms are added. Note that this approach works only for the products $H_n^{-1}\mathbf{q}$, but not to calculate the whole matrices H_n^{-1}.

Assembling all pieces, we arrive at the BFGS algorithm (cf. Algorithm 12.29).

Algorithm 12.31 (L-BFGS) Input: starting point $\mathbf{x}_0$, length $m \in \mathbb{N}$ of history. Output: approximation $\mathbf{x}_n$ of minimum.

1. Initialize the starting point $\mathbf{x}_0$.
2. Initialize the counter $n := 0$.
3. Repeat:

   a. Choose the initial approximation $H_{n,0}^{-1}$ of the inverse of the Hessian matrix, e.g., by using (12.28).
   b. Calculate the search direction

   $$\mathbf{s}_n := -H_n^{-1} \nabla f(\mathbf{x}_n)$$

   using Algorithm 12.30.
   c. Perform a line search to find the step size α_n as

   $$\alpha_n := \arg\min_{\alpha \in \mathbb{R}^+} f(\mathbf{x}_n + \alpha \mathbf{s}_n),$$

   where α_n is chosen to satisfy the Wolfe conditions (12.14) and (12.15).
   d. If $n > m$, free the memory for $\mathbf{d}_{n-m}$ and $\mathbf{y}_{n-m}$.
   e. Calculate

   $$\begin{aligned} \mathbf{x}_{n+1} &:= \mathbf{x}_n + \alpha_n \mathbf{s}_n, \\ \mathbf{d}_n &:= \mathbf{x}_{n+1} - \mathbf{x}_n, \\ \mathbf{y}_n &:= \nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n). \end{aligned}$$

   f. Increase the counter $n := n + 1$.
   g. Repeat until convergence.

4. Return the approximation $\mathbf{x}_n$ of a minimum.

The fact that L-BFGS only uses recent information and discards older iterations in contrast to BFGS, which uses information from all previous iterations, can be viewed as an advantage. While L-BFGS is significantly faster than BFGS when the number of variables is large and can perform nearly as well, the optimal choice of the parameter m in the algorithm depends on the problem class, which is a disadvantage of L-BFGS. When L-BFGS fails, one should therefore try to increase the parameter m.

Many variants of the L-BFGS method have been developed. For example, the limited-memory algorithm in [5, 21, 12] can solve optimization problems with simple bounds.

12.10 JULIA Packages

Many optimization packages are available under the `JuliaOpt` and `JuliaNLSolvers` umbrellas. The `Optim` package provides software for univariate and multivariate optimization written in JULIA with a focus on unconstrained optimization. The `LBFGSB` package is a JULIA wrapper for the L-BFGS-B nonlinear optimization code.

12.11 Bibliographical Remarks

A well written introduction to convex optimization with many applications is [2]. An in-depth treatment of convergence analysis in convex optimization can be found in [14]. Methods of unconstrained optimization (including quasi-Newton like methods such as BFGS) and constrained optimization as well as their practical aspects are discussed in [7]. Another excellent overview of optimization methods is [16].

Problems

12.1 Prove Theorem 12.5.

12.2 Prove Theorem 12.7.

12.3 Prove Corollary 12.12.

12.4 Prove Lemma 12.14.

12.5 Calculate the integral in (12.7) and plot the sum and its approximate upper bound used in the proof of Corollary 12.20.

12.6 $*$ Prove Theorem 12.21.

12.7 Prove (12.13).

12.8 Implement Algorithm 12.22.

12.9 Apply Algorithm 12.22 to a benchmark problem in Sect. 11.8 using a starting point you have found by global optimization.

12.10 Explain the meaning of the end points of the intervals $c_1 \in (0, 1)$ and $c_2 \in (c_1, 1)$ in the Wolfe conditions (12.14) and (12.15).

12.11 Implement Algorithm 12.26, which satisfies the Wolfe conditions (12.14) and (12.15).

12.12 Apply Algorithm 12.26 to a benchmark problem in Sect. 11.8 using a starting point you have found by global optimization.

12.13 Derive a rank-one update formula by substituting the definition (12.22) of H_{n+1} into the quasi-Newton condition (12.21) and following the text.

12.14 Prove the rank-two update formula (12.24) by substituting the definition (12.23) of H_{n+1} into the quasi-Newton condition (12.21) and following the text.

12.15 Prove (12.25) by showing that $H_{n+1}^{-1}H_{n+1} = I$ or by showing that $H_{n+1}H_{n+1}^{-1} = I$.

12.16 Prove: Suppose that H_n is symmetric; then H_{n+1} defined in (12.24) is symmetric as well.

12.17 ∗ Prove: Suppose that H_n is symmetric and positive definite and that $\mathbf{y}_n^\top \mathbf{d}_n > 0$; then H_{n+1} defined in (12.24) is positive definite as well.

12.18 Implement Algorithm 12.29.

12.19 Apply Algorithm 12.29 to a benchmark problem in Sect. 11.8 using a starting point you have found by global optimization.

12.20 ∗ Prove that the unique solution of the minimization problem (12.26) is the BFGS update (12.25).

12.21 Prove that (12.27) holds for H_n^{-1} defined by (12.25).

12.22 Prove that (12.29) yields Algorithm 12.30.

12.23 Implement Algorithm 12.30.

12.24 Implement Algorithm 12.31.

12.25 Apply Algorithm 12.31 to a benchmark problem in Sect. 11.8 using a starting point you have found by global optimization.

12.26 Compare Algorithm 12.29 and Algorithm 12.31 for different values of the parameter m.

References

1. Beck, A., Teboulle, M.: A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Imaging Sciences* **2**(1), 183–202 (2009)
2. Boyd, S., Vandenberghe, L.: *Convex Optimization*. Cambridge University Press, Cambridge, UK (2004)
3. Broyden, C.: The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics* **6**(1), 76–90 (1970)

4. Broyden, C.: The convergence of a class of double-rank minimization algorithms 2. the new algorithm. *IMA Journal of Applied Mathematics* **6**(3), 222–231 (1970)
5. Byrd, R., Lu, P., Nocedal, J., Zhu, C.: A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.* **16**(5), 1190–1208 (1995)
6. Fletcher, R.: A new approach to variable metric algorithms. *The Computer Journal* **13**(3), 317–322 (1970)
7. Fletcher, R.: *Practical Methods of Optimization*, 2nd edn. John Wiley and Sons (2000)
8. Goldfarb, D.: A family of variable-metric updates derived by variational means. *Mathematics of Computation* **24**(109), 23–26 (1970)
9. Lewis, A., Overton, M.: Nonsmooth optimization and quasi-Newton methods. *Math. Programming* **141**(1-2), 135–163 (2013)
10. Liu, D., Nocedal, J.: On the limited memory BFGS method for large scale optimization. *Math. Programming* **45**(1–3), 503–528 (1989)
11. Mokhtari, A., Ribeiro, A.: Global convergence of online limited memory BFGS. Journal of Machine Learning Research **16**(Dec), 3151–3181 (2015)
12. Morales, J., Nocedal, J.: Remark on "algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization". *ACM Transactions on Mathematical Software (TOMS)* **38**(1), article no. 7 (2011)
13. Nesterov, Y.: A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Dokl. Akad. Nauk SSSR* **269**, 543–547 (1983)
14. Nesterov, Y.: *Lectures on Convex Optimization*, 2nd edn. Springer (2018)
15. Nocedal, J.: Updating quasi-Newton matrices with limited storage. *Mathematics of Computation* **35**(151), 773–782 (1980)
16. Nocedal, J., Wright, S.: *Numerical Optimization*, 2nd edn. Springer (2006)
17. Shanno, D.: Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation* **24**(111), 647–656 (1970)
18. Shanno, D., Kettler, P.: Optimal conditioning of quasi-Newton methods. *Mathematics of Computation* **24**(111), 657–664 (1970)
19. Wolfe, P.: Convergence conditions for ascent methods. *SIAM Review* **11**(2), 226–235 (1969)
20. Wolfe, P.: Convergence conditions for ascent methods. II: some corrections. *SIAM Review* **13**(2), 185–188 (1971)
21. Zhu, C., Byrd, R., Lu, P., Nocedal, J.: Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)* **23**(4), 550–560 (1997)

Part IV
Algorithms for Machine Learning

Chapter 13
Neural Networks

Mama always said "son, if you had a brain, you'd be dangerous,"
guess it pays to be brainless.

—Eminem, Song *Brainless, The Marshall Mathers LP 2* (2013)

Abstract Artificial neural networks were first conceived decades ago and have been an important part of machine learning and artificial intelligence ever since. The basic idea behind neural networks is to combine linear combinations and nonlinear functions into layers such that they can approximate arbitrary functions. Neural networks have been used to great effect for example in image classification and recognition once the computational power and suitable algorithms to train large neural networks had become available. In this chapter, we implement a neural network and train it using backpropagation in a fully self-contained program. Using tens of thousands of scanned images, we train the neural network to recognize handwritten digits and discuss important aspects of training neural networks.

13.1 Introduction

The idea underlying an artificial neural network is to define a function taking a vector as the input and yielding a vector as the output as shown in Fig. 13.1. The circles indicate real numbers, the arrows indicate dependencies, and the rectangles indicate vectors. Between the input and output vectors (or layers in neural-network parlance), hidden layers are arranged such that the input layer influences only the first hidden layer, each hidden layer influences only the next hidden layer, and only the last hidden layer influences the output layer.

Much of the analogy of artificial neural networks obviously stems from the neurons and axons in the nervous system and the brain, although the dependencies or connections implemented by axons are in general much more com-

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_13

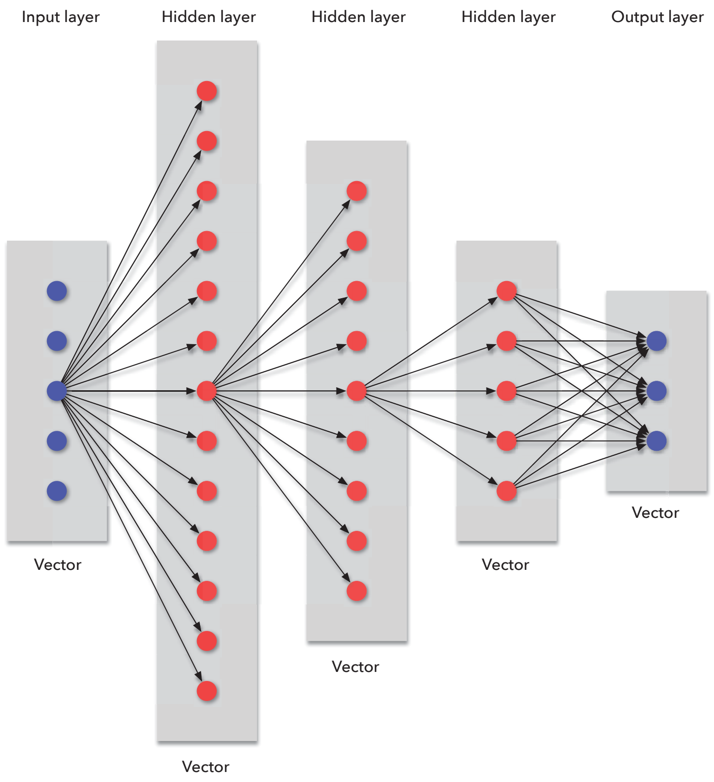

Fig. 13.1 Schematic diagram of an artificial neural network. Not all dependencies between adjacent layers are shown in order not to overload the figure; in general, each neuron influences each neuron in the next layer.

plicated (and still mostly unknown) than the ones between adjacent layers indicated in Fig. 13.1.

More general dependencies are of course possible in artificial neural networks and have been investigated. The computational reason why an arrangement in layers is preferable is that it makes training the neural network (see Sections 13.6 and 13.7) much easier than in the general case when circular dependencies are allowed.

Neural networks have become a vast field with many applications in supervised machine learning. Supervised learning means that the input cases and the output cases are known and that their (possibly highly complicated) functional relationship is to be learned while the data may be noisy. Since the arrangement in Fig. 13.1 resembles some structures in the visual cortex of the brain, it may be unsurprising that large enough neural networks with a certain number and structure of the hidden layers have been used very successfully in image-classification and image-recognition tasks. The field of deep learning is concerned with deep neural networks, i.e., those with many hidden layers.

In the following we study the structure of neural networks in more detail, state an important result from the theory of neural networks, learn how it is possible to train neural networks, and consider the example of handwriting recognition as a practical and nontrivial example. The JULIA code in this chapter implements neural networks of the form shown in Fig. 13.1 and is used to solve the handwriting-recognition problem.

13.2 Feeding Forward

Before we can evaluate the function given by a neural network, we still must specify the neural network more precisely. We denote the vector-valued output of layer $k-1$ by $\mathbf{x}^{(k-1)}$. The action of layer k is given by applying a linear (or more precisely, an affine) function and then a nonlinear function elementwise. The vector output $\mathbf{x}^{(k)}$ of layer k can therefore be written as

$$\mathbf{x}^{(k)} := \sigma(W^{(k)}\mathbf{x}^{(k-1)} + \mathbf{b}^{(k)}),$$

where the matrix $W^{(k)}$ is called the weight matrix, the vector $\mathbf{b}^{(k)}$ contains the so-called biases, and the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is applied elementwise to its argument.

It is an important feature of neural networks that the activation function is nonlinear. Otherwise, if it were linear, the whole network would just be a linear function as a composition of linear functions. There are two main considerations important for the choice of the activation function: first, it should facilitate the kind of functional relationship between the known input and output vectors of the learning problem at hand, and secondly, it should be expedient for the computations required for learning (see Sections 13.6 and 13.7).

The first requirement is generally hard to satisfy a priori and without any experience or experimentation. The question of the shape of the output layer in the handwriting-recognition problem is an example of such considerations and is discussed below.

The second requirement is especially important for deep neural networks, i.e., for networks with many hidden layers, and is discussed using the example of the choice of the activation function σ in the following.

Various popular choices of activation functions are the following. The first is the sigmoid function

$$\sigma_1(x) := \frac{1}{1+\mathrm{e}^{-x}},$$

also called the logistic function. The second is the leaky rectifier

$$\sigma_2(x) := \begin{cases} x, & x \geq 0, \\ \alpha x, & x < 0, \end{cases}$$

where α is a small parameter, e.g., $\alpha = 1/10$ or $\alpha = 1/100$. It is the leaky version of the rectifier

$$\sigma_3(x) := \max(x, 0).$$

An example of a smooth rectifier is the function

$$\sigma_4(x) := \ln(1 + \exp(x)).$$

A final example of an activation function is the hyperbolic tangent

$$\sigma_5(x) := \tanh(x).$$

Basic properties of these five functions are investigated in Problem 13.1. If an activation function is differentiable only almost everywhere, this fact does not pose a problem in practice; we can just use the limit from the left or from the right instead at any points where the derivative is not defined.

As seen in Problem 13.1, these activation functions and their derivatives differ substantially regarding their behavior away from zero. For example, the derivative of the sigmoid function becomes small away for zero, which impedes learning by gradient descent in deep networks, since small derivatives are multiplied, resulting in tiny gradients and hence slow learning (see Sections 13.6 and 13.7). The leaky rectifier σ_2 circumvents this problem and still enables learning for $x < 0$ – in contrast to the rectifier σ_3 – unless the parameter α is too small.

The output of all these activation functions is real-valued. Therefore, when performing regression using a neural network, the output layer provides the result. In classification problems, however, the output of the network must be one of the discrete classes, and hence the output must be discretized. If the number of classes is small, it is convenient to discretize the output of the activation function of a single neuron in the output layer. For example, if the activation function is σ_1 and there are two classes, then the discretization $([0, 1/2], (1/2, 1])$ suggests itself; if the activation function is tanh and there are three classes, then the discretization $([-1, -1/3], (-1/3, 1/3], (1/3, 1])$ suggests itself.

On the other hand, if the number of classes is relatively large, then it is much better to assign a class to each neuron in the output layer and to consider the index of the neuron with the largest value of the activation function to be the classification of the input vector. This is what the output layer in our example of recognizing handwritten digits looks like: there are ten neurons in the output layer and the index (minus one) of the neuron that fires most is the classification of the input image.

Neural networks are differentiable functions. It is an important feature that discrete problems such as classification problems are formulated such that learning becomes optimization of a differentiable function, which is much easier to solve computationally than a discrete optimization problem.

It is of course also possible to use different activation functions on different layers and to use layers with different connections between the neurons; these possibilities have indeed been pursued to great advantage in designing neural networks, e.g., in convolutional neural networks.

Having discussed these design considerations, we start to implement neural networks. We define a module called `NN` and a data structure `Activation`. Its field `f` contains the activation function, and its field `d` its derivative. The package `MLDatasets` contains sample data sets for machine learning, the package `Printf` provides facilities for printing numbers with a given precision, and the package `PyPlot` is one of the most popular plotting packages.

```
module NN

import GZip
import LinearAlgebra
import MLDatasets
import Printf
import PyPlot
import Random

struct Activation
    f::Function
    d::Function
end

function sigma(x)
    1/(1+exp(-x))
end

const sigmoid = Activation(
    sigma,
    x -> sigma(x) * (1-sigma(x)))
```

The next data type `Cost` contains the cost function to be used. In short, a cost function measures the difference between the output of the neural network and the correct output to be learned. We already define two cost functions at this point so that we can evaluate our neural network later. Cost functions are discussed in more detail in Sect. 13.5.

```
struct Cost
    f::Function
    delta::Function
end
```

```
function quadratic_cost(activation::Activation)::Cost
    Cost((a, y)    -> 0.5 * LinearAlgebra.norm(a-y)^2,
         (z, a, y) -> (a-y) .* activation.d.(z))
end

const cross_entropy_cost = Cost(
    (a, y)    -> sum(- y .* log.(a) - (1-y) .* log.(1-a)),
    (z, a, y) -> a-y)
```

The data structure `Network` contains the weights and biases of all layers with additional information. The `weights` are matrices and the `biases` are vectors. The vector `sizes` contains the numbers of neurons in each layer. The final four vectors record the progress during training as we will see later. The function `Network` is a custom constructor and only requires the sizes of the layers, but takes three keyword arguments. It calls the function `new`, only available in this context, to construct the instance (see Sect. 5.5).

The weights and biases are initialized with normally distributed random numbers. If a weight matrix is large, its product with the output of the previous layer tends to be large as well. This hinders learning with activation functions whose derivatives are small for large arguments. Therefore it is generally useful to scale the weight matrices such that products of the weight matrices with columns of all ones are still normally distributed with variance one; this is achieved by the scaling factor `sqrt(j)`, which is the square root of the size of the previous layer.

```
mutable struct Network
    activation::Activation
    cost::Cost
    n_layers::Int
    sizes::Vector{Int}
    weights::Vector{Array{Float64, 2}}
    biases::Vector{Vector{Float64}}
    training_cost::Vector{Float64}
    validation_cost::Vector{Float64}
    training_accuracy::Vector{Float64}
    validation_accuracy::Vector{Float64}

    function Network(sizes; activation = sigmoid,
                     cost = quadratic_cost(activation),
                     scale_weights = true)::Network
        new(activation, cost, length(sizes), sizes,
            [randn(i, j) / if scale_weights sqrt(j) else 1 end
                for (i, j) in zip(sizes[2:end], sizes[1:end-1])],
            [randn(i) for i in sizes[2:end]],
            [], [], [], [])
    end
end
```

Having defined these data structures, we can construct your first neural network by evaluating `Network([100, 10, 1])`.

Evaluating a neural network is commonly referred to as feeding forward. We loop over all weight matrices and bias vectors simultaneously using `zip`. In each iteration, the activation `a` of the previous layer is transformed linearly and then the activation function `nn.activation.f` is applied elementwise.

```
function feed_forward(nn::Network, input::Vector)::Vector
    local a = input

    for (W, b) in zip(nn.weights, nn.biases)
        a = nn.activation.f.(W*a + b)
    end

    a
end
```

13.3 The Approximation Property

Before we train our neural network, the question poses itself if neural networks such as the ones shown in Fig. 13.1 can approximate arbitrary functions to be learned or not. This question is fundamental: if arbitrary relationships between the input and output could not be represented by such functions, it would be absurd to try to train neural networks. Fortunately, the answer to this question is positive.

As the following theorem shows, neural networks ϕ without any hidden layer, but whose output consists of a linear combination of a sufficiently large number n of neurons, are already capable of approximating any given continuous function f arbitrarily well on compact subsets of $\mathbb{R}^d$ [1, 2]. The assumptions on the activation function σ are lenient. The restriction that the function to be approximated must be continuous is understandable in view of the fact that neural networks are continuous functions.

Theorem 13.1 (approximation property) *Suppose that K is a compact subset of $\mathbb{R}^d$, that f is an arbitrary function in $C(K, \mathbb{R})$, that $\epsilon \in \mathbb{R}^+$ is arbitrary, and that σ is a nonconstant, bounded, and monotonically increasing continuous function.*

Then there exist an integer n, constants $b_i \in \mathbb{R}$ and $v_i \in \mathbb{R}$, and vectors $\mathbf{w}_i \in \mathbb{R}^d$ for $i \in \{1, \dots, d\}$ such that the inequality

$$\max_{\mathbf{x} \in K} |\phi(\mathbf{x}) - f(\mathbf{x})| < \epsilon$$

holds, where the approximation ϕ is defined as

$$\phi(\mathbf{x}) := \sum_{i=1}^{n} v_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i),$$

i.e., the functions ϕ are dense in $C(K)$.

Functions in $C(K, \mathbb{R}^D)$ can be approximated by combining D neural networks, each neural network being responsible for one component.

Proof (sketch) Instead of giving a full proof of the theorem, we sketch a visual and intuitive argument for a slightly weaker result in the following. The intuitive argument works for neural networks with two hidden layers and for sigmoid activation functions.

In the first step, we approximate functions $g \in C([a, b], \mathbb{R})$, i.e., continuous functions with one-dimensional input and output. We start by considering a neural network with two neurons in a single hidden layer and one output neuron. Increasing the weight of the first hidden neuron makes its output approximate a step function, and changing the bias of the first hidden neuron changes the position of the step accordingly. We can thus approximate the output of the first hidden neuron by a step function with an adjustable step position. The same holds true for the second hidden neuron.

We can then combine the outputs of the two neurons in the hidden layer, which are two step functions, such that the neuron in the output layer is nonzero only between the two step positions. Furthermore, the height of the step can also be adjusted arbitrarily. This means that the neuron in the output layer before applying the activation function can be made approximately equal to

$$\psi_j(x) := \begin{cases} c_j, & x \in [a_j, b_j), \\ 0, & \text{otherwise.} \end{cases}$$

By using a hidden layer with $2m$ neurons, we can hence approximate any piecewise constant function

$$\psi(x) = \sum_{j=1}^{m} \psi_j(x) \approx \sigma^{-1} \circ g(x)$$

on the interval $[a, b]$. These piecewise constant functions approximate $g \in C([a, b])$ after applying the activation function σ in the output neuron as m increases. This concludes the first step, where we approximate continuous functions with one-dimensional input and output.

In the second step, we generalize this idea to approximate functions $f \in C(\mathbb{R}^d, \mathbb{R})$. In the two-dimensional case, we use two neurons for each of the two dimensions to construct piecewise constant functions of the form

$$\psi_{1j}((x_1,x_2)) := \begin{cases} c_{1j}, & x_1 \in [a_{1j}, b_{1j}), \\ 0, & \text{otherwise}, \end{cases}$$

$$\psi_{2k}((x_1,x_2)) := \begin{cases} c_{2k}, & x_2 \in [a_{2k}, b_{2k}), \\ 0, & \text{otherwise} \end{cases}$$

in the first hidden layer. These four neurons are combined in the second hidden layer to approximate functions of the form

$$\psi_{jk}((x_1,x_2)) := \begin{cases} c_{jk}, & (x_1,x_2) \in \big([a_{1j}, b_{1j}), [a_{2k}, b_{2k})\big), \\ 0, & \text{otherwise}, \end{cases}$$

which are nonzero only on a rectangle. Using these functions ψ_{jk}, we can approximate any continuous function $f \in C(\mathbb{R}^2, \mathbb{R})$ by the piecewise (on rectangles) constant function

$$\psi((x_1,x_2)) := \sum_{j=1}^{m_j} \sum_{k=1}^{m_k} \psi_{jk}((x_1,x_2)) \approx \sigma^{-1} \circ f((x_1,x_2))$$

and finally applying the activation function σ in the output layer. This argument can be generalized from two to d dimensions.

In the third and last step, we make the assumption that the neurons are approximated by step functions superfluous. In the one-dimensional case $C(\mathbb{R}, \mathbb{R})$, we choose intervals of same length. The error due to the smoothed step functions occurs where the intervals meet. We can make this error arbitrarily small by adding a large number N of shifted approximations of the function $f(x)/N$, because then each point x is only affected by the error due to one shifted approximation, which decreases as n increases. This ideas can be generalized to the multidimensional case. This concludes the sketch of the proof. $\square$

The universal approximation property in Theorem 13.1 is of great theoretic importance, since it motivates pursuing the goal of training neural networks to learn arbitrary functions. However, it still does not tell us how the parameters of the neural network, such as activation functions, the number of layers, and the numbers of neurons in every layer should be chosen. It is also clear that different classes of functions will require different parameter choices.

Theorem 13.1 may mislead us to believe that shallow neural networks are sufficient in practice. The opposite is true: the advantage of deep networks with many layers is their hierarchical structure. For example, in image recognition, there is a hierarchy comprising the recognition of pixels, edges, simple geometric shapes, larger objects consisting of the simple shapes, and even scenes containing multiple objects. In real-world problems, deep networks are far more useful than shallow networks.

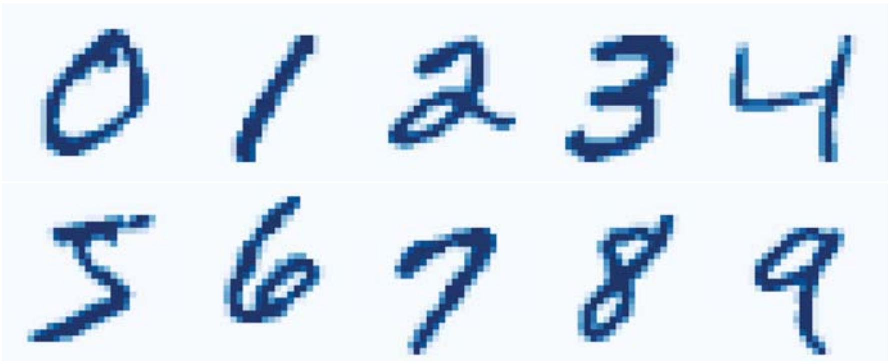

Fig. 13.2 The digits from 0 to 9 from the beginning of the MNIST training set.

13.4 Handwriting Recognition

To illustrate how neural networks learn, we consider a real-world example, namely how to recognize hand-written digits. Handwriting recognition is a classic, nontrivial task in artificial intelligence and machine learning and well-suited for neural networks, yet recognizing hand-written digits is still feasible as a first problem.

Fortunately, a large database of hand-written digits is available as the MNIST (modified National Institute of Standards and Technology) database, which contains 60 000 training images and 10 000 test images. The gray-scale images have a size of 28×28 pixels and were digitized from digits written by employees of the United States Census Bureau and American high-school students. Fig. 13.2 shows some of the first images in the database; local influence is seen in the shape of the digits 1 and 7. Even more fortunately, the MNIST database can be downloaded by simply installing the `MLDatasets` package and then evaluating `MLDatasets.MNIST.download()`.

Next, we read the whole database into six global variables, containing the images and labels of the training, validation, and test datasets. The MNIST images are split into these three datasets, and the reason for doing so will be discussed below. The global variables `MNIST_n_rows` and `MNIST_n_cols` contain the numbers of rows and columns of the gray-scale images. After reading each image in the function `MNIST_read_images`, its pixels are scaled to the interval $[0, 1]$.

```
global MNIST_file_training_images =
    joinpath(MLDatasets.MNIST.datadep"MNIST",
             MLDatasets.MNIST.TRAINIMAGES)
global MNIST_file_training_labels =
    joinpath(MLDatasets.MNIST.datadep"MNIST",
             MLDatasets.MNIST.TRAINLABELS)
global MNIST_file_test_images     =
    joinpath(MLDatasets.MNIST.datadep"MNIST",
             MLDatasets.MNIST.TESTIMAGES)
global MNIST_file_test_labels     =
    joinpath(MLDatasets.MNIST.datadep"MNIST",
             MLDatasets.MNIST.TESTLABELS)

global MNIST_n_rows, MNIST_n_cols

function MNIST_read_images(filename::String)
    GZip.open(filename, "r") do s
        local magic_number = bswap(read(s, UInt32))
        local n_items = Int(bswap(read(s, UInt32)))
        global MNIST_n_rows = Int(bswap(read(s, UInt32)))
        global MNIST_n_cols = Int(bswap(read(s, UInt32)))

        [Vector{Float64}(read!(s, Array{UInt8}(undef,
            MNIST_n_rows*MNIST_n_cols))) ./
                typemax(UInt8)
         for i in 1:n_items]
    end
end

function MNIST_read_labels(filename::String)
    GZip.open(filename, "r") do s
        local magic_number = bswap(read(s, UInt32))
        local n_items = Int(bswap(read(s, UInt32)))

        [Int(read(s, UInt8)) for i in 1:n_items]
    end
end
```

The function `vectorize` takes a label, i.e., an integer n between zero and nine, and yields a vector of length ten whose n-th element is equal to one, while all other elements are equal to zero. The reason for expanding the label into a vector was already discussed in Sect. 13.2: if the number of classes is large, then neural networks learn much better when each class corresponds to a designated neuron in the output layer. The class assigned to a given input is the one whose output neuron has the largest value.

```
function vectorize(n::Integer)
    local result = zeros(10)
    result[n+1] = 1
    result
end
```

Finally, the function `load_MNIST_data` yields the input and output values in six vectors, and six global variables are defined.

```
function load_MNIST_data()
    local train_x = MNIST_read_images(MNIST_file_training_images)
    local train_y = MNIST_read_labels(MNIST_file_training_labels)
    local test_x  = MNIST_read_images(MNIST_file_test_images)
    local test_y  = MNIST_read_labels(MNIST_file_test_labels)

    (train_x[1:50_000],      vectorize.(train_y[1:50_000]),
     train_x[50_001:60_000], train_y[50_001:60_000],
     test_x,                 test_y)
end

global (training_data_x,   training_data_y,
        validation_data_x, validation_data_y,
        test_data_x,       test_data_y) = load_MNIST_data()
```

The following function was used to plot the images in Fig. 13.2. The `PyPlot` packages was already imported at the beginning of the module.

```
function plot_digit(n::Int, file = nothing)
    local v

    if 1 <= n <= 50_000
        v = training_data_x[n]
    elseif 50_001 <= n <= 60_000
        v = validation_data_x[n-50_000]
    elseif 60_001 <= n <= 70_000
        v = test_data_x[n-60_000]
    end

    PyPlot.matshow(reshape(v, (MNIST_n_rows, MNIST_n_cols))',
                   cmap = "Blues")
    PyPlot.axis("off")

    if isa(file, String)
        PyPlot.savefig(file * string(n) * ".pdf",
                       bbox_inches = "tight", pad_inches = 0)
    end
end
```

13.5 Cost Functions

In order to train a neural network, we must be able to measure how well its output agrees with the given labels of the data. Functions that measure this difference are called cost functions, loss functions, or objective functions, and many choices exist.

We denote the items in the given training data by vectors $\mathbf{x} \in \mathbb{R}^{784}$, which in our application is $28 \cdot 28 = 784$ dimensional and represents an image. These items serve as input to the neural network. The corresponding labels in the training data are denoted by $y(\mathbf{x}) \in \mathbb{R}^{10}$, where each of the ten elements or neurons corresponds to one digit. In other words, the function $\mathbf{y} : \mathbb{R}^{784} \supset T \to \mathbb{R}^{10}$ represents the given training data. Furthermore, we denote the neural network by the function $\mathbf{a} : \mathbb{R}^{784} \to \mathbb{R}^{10}$, whose value is the activation of the output layer.

One of the most popular cost functions is the quadratic cost function

$$C_2(W, \mathbf{b}) := \frac{1}{2|T|} \sum_{\mathbf{x} \in T} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}(\mathbf{x})\|_2^2,$$

also called the mean squared error. The sum is over all $|T|$ elements $\mathbf{x}$ of the training set T. The cost function is a function of the parameters of the neural network, which are denoted by W and $\mathbf{b}$ for the collection of all weights and biases. The reason for the factor $1/|T|$ is that it makes the values of the cost function comparable whenever the number $|T|$ of the training items changes. The reason for the factor $1/2$ is that it removes the factor 2 in the derivative of the cost function, which we will use shortly. The factors are, of course, irrelevant when the cost function is minimized.

It goes without saying that other norms can be used instead of the Euclidean norm. Different choices of cost functions generally lead to different neural networks, and an expedient choice generally depends on the problem at hand.

Another popular cost function is the cross-entropy cost function

$$C_{\mathrm{CE}}(W, \mathbf{b}) := -\frac{1}{|T|} \sum_{\mathbf{x} \in T} \big(\mathbf{y}(\mathbf{x}) \cdot \ln \mathbf{a}(\mathbf{x}) + (\mathbf{1} - \mathbf{y}(\mathbf{x})) \cdot \ln(\mathbf{1} - \mathbf{a}(\mathbf{x}))\big), \tag{13.1}$$

where the logarithm is applied elementwise to its vector argument. The cost function, the activation function of the output layer, and how fast a network learns are closely related. This relationship is discussed at the end of the next section, where we will also see how the expression in C_{CE} is obtained.

13.6 Stochastic Gradient Descent

In order to improve a neural network, we aim to find weights and biases that minimize the cost function. Since neural networks generally have a large number of weights and biases, this is usually a high-dimensional optimization problem.

To minimize the cost function, we use a version of gradient descent called stochastic gradient descent. Other optimization methods can of course be used depending on the size of the optimization problem (see Chap. 11 and Chap. 12).

How does gradient descent work? Gradient descent was already discussed in Chap. 12, but we recapitulate the main idea here using the current notation. To simplify notation, we collect all parameters of the network, i.e., all weights and biases, in the vector **p**. Hence the gradient of the cost function C is

$$\nabla C = \begin{pmatrix} \frac{\partial C}{\partial p_1} \\ \vdots \\ \frac{\partial C}{\partial p_n} \end{pmatrix}.$$

The directional derivative

$$\frac{\partial C}{\partial \mathbf{e}}(\mathbf{p})$$

is the derivative of C at **p** in the direction of the unit vector **e** and can be written as

$$\frac{\partial C}{\partial \mathbf{e}}(\mathbf{p}) = \nabla C(\mathbf{p}) \cdot \mathbf{e}$$

using the gradient of C at **p**.

Starting at the point **p**, we would like to take a (small) step in the direction that minimizes $C(\mathbf{p})$ the most. How can we find a direction **e** in which the function C changes the most? The Cauchy–Bunyakovsky–Schwarz inequality, Theorem 8.1, implies the inequality

$$\left| \frac{\partial C}{\partial \mathbf{e}}(\mathbf{p}) \right| = |\nabla C(\mathbf{p}) \cdot \mathbf{e}| \le ||\nabla C(\mathbf{p})||,$$

since $||\mathbf{e}|| = 1$; equality in the Cauchy–Bunyakovsky–Schwarz inequality holds if and only if one vector is a multiple of the other. The inequality therefore means that the multiples of the gradient are the directions in which the directional derivative changes the most.

If the direction is $\mathbf{e} = \nabla C(\mathbf{p})$, then the directional derivative is

$$\frac{\partial C}{\partial \mathbf{e}}(\mathbf{p}) = \nabla C(\mathbf{p}) \cdot \nabla C(\mathbf{p}) = ||\nabla C(\mathbf{p})||^2 \ge 0,$$

and taking a step in this direction increases C; this is called gradient ascent. On the other hand, if the direction is $\mathbf{e} = -\nabla C(\mathbf{p})$, then the directional derivative is

$$\frac{\partial C}{\partial \mathbf{e}}(\mathbf{p}) = -\nabla C(\mathbf{p}) \cdot \nabla C(\mathbf{p}) = -||\nabla C(\mathbf{p})||^2 \le 0,$$

and taking a step in this direction decreases C; this is called gradient descent.

Since we aim to minimize the function C, we define the step $\Delta\mathbf{p}$ to take at the point **p** as

$$\Delta\mathbf{p} := -\eta \nabla C(\mathbf{p}),$$

where $\eta \in \mathbb{R}^+$ is called the learning rate. The directional derivative implies that the change ΔC in the function value is approximately given by

$$\Delta C \approx \nabla C(\mathbf{p}) \cdot \Delta \mathbf{p},$$

which yields

$$\Delta C \approx -\eta \nabla C(\mathbf{p}) \cdot \nabla C(\mathbf{p}) = -\eta \|\nabla C(\mathbf{p})\|^2 \leq 0$$

for our choice of the step $\Delta \mathbf{p}$; the function value indeed decreases.

Having discussed the methods of gradient ascent and descent, we now use a variant called stochastic gradient descent to adjust the parameters of the neural network. Stochastic gradient descent is the most common basic method to minimize the cost function. It just means that the training data are split randomly into batches and that gradient descent is performed for each batch. A reason for doing so is that in practice the number of training items is very large so that working with batches is more manageable. Reasonably large batches are also usually already sufficient to obtain a good approximation of the gradient, and the stochastic nature of stochastic gradient descent helps escape from local minima. Furthermore, the gradients can be computed in parallel for all batches.

Stochastic gradient descent is implemented by the function `SGD`. The number of steps in stochastic gradient descent is commonly called the number of epochs. The meaning of the parameter λ will be discussed in Sect. 13.9.1; we suppose that it vanishes for now. The function can also monitor the cost function and the accuracy (i.e., how many items are classified correctly) during the epochs. It can do so using the training data that must be supplied, but it can also use optional validation data. The reasons for this additional capability will be discussed in Sect. 13.8.

```
function SGD(nn::Network,
             training_data_x::Vector{Vector{Float64}},
             training_data_y::Vector{Vector{Float64}},
             epochs::Int, batch_size::Int, eta::Float64,
             lambda::Float64 = 0.0;
             validation_data_x::Vector{Vector{Float64}} = [],
             validation_data_y::Union{Vector{Int64},
                                      Vector{Vector{Float64}}} = [],
             monitor_training_cost       = true,
             monitor_validation_cost     = true,
             monitor_training_accuracy   = true,
             monitor_validation_accuracy = true)
    nn.training_cost       = []
    nn.validation_cost     = []
    nn.training_accuracy   = []
    nn.validation_accuracy = []

    for epoch in 1:epochs
        local perm = Random.randperm(length(training_data_x))
```

```
for k in 1:batch_size:length(training_data_x)
    update!(nn,
            training_data_x[perm[k:min(k+batch_size-1,
                                       end)]],
            training_data_y[perm[k:min(k+batch_size-1,
                                       end)]],
            eta, lambda, length(training_data_x))
end

@info @Printf.sprintf("epoch %d done", epoch)

if monitor_training_cost
    push!(nn.training_cost,
          total_cost(nn, training_data_x, training_data_y,
                     lambda))
    @info @Printf.sprintf("cost on training data:       %f",
                          nn.training_cost[end])
end

if monitor_validation_cost
    push!(nn.validation_cost,
          total_cost(nn, validation_data_x,
                     validation_data_y, lambda))
    @info @Printf.sprintf("cost on validation data:     %f",
                          nn.validation_cost[end])
end

if monitor_training_accuracy
    local a = accuracy(nn, training_data_x, training_data_y)
    local l = length(training_data_x)
    local r = a/l
    @info @Printf.sprintf("accuracy on training data:
        %5d / %5d = %5.1f%% correct", a, l, 100*r)
    push!(nn.training_accuracy, r)
end

if monitor_validation_accuracy
    local a = accuracy(nn, validation_data_x,
                       validation_data_y)
    local l = length(validation_data_x)
    local r = a/l
    @info @Printf.sprintf("accuracy on validation data:
        %5d / %5d = %5.1f%% correct", a, l, 100*r)
    push!(nn.validation_accuracy, r)
end
```

```
    end

    nn
end
```

The next four functions calculate the cost and the accuracy of a neural network. Each function has two methods, one for labels that are integers and one for vectorized labels.

```
function total_cost(nn::Network,
                    data_x::Vector{Vector{Float64}},
                    data_y::Vector{Int64}, lambda::Float64)::Float64
    total_cost(nn, data_x, vectorize.(data_y), lambda)
end

function total_cost(nn::Network,
                    data_x::Vector{Vector{Float64}},
                    data_y::Vector{Vector{Float64}},
                    lambda::Float64)::Float64
    sum(map((x, y) -> nn.cost.f(feed_forward(nn, x), y),
            data_x, data_y)) / length(data_x) +
                0.5 * lambda * sum(LinearAlgebra.norm(w)^2 for w in
                                        nn.weights) / length(data_x)
end

function accuracy(nn::Network,
                  data_x::Vector{Vector{Float64}},
                  data_y::Vector{Int64})::Integer
    count(map((x, y) -> y == argmax(feed_forward(nn, x)) - 1,
              data_x, data_y))
end

function accuracy(nn::Network,
                  data_x::Vector{Vector{Float64}},
                  data_y::Vector{Vector{Float64}})::Integer
    accuracy(nn, data_x, map(y -> argmax(y) - 1, data_y))
end
```

The function `update!` adds the gradient of the weights and biases multiplied by the learning rate η to the weights and biases of the network for each batch of training items. Again, we suppose that $\lambda = 0$ for now. The gradients are calculated by the function `propagate_back`, which will be discussed in the next section.

```
function update!(nn::Network,
                 batch_x::Vector{Vector{Float64}},
                 batch_y::Vector{Vector{Float64}},
                eta::Float64, lambda::Float64, n::Int)::Network
    local grad_W = [fill(0.0, size(W)) for W in nn.weights]
    local grad_b = [fill(0.0, size(b)) for b in nn.biases]

    for (x, y) in zip(batch_x, batch_y)
        (delta_grad_W, delta_grad_b) = propagate_back(nn, x, y)
        grad_W += delta_grad_W
        grad_b += delta_grad_b
    end

    nn.weights =
        (1-eta*lambda/n)*nn.weights - (eta/length(batch_x))*grad_W
    nn.biases -= (eta/length(batch_x)) * grad_b

    nn
end
```

13.7 Backpropagation

In this section, we calculate the gradient of a neural network. The function `propagate_back`, which yields the gradient, is thus the final missing piece before we can train our neural network. Since a neural network is the composition of its layers, we use the chain rule to calculate the derivatives.

We start by fixing the notation. We denote the activation of the l-th layer by

$$\mathbf{a}^{(l)} := \sigma(W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}),$$

where the function σ is applied elementwise to its vector argument, and we denote the weighted input to the neurons in layer l by

$$\mathbf{z}^{(l)} := W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}.$$

Of course, the equation $\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)})$ holds.

The backpropagation algorithm calculates all the partial derivatives

$$\frac{\partial C}{\partial w_{ij}^{(l)}} \quad \text{and} \quad \frac{\partial C}{\partial b_i^{(l)}}$$

of the cost function C with respect to all elements of the weight matrices $W^{(l)}$ and with respect to all elements of the bias vectors $\mathbf{b}^{(l)}$.

We make two assumptions on the cost function C, which are usually satisfied, namely that it can be written as an average

$$C(W,\mathbf{b}) = \frac{1}{|T|}\sum_{\mathbf{x}\in T} K(W,\mathbf{b},\mathbf{x})$$

over all training samples $\mathbf{x}$ in the training set T and that it is a function of the activation of the output layer of the neural network only, i.e., $C = C(\mathbf{a}^{(l)})$. Both cost functions in Sect. 13.5, the quadratic cost function and cross-entropy cost function, satisfy these two assumptions.

To help apply the chain rule, we denote the partial derivatives of the cost function C with respect to the weighted input $\mathbf{z}_i^{(l)}$ to neuron i in layer l by

$$\delta_i^{(l)} := \frac{\partial C}{\partial z_i^{(l)}}. \tag{13.2}$$

This value is often called the error of neuron i in layer l.

We start with the output layer L. Applying the chain rule to the definition (13.2) of the error yields

$$\delta_i^{(L)} = \sum_k \frac{\partial C}{\partial a_k^{(L)}} \frac{\partial a_k^{(L)}}{\partial z_i^{(L)}} \qquad \forall i,$$

where the sum is over all neurons k in the output layer L. Since the activation function σ is applied elementwise, the partial derivative $\partial a_k^{(L)}/\partial z_i^{(L)}$ is nonzero only if $k = i$. Therefore we have

$$\delta_i^{(L)} = \frac{\partial C}{\partial a_i^{(L)}}\sigma'(z_i^{(L)}) \qquad \forall i$$

or

$$\boldsymbol{\delta}^{(L)} = \frac{\partial C}{\partial \mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)}), \tag{13.3}$$

a formula for the error in the output layer, where $\odot$ denotes elementwise multiplication and σ' is applied elementwise to its vector argument.

Second, we derive an equation for the error $\boldsymbol{\delta}^{(l)}$ in terms of the error $\boldsymbol{\delta}^{(l+1)}$. The chain rule yields

$$\delta_i^{(l)} = \frac{\partial C}{\partial z_i^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = \sum_k \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} \delta_k^{(l+1)}.$$

Since

$$\mathbf{z}^{(l+1)} = W^{(l+1)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)} = W^{(l+1)}\sigma(\mathbf{z}^{(l)}) + \mathbf{b}^{(l+1)}$$

and hence

$$z_k^{(l+1)} = \sum_i w_{ki}^{(l+1)}\sigma(z_i^{(l)}) + b_k^{(l+1)}$$

hold by definition, we find

$$\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}} = w_{ki}^{(l+1)}\sigma'(z_i^{(l)})$$

and therefore

$$\delta_i^{(l)} = \sum_k w_{ki}^{(l+1)}\sigma'(z_i^{(l)})\delta_k^{(l+1)}$$

or

$$\boldsymbol{\delta}^{(l)} = \big((W^{(l+1)})^\top \boldsymbol{\delta}^{(l+1)}\big) \odot \sigma'(\mathbf{z}^{(l)}). \tag{13.4}$$

Third, we can find the partial derivatives of the cost function C with respect to all weights and biases using the errors $\boldsymbol{\delta}^{(l)}$. The chain rule yields

$$\frac{\partial C}{\partial w_{ij}^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{ij}^{(l)}},$$

$$\frac{\partial C}{\partial b_i^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_i^{(l)}}.$$

Furthermore, the equations $\partial C/\partial z_k^{(l)} = \delta_k^{(l)}$ and

$$z_k^{(l)} = \sum_j w_{kj}^{(l)} a_j^{(l-1)} + b_k^{(l)}$$

hold by definition. Differentiating the last equation yields

$$\frac{\partial z_k^{(l)}}{\partial w_{ij}^{(l)}} = \begin{cases} a_j^{(l-1)}, & i = k, \\ 0, & i \neq k, \end{cases}$$

$$\frac{\partial z_k^{(l)}}{\partial b_i^{(l)}} = \begin{cases} 1, & i = k, \\ 0, & i \neq k. \end{cases}$$

In summary, we have shown that

$$\frac{\partial C}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}, \tag{13.5a}$$

$$\frac{\partial C}{\partial b_i^{(l)}} = \delta_i^{(l)}. \tag{13.5b}$$

The four equations in (13.3), (13.4), and (13.5) constitute the four fundamental equations of backpropagation. All the partial derivatives of the cost function C are calculated via the errors $\boldsymbol{\delta}^{(l)}$ as well as the weighted inputs $\mathbf{z}^{(l)}$ and the activations $\mathbf{a}^{(l)}$. The weighted inputs and the activations are found by simply feeding the network forward. Then we calculate the errors $\boldsymbol{\delta}^{(l)}$ by starting with the error $\boldsymbol{\delta}^{(L)}$ of the last layer L given by (13.3) and working recursively towards lower layers by using (13.4) in order to obtain $\boldsymbol{\delta}^{(l)}$ from $\boldsymbol{\delta}^{(l+1)}$. Finally, (13.5) yields the partial derivatives $\partial C/\partial w_{ij}^{(l)}$ and $\partial C/\partial b_i^{(l)}$, since the errors and activations are now known.

The backpropagation algorithm is very efficient, since it comprises of only two passes over the layers to calculate all partial derivatives. As such it is approximately twice as costly as the feeding the network forward in order to evaluate it. In the forward pass, the weighted inputs and activations are calculated, while in the backward pass the errors and the partial derivatives are calculated.

Backpropagation is implemented by the function `propagate_back`. First, the two variables `grad_W` and `grad_b` for the partial derivatives as well as the two variables `z` and `a` for the weighted inputs and activations are allocated. Note that `a[l]` is equal to $\mathbf{a}^{(l-1)}$ such that `a[1]` records the input to the neural network.

The first loop is the forward pass, where all weights inputs and activations are calculated. Before the second loop, the variable `delta` is initialized as $\boldsymbol{\delta}^{(L)}$ given by (13.3). Since the expression depends on the cost function, it is calculated by the function stored in the `delta` field of the type `Cost`. The results `grad_W[end]` and `grad_b[end]` are given by (13.5) for $l = L$.

Then, in the second loop, the variable `delta` is updated recursively using (13.4) and the results `grad_W[l]` and `grad_b[l]` are calculated by (13.5). Again note that `a[l]` is equal to $\mathbf{a}^{(l-1)}$.

```
function propagate_back(nn::Network, x::Vector{Float64},
                        y::Vector{Float64})::Tuple
    local grad_W = [fill(0.0, size(W)) for W in nn.weights]
    local grad_b = [fill(0.0, size(b)) for b in nn.biases]

    local z = Vector(undef, nn.n_layers-1)
    local a = Vector(undef, nn.n_layers)

    a[1] = x
    for (i, (W, b)) in enumerate(zip(nn.weights, nn.biases))
        z[i] = W * a[i] + b
        a[i+1] = nn.activation.f.(z[i])
    end

    local delta = nn.cost.delta(z[end], a[end], y)
    grad_W[end] = delta * a[end-1]'
    grad_b[end] = delta

    for l in nn.n_layers-2:-1:1
        delta = (nn.weights[l+1]' * delta) .* nn.activation.d.(z[l])
        grad_W[l] = delta * a[l]'
        grad_b[l] = delta
    end

    (grad_W, grad_b)
end
```

At this point, the code of the module that implements our neural network is complete.

```
end # module
```

Next, we run an example. Whenever a program uses random numbers, it can be useful while developing and debugging to set the seed of the random-number generator using `Random.seed!`. This ensures that the same sequence of random numbers is generated every time.

```
import Random
Random.seed!(0)
NN.SGD(NN.Network([NN.MNIST_n_rows * NN.MNIST_n_cols, 30, 10]),
       NN.training_data_x, NN.training_data_y,
       100, 10, 3.0,
       validation_data_x = NN.test_data_x,
       validation_data_y = NN.test_data_y)
```

We use the package name `NN` as a prefix to access the symbols in this package. After a few minutes, we observe that the algorithm classifies more than 95% of the digits in the validation data correctly, while the accuracy in the training data is higher. The cost function is also smaller on the training data than on the validation data. Typical accuracies and costs for one hundred iterations of training are shown in Figures 13.3 and 13.4. We will discuss these curves thoroughly in the next section.

13.8 Hyperparameters and Overfitting

Training algorithm contain various parameters such as the learning rate η and the parameter λ here. The parameters that pertain to the training algorithm are called hyperparameters in order to distinguish them from the parameters of the neural network itself such as its number of levels, its weights, and its biases. Thus the question naturally arises how these parameters should be chosen.

Unfortunately, there is no general answer to this question. Much depends on the specific problem and the task the neural network shall solve. Whenever there are unknown (hyper-)parameters, the idea of optimizing these parameters suggests itself (see Chap. 11 and Chap. 12 for inspirations for optimization algorithms to be used). In addition to optimizing continuous parameters such as the

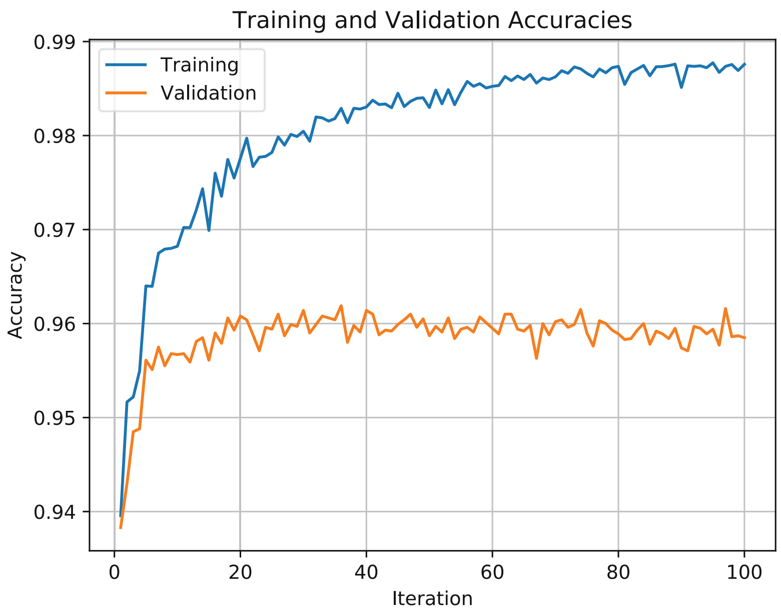

Fig. 13.3 Training and validation accuracies as functions of the iteration number. The training accuracies are generally larger than the validation accuracies.

learning rate, there are also discrete optimization problems. These concern discrete parameters such as the numbers of layers of the neural network and their sizes, the activation functions used in each layer, and even the type of the layer (e.g., in convolutional neural networks).

The hyperparameters are found either manually or automatically by an optimization algorithm using the validation dataset. The validation dataset is also useful in another regard. When training the network using stochastic gradient descent as in the example at the end of the previous section, the training accuracy is generally higher than the accuracy observed on an validation dataset (see Fig. 13.3). The same effect is observed in the values of the cost function: the cost function is smaller on the training set than on the validation dataset (see Fig. 13.4).

These observations are easily explained: the training data are used in gradient descent and therefore the cost function decreases and the accuracy increases on the training dataset, in this case up to the 100-th iteration. The situation is different on the validation dataset. There the accuracy stops to increase after about fifteen iterations, and the cost stops to decrease after the same number of iterations. This means that any improvement on the training data after this number of iterations is highly unlikely to translate into any improvement on new data. This effect is called overfitting.

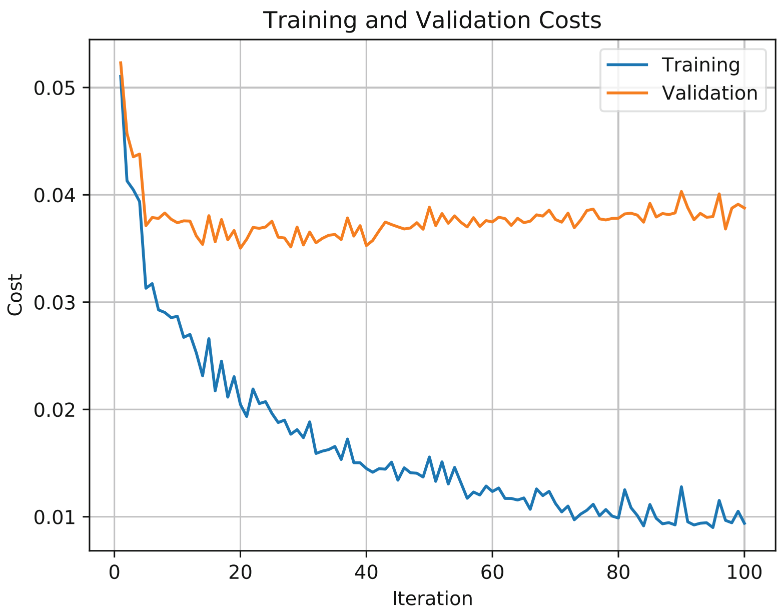

Fig. 13.4 Training and validation costs as functions of the iteration number. The training costs are generally smaller than the validation costs.

After this point, training fits the neural network only to idiosyncrasies and noise in the training data, which is not only a waste of computational resources, but it is even more importantly also detrimental for generalization. Generalization is the ultimate goal in machine learning; it means that the essence hidden in the data is learned and that the idiosyncrasies of the training data are discarded.

Since neural networks contain many parameters while the amount of available data is always limited, they are a very versatile tool, but at the same time the parameters are comparatively ill specified. Hence neural networks are prone to overfitting, as the many parameters are usually easily fitted to the training data. Therefore overfitting requires careful consideration.

Early stopping is a simple strategy to overcome overfitting and means that training is stopped as soon as the accuracy stops to decrease on the validation dataset. Unfortunately, it is not obvious when to stop since stochastic gradient descent is stochastic by its nature and because the accuracy of a neural networks may reach a plateau while training and then improve again.

Clearly, overfitting is reduced (and generalization is improved) by using more training data, but the amount of training data is often given and cannot be changed easily (but see Problem 13.12). We can also reduce the size of the neural network to reduce the number of parameters to be determined. This is generally a good approach, but care must be taken not to reduce it too much until it cannot learn the essence hidden in the data anymore.

After the hyperparameters have been chosen and training has been established while avoiding overfitting, the success of the whole procedure must be assessed on a third dataset never used before. This third dataset is called the test dataset, and it is the arbiter for the accuracy that has been achieved.

In summary, it is prudent to split all the available data into three sets, namely the training, the validation, and the test datasets, with the following purposes.

- The training dataset is used for minimizing the cost function.
- The validation dataset is used for finding hyperparameters and for preventing overfitting (e.g. using early stopping).
- The test dataset is used for assessing the success of the whole training procedure using untouched data.

13.9 Improving Training

In this section, we discuss two methods for improving the training of neural networks. The first is regularization, which we have implemented already, but which still needs to be explained. The second is the choice of cost function and how it affects training.

13.9.1 Regularization

Regularization is a method for reducing overfitting. The idea is to add a constraint to the parameters in order to keep them simple in a certain sense. In weight-decay regularization, parameters that are large carry a penalty in order to keep them simple, i.e., small. This has the added benefit of avoiding run-away parameters. To implement the penalty, a regularization term, which is often a multiple of a norm of the weights, is added to the cost function. If C_0 is the original cost function (see Sect. 13.5) and the vector $\mathbf{w}$ contains all weights $W_{ij}^{(l)}$, then the ℓ^2-regularized cost function is

$$\begin{aligned}
C_{\ell^2}(W,\mathbf{b},\lambda) &:= C_0(W,\mathbf{b}) + \frac{\lambda}{2|T|}||\mathbf{w}||_2^2 \\
&= C_0(W,\mathbf{b}) + \frac{\lambda}{2|T|}\sum_k w_k^2 \\
&= C_0(W,\mathbf{b}) + \frac{\lambda}{2|T|}\sum_{l,i,j}(W_{ij}^{(l)})^2,
\end{aligned}$$

where $\lambda \in \mathbb{R}_0^+$ is called the regularization parameter. The factor $1/|T|$ occurs since it is also part of C_0. The biases are not affected by regularization as explained below.

Of course, any other norm of the weights such as the ℓ^p-norms can be used, resulting in the more general definition

$$C_{\ell^p}(W, \mathbf{b}, \lambda) := C_0(W, \mathbf{b}) + \frac{\lambda}{p|T|}\|\mathbf{w}\|_p^p = C_0(W, \mathbf{b}) + \frac{\lambda}{p|T|}\sum_k |w_k|^p.$$

We first discuss how this modification of the cost function affects learning. It means that smaller weights are preferred all other things being equal. The size of the regularization parameter λ determines the relative importance of minimizing the original cost function C_0 and minimizing the weights.

Why does regularization reduce overfitting? Regularized neural networks tend to contain smaller weights, which implies that the output of the network does not change much when small perturbations are added to the input in contrast to unregularized neural networks with larger weights. In other words, it is more difficult for regularized neural networks to learn randomness in the training data; the smaller weights must learn the features present in the training data. In other words, the larger weights of an unregularized network can adjust better to noise and thus facilitate overfitting.

This explanation also justifies why the biases are not included in regularization: large biases do not affect the sensitivity of the neural network to perturbations or noise.

Regularization as modification of the cost function is implemented in a straightforward manner in backpropagation. Because of the derivatives

$$\begin{aligned}\frac{\partial C_{\ell^2}}{\partial \mathbf{w}} &= \frac{\partial C_0}{\partial \mathbf{w}} + \frac{\lambda}{|T|}\mathbf{w},\\ \frac{\partial C_{\ell^2}}{\partial \mathbf{b}} &= \frac{\partial C_0}{\partial \mathbf{b}},\end{aligned}$$

the steps in gradient descent become

$$\begin{aligned}\Delta\mathbf{w} &:= -\eta\frac{\partial C_0}{\partial \mathbf{w}} - \frac{\eta\lambda}{|T|}\mathbf{w},\\ \Delta\mathbf{b} &:= -\eta\frac{\partial C_0}{\partial \mathbf{b}}.\end{aligned}$$

After adding the step $\Delta\mathbf{w}$ to $\mathbf{w}$, the new weight is

$$\left(1 - \frac{\eta\lambda}{|T|}\right)\mathbf{w} - \eta\frac{\partial C_0}{\partial \mathbf{w}},$$

which is calculated at the end of the function `update!`. This concludes the explanation of the implementation.

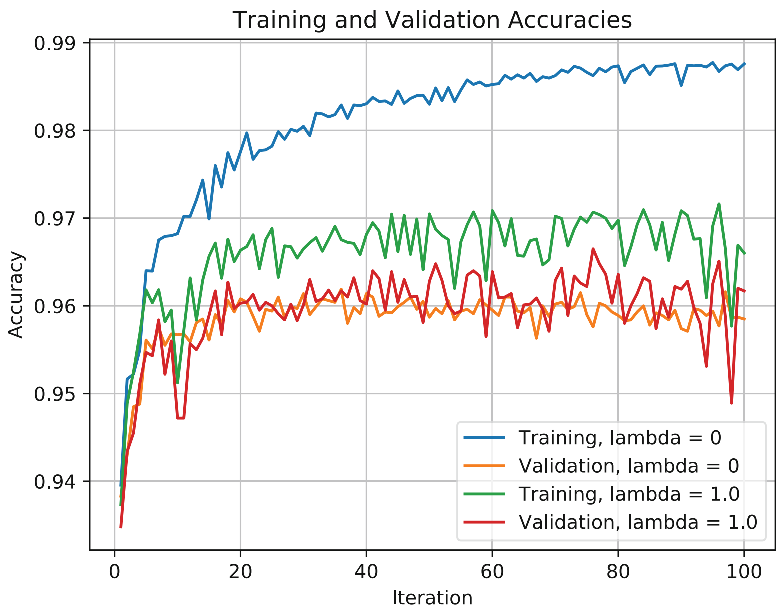

Fig. 13.5 Training and validation accuracies as functions of the iteration number with and without regularization.

Finally, we consider a numerical example. It is the same example as above, but now we use regularization with $\lambda = 1$.

```
NN.SGD(NN.Network([NN.MNIST_n_rows * NN.MNIST_n_cols, 30, 10]),
       NN.training_data_x, NN.training_data_y,
       100, 10, 3.0, 1.0,
       validation_data_x = NN.test_data_x,
       validation_data_y = NN.test_data_y)
```

Figures 13.5 and 13.6 show the accuracies and costs evaluated on training and validation data with and without regularization. Fig. 13.5 shows that continued overfitting to the training data is much reduced by regularization and that performance on the validation data has been improved as well. Fig. 13.6 again shows that continued overfitting to the training data has been reduced. It also shows that the cost does not increase on the validation data as training continues. Therefore even this first choice of hyperparameters has beneficial effects.

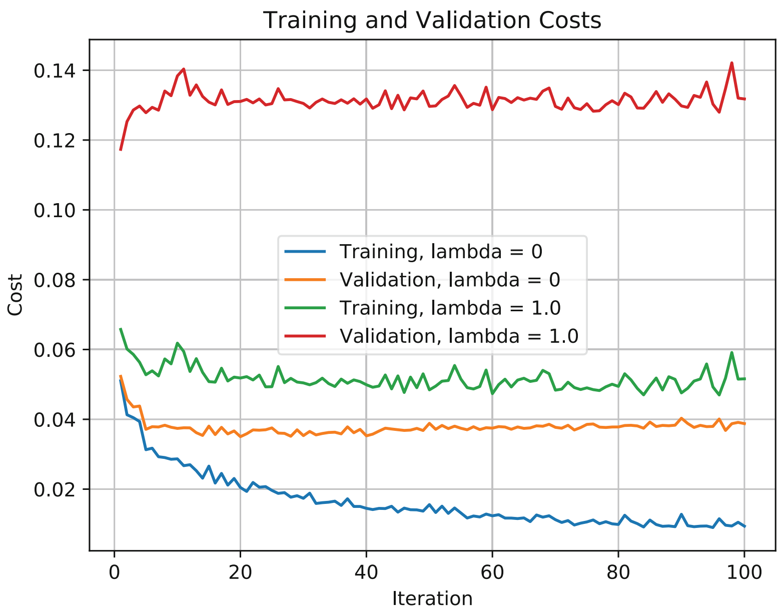

Fig. 13.6 Training and validation costs as functions of the iteration number with and without regularization.

13.9.2 Cost Functions

The choice of the cost function can strongly affect how fast a neural network learns. We discuss properties of the two cost functions in Sect. 13.5 that can now be understood in terms of the properties of the activation functions and their role in the backpropagation algorithm (see Sect. 13.7).

An effect due to the choice of the activation function is learning slowdown. As seen from (13.4), the errors $\boldsymbol{\delta}^{(l)}$ and therefore the gradients used while minimizing the cost function using stochastic gradient descent become small when $\sigma'(\mathbf{z}^{(l)})$ becomes small. Such a factor $\sigma'(\mathbf{z}^{(l)})$ occurs in every recursive use of (13.4) in backpropagation, which implies that deep neural networks become harder to train when the factors $\sigma'(\mathbf{z}^{(l)})$ become small. Therefore the leaky rectifier σ_2 is advantageous compared to the hyperbolic tangent σ_5, especially in deep neural networks.

The cost function can also be a cause of learning slowdown, again due to the factor $\sigma'(\mathbf{z}^{(l)})$ in the error $\delta^{(L)}$ in (13.3). Thus this interaction between the cost function and the activation function in the output layer may be detrimental to learning.

We can remedy the situation by choosing the cost function C such that this factor disappears. Considering the derivatives with respect to the biases, we would hence like to achieve that

$$\frac{\partial C}{\partial b} = a - y.$$

To simplify notation, we drop the sum over all training items in the cost function C for now. We also denote an element $b_i^{(L)}$ of the bias vector $\mathbf{b}^{(L)}$ in the output layer L by just b, the element $a_i^{(L)}$ by just a, and the element y_i by just y. The chain rule yields

$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}\sigma'(z).$$

For the activation function σ_1, we have $\sigma_1'(z) = \sigma_1(z)(1 - \sigma_1(z)) = a(1 - a)$. Therefore we have

$$a - y = \frac{\partial C}{\partial b} = \frac{\partial C}{\partial a}a(1 - a),$$

which is the ordinary differential equation

$$\frac{\partial C}{\partial a} = \frac{a - y}{a(1 - a)} = -\frac{y}{a} + \frac{1 - y}{1 - a}$$

for $\partial C/\partial a$. The last equation follows from a partial-fraction decomposition. Integration yields

$$C = -y \ln a - (1 - y)\ln(1 - a) + \text{const.}$$

for the cost function for a single training item, and hence the cross-entropy cost function C_{CE} defined in (13.1) for all training items.

We now check that this calculation for the partial derivatives with respect to the biases yields the desired form for all partial derivatives in the output layer. Starting from the cross-entropy cost function

$$C_{\text{CE}}(W, \mathbf{b}) = -\frac{1}{|T|}\sum_{\mathbf{x}\in T}\big(\mathbf{y}(\mathbf{x}) \cdot \ln \mathbf{a}^{(L)}(\mathbf{x}) + (\mathbf{1} - \mathbf{y}(\mathbf{x})) \cdot \ln(\mathbf{1} - \mathbf{a}^{(L)}(\mathbf{x}))\big),$$

we find the error in the output layer L by (13.3) to be

$$\begin{aligned}\boldsymbol{\delta}^{(L)} &= \frac{\partial C}{\partial \mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)}) \\ &= -\frac{1}{|T|}\sum_{\mathbf{x}\in T}\big(\mathbf{y}(\mathbf{x}) \oslash \mathbf{a}^{(L)}(\mathbf{x}) - (\mathbf{1} - \mathbf{y}(\mathbf{x})) \oslash (\mathbf{1} - \mathbf{a}^{(L)}(\mathbf{x}))\big) \odot \mathbf{a}^{(L)}(\mathbf{x}) \\ &\qquad \odot (\mathbf{1} - \mathbf{a}^{(L)}(\mathbf{x})) \\ &= \frac{1}{|T|}\sum_{\mathbf{x}\in T}(\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}(\mathbf{x})),\end{aligned}$$

where $\oslash$ denotes elementwise division of two vectors. Hence, by (13.5), the partial derivatives are

$$\frac{\partial C_{\text{CE}}}{\partial w_{ij}^{(L)}} = \delta_i^{(L)} a_j^{(L-1)} = \frac{1}{|T|} \sum_{\mathbf{x}\in T} (a_i^{(L)}(\mathbf{x}) - y_i(\mathbf{x})) a_j^{(L-1)}(\mathbf{x}),$$
$$\frac{\partial C_{\text{CE}}}{\partial \mathbf{b}^{(L)}} = \boldsymbol{\delta}^{(L)} = \frac{1}{|T|} \sum_{\mathbf{x}\in T} (\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{y}(\mathbf{x})).$$

Hence there are no factors $\sigma'(\mathbf{z}^{(L)})$ often responsible for learning slowdown.

There is a similar interaction between the quadratic cost function C_2 and so-called linear neurons in the output layer, meaning that the activation function σ of the output layer L is the identity function. Then $\mathbf{a}_j^{(L)} = \mathbf{z}_j^{(L)}$ and $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - \mathbf{y}$ by (13.3). Therefore (13.5) yields

$$\frac{\partial C_2}{\partial w_{ij}^{(L)}} = \frac{1}{|T|} \sum_{\mathbf{x}\in T} (a_i^{(L)}(\mathbf{x}) - z_i^{(L)}(\mathbf{x})) a_j^{(L-1)}(\mathbf{x}),$$
$$\frac{\partial C_2}{\partial \mathbf{b}^{(L)}} = \frac{1}{|T|} \sum_{\mathbf{x}\in T} (\mathbf{a}^{(L)}(\mathbf{x}) - \mathbf{z}^{(L)}(\mathbf{x})),$$

showing no detrimental factor in the output layer L for this choice of cost function and activation function.

These considerations imply that the cost function and the activation functions should not be chosen independently. It is worthwhile to study their interactions in order to arrive at efficient training algorithms.

13.10 Julia Packages

The package `MLDatasets` used above contains sample data sets for machine learning. The package `Flux` provides software for neural networks written in pure Julia and includes GPU support.

13.11 Bibliographical Remarks

Artificial neural networks were first implemented in the middle of the twentieth century and have become a standard method in machine learning and artificial intelligence. Thus there is a vast body of literature on this topic. A historic perspective can be found in [4], and a very accessible introduction can be found in [3]. The hyperparameters used in the examples are from [3].

Problems

13.1 Implement and plot the five activation functions σ_i, $i \in \{1,2,3,4,5\}$, along with their derivatives using JULIA. Furthermore, find the ten limits $\lim_{x\to\pm\infty} \sigma_i(x)$ for $i \in \{1,2,3,4,5\}$.

13.2 Find suitable hyperparameters for each of the five activation functions σ_i, $i \in \{1,2,3,4,5\}$, such that stochastic gradient descent works well for MNIST handwriting recognition.

13.3 Compare how well the neural network learns with and without scaling the initial weights.

13.4 Compare the best classification performance you can each achieve using each of the five activation functions σ_i, $i \in \{1,2,3,4,5\}$, in a neural network.

13.5 * Prove Theorem 13.1.

13.6 Implement an adaptive strategy for choosing the learning rate η. Choose $\mu \in \mathbb{R}^+$, $\mu \approx 1$, and change the learning rate to $\mu\lambda$, λ, or λ/μ depending on the improvement achieved by each of these three values.

13.7 Investigate further strategies for choosing the learning rate η adaptively by making it depend on the learning progress.

13.8 Parallelize the `for` loop over the all batches in the function `SGD`.

13.9 Use the validation dataset to optimize the hyperparameters of the training algorithm. Optimize single hyperparameters one after another in one-dimensional optimization problems. Which hyperparameters should be considered on a logarithmic scale?

13.10 Use the validation dataset to optimize the hyperparameters of the training algorithm. Optimize (as many as possible of) the hyperparameters simultaneously in a multidimensional optimization problem. Which hyperparameters should be considered on a logarithmic scale?

13.11 Implement early stopping.

13.12 More training data helps reduce overfitting. In certain problems, more training data can be generated by simply perturbing the available training data slightly. When dealing with image data, shifting and rotating the images slightly suggests itself. Implement and evaluate this idea to reduce overfitting.

13.13 Derive the formulas for ℓ^1-regularization and implement it (as an option).

13.14 Derive the formulas for ℓ^p-regularization and implement it (as an option).

13.15 Compare the performance of ℓ^1- and ℓ^2-regularization.

13.16 After using all ideas in this chapter and optimizing the hyperparameters using the validation data, what is the best accuracy you can achieve on the test data?

13.17 Implement a convolutional neural network and train it with the MNIST database. Which classification error can you achieve?

13.18 Extend the code in Sect. 13.10 to use a GPU.

References

1. Cybenko, G.: Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems* **2**(4), 303–314 (1989)
2. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. *Neural Networks* **2**(5), 359–366 (1989)
3. Nielsen, M.: Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/
4. Nilsson, N.J.: *The Quest for Artificial Intelligence*. Cambridge University Press, Cambridge, UK (2010)

Chapter 14
Bayesian Estimation

No other formula in the alchemy of logic has exerted more astonishing powers,
For it has established the existence of God from the premiss of total ignorance;
and it has measured with numerical precision the probability
that the sun will rise to-morrow.

—John Maynard Keynes, *A Treatise on Probability,* Chapter VII (1921)

Abstract Frequentist and Bayesian statistics and inference differ in their fundamental assumptions on the nature of probabilities and models. After a short discussion of the differences, we use the ideas of Bayesian inference to determine model parameters. The motivation for these considerations is the fact that models usually contain parameters that are unknown and often cannot be measured or determined directly. Thus they must be estimated by comparing the model to data. In this chapter, the Bayesian approach to the estimation of model parameters is developed, implemented, and applied to an example.

14.1 Introduction

Whenever we use a mathematical model to describe a realistic situation, it is very likely that the model will contain at least one parameter. Dimensional analysis offers a way to reduce the number of parameters and make equations dimensionless, but the fact remains. Therefore the Bayesian approach to estimating model parameters is discussed in this chapter, and one of most popular numerical approaches, namely Markov-chain Monte Carlo, is developed and implemented. It is assumed that the reader is familiar with the basics of probability theory, although the exposition is self-contained.

C. Heitzinger, *Algorithms with JULIA*,
https://doi.org/10.1007/978-3-031-16560-3_14

14.2 The Riemann–Stieltjes Integral

Theorems in discrete and continuous probability are often stated separately using two different notions, namely the (discrete) probability P of an event and the (continuous) probability density f of a random variable. In order to unify the treatment of discrete and continuous probabilities, we use the Riemann–Stieltjes integral as an elegant concept to cover both cases, the discrete and the continuous one, simultaneously.

To define the Riemann–Stieltjes integral, we need the concept of a partition. A partition of a set A is a set of subsets $A_i \subset A$ such that $\bigcup_i A_i = A$ and $A_i \cap A_j = \emptyset$ for all indices $i \neq j$. The definition of the Riemann–Stieltjes integral is a generalization of the Riemann integral with the additional notion of an integrator function. In a Riemann sum, each function value is multiplied by the subinterval length, whereas in the Riemann–Stieltjes integral the function values are more generally multiplied by the subintervals weighted by the integrator.

Definition 14.1 (Riemann–Stieltjes integral) The *Riemann–Stieltjes* integral

$$\int_a^b h(x)\mathrm{d}g(x)$$

of a function $h\colon [a,b] \to \mathbb{R}$ (the integrand) with respect to a function $g\colon [a,b] \to \mathbb{R}$ (the integrator) is the limit

$$\lim_{|P|\to 0} S(P,h,g)$$

of the Riemann–Stieltjes sum

$$S(P,h,g) := \sum_{i=0}^{n-1} h(\xi_i)(g(x_{i+1}) - g(x_i)).$$

Here

$$P := \big\{[x_0 := a, x_1), [x_1, x_2), \dots, [x_{n-1}, x_n := b]\big\}$$

is a partition of the interval $[a,b]$, where $x_i < x_{i+1}$ holds for all indices $i \in \{0,\dots,n-1\}$; the fineness

$$|P| := \max\big\{x_{i+1} - x_i \mid i \in \{0,\dots,n-1\}\big\}$$

of a partition P is the length of the longest of its subintervals; and the points ξ_i are chosen as $\xi_i \in [x_i, x_{i+1})$ for all indices $i \in \{0,\dots,n-1\}$.

If the limit above exists, then h is called Riemann–Stieltjes integrable with respect to g on the interval $[a,b]$.

If the integrator is smooth enough, then the Riemann–Stieltjes integral simplifies to the Riemann integral as the next theorem shows. Of course, if the integrator is the identity, then the Riemann–Stieltjes integral reduces to the Riemann integral.

Theorem 14.2 (Riemann–Stieltjes and Riemann integrals) *Suppose that the function $h\colon [a,b] \to \mathbb{R}$ is continuous and that the function $g\colon [a,b] \to \mathbb{R}$ is continuously differentiable. Then*

$$\int_a^b h(x)\mathrm{d}g(x) = \int_a^b h(x)g'(x)\mathrm{d}x$$

holds.

The usefulness in probability theory becomes apparent in the following theorem.

Theorem 14.3 (reduction of Riemann–Stieltjes integral to a finite sum) *Suppose that the function $h\colon [a,b] \to \mathbb{R}$ is piecewise continuous and that the function $g\colon [a,b] \to \mathbb{R}$ is a step function with the jumps*

$$g_i := g(x_i+) - g(x_i-)$$

at the points $x_i \in [a,b]$, $i \in \{1,\dots,n\}$, of discontinuity, where $g(a-) := g(a)$ and $g(b+) := g(b)$. Suppose further that at all points x_i not both h and g are discontinuous from the left and not both h and g are discontinuous from the right. Then the function h is Riemann–Stieltjes integrable with respect to g on the interval $[a,b]$, and the integral has the value

$$\int_a^b h(x)\mathrm{d}g(x) = \sum_{i=1}^{n} h(x_i)g_i.$$

To apply the Riemann–Stieltjes integral to probability theory, we view the integrator g as the cumulative probability distribution function

$$F_X(x) := P(X \le x)$$

of a random variable X. The derivative

$$f_X := F_X'$$

is its probability density. Then discrete random variables become special cases of continuous random variables: discrete random variables are just continuous random variables with piecewise constant cumulative probability distributions F_X, which are the integrators, implying that the probability densities f_X are sums of delta distributions.

Definition 14.4 (expectance, expected value) The *expectance* or *expected value* of a function h of a random variable X is defined as

$$\mathbb{E}[h(X)] := \int_{-\infty}^{\infty} h(x)\mathrm{d}F_X(x).$$

If the random variable X is continuous, then Theorem 14.2 yields the usual integral

$$\mathbb{E}[h(X)] = \int_{-\infty}^{\infty} h(x)\mathrm{d}F_X(x) = \int_{-\infty}^{\infty} h(x)f_X(x)\mathrm{d}x$$

of the expected value of a continuous random variable.

If the random variable X is discrete, then F_X is a step function with the points $x_i \in [a,b]$, $i \in \{1,\dots,n\}$, of discontinuity, and the jumps $P(X = x_i) = F_X(x_i+) - F_X(x_i-)$. The expected value $\mathbb{E}[h(X)]$ simplifies to the sum

$$\mathbb{E}[h(X)] = \int_{-\infty}^{\infty} h(x)\mathrm{d}F_X(x) = \sum_{i=1}^{n} h(x_i)P(X = x_i)$$

by Theorem 14.3, which is the usual definition of the expected value of discrete random variables. Furthermore, using delta distributions, we can write the probability density f_X, i.e., the derivative of the step function F_X, as

$$f_X(x) = \sum_{i=1}^{n} P(X = x_i)\delta(x = x_i).$$

14.3 Bayes' Theorem

We start with a basic definition.

Definition 14.5 (conditional probability) The *conditional probability* density function of Y given the occurrence of the value x of the random variable X is

$$f_Y(y \mid X = x) := \frac{f_{X,Y}(x,y)}{f_X(x)}$$

assuming that $f_X(x) > 0$, where $f_{X,Y}(x,y)$ is the joint density of the random variables X and Y and $f_X(x)$ is the marginal density of X.

In the discrete case, the conditional probability $P(A|B)$ of the event A occurring given that B with $P(B) > 0$ occurs is usually written as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

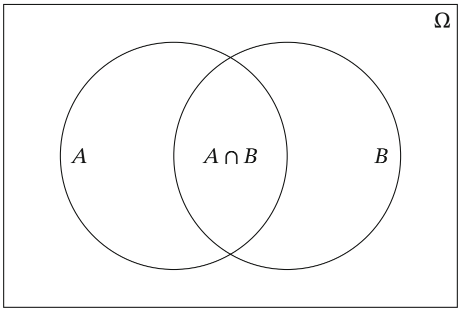

Fig. 14.1 Here two events A and B are illustrated as part of a probability space Ω. The conditional probability $P(A|B)$ corresponds to the ratio of the areas of $A \cap B$ and B.

(see Fig. 14.1).

The significance of Bayes' theorem is that it makes it possible to calculate $P(A|B)$ if $P(B|A)$ is known, i.e., it relates the two conditional probabilities $P(A|B)$ and $P(B|A)$. The basic form of Bayes' theorem is stated and proved in the following.

Theorem 14.6 (Bayes' theorem) *Suppose that A and B are two events and that $P(B) > 0$. Then the equation*

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

holds.

Proof We start with the case $P(A) = 0$. Then both sides of the equation vanish.

The remaining case is $P(A) > 0$. By Definition 14.5, $P(A|B) = P(A \cap B)/P(B)$ and $P(B|A) = P(B \cap A)/P(A)$ if $P(A) > 0$ and $P(B) > 0$. Since $P(A \cap B) = P(B \cap A)$, we find $P(A|B)P(B) = P(B|A)P(A)$. Division by $P(B) > 0$ yields the assertion. $\square$

There is an extended form of Bayes' theorem incorporating the law of total probability. The law of total probability states that

$$f_Y(y) = \int f_Y(y \mid X = x)\mathrm{d}F_X(x) = \int f_Y(y \mid X = x)f_X(x)\mathrm{d}x \qquad (14.1)$$

and follows easily from Definition 14.5. For discrete random variables, it reads

$$P(B) = \sum_i P(B|A_i)P(A_i)$$

if the events A_i are a partition of the whole sample space.

Using the law of total probability, the proof of the extended form of Bayes' theorem is similar to the proof of Theorem 14.6 above.

Theorem 14.7 (extended form of Bayes' theorem) *Suppose that X and Y are two random variables and that $f_Y > 0$. Then the equations*

$$f_X(x \mid Y = y) = \frac{f_Y(y \mid X = x) f_X(x)}{f_Y(y)} = \frac{f_Y(y \mid X = x) f_X(x)}{\int f_Y(y \mid X = x) \mathrm{d}F_X(x)}$$

hold.

Proof We start with the case $f_X(x) = 0$. Then all terms vanish.

The general case is $f_X(x) > 0$. By Definition 14.5, the equations

$$f_Y(y \mid X = x) f_X(x) = f_{X,Y}(x, y)$$
$$f_X(x \mid Y = y) f_Y(y) = f_{Y,X}(y, x)$$

hold if $f_X(x) > 0$ and $f_Y(y) > 0$. Since the two joint densities on the right-hand sides are identical, the first equation follows. The second equation uses (14.1) in the denominator. □

Corollary 14.8 (discrete, extended form of Bayes' theorem) *Suppose that the events A_i are a partition of the sample space and that B is an event with nonzero probability $P(B) \neq 0$. Then the equations*

$$P(A_i|B) = \frac{P(B|A_i)P(A_i)}{P(B)} = \frac{P(B|A_i)P(A_i)}{\sum_i P(B|A_i)P(A_i)}$$

hold.

14.4 Frequentist and Bayesian Inference

There are two perspectives in statistical inference, namely the frequentist and the Bayesian perspectives.

In the frequentist perspective, probabilities are the frequencies of the occurrences of an event if experiments are repeated many times. This definition is objective, since the frequencies are independent of the observer. The probabilities are not updated during data acquisition. The parameters of a model are unknown but deterministic. Estimators are constructed and confidence intervals are calculated. A confidence interval for a parameter contains the true value of the corresponding parameter in repeated sampling with a given probability or frequency, namely the confidence level. Since the unknown parameters are viewed as deterministic, parameter densities cannot be propagated through the model in order to quantify model uncertainties.

In the Bayesian perspective, probabilities are subjective and can be updated to incorporate new data or information. Probabilities are probability distributions and not a single frequency value. Model parameters are considered to be random variables. When a parameter is estimated, the probability density obtained is called the posterior probability density. This viewpoint is a natural one when uncertainties in model parameters are to be propagated through the models and quantified. Instead of confidence intervals, credible intervals are calculated; a credible interval contains the parameter with a given probability, namely the confidence level.

In Bayesian inference, new information can be incorporated into the knowledge of an observer, e.g., into the probabilities of parameters, as soon as it becomes available. In other words, it is a method for online learning. We denote the (unknown) model parameters by the random variable Q, which can be multidimensional in general. The data, measurements, or observations are denoted by the random variable D and can be multidimensional in general as well.

We rewrite Bayes' formula in Theorem 14.7 in the form

$$\pi(q|d) = \frac{\pi(d|q)\pi_0(q)}{\pi_D(d)} = \frac{\pi(d|q)\pi_0(q)}{\int \pi(d|q)\pi_0(q)\mathrm{d}q}, \tag{14.2}$$

where

$$\begin{aligned} \pi(q|d) &:= \pi_Q(q \mid D = d), \\ \pi(d|q) &:= \pi_D(d \mid Q = q) \end{aligned}$$

are defined to simplify notation. The indices referring to the random variables are usually dropped in Bayesian inference, since they are clear from the context.

The probability density

$$\pi_0(q) := \pi_Q(q)$$

is called the prior probability density and contains the previous knowledge about the random variable Q before the incorporation of new information. Furthermore, the probability density $\pi(q|d)$ on the left-hand side is called the posterior probability density and represents the updated knowledge after the realization $d = D(\omega)$ has been observed. Finally, $\pi(d|q)$ is called the likelihood, and the marginal density $\pi_D(d)$ is a normalization factor.

Therefore, new data informs the posterior density directly only through the likelihood $\pi(d|q)$. With this interpretation, equation (14.2) gives rise to an iterative algorithm.

Algorithm 14.9 (Bayesian inference)

```
1. Initialize the prior density $\pi_0(q)$.
2. While data are available:
    a. Calculate the posterior density $\pi(q|d)$ using (14.2). Realizations $d = D(\omega)$
       of the data inform the likelihood $\pi(d|q)$.
    b. Set $\pi_0(q)$ to be the posterior density $\pi(q|d)$ just calculated.
```

In the first step, the question which prior density should be used before any data are available arises immediately. If no prior information is available at all, one commonly uses a noninformative prior density or distribution. Straightforward choices are a uniform distribution if the parameter is known to lie in an interval and an unnormalized uniform distribution if the parameter is known to lie in an unbounded interval.

For example, if a parameter is known to be positive, then the unnormalized uniform density is $\pi_0(q) := [q > 0]$, where the notation means that $[S] = 1$ if the statement S is true and $[S] = 0$ if it is false. Of course, the integral $\int_{\mathbb{R}} \pi_0(q) \mathrm{d}q$ is unbounded, and hence this prior distribution is called improper.

We now consider a classical and simple, yet very instructive example. Suppose that you are tested positive for a rare disease that effects 1‰ of the population while knowing that the test correctly identifies 99% of the people who have the disease and incorrectly identifies only 1% of the people who do not have the disease. The numbers look dire, so you turn to mathematics as a last resort.

Fortunately, the discrete version of Bayes' theorem in Theorem 14.8 can immediately be applied to the problem at hand by writing Bayes' formula in the form

$$P(\text{dis.}|\text{pos.}) = \frac{P(\text{pos.}|\text{dis.})P(\text{dis.})}{P(\text{pos.}|\text{dis.})P(\text{dis.}) + P(\text{pos.}|\neg\text{dis.})P(\neg\text{dis.})},$$

where "dis." means you have the disease and "pos." means you were tested positive. This formula is easily implemented in Julia.

```julia
function posterior(prior, likelihood)
    likelihood * prior / (likelihood*prior
                          + (1-likelihood)*(1-prior))
end
```

The likelihood $P(\text{pos.}|\text{dis.})$ is given as 99% in this example. It is reasonable to choose the frequency of the disease in the population as the initial prior probability. After a few iterations, we obtain these numbers.

```julia
global p = 0.001
for i in 1:5
    global p = posterior(p, 0.99)
    @show (i, p)
end
```

```
(i, p) = (1, 0.09016393442622944)
(i, p) = (2, 0.9074999999999999)
(i, p) = (3, 0.9989714794017902)
(i, p) = (4, 0.9999896003148012)
(i, p) = (5, 0.9999998949515932)
```

We see that after the first test, the posterior probability of having the disease has increased from 1‰ to $\approx$ 9%. Why not more? This is explained by the relatively small correctness $P(\text{pos.}|\text{dis.}) = 99\%$ of the test compared to the small frequency of the disease in the population. Taking a simple of one thousand people, we expect one person to have the disease, while ten are expected to be tested positive. Out of these (approximately) eleven persons tested positive, only one has the disease, resulting in the probability of $\approx 9\%$ to have the disease after the first test.

If the prior probability is zero, the posterior probability will always be zero as well. This means that absolute confidence in the beginning remains unchanged in the Bayesian setting, unless the likelihood is equal to one. In this case, both the nominator and the denominator are zero and the quotient is undefined.

After a second, independent test, the posterior probability of having the disease increases to $\approx 91\%$, and so forth. The posterior probability converges quickly to 1 as more information becomes available.

When applying Bayes' Theorem like in this example, a few questions arise. What is the influence of the initial prior density? Can it change the result? And does the posterior density converge?

These questions can be answered as follows. The Bernstein–von-Mises theorem states that the posterior density converges to a normal distribution independent of the initial prior density in the limit of infinitely many, independent, and identically distributed realizations under certain conditions.

14.5 Parameter Estimation and Inverse Problems

Parameter-estimation or inverse problems are ubiquitous in science, technology, engineering, and applications. They occur whenever a model contains parameters which are a priori unknown.

14.5.1 Problem Statement

Parameter estimation and inverse problems refer to the kind of problems where model parameters are to be inferred given measurements or observations. The advantage of the Bayesian approach is that it also shows how well such an inverse problem can be solved. If the resulting probability distribution for a sought model parameter is spread out or multimodal, this parameter cannot be determined well. On the other hand, if the distribution is well localized around a certain value, the parameter can be calculated precisely. Having calculated probability distributions, confidence intervals are also easily found.

In contrast, classical methods that work by trying to find parameter values such that the distance in a certain norm between measurement and model output is minimized always yield a parameter value. This parameter value depends on the choice of norm in the minimization problem, but more importantly, additional considerations are necessary in order to assess how sensitive this minimum is to perturbations. Without additional work, we do not know how reliable the parameter values found are, which is especially important in the case of non-linear or so-called ill-posed inverse problems.

Because parameter estimation and inverse modeling must always deal with measurement noise and possibly with other uncertainties, it is therefore expedient to pose the problem within the context of probability theory from the beginning.

In order to apply Bayes' theorem, we start by considering the general statistical model

$$D_i := f(t_i, Q) + \epsilon_i, \tag{14.3}$$

where the function f represents the model; D is a random vector representing data, measurements, or observations; Q is a random vector representing parameters to be determined, i.e., the quantities of interest; and the random vector ϵ represents any unbiased, independent, and identically distributed errors such as measurement errors. The independent variable t of the model function f will represent time in the example below. The indices i number the points where data points D_i are available for the values t_i of the independent variable t of f. The errors are mutually independent from the parameters Q, and they are additive here. This model equation applies to any problem with additive errors, but the approach can of course also be formulated for multiplicative errors.

Equivalently, we can write

$$D_i \sim N(f(t_i, Q), \sigma^2),$$

i.e., the measurements are independent and normally distributed with mean $f(t_i, Q)$ and variance σ^2.

Before we apply Theorem 14.7 or (14.2), we define a model function in order to make things concrete and to discuss a non-trivial example that we will implement later.

14.5.2 The Logistic Equation as an Example

The logistic equation is the ordinary differential equation

$$f'(t) = q_1 f(t)\left(1 - \frac{f(t)}{q_2}\right), \qquad f(0) = q_3 \neq 0, \tag{14.4}$$

with the three positive parameters $q_1 \in \mathbb{R}^+$, $q_2 \in \mathbb{R}^+$, and $q_3 \in \mathbb{R}^+$. The equation models the increase and decrease of a population of size f as a function of time. The parameter q_3 is the initial population size at time $t = 0$. If the parameter q_2 is very large, then the second term on the right-hand side becomes negligible and the equation simplifies to $f'(t) = q_1 f(t)$ with the solution $f(t) = \mathrm{e}^{q_1 t}$, implying that the parameter q_1 is a growth rate. The meaning of the parameter q_2 will become apparent briefly, but it is already clear that the second, quadratic term $-(q_1/q_2)f(t)^2$ counteracts the growth term $q_1 f(t)$. (A *linear* second term could just be absorbed into the first term.)

Equation (14.4) can be solved by separation of variables; a short calculation shows that its solution is

$$f(t) = \frac{q_2 q_3}{q_3 + (q_2 - q_3)\mathrm{e}^{-q_1 t}}, \tag{14.5}$$

called the logistic function. It is straightforward to see that

$$\lim_{t\to\infty} f(t) = q_2,$$

implying that the population size always tends to the parameter q_2, which is hence usually called the carrying capacity. The larger the carrying capacity is, the smaller the second term $-(q_1/q_2)f(t)^2$ in (14.4) responsible for population decrease is.

A realistic example of a population governed by the logistic equation is the growth of a bacterial colony. We can measure the number of bacteria in a Petri dish as time progresses or we can create synthetic measurements by defining

$$d_i := f(t_i, q_1, q_2, q_3) + \epsilon_i, \qquad \epsilon_i \sim N(0, \sigma^2),$$

after recalling (14.3). Here the times $t_i := i\Delta t$ when measurements are taken are equidistant, and the measurement errors ϵ_i are normally distributed with variance σ^2.

The following JULIA function implements the model. It is simple, since we could solve the underlying logistic equation explicitly, and its solution is given by (14.5).

```julia
function f(i, dt, sigma, q1, q2, q3)
    q2*q3 / (q3 + (q2-q3)*exp(-q1*i*dt)) + sigma*randn(Float64)
end
```

Fig. 14.2 shows synthetic measurements generated by this function. The population starts with a small number of individuals and then approaches the carrying capacity.

In order to generate the synthetic measurements, we had to choose values for the three parameters q_1, q_2, and q_3 of interest. After having generated the data, we forget these three values before proceeding with the parameter estimation.

14.5.3 The Likelihood

In order to apply Bayes' theorem in the form (14.2) and to calculate the sought posterior density $\pi(q|d)$, we must know the likelihood $\pi(d|q)$ (and the prior density $\pi_0(q)$). The likelihood function depends on the assumptions made on the

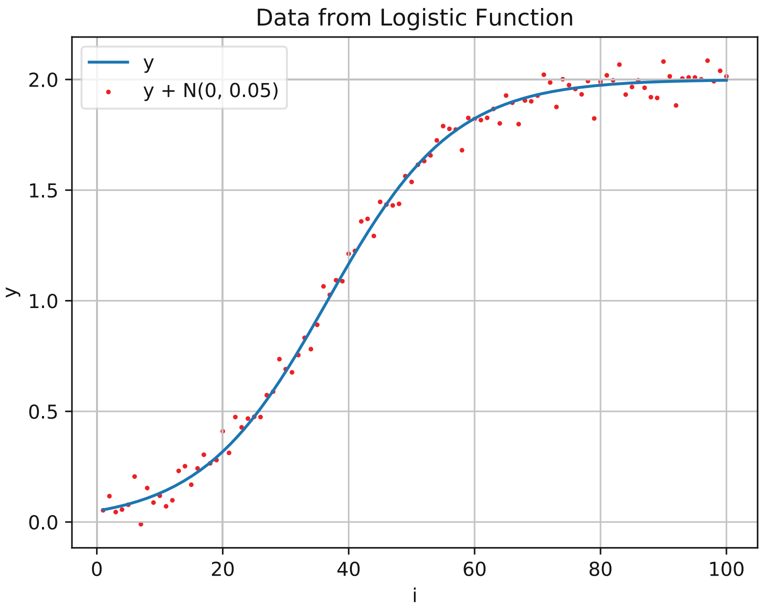

Fig. 14.2 Synthetic measurements generated using the logistic function as implemented by `f` for $i \in \{1, \dots, 100\}$, $\Delta t := 1$, $\sigma = 0.05$, $q_1 := 0.1$, $q_2 := 2$, $q_3 := 0.05$.

distribution of the errors. In the case of independent and identically distributed errors $\epsilon_i \sim N(0, \sigma^2)$, which we assume in this example, the likelihood is

$$\pi(d|q) := L(q, \sigma^2 \mid d) := \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{-(d_i - f(t_i,q))^2/(2\sigma^2)}$$

$$= \frac{1}{(2\pi\sigma^2)^{N/2}} \mathrm{e}^{-S(q)/(2\sigma^2)}, \quad (14.6)$$

where D is a N-dimensional random vector meaning that there are N data points $(d_1, \dots, d_N)$ and we have defined

$$S(q) := \sum_{i=1}^{N} (d_i - f(t_i, q))^2.$$

14.5.4 Markov-Chain Monte Carlo

Furthermore, in order to apply Bayes' Theorem in the form (14.2), the integral in the denominator must be evaluated. This numerical integration remains a challenge if the number of parameters, i.e., the dimension of the random variable Q, is large, although many methods for high-dimensional numerical integration have been developed. Therefore we follow an alternative approach here.

The alternative is to construct a Markov chain whose stationary distribution is equal to the posterior density. To do so, we start by defining Markov chains.

Definition 14.10 (Markov chain, Markov property) A *Markov chain* is a sequence of random variables X_n, $n \in \mathbb{N}$, that satisfy the *Markov property,* namely that X_{n+1} depends only on its predecessor X_n for all $n \in \mathbb{N}$, i.e.,

$$\begin{aligned} &P(X_{n+1} = x_{n+1} \mid X_1 = x_1, \dots, X_n = x_n) \\ &\qquad\qquad = P(X_{n+1} = x_{n+1} \mid X_n = x_n) \qquad \forall n \in \mathbb{N}. \end{aligned}$$

The set of all possible realizations of the random variables X_n is called the state space of the Markov chain.

Markov chains can be realized if three pieces of information are known:

1. its state space,
2. its initial distribution p^0, i.e., the distribution of X_0, and
3. its transition or Markov kernel, which gives the probability

   $$p_{ij} := P(X_{n+1} = x_j \mid X_n = x_i)$$

   of transitioning from state x_i to x_j and thus defines how the chain evolves. If the state space is finite, the Markov chain is called finite and the entries of the transition matrix P are the probabilities p_{ij}.

Here we assume that the transition probabilities p_{ij} that constitute the transition kernel are independent of time or iteration n. Markov chains with this property are called homogeneous Markov chains.

Clearly, the entries of the initial distribution p^0 and of the transition matrix P are nonnegative, and the elements of p^0 and the rows of P sum to one. The distributions of the states as time progresses are given by

$$\begin{aligned} &p^0, \\ &p^1 := p^0 P, \\ &p^2 := p^1 P = p^0 P^2, \\ &\quad\vdots \\ &p^n := p^{n-1} P = p^0 P^n, \end{aligned}$$

i.e., in each iteration the distribution p^n is multiplied by the transition matrix P.

A natural question to ask is whether the random variables of a Markov chain converge and have a limit. It turns out that convergence in distribution is the right notion of convergence to use to answer this question.

Definition 14.11 (convergence in distribution) Let X_n, $n \in \mathbb{N}$, be a sequence of random variables with distributions F_{X_n}. If F_X is a distribution function, if C is the set where F_X is continuous, and if

$$\lim_{n\to\infty} F_{X_n}(x) = F_X(x) \qquad \forall x \in C$$

holds, then the sequence $\langle X_n \rangle_{n\in\mathbb{N}}$ is said to *converge in distribution* to the limiting random variable X. Convergence in distribution is written as

$$X_n \xrightarrow{D} X.$$

Applying convergence in distribution to a Markov chain therefore means considering the limit

$$\pi := \lim_{n\to\infty} p^n.$$

We want to decide whether the limiting distribution π exists and – if it does – to calculate it. The calculation

$$\pi = \lim_{n\to\infty} p^0 P^n = \lim_{n\to\infty} p^0 P^{n+1} = \left(\lim_{n\to\infty} p^0 P^n\right) P = \pi P$$

implies that if the limit π exists it must satisfy the equation

$$\pi = \pi P.$$

This observation motivates the following definition.

Definition 14.12 (stationary or equilibrium distribution) If a Markov chain has the transition matrix P, then distributions that satisfy the equation

$$\pi = \pi P$$

are called *stationary or equilibrium distributions* of the Markov chain.

Every homogeneous Markov chain on a finite state space has at least one stationary distribution. A stationary distribution, however, may not be unique and it may not be equal to $\lim_{n\to\infty} p^n$.

There are two kinds of homogeneous Markov chain that we must exclude to ensure the unique existence of a stationary distribution. The first kind to be excluded are Markov chains in which not the whole state space is reachable after some time. This kind is excluded by stipulating that the Markov chain is irreducible.

Definition 14.13 (irreducible Markov chain, reducible Markov chain) A Markov chain is called *irreducible* if any state x_i can be reached from any other state x_j in a finite number of steps with nonzero probability. A *reducible* Markov chain is a not irreducible one.

The second kind of Markov chain to be excluded in order to obtain a stationary distribution are periodic ones. In a periodic Markov chain, parts of the state space are visited at regular time intervals.

Definition 14.14 (aperiodic Markov chain, periodic Markov chain) The period of a Markov chain is defined as

$$\gcd(\{m \in \mathbb{N} \mid P(X_{n+m} = x_i \mid X_n = x_i) > 0\}).$$

A Markov chain is called *aperiodic* if its period is equal to one and it is called *periodic* if its period is greater than one.

The following theorem answers the question which properties of a Markov chain ensure that it has a unique stationary distribution π.

Theorem 14.15 (unique stationary distribution) *Every finite, homogeneous, irreducible, and aperiodic Markov chain has a unique stationary distribution π. Furthermore, the Markov chain converges in distribution, i.e.,*

$$X_n \xrightarrow{D} X$$

and

$$\lim_{n\to\infty} p^n = \pi,$$

to this limiting stationary distribution π for every initial distribution p^0.

Knowing that a Markov chain has a unique stationary distribution π, we could calculate it using the defining equation $\pi = \pi P$ in Definition 14.12 together with the condition $\sum_i \pi_i = 1$. Unfortunately, this is often difficult. An alternative that we discuss next is to use the detailed balance condition in the following definition.

Definition 14.16 (reversible Markov chain, detailed balance) A Markov chain with transition matrix P and limiting distribution π is called *reversible* if the condition

$$\pi_i p_{ij} = \pi_j p_{ji} \qquad \forall\forall i, j$$

of *detailed balance* is satisfied.

The next result is easy to prove.

Theorem 14.17 *Every reversible Markov chain is stationary.*

Proof By definition, the limiting distribution of a reversible Markov chain exists. Using the detailed-balance condition, we calculate

$$\pi P = \sum_i \pi_i p_{ij} = \sum_i \pi_j p_{ji} = \pi_j \sum_i p_{ji} = \pi_j,$$

which implies

$$\pi P = \pi,$$

which is the definition of stationarity. □

The detailed-balance condition helps calculate the stationary distribution. Suppose a finite and homogeneous Markov chain is irreducible and aperiodic. Then Theorem 14.15 implies that there exists a *unique* stationary distribution *irrespective* of the initial distribution. We can find the limiting stationary distributing by identifying a candidate distribution and checking that it is stationary using the detailed-balance condition in Definition 14.16 and Theorem 14.17. Then the candidate distribution must be identical to the limiting distribution, since it is unique and stationary.

In other words, the detailed-balance condition in Definition 14.16 and Theorem 14.17 is only a sufficient condition for the existence of a stationary distribution. The uniqueness of the stationary distribution can be guaranteed by Theorem 14.17, when the important assumptions that the Markov chain is irreducible and aperiodic are satisfied.

The basic idea of Markov-chain Monte Carlo is to construct Markov chains whose stationary distribution is the sought posterior density. Evaluating realizations of the Markov chain hence is the same as sampling the posterior and yields densities for the parameter values.

14.5.5 The Metropolis–Hastings Algorithm

The Metropolis and Metropolis–Hastings algorithms make it possible to implement this idea. In parameter estimation and inverse problems, the random variables X_n in the Markov chain are the parameters of the model, and therefore we denote the random variables by Q_n and the realizations by q_n from now on.

The Metropolis and Metropolis–Hastings algorithms compute Markov chains whose unique stationary distributions are any given distribution ω. For our purposes, we will later define ω to be the posterior distribution, i.e., we will use

$$\omega(q) := \pi(q|d) = \frac{\pi(d|q)\pi_0(q)}{\int \pi(d|q)\pi_0(q)\mathrm{d}q}. \tag{14.7}$$

But for now it is more convenient to derive the algorithm for a general distribution ω that should become the stationary distribution of the Markov chain.

We construct the transition probability $P(q'|q)$ for going from state q to state q' such that it satisfies the detailed-balance condition (see Definition 14.16), of course. Given ω, this means that the transition probability $P(q'|q)$ must satisfy

$$P(q'|q)\omega(q) = P(q|q')\omega(q')$$

or, equivalently,

$$\frac{P(q'|q)}{P(q|q')} = \frac{\omega(q')}{\omega(q)}. \tag{14.8}$$

The transition from state q to state q' happens in two steps: first, a new state is proposed by a proposal or jumping distribution $J(q'|q)$, which is the conditional probability of proposing the new state q' given state q, and second the acceptance probability $A(q'|q)$ is the probability of accepting the proposed state q'. If it is rejected, the old state q is repeated. In summary, this means that we try to find J and A such that

$$P(q'|q) = J(q'|q)A(q'|q).$$

Substituting this form of $P(q'q)$ into (14.8) yields the form

$$\frac{A(q'|q)}{A(q|q')} = \frac{\omega(q')J(q|q')}{\omega(q)J(q'|q)} \tag{14.9}$$

of the detailed-balance condition. At this point, we can choose any proposal distribution J. There are many choices, but the choice is important for the numerical behavior of the Markov chain and will be discussed later. Having chosen the proposal distribution, we must define a suitable acceptance probability. The Metropolis acceptance probability

$$A(q'|q) := \min\left(1, \frac{\omega(q')J(q|q')}{\omega(q)J(q'|q)}\right)$$

is common.

We can check that it works by setting

$$r := \frac{\omega(q')J(q|q')}{\omega(q)J(q'|q)}$$

and calculating

$$\begin{aligned}\frac{A(q'|q)}{A(q|q')} = \frac{\min(1,r)}{\min(1,1/r)} &= \begin{cases}\frac{1}{1/r}, & r \geq 1,\\ r, & r < 1\end{cases}\\ &= r,\end{aligned}$$

which shows that (14.9) and therefore the detailed-balance condition is satisfied. Hence we can indeed construct a Markov chain whose stationary distribution is the given, arbitrary distribution ω.

The difference between the Metropolis and the Metropolis–Hastings algorithm lies only in the proposal distribution J. If it is symmetric, i.e., if $J(q', q) = J(q, q')$, then the algorithm is called a Metropolis algorithm. If it is not symmetric, it is called a Metropolis–Hastings algorithm, which is therefore slightly more general.

We can now formulate the Metropolis–Hastings algorithm. The similarity to simulated annealing (see Sect. 11.3) is not a coincidence, but due to their common root. The algorithm works for general distributions ω, but in the formulation of the algorithm we also note what happens when ω is given by (14.7), because this is the application we are interested in.

Algorithm 14.18 (Metropolis–Hastings)

1. Initialization: choose an initial state q_1 (such that $\pi(q_0|d) > 0$).
2. Repeat:

   a. Generate a candidate state

   $$q' := q_n + Rz \tag{14.10}$$

   after choosing $z \sim N(0,1)$, where R is the Cholesky factorization of D or V in (14.12) below. This definition of q' ensures that

   $$q' \sim N(q_n, D) \quad \text{or} \quad q' \sim N(q_n, V),$$

   respectively, because of Theorem 14.20 below.

   b. Calculate the acceptance probability

   $$\begin{aligned} A(q'|q_n) &:= \min\left(1, \frac{\omega(q')J(q_n|q')}{\omega(q_n)J(q'|q_n)}\right) = \min\left(1, \frac{\pi(q'|d)J(q_n|q')}{\pi(q_n|d)J(q'|q_n)}\right) \\ &= \min\left(1, \frac{\frac{\pi(d|q')\pi_0(q')}{\int \pi(d|q)\pi_0(q)\mathrm{d}q}J(q_n|q')}{\frac{\pi(d|q_n)\pi_0(q_n)}{\int \pi(d|q)\pi_0(q)\mathrm{d}q}J(q'|q_n)}\right) = \min\left(1, \frac{\pi(d|q')\pi_0(q')J(q_n|q')}{\pi(d|q_n)\pi_0(q_n)J(q'|q_n)}\right). \end{aligned} \tag{14.11}$$

   c. Accept or reject the candidate by generating a uniformly distributed random number $u \sim U(0,1)$ from the interval $[0,1]$ and defining the next value in the Markov chain as

   $$q_{n+1} := \begin{cases} q', & u \le A(q'|q_n), \\ q_n, & u > A(q'|q_n). \end{cases}$$

In other words, the candidate value q' is used as the next value q_{n+1} in the Markov chain with the acceptance probability, and otherwise it is rejected and the old value q_n is repeated.

d. Repeat until the chain is long enough to estimate the parameter q after discarding a sufficiently long burn-in period at the beginning. Compute any statistic of interest from the Markov chain without the burn-in period.

The derivation of the acceptance probability above showed that the values in a Markov chain calculated by the Metropolis–Hastings algorithm satisfy the detailed-balance condition by construction and thus the Markov chain is reversible. We have hence shown the following theorem.

Theorem 14.19 (detailed balance in Metropolis–Hastings algorithm) *The Markov chain constructed by Algorithm 14.18 satisfies the detailed-balance condition and is thus reversible.*

If the proposal distribution used in the Metropolis–Hastings algorithm additionally renders the Markov chain irreducible and aperiodic, then Theorem 14.15 ensures that the stationary distribution is unique. Intuitively, this means that the proposal distribution samples the whole space without prejudice.

Before we implement the Metropolis–Hastings algorithm, we discuss a few important points.

It is maybe the most important feature of the algorithm that the integral $\int \pi(d|q)\pi_0(q)\mathrm{d}q$ cancels in the acceptance probability (14.11). The integral only serves as a normalization factor in Bayes' theorem, but in order to apply Bayes' theorem directly it must be evaluated, which is time-consuming in multidimensional problems. Rendering this integral over the whole parameter space superfluous is the main computational appeal of the Metropolis–Hastings algorithm and other Markov-chain Monte Carlo methods. Instead of evaluating the integral, a sufficiently long Markov chain is computed.

Many choices for the proposal distribution J are possible. Two common choices that ensure that Theorem 14.15 can be applied are the normal distributions

$$J(q'|q_n) := N(q_n, D), \tag{14.12a}$$

$$J(q'|q_n) := N(q_n, V), \tag{14.12b}$$

where D is a diagonal matrix and V is the covariance matrix for the parameter vector q. In the first choice, the elements of the diagonal matrix reflect the scale associated with each parameter. In the second choice, the scale of each parameter can depend on the other parameters via the covariance matrix. Considerations regarding the choices of D or V will be discussed later.

In both these choices for the proposal distributions J, J is symmetric as the calculation

$$\begin{aligned} J(q', q_n) &= \frac{1}{\sqrt{(2\pi)^N|V|}} \mathrm{e}^{-\frac{1}{2}(q'-q_n)V^{-1}(q'-q_n)^\top} \\ &= \frac{1}{\sqrt{(2\pi)^N|V|}} \mathrm{e}^{-\frac{1}{2}(q_n-q')V^{-1}(q_n-q')^\top} = J(q_n, q') \end{aligned}$$

shows.

It is obvious that in a Metropolis algorithm (where the proposal distribution J is symmetric by definition) the acceptance probability (14.11) simplifies to

$$A(q'|q_n) = \min\left(1, \frac{\pi(d|q')\pi_0(q')}{\pi(d|q_n)\pi_0(q_n)}\right).$$

When generating a candidate state q' (which may be a vector), equation (14.10) shows how to calculate a state q' distributed according to a general normal distribution $N(q_n, V)$ using only a random-number generator that produces values distributed according to $N(0,1)$. The definition in (14.10) indeed calculates a suitable candidate state from $N(q_n, V)$ due to the following theorem.

Theorem 14.20 (construction of $N(\mu, V)$**)** *Suppose that $Y \sim N(\mu, V)$ and $Z \sim N(0, I)$ are normally distributed n-dimensional random vectors, where the matrix V is a positive definite and I is the $n \times n$ identity matrix. Then*

$$Y = RZ + \mu,$$

where

$$V = RR^\top$$

and R is a lower triangular matrix.

A proof can be found in [5]. The factorization $V = RR^\top$ can be efficiently computed using a Cholesky factorization (see Theorem 8.10).

The likelihood $\pi(d|q)$ used in Algorithm 14.18 was already discussed in Sect. 14.5.3 and contains an assumption on the error distribution.

Finally, the particular choice of the initial parameter value should not have any influence on any statistic of the Markov chain, since a sufficiently large burn-in period should be discarded anyway.

14.5.6 Implementation of the Metropolis–Hastings Algorithm

Having shown that the Metropolis–Hastings algorithm can be used to construct the posterior distribution, we discuss an implementation for one parameter. We start by importing a few packages.

```
import ProgressMeter
import Statistics
import StatsBase
```

To run Algorithm 14.18, we must provide three arguments, the length `M` of the Markov chain to be calculated, the variance `var` of the proposal distribution, and a specification of the parameter value. The macro `@showprogress` in the `ProgressMeter` package takes three arguments: the time in seconds between updates of the progress bar, a description, and the **`for`** loop whose progress is to be shown.

```
function MH_1D(prior::Function, likelihood::Function,
               proposal::Function,
               M::Int, var::Float64,
               q_min::Float64, q_max::Float64,
               q_init::Float64)::Vector{Float64}
    local q = fill(NaN, M)
    q[1] = q_init

    ProgressMeter.@showprogress 1 "Iterations: " for n in 1:M-1
        local qq::Float64 = q[n] + sqrt(var) * randn(Float64)

        if qq < q_min
            qq = q_min
        end
        if qq > q_max
            qq = q_max
        end

        local A::Float64 =
            min(1, (likelihood(qq) * prior(qq)
                    * proposal(q[n], qq, var)) /
                        (likelihood(q[n]) * prior(q[n])
                         * proposal(qq, q[n], var)))

        q[n+1] =
            if rand(Float64) <= A
                qq
            else
                q[n]
            end
    end

    q
end
```

The function `f` yields the exact value of the model, here the logistic equation, at time `i` times `dt` for given parameter values `q1`, `q2`, and `q3`.

```
function f(i::Int, dt::Float64, q1::Float64, q2::Float64,
           q3::Float64)::Float64
    @assert i >= 0
    @assert dt > 0
    @assert q1 >= 0
    @assert q2 >= 0
    @assert q3 >= 0

    q2*q3 / (q3 + (q2-q3) * exp(-q1*i*dt))
end
```

The function `model` wraps evaluations of `f` for equidistant points in time and returns a vector.

```
function model(q1::Float64, q2::Float64,
               q3::Float64)::Vector{Float64}
    Float64[f(i, dt, q1, q2, q3) for i in 0:N-1]
end
```

You may want to experiment with different values for the number of points and the time step by changing the global variables `N` and `dt` below.

Next we define some global variables. Since we use synthetic measurements, we define the exact parameter values. We will extract the parameter `q2` and assume that it lies in the interval $[0, 5]$. We also know the standard deviation σ of the additive noise. In a real-world example, it would correspond to the measurement error. Based on these constants, we produce the synthetic data by evaluating the `model` function and adding the noise.

```
global N = 50
global dt = 1.0
global q1_exact = 0.1
global q2_exact = 2.0
global q2_min = 0.0
global q2_max = 5.0
global q3_exact = 0.05
global sigma = 0.05
global data =
    model(q1_exact, q2_exact, q3_exact) + sigma * randn(Float64, N)
```

To complete the description of our inverse problem, we define the prior, the likelihood, and the proposal distribution. We use a uniformly distributed prior here.

```
function prior(q::Float64)::Float64
    1 / (q2_max - q2_min)
end
```

The ratio $\pi_0(q')/\pi_0(q_n)$ in the acceptance probability (14.11) may simplify if the prior distribution has a suitable form. In fact, the factor $\pi_0(q')/\pi_0(q_n)$ in (14.11) simplifies to one here.

The likelihood (14.6) uses the standard deviation σ defined above.

```
function likelihood(q::Float64)::Float64
    local S = sum((model(q1_exact, q, q3_exact) .- data).^2)

    exp(-S / (2*sigma^2)) / (2*pi*sigma^2)^(N/2)
end
```

If the likelihood is a normal distribution (as is the case in (14.6) in Sect. 14.5.3), a numerical improvement is possible. Then in the quotient $\pi(d|q')/\pi(d|q_n)$ in the acceptance probability (14.11), the normalization factor $1/(2\pi\sigma^2)^{N/2}$ can be cancelled so that we have

$$\frac{\pi(d|q')}{\pi(d|q_n)} = \mathrm{e}^{(S(q_n)-S(q'))/(2\sigma^2)}. \tag{14.13}$$

This form of the ratio has the advantage that the division of two numbers possibly very close to zero is avoided and thus numerical accuracy is improved. You may want to implement this improvement when possible (see Problem 14.13).

The third function to be defined is the proposal distribution.

```
function proposal(q1::Float64, q2::Float64, var::Float64)::Float64
    @assert var > 0

    exp(-(q1-q2)^2 / (2*var)) / sqrt(2*pi*var)
end
```

Now everything is in place to compute a Markov chain for the parameter q2. The following function call returns a long vector, still containing the burn-in period. The proposal distribution has variance 0.01, and the initial state is the interval midpoint.

```
MH_1D(prior, likelihood, proposal,
      10^5, 0.01,
      q2_min, q2_max, (q2_min+q2_max)/2)
```

In postprocessing (see Problem 14.14), the burn-in period is discarded and the values of the Markov chain are sorted into bins. You may want to experiment with different values for the arguments of MH_1D and for the various (global) variables to see how they affect the posterior distribution.

The results from this function call for determining the parameter q_2 in the logistic equation are shown in Figures 14.3, 14.4, and 14.5. Using a burn-in period of length 10^3 and a bin width of 0.005 for the final histogram, the maximum-a-posterior (MAP) estimate is 2.0025, which is reasonably close to the exact value 2. In other words, the most likely bin is centered around 2.0025. The mean value is

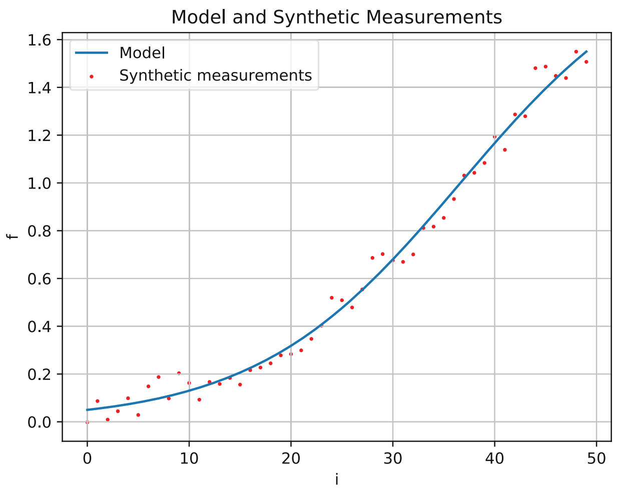

Fig. 14.3 Exact solution of the model equation and 50 synthetic measurements with additive noise ($\sigma = 0.05$).

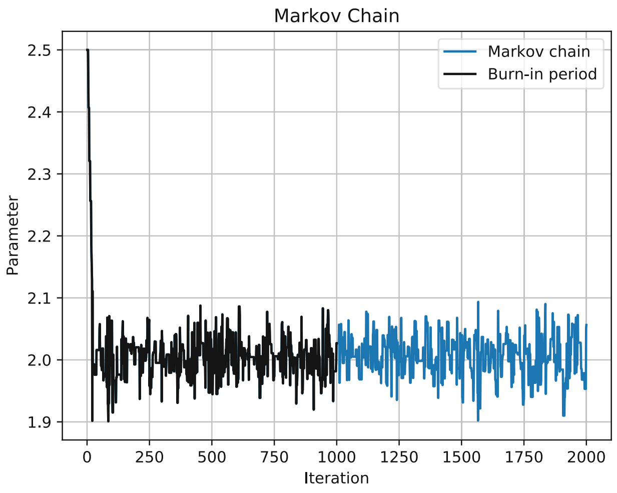

Fig. 14.4 Beginning of a Markov chain of length 10^5 for the measurement data shown in Fig. 14.3. The burn-in period discarded when plotting the next figure, Fig. 14.5, is shown in black.

≈ 2.004 and the median is ≈ 2.004. In all three values, the first three digits agree with the exact value.

In Problem 14.15, you are asked to extend the implementation to multiple dimensions.

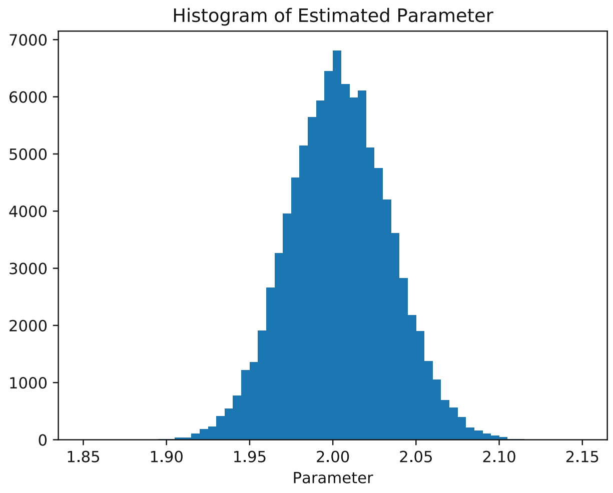

Fig. 14.5 Histogram of the parameter values found in the Markov chain shown in Fig. 14.4 after the burn-in period.

14.5.7 Maximum-a-Posteriori Estimate and Maximum-Likelihood Estimate

The posterior density $\pi(q|d)$ provides complete information about the model parameters calculated from the measurements or observations. From this density, point estimates such as the mean, the median, or a mode can be calculated. Furthermore, confidence intervals are easily calculated as well.

A mode of a continuous probability distribution is a local maximum of its density. A mode of the posterior density $\pi(q|d)$ is called a maximum-a-posterior (MAP) estimate and can be written as

$$q_{\mathrm{MAP}} := \arg\max_q \pi(q|d) = \arg\max_q \pi(d|q)\pi_0(q)$$

by (14.2), since $\pi_D(d)$ is constant with respect to q.

If the prior π_0 is uniform, the MAP estimate q_{MAP} is identical to the maximum-likelihood (ML) estimate

$$(q_{\mathrm{ML}}, \sigma^2_{\mathrm{ML}}) := \arg\max_{q\in Q,\ \sigma^2\in\mathbb{R}^+} L(q, \sigma^2 \mid d),$$

since $\pi(d|q) = L(q, \sigma^2 \mid d)$ by (14.6).

14.5.8 Convergence

Having shown that the Metropolis–Hastings algorithm yields the stationary distribution of the Markov chain and having implemented the basic algorithm, numerical questions still remain. The two main questions are the following. How should the proposal distribution be chosen? And how long should the Markov chain be?

The variance of the proposal distribution affects the Markov chain in an important way. If the variance is too large, a large proportion of the candidate states is rejected because they have smaller likelihoods and the chain stagnates for many iterations. On the other hand, if the variance is too small, the acceptance probability is large, but the chain explores the parameter space only slowly.

In multidimensional problems, the individual parameters should in general be explored at different speeds or scales. This is the reason why covariance matrices D or V are used in the proposal distributions in (14.12) instead of just a multiple of the identity matrix, which would explore all parameters at the same scale. We still do not know a good, automatic method to find such a matrix D or V beyond checking the resulting Markov chain, but we will return to this question in the next section.

How long should the Markov chain be to ensure convergence and to adequately sample the posterior distribution? This is a difficult question, as analytic convergence and stopping criteria are lacking. Convergence of Markov-chain Monte Carlo algorithms can be falsified, but not verified in general. We mention some tests to instill confidence in the convergence of a Markov chain, while more on this subject can be found, for example, in [2, 4].

The simplest and most straightforward method to assess the burn-in period and the convergence behavior is to plot or to statistically monitor the marginal paths of the unknown parameters as in Fig. 14.4. Unfortunately, the chain may meander around a local minimum for a long time before it transitions close to another local minimum or, hopefully, close to a global minimum. But depending on the problem and on the starting point, this may take a very, very long time.

Furthermore, it is possible – at least when the number of unknown parameters is sufficiently small – to compare the parameter values resulting from the Markov chain with the parameter values that stem from applying Bayes' formula (14.2) by calculating the integral directly, e.g. by sparse-grid quadrature.

A more statistical test is to keep track of the ratio of accepted states to the total number of states, which is called the acceptance ratio. Depending on the problem, a rather large range of acceptance ratios can be reasonable, but acceptance ratios between 0.1 and 0.5 are usually considered reasonable (see Problem 14.17). Knowing the acceptance ratio helps tuning the proposal density J.

Another statistical test is to calculate the autocorrelation between subchains of length L of the Markov chain with lag h. While adjacent subchains are likely correlated because of the Markov property, low autocorrelation often indicates fast convergence since in this case independent samples are produced and mixing is good. The autocorrelation function

$$\mathrm{ACF}(L,h) := \frac{\sum_{i=1}^{L-h}(q_i-\bar{q})(q_{i+h}-\bar{q})}{\sum_{i=1}^{L}(q_i-\bar{q})^2}$$

is the ratio of the estimate

$$\frac{1}{L}\sum_{i=1}^{L-h}(q_i-\bar{q})(q_{i+h}-\bar{q}) \tag{14.14}$$

of the autocovariance and the estimate

$$\frac{1}{L}\sum_{i=1}^{L}(q_i-\bar{q})^2$$

of the variance, where $\bar{q}$ is the sample mean. Although the estimate

$$\frac{1}{L-h}\sum_{i=1}^{L-h}(q_i-\bar{q})(q_{i+h}-\bar{q})$$

of the autocovariance suggests itself, the estimate (14.14) with the factor $1/L$ instead of $1/(L-h)$ is often used, since it can be shown to be biased with a bias of (only) order $1/L$ (thus being asymptotically unbiased) and it has the useful property that its finite Fourier transform is nonnegative, among other properties [3, Section 4.1].

14.5.9 The Delayed-Rejection Adaptive-Metropolis (DRAM) Algorithm

We now revisit the question how the proposal distribution can be chosen automatically to ensure expedient parameter scaling during the learning progress. The answer is provided by adaptive Metropolis algorithms [1, 7, 8, 10] such as the DRAM algorithm [6].

Since adaptive Metropolis algorithms change the proposal distribution using the chain history, they violate the Markov property and no longer yield a Markov process. Therefore establishing their convergence to the posterior distribution requires further thought. Examples are criteria such as the diminishing-adaptation condition and the bounded-convergence condition [1, 7, 8].

The Metropolis algorithm becomes adaptive in the DRAM algorithm in the following manner. In the beginning, the covariance matrix is initialized as $V_1 = D$ (diagonal) or $V_1 = V$. Afterwards, the covariance matrix in the n-th step is computed as

$$V_n := s_p(\mathrm{cov}(q_1,\dots,q_n)+\epsilon I_p). \tag{14.15}$$

Here p is the dimension of the parameter space, and the parameter s_p is commonly chosen to be $s_p := 2.38^2/p$ [6]. The initial period without adaptation should be chosen long enough to be sufficiently diverse to make the covariance matrix nonsingular. The purpose of the second term ϵI_p, where I_p is the p-dimensional identity matrix and $\epsilon \geq 0$, is to ensure that V_n is positive definite; it is often possible to set $\epsilon := 0$.

The most straightforward way to calculate the covariance in the formula above is to use the formula for the empirical covariance. However, this becomes increasingly computationally expensive as n increases. A much faster way is to use recursive formulas.

First, the definition of and a recursive formula for the sample mean $\bar{q}_n$ in the n-th step are

$$\bar{q}_n := \frac{1}{n}\sum_{i=1}^{n} q_i = \frac{1}{n} q_n + \frac{n-1}{n}\bar{q}_{n-1}, \tag{14.16}$$

which can be interpreted as a weighted average (see Problem 14.19).

Using the sample mean, the definition and a direct formula for the empirical covariance in the n-th step are

$$\begin{aligned} \operatorname{cov}(q_1,\dots,q_n) &:= \frac{1}{n-1}\sum_{i=1}^{n}(q_i-\bar{q}_n)(q_i-\bar{q}_n)^\top & \text{(14.17a)}\\ &= \frac{1}{n-1}\Big(\sum_{i=1}^{n} q_i q_i^\top - n\bar{q}_n\bar{q}_n^\top\Big) & \text{(14.17b)}\end{aligned}$$

(see Problem 14.20). Based on this direct formula, the recursive formula

$$\begin{aligned}\operatorname{cov}(q_1,\dots,q_n) = {}& \frac{n-2}{n-1}\operatorname{cov}(q_1,\dots,q_{n-1})\\ &+ \frac{1}{n-1} q_n q_n^\top - \frac{n}{n-1}\bar{q}_n\bar{q}_n^\top + \bar{q}_{n-1}\bar{q}_{n-1}^\top \end{aligned} \tag{14.18}$$

for the empirical covariance can be shown (see Problem 14.21).

Using this recursion for the empirical covariance $\operatorname{cov}(q_1,\dots,q_n)$ occurring in (14.15), we find the recursive formula

$$V_n = \frac{1}{n-1}\big((n-2)V_{n-1} + s_p(q_n q_n^\top - n\bar{q}_n\bar{q}_n^\top + (n-1)\bar{q}_{n-1}\bar{q}_{n-1}^\top + \epsilon I_p)\big) \tag{14.19}$$

equivalent to (14.15) above (see Problem 14.22).

The efficiency of the algorithm is further improved if the proposal distribution is adapted only from time to time as in Algorithm 14.21 below.

Delayed rejection is the second aspect of the DRAM algorithm. It means that another candidate state q'' is constructed and given a chance instead of retaining the previous value whenever a candidate state q' has been rejected. This so-called second-stage candidate q'' can be chosen using the proposal function

$$J_2(q'' \mid q_n, q') := N(q_n, \gamma^2 V_n),$$

where V_n is the covariance matrix calculated in the adaptive part of the algorithm above and $\gamma \in (0,1)$ is a constant [6]. Since the constant γ is smaller than one, the proposal function J_2 for the second state is narrower than the original one, which increases mixing. A popular choice for γ is $1/5$.

This proposal function J_2 must be accompanied by a matching acceptance probability A_2 in order to ensure that the detailed-balance condition is satisfied. In Sect. 14.5.5, we used the detailed-balance condition to define a suitable acceptance probability after having decided which proposal distribution to use. If the first proposed state q' is accepted, the detailed-balance condition holds by the calculations in Sect. 14.5.5. Otherwise, if it is rejected, the transition probability is

$$\begin{aligned}P(q''|q_n) &= P(q' \text{ proposed})P(q' \text{ rejected})P(q'' \text{ proposed})P(q'' \text{ accepted})\\ &= J(q'|q_n)(1 - A(q'|q_n))J_2(q'' \mid q_n, q')A_2(q'' \mid q_n, q'),\end{aligned}$$

where $A_2(q'' \mid q_n, q')$ is the second-stage acceptance probability for the proposed state q'' after being in state q_n and having proposed the state q' in the first stage. Substituting this transition probability $P(q''|q_n)$ and the desired stationary distribution $\omega(q) := \pi(q|d)$, i.e., the posterior distribution, into the detailed-balance condition, which now reads

$$P(q''|q_n)\omega(q_n) = P(q_n|q'')\omega(q''),$$

yields

$$\begin{aligned}&\pi(q_n|d)J(q'|q_n)(1 - A(q'|q_n))J_2(q'' \mid q_n, q')A_2(q'' \mid q_n, q')\\ &\qquad = \pi(q''|d)J(q'|q'')(1 - A(q'|q''))J_2(q_n \mid q'', q')A_2(q_n \mid q'', q')\end{aligned}$$

and hence

$$\frac{A_2(q'' \mid q_n, q')}{A_2(q_n \mid q'', q')} = \frac{\pi(q''|d)J(q'|q'')(1 - A(q'|q''))J_2(q_n \mid q'', q')}{\pi(q_n|d)J(q'|q_n)(1 - A(q'|q_n))J_2(q'' \mid q_n, q')} =: r.$$

We can proceed similarly to the case with only one stage in Sect. 14.5.5 to find such an A_2. A suitable acceptance probability is

$$A_2(q'' \mid q_n, q') := \min(1, r), \tag{14.20}$$

since

$$\frac{A_2(q'' \mid q_n, q')}{A_2(q_n \mid q'', q')} = \frac{\min(1, r)}{\min(1, 1/r)} = r.$$

As you will have guessed, these ideas can be extended and third-order, fourth-order, etc. candidate states can be constructed recursively, together with their proposal densities and acceptance probabilities as we just did.

In summary, combining both adapting the proposal distribution and delaying rejection, the DRAM algorithm is the following. The adaptive mechanism ensures that information learned about the posterior distribution is remembered in the long term as the chain progresses. The delayed-rejection part acts in the short term to improve mixing and to avoid stagnation of the chain.

Algorithm 14.21 (delayed-rejection adaptive Metropolis (DRAM))

1. Initialization: choose an initial state q_1 (such that $\pi(q_0|d) > 0$), e.g., as

$$q_1 := \arg\min_q S(q);$$

   choose the number K of steps after which the proposal distribution is adapted; choose the parameter ϵ; choose the initial covariance matrix V_1 (diagonal or symmetric) in the proposal distribution; choose the factor γ (often $\gamma := 1/5$) for the second-stage proposal distribution; and set the iteration number $n := 1$.
   If the likelihood function (14.6) is used, the variance σ^2 must be known. If it is not known a priori as is often the case, there are two options:

   a. use the empirical estimate

$$\sigma^2 := \frac{1}{N-p} \sum_{i=1}^{N} (d_i - f_i(q))^2$$

      for σ_n, where N is the number of observations and p is the dimension of the unknown parameter vector, throughout the iteration or
   b. sample it through realizations of the Markov chain; then the parameters σ_s and n_s (usually $n_s \in [0.01, 1]$) must be chosen during initialization.

2. Iteration:

   a. Every K steps, update the covariance matrix V_n of the proposal distribution using (14.19).
   b. Generate a first-stage candidate state

$$q' := q_n + R_n z$$

      after choosing $z \sim N(0,1)$, where R_n is the Cholesky factorization of V_n (ensuring $q' \sim N(q_n, V_n)$ because of Theorem 14.20).
   c. If σ^2 is sampled by the Markov chain, update σ_n as

$$\sigma_n \sim \mathrm{InvGamma}\left(\frac{n_s + N}{2}, \frac{n_s\sigma_s^2 + S(q_n)}{2}\right),$$

      where InvGamma is the inverse-gamma distribution.

d. Calculate the first-stage acceptance probability

$$A(q'|q_n) := \min\left(1, \frac{\pi(d|q')\pi_0(q')J(q_n|q')}{\pi(d|q_n)\pi_0(q_n)J(q'|q_n)}\right).$$

The first fraction $\pi(d|q')/\pi(d|q_n)$ can be simplified to

$$\frac{\pi(q'|d)}{\pi(q_n|d)} = \mathrm{e}^{(S(q_n)-S(q'))/(2\sigma_n^2)}$$

as in (14.13) if the likelihood has the form (14.6). The second fraction $\pi_0(q')/\pi_0(q_n)$ is equal to one in the case of a uniform prior distribution. The third fraction $J(q_n|q')/J(q'|q_n)$ is equal to one if the proposal distribution is symmetric.

e. Accept or reject the first-stage candidate q' by generating a uniformly distributed random number $u \sim U(0,1)$ from the interval $[0,1]$. If $u \le A(q'|q_n)$, the candidate q' is accepted and we set $q_{n+1} := q'$.

f. If the first-stage candidate was rejected, accept or reject a second-stage candidate.

   i. Generate a second-stage candidate

   $$q'' := q_n + \gamma R_n z,$$

   where $z \sim N(0,1)$ and R_n is the Cholesky factorization of V_n as above.

   ii. Calculate the second-stage acceptance probability $A_2(q'' \mid q_n, q')$ using (14.20), where the fraction $\pi(q''|d)/\pi(q_n|d)$ can be simplified to

   $$\frac{\pi(q''|d)}{\pi(q_n|d)} = \mathrm{e}^{(S(q_n)-S(q''))/(2\sigma_n^2)}$$

   if the likelihood is given by (14.6).

   iii. Accept or reject the second-stage candidate q'' by generating a uniformly distributed random number $u \sim U(0,1)$ from the interval $[0,1]$. The next state becomes

   $$q_{n+1} := \begin{cases} q'', & u \le A_2(q'' \mid q_n, q'), \\ q_n, & u > A_2(q'' \mid q_n, q'). \end{cases}$$

g. Increase the iteration number $n := n + 1$.

3. Iterate until the chain is long enough to estimate the parameter q after discarding a sufficiently long burn-in period at the beginning. Compute any statistic of interest from the Markov chain without the burn-in period.

We close the discussion of the algorithm with a comment on how to treat the variance σ^2 on the right side in (14.6) as a random parameter to be sampled by the Markov chain. The likelihood

$$\pi(d, q \mid \sigma^2) := \frac{1}{(2\pi\sigma^2)^{N/2}} \mathrm{e}^{-S(q)/(2\sigma^2)}$$

is inverse-gamma distributed. Note that $\pi(d, q \mid \sigma^2)$ is different from $\pi(d|q)$. The conjugated prior is

$$\pi_0(\sigma^2) \propto (\sigma^2)^{-(\alpha+1)} \mathrm{e}^{\beta/\sigma^2}$$

with the two parameters α and β. The posterior density is

$$\pi(\sigma^2 \mid d, q) \propto (\sigma^2)^{-(\alpha+1+N/2)} \mathrm{e}^{-(\beta+S(q)/2)/\sigma^2},$$

implying that

$$\begin{aligned} \sigma^2 \mid (d, q) &\sim \mathrm{InvGamma}(\alpha + N/2, \beta + S(q)/2) \\ &= \mathrm{InvGamma}\left(\frac{n_s + N}{2}, \frac{n_s\sigma_s^2 + S(q_n)}{2}\right) \end{aligned}$$

with the two new parameters $n_s := 2\alpha$ and $\sigma_s^2 := \beta/\alpha$. The parameter n_s can be interpreted as the number of observations used in the prior distribution, and the parameter σ_s^2 is the mean squared error of the observations [4]. Usually n_s is chosen to be small, which corresponds to a noninformative prior distribution.

14.6 Julia Packages

The packages under the `Turing` umbrella provide Bayesian inference with general-purpose probabilistic programming. Similarly, the `Mamba` package implements Markov-chain Monte Carlo methods.

14.7 Bibliographical Remarks

A standard textbook on Bayesian inference is [4]. A very good introduction to Bayesian techniques and uncertainty quantification in general can be found in [9].

Problems

14.1 Prove Theorem 14.2.

14.2 Prove Theorem 14.3.

14.3 Consider the example of a fair dice, all of whose six faces have the probability $1/6$. Write down and sketch the probability density and the cumulative probability distribution as discussed in Sect. 14.2. What are the points of discontinuity and the probabilities $P(X = x_i)$? At which points is F_X continuous from the left? At which points from the right? Furthermore, calculate the expected values $\mathbb{E}[X]$ and $\mathbb{E}[X^2]$ as well as the variance $\mathbb{V}[X] = \mathbb{E}[(X - \mathbb{E}(X))^2]$ as Riemann–Stieltjes integrals.

14.4 For the example at the end of Sect. 14.4, plot the iterated posterior probability for different values for the initial prior probability and the likelihood.

14.5 Use separation of variables to solve the logistic equation (14.4).

14.6 Find an example of an irreducible Markov chain and of a reducible one.

14.7 Find an example of a reducible Markov chain with two different stationary distributions.

14.8 Find an example of an aperiodic Markov chain and of a periodic one.

14.9 Find an example of a periodic Markov chain whose limiting distribution $\lim_{n\to\infty} p^n$ does not exist.

14.10 Prove Theorem 14.15.

14.11 Prove Theorem 14.19.

14.12 Prove Theorem 14.20.

14.13 Implement the special form of the acceptance probability for the case of a normally distributed likelihood.

14.14 Write a function that – given a Markov chain and the length of the burn-in period – calculates a histogram (given the bin width), the maximum-a-posteriori (MAP) estimate, and a (symmetric) confidence interval around the MAP estimate based on the histogram and given the percentage of samples to be found in the confidence interval.

14.15 Implement a multidimensional version of the Metropolis–Hastings algorithm in Sections 14.5.5 and 14.5.6.

14.16 Investigate how the parameters of the Metropolis–Hastings algorithm affect the results at the hand of a parameter-estimation problem of your choice.

14.17 Extend the implementations of the Markov-chain Monte Carlo algorithm to calculate and return the acceptance ratio. Observe in an example how the proposal distribution affects the acceptance ratio.

14.18 Extend the implementations of the Markov-chain Monte Carlo algorithms to calculate and return the autocorrelation. Observe in an example how the proposal distribution affects the autocorrelation.

14.19 Prove equation (14.16).

14.20 Prove equation (14.17b).

14.21 Prove equation (14.18).

14.22 Prove equation (14.19).

14.23 Implement the DRAM algorithm Algorithm 14.21 for a one-dimensional parameter.

14.24 Implement the DRAM algorithm Algorithm 14.21 for parameter vectors.

14.25 Consider the example of estimating parameters in the logistic equation, choose the value σ^2 in (14.6), and use both variants of the DRAM algorithm for known and unknown σ^2 to estimate a parameter.

1. Do the parameter values found by both variants differ?
2. When using the variant for unknown σ^2 while pretending to not know the value of σ^2, is the true value of σ^2 approximated?
3. How do n_s and σ_s^2 affect the results?

14.26 Investigate how the parameters of the DRAM algorithm affect the results at the hand of a parameter-estimation problem of your choice.

14.27 Compare the performance of the Metropolis–Hastings and the DRAM algorithms. Which one is easier to use?

14.28 Second-order ordinary differential equations describe physical systems such as spring-mass systems and electrical circuits.

1. Implement a numerical method in Chap. 9 to solve the (general form of the) second-order ordinary differential equation

$$y''(t) + q_1 y'(t) + q_2 y(t) = 0, \qquad y(0) = q_3, \quad y'(0) = q_4 \tag{14.21}$$

   with the four unknown parameters $q = (q_1, q_2, q_3, q_4)$ and use it to generate synthetic measurements as in Sect. 14.5.
2. Use the Metropolis–Hastings and DRAM algorithms

   a. to estimate one of the parameters,
   b. to estimate two of the parameters,

   c. to estimate three of the parameters, and
   d. to estimate all four parameters,

   and compare their performance. Which one is easier to use?
3. Investigate how the parameters of the algorithms affect the results.
4. Compare the results for known and unknown σ^2.

References

1. Andrieu, C., Thoms, J.: A tutorial on adaptive MCMC. *Statistics and Computing* **18**, 343–373 (2008)
2. Brooks, S., Roberts, G.: Convergence assessment techniques for Markov chain Monte Carlo. *Statistics and Computing* **8**(4), 319–335 (1998)
3. Chatfield, C.: *The Analysis of Time Series*, 6th edn. Chapman & Hall (2003)
4. Gelman, A., Carlin, J., Stern, H., Dunson, D., Vehtari, A., Rubin, D.: *Bayesian Data Analysis*, 3rd edn. Taylor & Francis Group, Boca Raton, FL (2013)
5. Golberg, M., Cho, H.: *Introduction to Regression Analysis.* WIT Press, Southampton, UK (2004)
6. Haario, H., Laine, M., Mira, A., Saksman, E.: DRAM: efficient adaptive MCMC. *Statistics and Computing* **16**(4), 339–354 (2006)
7. Haario, H., Saksman, E., Tamminen, J.: An adaptive Metropolis algorithm. *Bernoulli* **7**(2), 223–242 (2001)
8. Roberts, G., Rosenthal, J.: Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics* **18**(2), 349–367 (2009)
9. Smith, R.C.: *Uncertainty Quantification.* SIAM, Philadelphia, PA (2014)
10. Vihola, M.: Robust adaptive Metropolis algorithm with coerced acceptance rate. *Statistics and Computing* **22**(5), 997–1008 (2012)

Index