Numerical Methods for Data Science

David Bindel

Table of contents

Preface

This is an incomplete draft of a text in progress on *Numerical Methods for Data Science* that I am working on during my sabbatical in Spring 2024. It is being written using Quarto as a typesetting system and Julia as the programming language for most of the computational examples. This project began its life as a text to accompany Cornell CS 6241: Numerical Methods for Data Science, a graduate course at Cornell designed for a target audience of beginning graduate students with a firm foundation in linear algebra, probability and statistics, and multivariable calculus, along with some background in numerical analysis. The focus is on numerical methods, with an eye to how thoughtful design of numerical methods can help us solve problems of data science.

Over several iterations of the course, I have seen students from a variety of backgrounds. They often have some grounding in computational statistics, machine learning, or data analysis in other disciplines, which is helpful. But the majority have not had even a semester introductory survey in numerical analysis, let alone a deeper dive. It also became clear that the list of topics was overly ambitious for a one-semester course, even for students with a strong numerical analysis background.

Hence, the new goal of this project is a textbook with three parts. The first piece is an expanded version of a "background notes" handout that I have developed over several years of teaching. This material might be covered in previous courses in computing and mathematics, but many students can use a refresher of some of the details. The second part is what I think of as "Numerical Methods *applied to* Data Science." This corresponds roughly to methods covered in an undergraduate survey course covering standard topics (close to what I usually cover in Cornell CS 4220, though with some changes), with an attempt to provide examples and problems that are identifiably connected to data science. The third part is "Numerical Methods *for* Data Science," and covers more advanced topics, and is appropriate for a graduate course. Both the second and the third parts are a little longer than what I can comfortably cover in a single semester, allowing the reader (or the instructor) some flexibility to pick and choose topics.

1 Introduction

1.1 Overview and philosophy

The title of this course is *Numerical Methods for Data Science*. What does that mean? Before we dive into the course technical material, let's put things into context. I will not attempt to completely define either "numerical methods" or "data science," but will at least give some thoughts on each.

Numerical methods are algorithms that solve problems of continuous mathematics: finding solutions to systems of linear or nonlinear equations, minimizing or maximizing functions, computing approximations to functions, simulating how systems of differential equations evolve in time, and so forth. Numerical methods are used everywhere, and many mathematicians and scientists focus on designing these methods, analyzing their properties, adapting them to work well for specific types of problems, and implementing them to run fast on modern computers. Scientific computing, also called Computational Science and Engineering (CSE), is about applying numerical methods — as well as the algorithms and approaches of discrete mathematics — to solve "real world" problems from some application field. Though different researchers in scientific computing focus on different aspects, they share the interplay between the domain expertise and modeling, mathematical analysis, and efficient computation.

I have read many descriptions of *data science*, and have not been satisfied by any of them. The fashion now is to call oneself a data scientist and (if in a university) perhaps to start a master's program to train students to call themselves data scientists. There are books and web sites and conferences devoted to data science; SIAM even has a journal on the Mathematics of Data Science. But what is data science, really? Statisticians may claim that data science is a modern rebranding of statistics. Computer scientists may reply that it is all about machine learning[1] and scalable algorithms for large data sets. Experts from various scientific fields might claim the name of data science for work that combines statistics, novel algorithms, and new sources of large scale data like modern telescopes or DNA sequencers. And from my biased perspective, data science sounds a lot like scientific computing!

Though I am uncertain how data science should be defined, I am certain that a foundation of numerical methods should be involved. Moreover, I am certain that advances in data science, broadly construed, will drive research in numerical method design in new and interesting

[1] The statisticians could retort that machine learning is itself a modern rebranding of statistics, with some justification.

directions. In this course, we will explore some of the fundamental numerical methods for optimization, numerical linear algebra, and function approximation, and see the role they play in different styles of data analysis problems that are currently in fashion. In particular, we will spend roughly two weeks each talking about

- Linear algebra and optimization concepts for ML.
- Latent factor models, factorizations, and analysis of matrix data.
- Low-dimensional structure in function approximation.
- Function approximation and kernel methods.
- Numerical methods for graph data analysis.
- Methods for learning models of dynamics.

You will not strictly need to have a prior numerical analysis course for this course, though it will help (the same is true of prior ML coursework). But you should have a good grounding in calculus, linear algebra, and probability, as well as some "mathematical maturity."

1.2 Readings

In the next chapter, we give a lightning review of some background material, largely to remind you of things you have forgotten, but also perhaps to fill in some things you may not have seen. Nonetheless, I have never believed it is possible to have too many books, and there are many references that you might find helpful along the way. All the texts mentioned here are either openly available or can be accessed as electronic resources via many university libraries. I note abbreviations for the books where there are actually assigned readings.

1.2.1 General Numerics

If you want to refresh your general numerical analysis chops and have fun doing it, I recommend the *Afternotes* books by Pete Stewart. If you would like a more standard text that covers most of the background relevant to this class, you may like Heath's book (expanded for the "SIAM Classics" edition). I was involved in a book on many of the same topics, together with Jonathan Goodman at NYU. O'Leary's book on *Scientific Computing with Case Studies* is probably the closest of the lot to the topics of this course, with particularly relevant case studies. And Higham's *Accuracy and Stability of Numerical Methods* is a magesterial treatment of all manner of error analysis (highly recommended, but perhaps not as a starting point).

- Afternotes on Numerical Analysis and Afternotes Goes to Graduate School, Stewart
- Scientific Computing: An Introductory Survey, Heath
- Principles of Scientific Computing, Bindel and Goodman
- Scientific Computing with Case Studies, O'Leary
- Accuracy and Stability of Numerical Algorithms, Higham

1.2.2 Numerical Linear Algebra

I learned numerical linear algebra from Demmel's book, and still tend to go to it as a reference when I think about how to teach. Trefethen and Bau is another popular take, created from when Trefethen taught at Cornell CS. Golub and Van Loan's book on Matrix Computations ought to be on your shelf if you decide to do this stuff professionally, but I also like the depth of coverage in Stewart's *Matrix Algorithms* (in two volumes). And Elden's *Matrix Methods in Data Mining and Pattern Recognition* is one of the closest books I've found to the spirit of this course (or at least part of it).

- ALA: Applied Numerical Linear Algebra, Demmel
- Numerical Linear Algebra, Trefethen and Bau
- Matrix Algorithms, Vol 1 and Matrix Algorithms, Vol 2, Stewart
- Matrix Methods in Data Mining and Pattern Recognition, Elden

1.2.3 Numerical Optimization

My go-to book on numerical optimization is Nocedal and Wright, with the book by Gill, Murray, and Wright as a close second (the two Wrights are unrelated). For the particular case of convex optimization, the standard reference is Boyd and Vandeberghe. And given how much of data fitting revolves around linear and nonlinear least squares problems, we also mention an old favorite by Bjorck.

- NO: Numerical Optimization, Nocedal and Wright
- Practical Optimization, Gill, Murray, and Wright
- Convex Optimization, Boyd and Vandenberghe
- Numerical Methods for Least Squares Problems, Bjorck

1.2.4 Machine Learning and Statistics

This class is primarily about numerical methods, but the application (to tasks in statistics, data science, and machine learning) is important to the shape of the methods. My favorite book for background in this direction is Hastie, Tribshirani, and Friedman, but the first book I picked up (and one I still think is good) was Bishop. And while you may decide not to read the entirety of Wasserman's book, I highly recommend at least reading the preface, and specifically the "statistics/data mining dictionary".

- ESL: Elements of Statistical Learning, Hastie, Tribshirani, and Friedman
- Pattern Recognition and Machine Learning, Bishop
- All of Statistics, Wasserman

1.2.5 Math Background

If you want a quick refresher of "math I thought you knew" and prefer something beyond my own notes, Garrett Thomas's notes on "Mathematics for Machine Learning" are a good start. If you want much, much more math for ML (and CS beyond ML), the book(?) by Gallier and Quaintance will keep you busy for some time.

- Mathematics for Machine Learning, Thomas
- Much more math for CS and ML, Gallier and Quaintance

Part I

Background Plus a Bit

Throughout the book, we assume the fundamentals of linear algebra, multivariable calculus, and probability. We also assume some facility with programming and either a knowledge of the basics of Julia[2] or a willingness to pick it up. From this perspective, the chapters in this section of the book should be treated as review, and an impatient reader might reasonably decide to skip over them. But we counsel most readers to be patient enough to at least skim these chapters first, for two reasons:

- We introduce notation that will be useful later in the book but is not used in all introductory courses, even when the topics are standard. This includes things like quasimatrix notation in our discussion of linear algebra and variational notation in our discussion of calculus.
- In our experience, classes vary enough so that most students coming into our courses will not have seen everything in these chapters (or at least they do not remember everything), even if they took all the relevant background courses. My usual advice in such cases is that having a few such gaps in technical background is normal, but diagnosing them and filling them in will help in having a richer understanding of the material at hand.

Of course, even for those readers who have seen all the contents of these chapters before, we hope there will be some value in seeing the material again from a different prspective.

We begin the section with two chapters on computing. In Chapter 2, we discuss the Julia programming language, focusing on the mechanics of the language along with some tips on Julia style. Those readers who are familiar with Julia (or choose to program in some other language) may still appreciate Chapter 3, where we give some basics of performance analysis of computer codes. Much of this chapter is independent of the Julia programming language, though we do also give some pointers on writing fast Julia code.

After discussing computing, we turn to mathematics. In Chapter 4, we review some standard topics in linear algebra, as well as a few topics in multilinear algebra. Unlike our discussion later in the book, where we will often focus on specific bases and concrete spaces, we try to keep an eye on the "basis free" properties of linear spaces and maps upon them. We also try to avoid being overly abstract by regularly tying concepts back to spaces of polynomials and providing code that implements those concepts. Similarly, regular examples in code will feature in our chapters on calculus and analysis (Chapter 5) and probability theory (Chapter 7).

[2]The examples in this book will be in Julia. If you are unfamiliar with Julia but familiar with MATLAB or Octave, you should be able to read most of the code. The syntax may be slightly more mysterious if you primarily program in some other language, but I will generally assume that you have the computational maturity to figure things out.

2 Julia programming

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

Entering this chapter, we assume you have written code in some other languages, but not programmed much (or at all) *in Julia*, the language that we will use for the code in this book. Julia is a relatively young language initially released in 2012. In comparison the first releases of MATLAB and Python were 1984 and 1991, respectively. Despite its relative youth, Julia has become popular in scientific computing for its speed, simple MATLAB-like matrix syntax, and support for a variety of programming paradigms. Among other reasons, we will use Julia for our codes because

- Julia has strong support for interactive computation, similar to Python, R, or MATLAB. I can try things out on a regular command line, or inside a code editor like VSCode, or in a notebook environment (using Jupyter or Pluto).
- Carefully written Julia code can run as fast as codes written in C, C++, or Fortran.
- Like MATLAB or modern Fortran, Julia provides a concise and convenient syntax for linear algebra computations. Julia also provides native support for Unicode, which is nice for writing mathematics in code the same as one would write it on paper.
- Julia has an easy-to-use package manager and a growing ecosystem of useful package. Because so many users of Julia come from scientific computing and data science, there tend to be many good packages for numerical computing.
- Julia treats functions (and closures) as first-class objects and supports a variety of functional programming features.
- Julia has an expressive type system that is useful for code correctness, for writing expressive code via multiple dispatch (different specialized methods with the same name but different type signatures), and for fast execution.
- Julia provides a rich metaprogramming facility, explicitly modeled on Lisp macros. This makes it easy to write Julia programs that write programs, and can be a powerful way to extend the language.

The rest of this chapter is intended to get you started with Julia. It is not comprehensive, and we recommend that the curious reader explore further with the excellent Julia documentation. But we also includes some treatment of more advanced corners of the language.

2.1 Interacting with Julia

We can use Julia non-interactively: typing `julia filename.jl` at the command line will execute any Julia code in the text file `filename.jl`. But both for developing new code and for learning the language, it is very useful to work with Julia interactively. The two most common approaches are

- Typing `julia` at the command line will start an interactive Julia session, sometimes known as a *read-evaluate-print loop* (REPL). One can also start an interactive Julia session from within in some editors, such as Visual Studio.
- One can interact with Julia using a *notebook* interface, such as Jupyter (likely the most popular) or Pluto. We view notebooks using a web browser.

2.1.1 The Julia prompt

Running `julia` with no arguments takes us to the Julia prompt. From here, we can type and run general Julia code, though for anything too long we would usually write the code in a file. But even typing simple commands can have some surprises!

The Julia prompt supports many of the amentities of modern command prompts, including:

- *History*: One can scroll through past commands using the up and down arrows.
- *Emacs-style editing*: Code typed into the command prompt can be edited using navigation key bindings similar to the Emacs editor. For example, Ctrl-A and Ctrl-E jump to the beginning and end of the current line, Ctrl-K "kills" text (putting it into a buffer called the "kill ring"), and Ctrl-Y retrieves the last line from the kill ring.
- *Tab completion*: Typing a partial command into Julia and then pressing the tab key will lead to Julia either finishing the command (if it is unambiguous). If the completion is ambiguous, pressing tab a second time will produce a list of suggested possible completions (if the completion is ambiguous).

Julia allows us to use Unicode characters in code, including code written at the command line. The simplest way to enter these characters is to type a LaTeX special character command and then tab; for example `\pi` followed by tab produces the character π (and in Julia, as in mathematics, this represents 3.14159... unless superceded by another local definition). While it is possible to use Julia without taking advantage of Unicode, in many cases it is the most concise way of writing expressions. For example, writing `isapprox(x, y)` or `x ≈ y` (rendering the approximately equal symbol by tab-completion on `\approx` are equivalent in functionality. But the latter is certainly shorter and arguably more idiomatic. Several popular editors also have Julia modes that support Unicode via tab-completion of LaTeX commands, including Visual Studio, Emacs, and Vim.

From the Julia prompt, one can also access

- *Help*: We access the help system by typing the `?` character and then the name of a function (or module, global variable, etc). Tab completion works in the help system as well, both for completing command names and for entering Unicode. For example, typing `?\pi` followed by a tab will get system help on the constant π.
- *Shell*: If we would like to list directories or quickly run other programs, we can access the shell by typing the `;` character. To exit shell mode, we type the backspace character at an otherwise-empty shell prompt.
- *Packages*: To enter the package manager, we type the `]` character; when done, we can exit the package manager by typing a backspace, as with shell mode. We will have more to say about the package manager later in this chapter.

Pressing Ctrl-C lets us break out of a running Julia command. We can exit from Julia by typing `exit()` at the prompt or pressing Ctrl-D.

2.1.2 Jupyter notebooks

The Jupyter notebook system was first released in 2015 as a multi-language extension to the earlier IPython notebooks. The Jupyter system supports "computational notebooks" built from cells containing text documentation written in Markdown; code written in Julia or another supported language; and code outputs, which may include both text and graphics. Early versions of Jupyter supported Julia, Python, and R (hence the name),though support for many other languages has been added since. In 2018, a more full-featured environment called JupyterLab was released, and today lives alongside the "classic" Jupyter notebook interface. Both classic Jupyter and JupyterLab use the same file formats `.ipynb` files), which may contain all cell types, including computed outputs.

Users typically interact with Jupyter through a web browser interface, though there is also support for working with Jupyter notebooks in editors like Visual Studio Code and Emacs. An example of a running notebook is shown in **?@fig-tbd**. The actual computations for a Jupyter noteook run in a separate "kernel" that communicates with the web browser via a network protocol. The kernel and the brower interface are separate; they may run on the same computer, but are not required to. The kernel responds as though individual cells were typed into the command prompt in the order that the user chooses to execute them, with output then redirected to be shown in the web browser. Because the order of cell execution is chosen by the user, it is not always possible to tell how a notebook arrived at some result just by looking at the output: the user might have executed cells in some unintuitive order, or perhaps a cell was run and then deleted. For this reason, when preparing to save a notebook (whether to share with collaborators or because it is time to quit for the day), we recommend restarting the kernel and re-running all cells from the start so that "what you see is what you get."

The Julia kernel for Julia is provided in the `IJulia` package. This can be installed from the Julia command prompt by entering the package mode (by typing `]` at the prompt) and typing

```
add IJulia
```

Once the `IJulia` package is installed and we have exited back to the normal Julia prompt, we can run Jupyter with a Julia kernel from the Julia prompt by typing

```
using IJulia
IJulia.notebook()
```

Running Jupyter in this way results in a new installation of the Jupyter system, independent of any installations of Jupyter that may already exist on your computer. If you already have Jupyter installed (e.g. as part of a Python installation), you can tell it about the `IJulia` kernel by making sure the `JUPYTER` environment variable points to the desired `jupyter` executable (e.g. by assigning the path to `ENV["JUPYTER"]` from the Julia prompt), and either running `build IJulia` at the package prompt, or at the Julia prompt running

```
using IJulia
installkernel()
```

Once the kernel is installed, one can start Jupyter as normal (e.g. by typing `jupyter lab` at the shell), and then select the Julia kernel when starting a new notebook or loading an existing one.

2.1.3 Pluto notebooks

The Pluto notebook is a Julia-specific take on the computational notebook concept. As with Jupyter, the user interacts with Julia computations through a web interface, with computations executing in a separate Julia process. However, there are several key differences between the two systems:

- Pluto does not distinguish between "code cells" and "documentation cells." The Pluto equivalent of a documentation cell is a code cell that renders Markdown text, which is then displayed as the cell output.
- A single Pluto cell can only contain one statement, though this includes things like function definitions or `begin`/`end` blocks.

- Pluto maintains a *dependency graph* between cells that is used to determine execution order. If cell A defines a symbol that is used within cell B, then executing any update to cell A will also trigger an update of cell B, along with any other cells that depend on cell A, whether directly or indirectly. This mostly addresses the issue of uncertain execution order that sometimes causes headaches for Jupyter users. However, the dependency graph construction is not perfect, and can sometimes be confused by too-clever macros. When this happens, or when there is a change in the packages used by a Pluto notebook, it is usually a good idea to recompute the whole notebook from scratch.
- Pluto notebooks are saved as regular Julia files, with special comments to delimit the cells and indicate the order in which they are displayed. Within the file, the cells are ordered according to the depencency graph, so that a Pluto notebook can also be run as an ordinary Julia program.

2.2 Simple expressions

We start our deeper dive into Julia with a treatment of simple expressions, the very simplest of which are literals (Table 2.1).

Table 2.1: Examples of literals in Julia

Type	Examples
`Int`	`123`, `1_234`
`UInt`	`0x7B`, `0o173` `0b1111011`
`Float64`	`10.1`, `10.1e0`, `1.01e1`
`Float32`	`10.1f0`, `1.01f1`
`Logical`	`true`, `false`
`Char`	`'c'`
`String`	`"string"`
`Symbol`	`:symbol`

Ordinary decimal integer expressions (e.g. `123`) are interpreted as type `Int`, the system signed integer type. In principle, this will be 32-bit or 64-bit depending on the preferred default integer size for the system. In practice, most current systems are 64-bit. Unsigned integer literals can be written in hexadecimal, octal, or binary; these listerals default to the smallest type that will contain them. Floating point literals default to 64-bit, unless an exponent is included. For both integer and floating point types, underscores can be included on input as spacers; that is, `1_234` and `1234` are interpreted the same way.

In addition to standard numerical literals, Julia provides a few constants, including π and e; we type Unicode symbols for these with tab completion on `\pi` and `\euler`. There are

also special constants for exceptional floating point values. For example, `Inf` or `Inf64` denote double-precision floating point infinity and `NaN` or `NaN64` denote double-precision not-a-number encodings; the single-precision (32-bit) analogues are `Inf32` and `NaN32`.

Julia supports string literals surrounded by either single quotes or triple quotes. Triple-quoted strings can include regular quote characters and newlines with no special characters, e.g.

```
box_quote = """
"All models are wrong, but some models are useful."
    -- George Box"""
```

Julia characters correspond to Unicode codepoints, and strings likewise use Unicode encodings. In particular, Julia string literals are represented in program code and in memory using UTF-8, which defines a variable-length encoding for characters. This leads to some complexities in Julia string processing, which we address in Section 2.5.3.

A *symbol* in Julia is represented in memory as a special type of string that is entered in a global dictionary for fast comparisons. Symbols can be used like strings that admit particularly fast comparison, but they are more important for the role that they play in metaprogramming, where evaluating a symbol corresponds to evaluating a variable named in the program.

2.2.1 Assignment, scopes, and variable types

Like many other languages, the Julia assignment operation `=` represents binding a name to a value. For example,

```
x = 10
```

assigns a name `x` to the value `10`. Julia also allows *destructuring* assignments in which an ordered collection (a tuple or vector) is unpacked into multiple outputs, e.g.

```
L, U, p = lu(A)  # Assign multiple outputs
```

There is also a form of destructuring assignment for when the right hand side has named quantities, e.g.

```
p = (x = 1.0, y = 2.0, z = 3.0)
(; y, z) = p  # Sets y = p.y, z = p.z
```

This format works for named tuples and for structures.

What is more complicated is the *scope*, i.e. the part of the code in which this assignment holds.

2.2.1.1 Scope

Global variables in Julia belong to a "top-level" namespace called the *global scope*. Technically, there is a separate global scope for each *module*, as we will describe in more detail in **?@sec-tbd**. But when we start a Julia session, we are working with a new (anonymous) module with its own global scope, and when we refer to "the" global scope, this is the namespace that we mean.

Certain types of code blocks create new *local scopes*, or variable namespaces with bindings that are only meaningful inside the code block. Julia is *lexically scoped*, so the visibility of variable bindings depends on the code structure, not the execution path through the code. Local scopes can (and often are) nested. The `let` statement in Julia serves the sole purpose of creating a new scope with local bindings; for example:

```
let
    p = 0
    let p = 1      # New version of p
        println(p) # Prints 1 (p from inner scope)
    end
    println(p)   # Prints 0 (p from outer scope)
end
```

```
1
0
```

The `let` statement actually has two parts:

- A series of assignment statements on the same line as the `let`, which result in bindings of variable names that are purely local to the `let`;
- A code block that by default makes use of the bindings in the assignment block *and* any bindings in the outer scope.

The difference can be somewhat subtle. For example, a single newline can change the semantics of the previous example by moving the assignment `p=0` from the assignment block of the `let` (where it creates a new version of `p`) to the code block of the `let` (where it re-binds the symbol `p` from the enclosing scope):

```
let
    p = 0
    let
        p = 1      # Reassign outer version of p
        println(p) # Prints 1
    end
```

```
    println(p)   # Prints 1 again
end
```

```
1
1
```

At the same time, we can use the `local` keyword to make a version of the variable that is explicitly local to the `let` code block:

```
let
    p = 0
    let
        local p = 1  # Create local version of p
        println(p)   # Prints 1
    end
    println(p)   # Prints 0
end
```

```
1
0
```

Julia distinguishes between constructs that create "hard" local scopes (such as function and macro definitions or the assignment part of `let` blocks) and those that create "soft" local scopes (such as loops, conditionals, the code part of `let` statements, and `try` blocks), with the two differing in their treatment of assignments to existing names. For example, a `for` statement establishes a soft local scope, in which assignments to existing variables in enclosing scopes do not create a shadow variable:

```
function do_more_stuff()
    x = 0
    for k = 1:5
        x = k    # x is the same variable as before
    end
    println(x)   # Prints 5
end
```

However, if `x` were not declared before the `for` loop, the version inside the loop body would be purely local to the loop:

```
function do_more_stuff()
    for k = 1:5
        x = k    # x is local to this iteration
    end
    println(x)   # Error -- x doesn't exist here!
end
```

In any local scope, we can explicitly declare that we want to use a global variable. For example:

```
function modify_global_xyz()
    xyz = 0
    let
        global xyz
        xyz = 1
    end
end

modify_global_xyz()
println(xyz) # Prints 1
```

```
1
```

However, if we try to assign to a global variable name in a local scope without using the `global` keyword, different things will happen depending not only on whether the local scope is "hard" or "soft," but also on whether we are in an interactive session or running code non-interactively:

- Hard local scope: Always create a shadow of the global variable.
- Soft local scope: If interactive, assign to the global variable. Otherwise, create a shadow variable and issue a warning to the user.

2.2.1.2 Variable types

All *values* in Julia have concrete types. When a *variable* in Julia is given a type, it restricts the types of values that can be assigned to the variable to only those that are compatible. For example, if we would like `x` to represent a floating point number, we could write:

```
x :: Float64 = 0  # Implicitly converts the integer 0 to the floating point 0.0
```

Once the type of `x` has been specified, we cannot assign an incompatible value to `x`:

```
x :: Float64 = 0  # OK so far, x is now floating point 0.0
x = "Oops"        # Error!
```

Though values must have concrete types, we can give variables *abstract* types. For example, Julia has many numeric types, all of which are subtypes of type `Number`. We can assign any numeric value to a variable declared to have type `Number`, but non-numeric values are not allowed:

```
x :: Number = 0   # OK, x is now Int(0), and Int is a subtype of Number
x = 0.0           # Still OK, Float64 is a subtype of Number
x = "Oops"        # Error!
```

We will have much more to say about types in **?@sec-tbd**.

2.2.2 Arithmetic operations

Table 2.2: Standard arithmetic operations

Syntax	Description
`+x`	Unary plus (identity)
`-x`	Unary minus
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x ÷ y`	Integer division
`x \ y`	Inverse divide (≡ `y/x`)
`p // q`	Rational construction
`x ^ y`	Power
`x % y`	Remainder

We show some of the standard Julia arithmetic operations in Table 2.2. These mostly behave similarly to other languages, but with some subtleties around division. In particular:

- The ordinary division operator produces a floating point result even when the inputs are integers and the result is exactly representable as an integer. For example, `6/3` yields `2.0`.
- Integer division always rounds toward zero; for example, `14 ÷ 5` yields `2`, while `-14 ÷ 5` yields `-2`. The remainder operation is consistent with integer division.

- Integer division and remainder can also be used with non-integer types; for example, `14.1 ÷ 5` yields `2.0`, and `14.1 % 5` yields `4.1`.
- The backslash (inverse divide) operation can be used with any numeric types, but is most frequently used in the context of linear algebra. When `A` and `b` are a matrix and a vector, we use `A\b` to compute $A^{-1}b$ without computing an explicit inverse.
- The construct `p // q` is used to construct a *rational* number. The numerator and denominator must be integers.

Julia also allows for implicit multiplication when a literal number (integer or floating point) precedes a symbol with no space. For example, `2.0im` produces a complex floating point value equal to $2i$.

There are *updating* operations of all the arithmetic operations (except for rational construction), formed by combining the operation name with an equal sign. These are essentially equivalent to a binary operation followed by an assignment; for example,

```
x += 1  # Equivalent to x = x + 1
x *= 2  # Equivalent to x = x * 2
```

2.2.3 Logical operations

Julia supports the usual logical and `p && q`, or `p || q`, and `!p` constructions. The arguments to these operations must be logical; unlike some languages, we cannot interpret integers or lists as logical values, for example. The and and or operators are *short-circuited*, meaning that the second argument is only executed if the first argument is insufficient to determine the truth value. For example, consider the code fragment

```
let
    x = true || begin println("Hello 1"); true end
    y = false || begin println("Hello 2"); true end
    x, y
end
```

```
Hello 2
```

```
(true, true)
```

Both `x` and `y` are assigned to be true, but only the string `"Hello 2"` is printed.

Julia also supports a ternary *conditional* operator: `p ? x : y` evaluates `p` and returns either `x` or `y` depending whether `p` is true or false, respectively. Like the logical and and or, the ternary

conditional operator is short-circuiting; only one of the `x` and `y` clauses will be evaluated. The conditional operator is equivalent to a Julia `if` statment; that is, these two statements do the same thing:

```
z1 = p ? x : y
z2 = if p then x else y end
```

2.2.4 Comparisons

Julia provides several different comparison operations. Most types support tests for structural equality (`==`) or inequality (`!=` or `≠`). This is a forgiving sort of test, allowing for things like promotion of the arguments to a common type. For example, all the following expressions evaluate to `true`:

```
1 == 1   &&
1 != 2   &&
2 == 2.0 &&
[1; 2] == [1.0; 2.0]
```

```
true
```

The substitutional equality operation (`===`) and inequality operation (`!==`) test whether two entities are *indistinguishable*. This is usually a more restrictive notion of equality. For example, all the following expressions evaluate to `true`:

```
1 === 1  &&
1 !== 2  &&
2 !== 2.0
```

```
true
```

Values that correspond to *mutable* objects never satisfy `===`, unless they are exactly the same object. For example, Julia allows us to change the contents of an array (it is *mutable*), but not the contents of a string (it is *immutable*). Therefore, both the following expressions are `true`:

```
"12" === "12"  &&
[1; 2] !== [1; 2]
```

```
true
```

For the most part, substitutional equality implies structural equality, but not vice-versa. We say "for the most part" because of the example of the default *not-a-number* (`NaN`). According to the standard floating point semantics (Chapter 9), not-a-number is not equal to anything in the floating point system, including itself. But the `NaN` symbol does satisfy substitutional equality. Therefore, both the following are `true`:

```
NaN != NaN  &&
NaN === NaN
```

```
true
```

We will have more to say about the vagaries of floating point equality when we get to Chapter 9. For now we simply mention that Julia also supports an *approximate* equality operation that can be used for comparing whether floating point quantities are within some tolerance of each other. For example, the following statements are both `true`:

```
1.0 ≈ 1.00000001 &&
1.0 ≠ 1.00000001
```

```
true
```

In addition to testing for equality or inequality, we can test *ordered* types with `<`, `>`, `<=` (or `≤`), and `>=` (or `≥`). Ordered types include integers, floating point numbers, strings, symbols, and logicals. For strings and symbols, we use *lexicographic* ordering (aka alphabetic ordering); for logicals, we have `false < true`.

```
:a < :b   &&
"a" < "b" &&
false < true
```

```
true
```

2.2.5 Simple calls

Simple calls in Julia have the form `function_name(args)`. The arguments are comma-separated, starting with any *positional* arguments followed by any *keyword* arguments. Julia uses the types of the provided positional arguments to distinguish between different *methods* with the same name and number of arguments. Most operators in Julia are just a thin layer of "syntactic sugar" that hide ordinary method calls. For example, writing `1+2` and `+(1,2)` are equivalent;

but `1+2` and `1.0+2.0` invoke different methods, since the types are different (both `Int` for the former expression, both `Float64` in the latter).

A call may provide only the leading positional arguments if the remaining arguments are assigned default values. Keyword arguments always have a default value. For example, the linear algebra routine `cholesky` takes positional arguments: a matrix `A` and a flag variable that indicates whether the computation should use pivoting or not (defaulting to `NoPivot()`). In addition, Cholesky takes a logical keyword argument `check` that indicates whether the routine should produce an exception on failure (`check == true`, the default) or should produce no exception (`check == false`). Therefore, all the following are valid calls to `cholesky`

```
C = cholesky(A)  # without pivoting, check result
C = cholesky(A, check=false)  # without pivot, no check
C = cholesky(A, RowMaximum()) # pivoted, check result
C = cholesky(A, RowMaximum(), check=false), # pivot, no check
```

Julia passes arguments using a "call-by-sharing" convention, in which names within the function are bound to the values passed in. This means that if a mutable value (e.g. an array) is passed as an argument to a function, the function is allowed to modify that argument. By convention, functions that modify their arguments end with an exclamation mark. For example, calling `cholesky(A)` function produces a new output matrix and leaves `A` alone, while `cholesky!(A)` modifies the matrix `A`.

Sometimes, it is convenient to package the positional arguments to a function into a tuple. A tuple can be *unpacked* into an argument list by following it with `...`, sometimes called the *splat* operation. This is sometimes useful in function calls; for example

```
a = (1.0, 1.00000001)
isapprox(a...)      # Same as isapprox(1.0, 1.00000001)
```

While it is mostly used for calling, Julia does also allow this sort of unpacking in other contexts than function calls; for example

```
b = (a, 2)          # Yields ((1.0, 1.00000001), 2)
bunpack = (a..., 2) # Yields (1.0, 1.00000001, 2)
```

2.2.6 Broadcasting operations

Function calls and operations can be applied elementwise to arrays using broadcasting syntax. This involves putting a dot before the operator name or after the function name. For example:

```
θs = [0; π/4; π/2; 3π/4; π] cos.(θs) # [1; sqrt(2)/2; 0; -sqrt(2)/2; -1]
θs .+ 1                              # [1; π/4+1; π/2+1; 3π/4+1; π+1]
```

The `@.` macro can be used to "dot-ify" everything in an expression, e.g.

```
xs =range(0, 1, length=100) y = @. xs^2 + 2xs + 1.0
```

evaluates the map $x \mapsto x^2 + 2x + 1$ to every element of the vector `xs`.

Vectorized operations are critical to performance in some languages. In Julia, they are not so performance critical. However, they are convenient for concise code.

2.2.7 Indexing and slicing

Square brackets are used to access elements of an array or similar object,or to refer to sub-arrays of an array (slices). Indices are one-based; and in the context of indexing, the symbol `end` refers to the final index. For example,

```
lst = [5; 2; 2; 3]
lst[2]        # Evaluates to 2
lst[end]      # Evaluates to 3
lst[1:2]      # Evaluates to [5; 2]
lst[[2; 1]]   # Evaluates to [2; 5]
lst[2:end]    # Evaluates to [2; 2; 3]
lst[2:3] .= 4 # Now lst is [5; 4; 4; 3]
```

We can index into an array to either get elements or subarrays or to assign to them. When getting elements or subarrays, the default behavior is to get a copy of the extracted data; so, for example

```
lst = [5; 2; 2; 3]
sublst = lst[1:2]    # New storage for sublst
sublist[:] = [3; 4]  # Changes contents of sublst, not lst
```

If we want to modify a subarray in the main array from which it was extracted, we can use a *view*. A *view* has its own indexing, but uses the storage of another object. For example:

```
lst = [5; 2; 2; 3]
sublst = view(lst, 1:2)  # Provides a view into lst
sublist[:] = [3; 4]      # lst is now [3; 4; 2; 3]
```

We frequently provide indices that are integers (to get individual elements) or arrays of integers (whether ranges like `1:2` or concrete index lists like `[1 4 2]`). But we can also use *logical* indexing, in which an iterable boolean list tells us whether the corresponding entry should be kept in the result or excluded. For example:

```
lst = [5; 2; 2; 3]
idx_big = (lst .> 2)  # [true, false, false, true]
lst[idx_big]          # Results in [5; 3]
```

In some circumstances, we also have index spaces that are fundamentally non-integer (or can be). An example is with *named tuples*, which can be accessed either by indexing positionally or by name. For example, consider the following tuple:

```
xyz = (x=4, y=5, z=6)  # Create a named tuple
xyz[1]                 # Returns 4 (the value of x)
xyz[:y]                # Returns 5 (the value of y)
```

Our examples so far have been for single-index arrays (vectors), but everything generalizes natural to multi-dimensional arrays.

2.2.8 Structure access

Several compound data types contain named *fields*, including record types (defined with the `struct` keyword) and named tuples (mentioned in the previous section). These fields can be accessed with the syntax `val.field`. For example, consider again the `xyz` tuple from the previous example:

```
xyz = (x=4, y=5, z=6)  # Create a named tuple
xyz.z                  # Returns 6 (structure access syntax)
```

Many compound data types are immutable, so that fields cannot be assigned. For those that are mutable, though, we can also use `val.field` on the left hand side of an assignment statement.

2.2.9 Strings

One particularly convenient feature of Julia is *string interpolation*. For example, consider the following Julia code:

```
let
  sqrt2approx = 99.0/70.0
  "The error in sqrt(2)-$sqrt2approx is $(sqrt(2)-sqrt2approx)"
end
```

```
"The error in sqrt(2)-1.4142857142857144 is -7.215191261922271e-5"
```

In evaluating the string, Julia replaces `$sqrt2approx` with the value of the variable `sqrt2approx`, and it replaces `$(sqrt(2)-sqrt2approx)` with the string representation of the number `sqrt(2)-sqrt2approx`. For more controlled formatting (e.g. printing only a few digits of the error), the standard recommendation is to use the `@sprintf` macro from the `Printf` package.

In addition to building strings by interpolation, Julia lets us build strings by *concatentation*. The operator `*` concatenates two strings; for example `"Hello " * "world"` produces `"Hello world"`. The `^` operation on strings can also be used for repetition; for example `"Hello "^2` produces `"Hello Hello "`.

2.3 Control flow

We have already discussed one form of control flow, with short-circuit evaluation of logical and conditional operators. Beyond this, we have conditionals and loops. We treat exception handling in Section 2.6.

2.3.1 Conditional statements

Julia has an `if`/`elseif`/`else`/`end` statement structure. The statement is actually an expression, which evaluates to whatever branch is taking. For example

```
is_leap_year(year) =
    if year % 400 == 0
        true # Multiples of 400 years are leap years
    elseif year % 100 == 0
        false # Multiples of 100 years otherwise are not
    elseif year % 4 == 0
        true  # Which is an exception to "every fourth year"
    else
        false # Otherwise, not a leap year
    end
```

If none of the conditions are satisfied and there is no `else` clause, the result evaluates to `nothing`.

Unlike loops, `if` statements do not introduce a local scope.

2.3.2 Loops

A `while` loop executes while the loop condition is true

```
function gcd(a,b)
    while b != 0
        a, b = b, a%b
    end
    a
end
```

A `for` loop iterates over a collection. Perhaps most frequently, this is an integer range:

```
function mydot(x, y)
    @assert length(x) == length(y)
    result = 0.0
    for i = 1:length(x)
        result += x[i]*conj(y[i])
    end
    result
end
```

However, we can also iterate over other collections, whether they are concrete or abstract. For example, the `zip` function creates an abstract collection of tuples constructed from elements of parallel lists:

```
function mydot2(x, y)
    @assert length(x) == length(y)
    result = 0.0
    for (xi, yi) in zip(x, y)
        result += xi*conj(yi)
    end
    result
end
```

In the `for` loop syntax, we can use `=`, `in`, or `∈` between the loop variable name and the collection to be iterated over. We can define new things that a `for` loop can iterate on by overloading the `Base.iterate` function.

The `break` statement jumps out of a loop in progress.

Both `while` loops and `for` loops produce "soft" local scope: references to already-used names from surrounding scopes will not be shadowed, but any new names will be bound in the local scope.

2.4 Functions and methods

2.4.1 Defining functions

Julia provides three primary syntactic variants for defining functions:

- Using the `function` keyword
- Using an equational definition
- Using the `->` operator

The `function` keyword is the standard choice for defining complicated functions that might involve several statements. We have seen a few examples already, but for now let's consider again the Euclidean algorithm for the greatest common denominator, seen in Section 2.3.2:

```
function gcd(a, b)
    while b != 0
        a, b = b, a%b
    end
    a
end
```

The defined function `gcd` takes two positional arguments (`a` and `b`). On a function call, the concrete arguments to the call are bound to the names `a` and `b` in a local scope, and then the body of the function is then evaluated to obtain a return value. We note that as with other compound statements in Julia (e.g. `begin/end` blocks and `let/end` blocks), the value of the function body is the last expression in the body – in this case, the final value of `a`. We could also write `return a`, but this is not strictly necessary in this case. In general, `return` statements are only really needed when we want to exit before reaching the end of a function body.

For short expressions, it is more concise to define functions using Julia's equational syntax:

```
N0(x) = (x-1.0)*(x-2.0)*(x-3.0)/6.0
```

The function body on the right-hand side of the function definition can be any valid Julia expression, including a compound statement inside a `begin/end` constrution. Conventionally, though, we use simple expressions for the function bodies when using this syntax.

Our examples thus far have all involved *named* functions, but Julia also supports *anonymous* functions (sometimes called *lambdas*). We can either write anonymous functions with the `function` keyword, or using the `->` syntax:

```
f1 = function(x) x^2 + 2x + 1 end
f2 = x -> x^2 + 2x + 1
```

The body in the `function` version is a compound expression ending with `end`. The body in the `->` version is a single expression, similar to what we saw with the equational syntax.

2.4.2 Defaults and keywords

Most Julia functions take arguments. We can define arguments later in the list to have default values; for example:

```
increment(x, amount=1) = x + amount
```

Default argument values may depend on earlier arguments.

Julia functions can take *keyword* arguments as well as positional arguments. Keyword arguments must have a default value. In the function definition, a semicolon separates the positional argument list from the keyword argument list. For example, consider the following routine to print a list in sorted order:

```
function print_sorted(l; by=identity)
    l = sort(l, by=by)
    for v in l
        println(v)
    end
end
```

The keyword argument `by` indicates a function that returns the sort key, and is passed through to the call to the `sort` function in the first line. We need the semicolon separator in the function definition to distinguish keyword arguments from positional arguments with defaults. However, there is no such requirement when calling the function, so the call to `sort` only uses commas to separate arguments.

2.4.3 Closures

When a Julia function is created inside an enclosing local scope, it may refer to variables that were created inside that scope. In this case, Julia creates a *closure* of the function definition together with variable bindings from the surrounding environment.

For example, suppose we have a naive Newton iteration for solving a 1D equation $f(x) = 0$:

```
"""    naive_newton(f, df, x; maxiter=10, ftol=1e-8)

Run Newton iteration to find a zero of `f`
(with derivative `df`) from the starting
point `x`.  Returns when the magnitude of `f`
is less than `ftol`, otherwise errors out
after running for `maxiter` steps.
"""
function naive_newton(f, df, x; maxiter=10, ftol=1e-8)
    fx = f(x)
    for _ = 1:maxiter
        if abs(fx) < ftol
            return x
        end
        x -= fx/df(x)
        fx = f(x)
    end
    if abs(fx) < ftol
        return x
    end
    error("Did not converge in $maxiter steps")
end
```

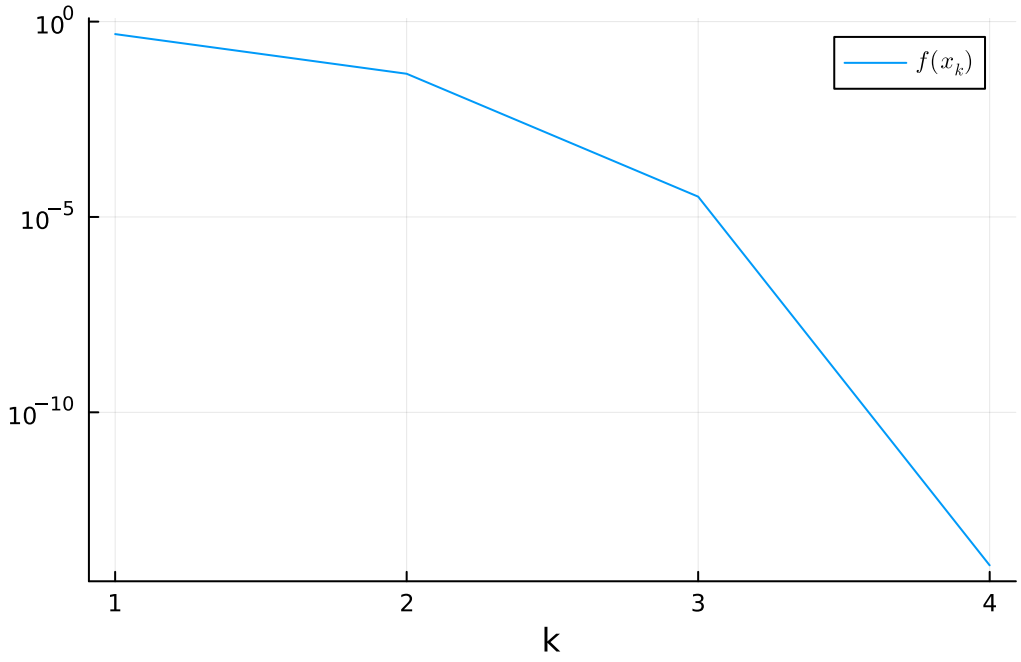
We would like to evaluate the list of function values in order to plot the convergence. We could do this by modifying the `naive_newton` routine, or we could record each call to the function.

```
function plot_naive_newton(f, df, x; maxiter=10, ftol=1e-8)

    # Create a list of values and a closure that
    # records function evaluations to the list
    fs = []
    function frecord(x)
        fx = f(x)
        push!(fs, fx)
        fx
    end

    x = naive_newton(frecord, df, x,
                     maxiter=maxiter, ftol=ftol)
    plot(filter(fx->fx > 0, abs.(fs)),
         xlabel="k",
         yscale=:log10, label="\$f(x_k)\$")
```

```
end
```

Now a call to `plot_naive_newton(sin, cos, 0.5)` (for example) will show a plot of the function value magnitudes at each step of the iteration.

```
plot_naive_newton(sin, cos, 0.5)
```



2.4.4 Methods

Julia allows a single function name to provide a common interface to different implementations (methods) based on the number and type of the arguments. The process of looking up the appropriate method from the invocation can take into account the types of more than one of the arguments (multiple dispatch). For example, consider the definitions

```
myrotate(z :: Complex, θ) = exp(1im*θ)*z
myrotate(xy :: Vector, θ) =
    let c = cos(θ), s = sin(θ), x = xy[1], y = xy[2]
        [c*x-s*y; s*x+c*y]
    end
```

Both versions of `myrotate` correspond to a rotation of a plane, but we have different implementations depending on whether the first argument is a complex number or a vector.

```
println(myrotate(1.0 + 0.0im, π/4))
println(myrotate([1.0; 0.0], π/4))
```

```
0.7071067811865476 + 0.7071067811865475im
[0.7071067811865476, 0.7071067811865475]
```

Methods are also very useful for defining structurally recursive functions to process data structures. For example, consider "flattening" a structure of nested tuples into a vector:

```
function flatten_tuple(t)
    result = []
    process(x :: Tuple) = process.(x)
    process(x) = push!(result, x)
    process(t)
    result
end

let
    t = ((1,2), (3,(4,5)), 6)
    flatten_tuple(t)
end
```

```
6-element Vector{Any}:
 1
 2
 3
 4
 5
 6
```

We will have more to say about Julia's type system in Section 2.5.

2.4.5 Operator overloading

As mentioned previously, operators in Julia are just a special type of function. We can add methods to these functions in order to implement operators acting on new types. For example, consider a new type `Polynomial` representing polynomials expressed in the monomial basis (we will return to this in Section 2.5):

```
struct Polynomial
    coeffs :: Vector{Float64}
end
```

We would like to be able to add, subtract, and multiply polynomials. For example, to add implementation of the `+` and `*` operators from `Base` that works with a `Polynomial`, we would write

```
function Base.:+(p :: Polynomial, q :: Polynomial)
      n = max(length(p.coeffs), length(q.coeffs))
      c = zeros(n)
      c[1:length(p.coeffs)] = p.coeffs
      c[1:length(q.coeffs)] += q.coeffs
      Polynomial(c)
end

function Base.:*(p :: Polynomial, q :: Polynomial)
      n = length(p.coeffs) + length(q.coeffs) - 1
      c = zeros(n)
      for (i, pi) in enumerate(p.coeffs)
            c[i:length(q.coeffs)+i-1] += pi*q.coeffs
      end
      Polynomial(c)
end
```

If we would like to treat constants as a type of polynomial, we need to add a conversion routine and some additional methods for the `+` and `*` functions.

```
Base.convert(::Type{Polynomial}, c :: Number) = Polynomial([c])
Base.:+(p :: Polynomial, q) = p + convert(Polynomial, q)
Base.:+(p, q :: Polynomial) = convert(Polynomial, p) + q
Base.:*(p :: Polynomial, q) = p * convert(Polynomial, q)
Base.:*(p, q :: Polynomial) = convert(Polynomial, p) * q
```

We might also want to be able to evaluate a polynomial, since it represents a function:

```
function (p :: Polynomial)(x)
    px = 0*x
    for c in reverse(p.coeffs)
        px = px*x+c
    end
    px
end
```

With these definitions in place, we can write code like

```
let
    x = Polynomial([0; 1])  # x
    p = x*(2*x+1)
    p(2)
end
```

```
10.0
```

The ambitious reader might also try writing subtraction, polynomial division and remainders.

2.4.6 Higher-order functions

A *higher-order* function is a function that takes another function as an argument. Higher-order functions like `map`, `filter`, and `reduce` are the bread-and-butter of functional programming languages, and Julia supports these as well:

```
map(x->x+1, [1; 2; 3])           # Yields [2; 3; 4]
filter(x->x > 0, [1; -1; 2; 3])  # Yields [1; 2; 3]
reduce(+, [1; 2; 3])             # Yields 6
```

Higher-order functions are quite common in numerical methods, as many of the operations that we want to perform are naturally expressed in terms of operations on functions (computing integrals and derivatives, optimizing, finding zeros). Indeed, we already saw a simple example of a higher-order function with the `naive_newton` example in Section 2.4.3.

2.4.7 do syntax

A common pattern in higher-order functions is to have a function that takes one function argument and potentially a few other operations. When the argument function is small, it is convenient to use the `->` operation to define a short anonymous function, as we did in Section 2.4.6. But sometimes when the argument function is not small, it gets clunky to write it in place as an anonymous function. For example, suppose we wanted a verbose version of the `filter` example above, one that prints out a description of what is going on at each step. Using the notation we have written so far, we could write something like

```
filter(function(x)
           ispositive = x > 0
           action = ispositive ? "Keep it" : "Toss it"
           println("Testing x = $x: $action")
           ispositive
       end, [1; -1; 2; 3])
```

```
Testing x = 1: Keep it
Testing x = -1: Toss it
Testing x = 2: Keep it
Testing x = 3: Keep it

3-element Vector{Int64}:
 1
 2
 3
```

Because this pattern occurs so often in Julia, however, there is a special syntax that makes it easier to write:

```
filter([1; -1; 2; 3]) do x
    ispositive = x > 0
    action = ispositive ? "Keep it" : "Toss it"
    println("Testing x = $x: $action")
    ispositive
end
```

```
Testing x = 1: Keep it
Testing x = -1: Toss it
Testing x = 2: Keep it
Testing x = 3: Keep it

3-element Vector{Int64}:
 1
 2
 3
```

Behind the scenes, Julia rewrites constructions of the form

```
f(fargs) do args
    body
end
```

to

```
f(function(args) body end, fargs)
```

Beyond being used with functions like `map` and `reduce`, the `do` syntax is often used in Julia for tasks like I/O that involve a required setup and teardown phase. For example, writing to a file in Julia is often done using syntax like the following:

```
open("foo.txt", "w") do f
    println(f, "Hello, world")
end
```

The version of the `open` method that takes a function argument in the terse syntax will call the function with the file handle, then close the file afterward.

2.4.8 Composition and pipes

In addition to nested calls (e.g. `f(g(x))`), Julia supports two special syntax constructs for function composition:

- The composition operator (`∘`) acts like composition in mathematics.
- The pipeline operator (`|>`) chains together inputs and outputs like the pipe symbol in a Unix shell command or pipes in R.

For example:

```
sqr(x) = x.^2
norm2a(v) = sqrt(sum(sqr(v)))
norm2b = sqrt ∘ sum ∘ sqr
norm2c(v) = v |> sqr |> sum |> sqrt
```

2.4.9 Generators and comprehensions

Generators and *comprehensions* are a final type of specialized use of anonymous functions. A generator expression has the form `expr for var in collection`, with the option of additional comma-separated `var in collection` expressions. The expression evaluates to an iterable collection that is lazily evaluated. For example, consider the generator expression

```
g = ((i,j) for i=1:2, j=1:2)
```

We can loop over the generator the same as we would over any other collection; with each iteration, the expression `(i,j)` is evaluated with a different binding of `i` and `j` from the iteration space, visited in column-major order (early indices change fastest). For example, consider the output of the following code fragment:

```
let
    g = ((i,j) for i=1:2, j=1:2)
    for v in g
        println(v)
    end
end
```

```
(1, 1)
(2, 1)
(1, 2)
(2, 2)
```

Iterable expressions can be used in many places where Julia would accept a list or vector. For example:

```
sum(i^2 for i = 1:10)
```

```
385
```

An *array comprehension* is similar to a generator, but with surrounding square brackets. Rather than producing a lazily-evaluated collection, it produces a concrete array of results. For example, if $k(x, y)$ is a bivariate function of interest, we can use a generator to concisely produce a matrix of function samples over a regular grid:

```
xgrid = range(0.0, 1.0, length=20)
kxx = [k(x,y) for x in xgrid, y in xgrid]
```

2.5 Types

At a certain level, computers only manipulate sequences of bits. But programmers usually work at a more abstract level, where those bit sequences are interpreted as different *types* of values: these 64 bits represent an integer, those 64 bits represent a floating point number, that other set of 64 bits represents the string `"program"` (with a null terminator). One of the classic distinctions drawn among programming languages is how types are treated. In *statically-typed* languages like C++ and Fortran, every variable and expression is associated with a particular type that can be determined at compile time[1]. In *dynamically-typed* languages like MATLAB, Python, and R, values have types but variables are not restricted in the types of values to which they may refer. But, of course, things are not so simple — C++ is mostly statically typed, but run-time polymorphism allows some elements of dynamic typing; while modern Python is mostly dynamically typed, but allows optional *type annotations*.

Julia is a dynamically-typed language, in which variables are just names that can be bound to values of different types. At the same time, Julia has a rich type system with optional type declarations using the `::` syntax as described in Section 2.2.1.2. These type declarations serve three purposes:

- *Correctness*: Sometimes an operation only makes sense with certain types of inputs, and declaring that explicitly makes it possible for the system to provide an early error rather than an error at runtime.
- *Expressivity*: Declaring the types of arguments to functions allows the same function name to dispatch to multiple methods, as discussed in Section 2.4.4.
- *Performance*: When the Julia system can infer types in a function call, the just-in-time compiler can create faster specialized versions of the code, much like what we would find in C or Fortran.

While type declarations is often helpful, Julia also benefits from not requiring them. As an example, consider the `gcd` function from Section 2.3.2. We might argue that it is appropriate to restrict the arguments to be `Integer` types, i.e.

```
function gcd(a :: Integer, b :: Integer)
    while b != 0
        a, b = b, a%b
    end
    a
end
```

[1] In the beginning, the creators of the C language allowed the same piece of memory to be treated as belonging to different types. This has made a lot of people very angry and been widely regarded as a bad move.

However, the Euclidean algorithm also makes sense in more general Euclidean domains, such as polynomials. So we might extend the operations on the `Polynomial` type from Section 2.4.5 so that comparisons with zero and the remainder operator are both allowed. With this extension, the original `gcd` function (without the `:: Integer` annotations) would produce the correct result.

2.5.1 Type hierarchy

Julia types form a tree-shaped hierarchy, where each abstract type may have multiple *subtypes*. For example, `Integer` is a subtype of `Number`, and concrete integer types like `Int64` are subtypes of `Integer`. The syntax `<:` is used to denote a subtype relation; for example, we would write `Integer <: Number` to test that `Integer` is indeed a subtype of `Number`. At the root of the type hierarchy is the type `Any`.

Only *abstract* types may serve as supertypes in Julia. Concrete types, whether primitive types (like `Logical`, `UInt16`, or `Float64`), or various compound types, are always leaves of the hierarchy.

It is possible (and sometimes useful) to declare new abstract types, e.g.

```
abstract type MyAwesomeAbstraction end
```

2.5.2 Numeric types

The *primitive* concrete numeric types in Julia are

- Floating point numbers `Float16`, `Float32`, and `Float64`. These are all subtypes of `AbstractFloat`. The 32-bit and 64-bit formats are supported in hardware, but operations with the 16-bit format are software only. We will have much more to say about floating point in Chapter 9.
- Signed integer types `Int8`, `Int16`, `Int32`, `Int64`, and `Int128`. These are all subtypes of `Signed`, which is a subtype of `Integer`. The type `Int` is an alias for `Int64` or `Int32`, depending on whether Julia is being run on a 64-bit or a 32-bit system.[2]
- Unsigned integer types `UInt8`, `UInt16`, `UInt32`, `UInt64`, and `UInt128`. These are all subtypes of `Unsigned`, which is a subtype of `Integer`.

[2]At the time of this writing, the default `Int` type in Julia is almost always 64 bits. It is not so easy to find 32-bit systems any more.

Technically, type `Logical` is also a subtype of type `Integer`, and it can be used in arithmetic expressions like an integer – for example, `true + true` evaluates to `2`, while `1.0 * false` evaluates to `0.0`. However, we do not recommend using logical variables in this way.

In addition to the primitive types, Julia supports parameterized types for rational and complex numbers. For example, `3 // 4` produces a representation of 3/4 of type `Rational{Int64}`, while `1.0 + 1.0im` produces a representation of $1 + 1i$ of type `Complex{Float64}`.

Julia also supports arbitrary-precision `BigInt` and `BigFloat` types, though we will rarely make use of these.

2.5.3 Strings and symbols

Strings in Julia inherit from the `AbstractString` type, and are logically sequences of Unicode characters[3]. Unicode characters (or "code points") in turn inherit from the `AbstractChar` type, and are associated with 32-bit unsigned integers. The character `A`, for example, corresponds to code point U+0041; this is character 65 in decimal, but Unicode codepoints are typically written in hexadecimal. More generally, code points U+0000 through U+007F correspond to the classic ASCII character set, which includes Latin characters and punctuation used in standard English.

The default character type `Char` in Julia takes 32 bits in memory. The default `String` type is *not* an array of 32-bit `Char` entities; instead, Julia uses UTF-8 strings, which use a variable-length encoding. Unicode codepoint numbers corresponding to ASCII are encoded with one byte per character, and other characters may take more than one byte to represent. Hence, the `length` of a string (in characters) and the `sizeof` a string (in bytes) mean different things in Julia. These are also different complexity: computing the length (in codeunits) in a UTF-8 string requires that we scan the string from start to end. To avoid the time cost of a scan, *indexing* into a standard string is done by bytes. For example, in the string `s = "sπot"`, we can safely refer to `s[1]` (s), `s[2]` ( ), `s[4]` (o), and `s[5]` (t); but accessing `s[3]` will cause a runtime exception.

Symbols inherit from type `Symbol`. They are encoded as "interned" UTF-8 strings, which means that the string is stored in a hash table, and the data passed around the Julia environment is the location in that hash table.

[3] In 2003, Joel Spolsky wrote a blog post on "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)", which includes the memorably line "if you are a programmer working in 2003 and you don't know the basics of characters, character sets, encodings, and Unicode, and I *catch* you, I'm going to punish you by making you peel onions for 6 months in a submarine. I swear I will." That was two decades ago at the time of this writing. Unicode as a concept and UTF-8 as an encoding are both ubiquitous. There is no excuse for not knowing a little about it.

2.5.4 Tuples

The type of a tuple is a Cartesian product of the tuple elements. In Julia, this is represented as `Tuple`, a parameterized type whose parameters are the component types. For example, the type of `(1.0, 2)` is `Tuple{Float64,Int64}`. Named tuples are similar; the type of `(x=1.0, y=2)` is `@NamedTuple{x :: Float64, y :: Int64}`.

2.5.5 Structs and parameterization

We declare a structure type (sometimes called a record type) with the `struct` keyword; for example, we might define a type for rational numbers as

```
# Version 0
struct MyRational <: Number
    p
    q
end
```

Then `MyRational(1,2)` will give us an instance of `MyRational` with `p` set to 1 and `q` set to 2. However, this definition also allows us to legitimately write `MyRational("bad", "idea")`, and we probably do not want to allow a ratio of two strings. A second attempt at a definition might be

```
# Version 1
struct MyRational <: Number
    p :: Integer
    q :: Integer
end
```

With this definition, we are restricted to integer values for `p` and `q`. However, we may still be unhappy with this if we read about Julia performance, and see the advice to make fields be concrete types to produce efficient code (`Integer` is an abstract type). This brings us to our third attempt, which involves making `MyRational` a *parametric* type:

```
struct MyRational{T<:Integer} <: Number
    p :: T
    q :: T
end
```

Writing `{T<:Integer}` after the structure name says that we want to have a type parameter `T` which is a subtype of `Integer`. With this definition, the constructor `MyRational(1,2)` will give us an instance of type `MyRational{Int64}` (assuming a 64-bit machine).

We may decide that we would like to make `p` and `q` have reduced form, and disallow a zero denominator. To do this, we would create a new *inner constructor* function:

```
struct MyRational{T<:Integer} <: Number
    p :: T
    q :: T
    MyRational(p :: Integer, q :: Integer) =
        if q == zero(q)
            error("Divide by zero")
        else
            p, q = promote(p, q)
            r = gcd(p, q)
            new{typeof(p)}(p÷r, q÷r)
        end
end
```

There are several things to unpack here:

- The `MyRational` takes two abstract `Integers`. These could be different types of integers! In order to gracefully deal with this, we *promote* the arguments to a common type with the `promote` command (see Section 2.5.12)
- Inside the `MyRational` constructor, we cannot just call another constructor called `MyRational`. Instead, we call `new` to create the object. This is only used inside of inner constructors.
- The `new` call needs a type parameter, which we get from the type of `p`.

In addition to inner constructors, we can also define an *outer constructor*, so named because the definition is outside the `struct` statement. For example, we might want to define a constructor that takes just one argument (with the second being an implicit 1):

```
MyRational(p :: Integer) = MyRational(p, one(p))
```

As described earlier, we might also decide that we wanted to overload the operators. Note that this does not require any special magic with the type parameters to `MyRational`:

```
Base.:+(a :: MyRational, b :: MyRational) =
    MyRational(a.p*b.q + a.q*b.p, a.q*b.q)
Base.:-(a :: MyRational, b :: MyRational) =
    MyRational(a.p*b.q - a.q*b.p, a.q*b.q)
Base.:*(a :: MyRational, b :: MyRational) =
    MyRational(a.p*b.p, a.q*b.q)
Base.:/(a :: MyRational, b :: MyRational) =
    MyRational(a.p*b.q, a.q*b.p)
```

What if we want to get the type associated with `p` and `q`? We can, of course, use `typeof(p)` to do this, but we can also write a method to extract the type. Unlike the overloaded operators, this method *does* explicitly refer to the type parameter, and so we need some additional syntax:

```
inttype(:: MyRational{T}) where {T} = T
```

Unpacking this construction, we have

- An anonymous argument of type `MyRational{T}`. We do not need to give the argument a name, because we do not care about the value of the variable; we only care about its type.
- The `where {T}` clause to specify that this method definition is parameterized by the type `T`.

2.5.6 Structs and inheritance

Inheritance from abstract types can be a used to provide shared functionality in two ways:

- We can keep some information in abstract type parameters.
- We can write generic methods for an abstract type that use specific implementations for concrete types (Julia's version of "duck typing" from languages like Python).

As a somewhat silly example, let's consider an abstraction around a list of points in d-dimensional space, where each point is considered to be a length-d tuple.

```
abstract type Points{d,T} end
dim(p :: Points{d,T}) where {d,T} = d
Base.eltype(p :: Points{d,T}) where {d,T} = T
```

The dimension `d` and the element type `T` are type parameters for the abstract type, and nothing special needs to be done for types inheriting from `Points{d,T}` to use the `dim` or `eltype` functions.

The abstraction of "list of points" can be implemented with multiple data structures and associated versions of the `Base` functions `length` and `getindex`. For example, we could use parallel arrays for the coordinates:

```
struct ParallelPoints2d{T} <: Points{2,T}
    x :: Vector{T}
    y :: Vector{T}
end
Base.length(ps :: ParallelPoints2d) = length(ps.x)
Base.getindex(ps :: ParallelPoints2d, i) = (ps.x[i], ps.y[i])
```

Or we could keep the points in a matrix, where each column represents a different point:

```
struct MatrixPoints{d,T} <: Points{d,T}
    xy :: Matrix{T}
end
Base.length(ps :: MatrixPoints) = size(ps.xy,2)
Base.getindex(ps :: MatrixPoints, i) = Tuple(view(ps.xy,:,i))
```

For the case of `MatrixPoints`, it is redundant to explicitly specify the dimension of the space when we provide a matrix of points to initialize the object. Julia can infer from the type of the matrix what `T` should be; but to get `d`, it needs some help.

```
MatrixPoints(xy :: AbstractMatrix) =
    MatrixPoints{size(xy,1),eltype(xy)}(xy)
```

Just using the `length` and `index` (and the `dim` function from above), we can write generic implementations of operations like converting a list of points to a matrix, whatever the underlying data structure.

```
function to_matrix(ps :: Points)
    result = zeros(eltype(ps), dim(ps), length(ps))
    for j = 1:length(ps)
        result[:,j] .= ps[j]
    end
    result
end

to_matrix(ParallelPoints2d([1, 2, 3], [4, 5, 6]))
```

```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Or we might like to be able to iterate over a list of points; for this, we again only need to know that `length` and `getindex` are implemented to be able to provide a generic iterator.

```
Base.iterate(ps :: Points, i=1) =
    i > length(ps) ? nothing : (ps[i], i+1)
```

There are some times when we might want slightly more specialized behavior based on the type. For example, for output, we might want to list 1D points as just individual numbers rather

than as tuples (i.e. output `1` rather than `(1,)`). To do this, we can write a generic version of the `show` routine (used for output), but also one that specializes to the one-dimensional case. Julia will choose the more specific version for a given type.

```
# Generic version of show
function Base.show(io :: IO, ps :: Points)
    print(io, "[ ")
    for p in ps
        print(io, p)
        print(io, " ")
    end
    print(io, "]")
end

# Version of show for 1D points
function Base.show(io :: IO, ps :: Points{1,T}) where {T}
    print(io, "[ ")
    for p in ps
        print(io, p[1])
        print(io, " ")
    end
    print(io, "]")
end

# Illustrate printing in 1D or 2D with different structs
let
    println(ParallelPoints2d([1, 2, 3], [4, 5, 6]))
    println(MatrixPoints([1 2 3; 4 5 6]))
    println(MatrixPoints([1 2 3 4 5 6]))
end
```

```
[ (1, 4) (2, 5) (3, 6) ]
[ (1, 4) (2, 5) (3, 6) ]
[ 1 2 3 4 5 6 ]
```

Between type parameters and dispatch to different methods for concrete types, the Julia language provides most of the amenities that would be implemented via inheritance in other languages. Perhaps the biggest exception to this is that Julia provides no direct support for identifying data fields that should be implemented for all structs inheriting from some abstract type – though one can write macros that provide such support.

2.5.7 Mutable structs

By default, the entries of a struct are *immutable*. Using the example from the previous section, once we have created a `MyRational` with a given `p` and `q`, we cannot change those values. Note that if any field of a structure is a writeable container (like an array), we *can* write into that container; the container is immutable, but its contents are not. Still, sometimes we would genuinely like to be able to update a structure, and for this we have *mutable* structures, which are declared with the `mutable` keyword. For example, consider the following counter struct:

```
mutable struct FooCounter
    p :: Int
    FooCounter() = new(0)
end
```

Now we can write an increment operator that updates this structure (using an exclamation mark in the name, as is the convention for functions that change their inputs):

```
increment!(f :: FooCounter) = (f.p += 1)

foo = FooCounter()        # foo.p initialized to 0
println(increment!(foo))  # Prints 1
println(increment!(foo))  # Prints 2
```

2.5.8 Array types

An `Array{T,d}` is a d-index array of type `T`. The type aliases `Vector{T}` and `Matrix{T}` correspond to arrays with `d` set to 1 and 2, respectively. It is possible to make `T` an abstract type, but it is generally much more efficient to make `T` a concrete type.

In addition to arrays, there are several constructs that are subtypes of the `AbstractArray` interface. These includes:

- `Range`: An abstraction for a regularly-spaced range, which can be constructed by the colon operator (e.g. `1:3`) or with the `range` command.
- `Adjoint`: (Conjugate) transpose of a matrix.
- `BitArray`: A compact storage format for an array of logical values.
- `SparseMatrixCSC`: A sparse matrix format.
- `SubArray`: A submatrix of an existing matrix, which can be constructied using the `view` macros.
- `StaticArray`: An fast type for small arrays whose dimensions are part of the type.

These constructs can all be indexed, sliced, iterated over, and otherwise treated much like ordinary arrays (though not all are writeable).

2.5.9 Other collections

In addition to tuples and array-like collections, Julia supports dictionary and set types. A dictionary contains key-value pairs, written as `key => value` in Julia; the `=>` operator constructs a `Pair` object. The structure supports addition and deletion of pairs, testing for keys, iteration, and lookup (either with `get`, which supports default values, or with `dict[key]`, which does not). There are a few variants, the most frequently used of which is the ordinary dictionary (`Dict`).

Julia also supports a `Set` type, which supports addition and removal, testing for inclusion or subset relations, and operations like union, intersection, and set differences.

2.5.10 Unions, Nothing, and Missing

Julia allows `Union` types whose values may be one of a selection of types. Two specific examples have to do with data that may be "null" or missing[4]. In Julia, the type `Nothing` (with instance `nothing`) is used the way that programmers use "null" in some other languages, while the type `Missing` (with instance `missing`) is used to indicate missing data. These types are most often used in the context of `Union{T, Nothing}` or `Union{T, Missing}` types. For example, values of type `Union{T, Missing}` can either be something of type T or the value `missing`.

2.5.11 Types of types and value types

Types are themselves things that can be represented and manipulated in Julia. Each type has type `DataType`, which is a subtype of the parametric abstract type `Type{T}`. In general, a type `T` (and no other type) is an instance of `Type{T}`. This can be useful for certain special types of method dispatch situations. Similarly, *value types* (`Val{T}`) are parameterized types that take a value as a parameter, and can be used for specialized method dispatch or for helping the compiler with performant code.

2.5.12 Type conversions and promotions

The `convert` function in Julia is used to convert values of one type to another type (when possible). The system calls `convert` automatically when assigning a value of one type to a variable or field of another type. The first argument is the target type, and the second is the

[4]The notion of "null" is surprisingly subtle. This can refer to something that is "deliberately left blank" (e.g. a default value for an optional argument), something that is "well-defined but unknown" (e.g. a record of someone's age) or something that is inappropriate to the task at hand (e.g. spouses' name when not married).

value to be converted. Defining new methods for `convert` is a good use of the `Type` type; for example, we might define a conversion from an `Integer` to the `MyRational` type defined earlier as:

```
Base.convert(::Type{MyRational}, p :: Integer) = MyRational(p)
```

A more elaborate example might involve converting `MyRational` values with different integer parameter types:

```
Base.convert(::Type{MyRational{T}}, r :: MyRational) where {T} =
    MyRational(T(r.p), T(r.q))
```

The `promote` function in Julia takes a list of values and converts them both to a compatible greatest type. We typically would not write new methods for `promote`; instead, we write new methods for `promote_rule`. Again using `MyRational` as an example, we might have

```
Base.promote_rule(::Type{MyRational{T}}, ::Type{S}) where {T<:Integer,S<:Integer} =
    MyRational{promote_type(T,S)}
```

2.6 Exception handling

Julia has throw/catch system for handling exceptional conditions, similar to many other modern languages (Python, MATLAB, R, Java, C++). Information about exceptions is encoded in via a struct that is a subtype of `Exception`, which we `throw` when the error is detected:

```
"""    imlog(x)
Compute the imaginary part of the principal branch
of the log of negative number x.
"""
imlog(x) =
    (x >= zero(x)) ?
    throw(DomainError("Argument must be negative")) :
    log(-x)
```

The `error` function throws an instance of an `ErrorException`.

Exceptions that are thrown at some point in a call chain can be *caught* by callers using a `try`/`catch` statement.

```
"""    generous_log(x :: Number)
Computes the log of a number; if a domain error is
thrown for the log of a negative real number, tries
with a complex version of the number.
"""
generous_log(x :: Number) =
    try
        log(x)
    catch e
        if e isa DomainError
            log(Complex(x))
        else
            throw(e)
        end
    end
```

These statements may also have an optional `else` clause that is executed when the `try` statement succeeded without exceptions; and an optional `finally` clause that is executed when exiting the `try`/`catch`, independent of how the exit occurred.

2.7 Documentation

Functions, types, and modules can all have *documentation strings* just before them, which are referenced by the Julia help system. Documentation strings are formatted using Markdown. A fenced code block beginning with `jldoctest` can be used to produce a usage example that is suitable for automatic testing.

2.8 Vectors, matrices, and linear algebra

We have already discussed some of Julia's array support. However, one of the attractive features of Julia is the support not only for arrays, but for numerical linear algebra computations. Many of these facilities are in the `LinearAlgebra` package, which we will almost always use throughout this book.

2.8.1 Building matrices and vectors

By default, we think of a one-dimensional array as a column vector, and a two-dimensional array as a matrix. We can do standard linear algebra operations like scaling (`2*A`), summing like types of objects (`v1+v2`), and matrix multiplication (`A*v`). The expression

```
w = v'
```

represents the adjoint of the vector `v` with respect to the standard inner product (i.e. the conjugate transpose). The tick operator also gives the (conjugate) transpose of a matrix. We note that the tick operator in Julia does not actually copy any storage; it just gives us a re-interpretation of the argument. This shows up, for example, if we write

```
let
    v = [1, 2]  # v is a 2-element Vector{Int64} containing [1, 2]
    w = v'      # w is a 1-2 adjoint(::Vector{Int64}) with eltype Int64
    v[2] = 3    # Now v contains [1, 3] and w is the adjoint [1, 3]'
end
```

```
3
```

Julia gives us several standard matrix and vector construction functions.

```
Z = zeros(n)   # Length n vector of zeros
Z = zeros(n,n) # n-by-n matrix of zeros
b = rand(n)    # Length n random vector of U[0,1] entries (from Random)
e = ones(n)    # Length n vector of ones
D = diagm(e)   # Construct a diagonal matrix
e2 = diag(D)   # Extract a matrix diagonal
```

The identity matrix in Julia is simply `I`. This is an abstract matrix with a size that can usually be inferred from context. In the rare cases when you need a *concrete* instantiation of an identity matrix, you can use `Matrix(I, n, n)`.

2.8.2 Concatenating matrices and vectors

In addition to functions for constructing specific types of matrices and vectors, Julia lets us put together matrices and vectors by horizontal and vertical concatenation. This works with matrices just as well as with vectors! Spaces are used for horizontal concatenation and semicolons for vertical concatenation.

```
y = [1; 2]        # Length-2 vector
y = [1 2]         # 1-by-2 matrix
M = [1 2; 3 4]    # 2-by-2 matrix
M = [I A]         # Horizontal matrix concatenation
M = [I; A]        # Vertical matrix concatenation
```

Julia uses commas to separate elements of a list-like data type or an array. So `[1, 2]` and `[1; 2]` give us the same thing (a length 2 vector), but `[I, A]` gives us a list consisting of a uniform scaling object and a matrix — not quite the same as horizontal matrix concatenation.

2.8.3 Transpose and rearrangemenent

Julia lets us rearrange the data inside a matrix or vector in a variety of ways. In addition to the usual transposition operation, we can also do "reshape" operations that let us interpret the same data layout in computer memory in different ways.

```
# Reshape A to a vector, then back to a matrix
# Note: Julia is column-major
avec = reshape(A, prod(size(A)));
A = reshape(avec, n, n)

idx = randperm(n)   # Random permutation of indices (need to use Random)
Ac = A[:,idx]       # Permute columns of A
Ar = A[idx,:]       # Permute rows of A
Ap = A[idx,idx]     # Permute rows and columns
```

2.8.4 Submatrices, diagonals, and triangles

Julia lets us extract specific parts of a matrix, like the diagonal entries or the upper or lower triangle. Some operations make separate copies of the data referenced:

```
A = randn(6,6)    # 6-by-6 random matrix
A[1:3,1:3]        # Leading 3-by-3 submatrix
A[1:2:end,:]      # Rows 1, 3, 5
A[:,3:end]        # Columns 3-6

Ad = diag(A)      # Diagonal of A (as vector)
A1 = diag(A,1)    # First superdiagonal
Au = triu(A)      # Upper triangle
Al = tril(A)      # Lower triangle
```

Other operations give a *view* of the matrix without making a copy of the contents, which can be much faster:

```
A = randn(6,6)          # 6-by-6 random matrix
view(A,1:3,1:3)         # View of leading 3-by-3 submatrix
view(A,:,3:end)         # View of columns 3-6
Au = UpperTriangular(A) # View of upper triangle
Al = LowerTriangular(A) # View of lower triangle
```

2.8.5 Matrix and vector operations

Julia provides a variety of *elementwise* operations as well as linear algebraic operations. To distinguish elementwise multiplication or division from matrix multiplication and linear solves or least squares, we put a dot in front of the elementwise operations.

```
y = d.*x   # Elementwise multiplication of vectors/matrices
y = x./d   # Elementwise division
z = x + y  # Add vectors/matrices
z = x .+ 1 # Add scalar to every element of a vector/matrix

y = A*x    # Matrix times vector
y = x'*A   # Vector times matrix
C = A*B    # Matrix times matrix

# Don't use inv!
x = A\b    # Solve Ax = b *or* least squares
y = b/A    # Solve yA = b or least squares
```

2.8.6 Things best avoided

There are few good reasons to compute explicit matrix inverses or determinants in numerical computations. Julia does provide these operations. But if you find yourself typing `inv` or `det` in Julia, think long and hard. Is there an alternate formulation? Could you use the forward slash or backslash operations for solving a linear system?

2.9 Useful packages

Beyond the `LinearAlgebra` package mentioned in Section 2.8, there are several other packages that we will use or recommend in this book:

2.9.1 Statistics

The `Statistics` package provides a variety of standard statistics computations (mean, median, etc). There is also a `StatsBase` package that provides some additional statistics. There has been discussion of refactoring these.

The `StatsFuns` package provides a variety of special functions that occur in statistical computations (e.g. the log gamma function or various common pdf, cdf, and inverse cdf functions). The `Distributions` package is recommended as providing a more convenient interface, but sometimes we will want the low-level interface.

2.9.2 Sparse matrices

The `SparseArrays` package is generally useful for setting up sparse matrix and vectors. In addition, Julia provides packages for a variety of sparse direct and iterative solvers, including the `SuiteSparse` package (direct solvers) and the `IterativeSolvers` package (iterative solvers).

2.9.3 DataFrames

The `DataFrames` package provides functionality for manipulating tabular data similar to the Python `Pandas` package.

2.9.4 Automatic differentiation

Though we will only make limited use of these within the book, there are several Julia packages that support automatic differentiation, including `ForwardDiff`, `ReverseDiff`, `Zygote`, and `Enzyme`.

2.9.5 Plots and tables

The `Plots` package provides a standard interface for producing plots with a variety of plotting backends. There are other plotting packages in Julia (the second-most popular of which is probably `Makie`), but the `Plots` package makes a good starting point.

For text output of tabular data, we recommend the `PrettyTable` package. This can also write to a variety of output formats, including HTML, Markdown, and LaTeX.

2.9.6 I/O

The `FileIO` package provides a common interface for reading (and writing) a wide variety of data file types, including text, tabular, image, sound, and video formats.

2.9.7 Development tools

The `Test` package in Julia provides a variety of useful tools for writing unit tests. When tests fail and it is necessary to step through code, the `Debugger` package is useful. For performance evaluation, we recommend the `BenchmarkTools` and `Profile` packages, as well as `DispatchDoctor` for checking type stability.

2.10 Macros and metaprogramming

Julia provides a LISP-like *metaprogramming* facility that allows us to manipulate Julia code as data. This is a very powerful tool, allowing us to (for example) write *macros*, i.e. code that generates new code on our behalf. Metaprogramming requires care to get right, and the usual advice is to only use it when other tools do not suffice. However, there are some times when it really is the right tool, and so we will touch on it here at least lightly.

2.10.1 Code as data

The heart of metaprogramming is the understanding that code is a type of data, and can be manipulated as such. We distinguish code that we want to treat as data by *quoting* it, which involves bracketing the code by `:(` and `)` or by `quote` and `end`. For example, the expression

```
quoted_sum = :(1 + 1)
```

```
:(1 + 1)
```

assigns `quoted_sum` to the unevaluated code for `1 + 1`. To evaluate the code, we can call the `eval` function

```
eval(quoted_sum)
```

```
2
```

Quoted expressions are printed by the Julia interpreter in natural Julia syntax. Internally, though, these expressions are not represented by strings, but by *expression trees*. We can see the structure of the expresion tree with the `dump` command, e.g.

```
dump(quoted_sum)
```

```
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
```

The expression tree data structure involves internal nodes of type `expr` that have a `head` field indicating the type of operation represented, and an `args` array indicating the arguments of the operation. The leaves in the expression tree are literals, including symbols.

Just as we can use *string interpolation* to build strings that include computations, we can also use interpolation with quoted code; for example

```
let
    x = 1
    :($x + 1)
end
```

```
:(1 + 1)
```

We can produce the same expression tree without quoting by explicitly using the constructor for `Expr` objects, e.g.

```
let
    x = 1
    Expr(:call, :+, x, 1)
end
```

```
:(1 + 1)
```

2.10.2 Macros

For the reader who *really* wants to learn to write macros, we recommend the documentation together with the book *On Lisp* (Graham 1993). We will write small macros a few times in order to simplify our life; and in any case, we also use macros written by others often enough that we are obliged to say a few words from a user's perspective.

A macro maps a tuple of input expressions to an output expression, which is then compiled. The actual work of the macro code occurs at compile time; at run time, the system evaluates whatever output expression was generated by the macro. Macros are frequently used for various types of programming utilities. Debugging, timing, assertions, and formatted output in Julia are all conventionally done with macros.

The argument expressions can be parenthesized or not; for example, the following two lines are functionally identical invocations of the `@assert` macro:

```
@assert(f(), "Call to f failed")
@assert f() "Call to f failed"
```

2.10.2.1 A toy definition

As a toy example, of a macro definition, consider a macro that prints the input expression

```
macro printarg(e)
    s = "$e = "
    quote
        let v = $e
            println($s, v)
            v
        end
    end
end
```

```
@printarg (macro with 1 method)
```

When calling a macro, we preface the name with an `@` sign; for example

```
@printarg 1+1
```

```
1 + 1 = 2
```

```
2
```

This is essentially the functionality of the built-in `@show` macro, except that our `printarg` macro fails when the argument involves an expression with a variable. For example, `@show` works fine in this context, while `@printarg` would fail:

```
let
    x = 1
    @show x + x
end
```

```
x + x = 2
```

```
2
```

The problem has to do with the fact that Julia macros and rename variables internally when there is a possibility of accidental conflict with names in an outer environment in a local scope. To deliberately refer to a variable in the outer environment, that variable must be *escaped*. We can see the difference by expanding both the `@printarg` and the `@show` macros on the same input using the `@macroexpand` macro:

```
@macroexpand @printarg x + x
```

```
quote
    #= In[47]:4 =#
    let var"#153#v" = Main.x + Main.x
        #= In[47]:5 =#
        Main.println("x + x = ", var"#153#v")
        #= In[47]:6 =#
        var"#153#v"
    end
end
```

We can see that all the variable names in this expression are explicitly scoped to the `Main` module. In contrast, for `@show`, we have

```
@macroexpand @show x + x
```

```
quote
    Base.println("x + x = ", Base.repr(begin
                #= show.jl:1232 =#
                local var"#155#value" = x + x
            end))
    var"#155#value"
end
```

Here we see that a local variable (with a name that will not conflict with any other name in the system) is assigned to the expression `x+x`.

2.10.2.2 A modest example

As an example of a more significant small utility macro, consider a closure defined inside of an outer function. Such closures sometimes suffer[5] from a performance issue because the compiler analysis does not determine that variables from the outer context (captured variables) are "type stable" (see Chapter 3). Putting the function into a `let` block can help address the issue, e.g. replacing

```
f = x -> C*x
```

with

```
f = let C=C x -> C*x end
```

This does make a semantic change to the program; for example, the code

```
C = 10
f = x -> C*X
C = 20
f(5)
```

will produce 100, while introducing an enclosing `let` block would produce 50. Nonetheless, in many cases captured variables remain unchanging for the useful lifetime of a closure, and this is a valid transformation.

Let us assume we are interested in automatically producing a surrounding `let` block. In this case, we first need a list of variables referenced in an expression. We do this by recursing through the call tree and adding any symbols we encounter (except for function names) to a set. Note that we can make this code very concise by defining different methods associated with the type of the first argument (dispatching based on an `Expr`, a `Symbol`, or another type)

[5] As the Julia documentation points out, improvements to the compiler may mean this will no longer be an issue soon. This does not change the value of this as an example.

```
symbols!(e, s = Set{Symbol}()) = nothing

symbols!(e :: Symbol, s = Set{Symbol}()) = push!(s, e)

function symbols!(e :: Expr, s = Set{Symbol}())
    args = e.head == :call ? e.args[2:end] : e.args
    for arg in args
        symbols!(arg, s)
    end
    s
end
```

Now we are in a position to define the macro:

```
macro letclosure(e)

    # Check that this is a function definition with ->
    @assert e.head == :(->)

    # Get the symbols that are not in the argument list
    s = Set{Symbol}()
    sarg = Set{Symbol}()
    symbols!(e.args[2], s)
    symbols!(e.args[1], sarg)
    setdiff!(s, sarg)

    # Bind local variables to the outer version
    bindings = [quote $v = $(esc(v)) end for v in s]

    # Put the expression into the let block
    quote
        let
            $(bindings...)
            $e
        end
    end
end
```

There are a few points to note about our definition:

- For local variables, the `macro` system produces "local" versions of names that do not coincide with any other names in the module. When we *want* such a coincidence, we need to explicitly escape a symbol with the `esc` function.

- As in other contexts, the "splatting" operator (`$(bindings...)`) expands the contents of a list in place.

When we evaluate the macro, the code is transformed and the result is inserted into our program; for example:

```
let
    C = 10
    f = @letclosure x -> C*x
    C = 20
    f(5)
end
```

```
50
```

If we want to see how a macro expands out to regular code, we can use the `@macroexpand` macro. Continuing with our example above, we have:

```
@macroexpand @letclosure x -> C*x
```

```
quote
    #= In[53]:18 =#
    let
        #= In[53]:19 =#
        begin
            #= In[53]:14 =#
            var"#159#C" = C
        end
        #= In[53]:20 =#
        (var"#161#x",)->begin
                #= In[55]:1 =#
                var"#159#C" * var"#161#x"
            end
    end
end
```

Note in particular that in the variable bindings, the regular `C` appears on the right hand side, but a local version of `C` is assigned to. All the other variables in the generated code likewise have purely local names that are guaranteed not to coincide with other names in the module.

2.10.3 Macros for structs

One simple use for macros is to define "boilerplate" code, such as data fields that must be defined for every struct inheriting from a particular abstract data type. For example, we might assume that every instance of a `User` has a name and ID, but some types of users might have additional information.

```
abstract type User end

macro make_user_type(T, fields)
    esc(quote
            struct $T <: User
                lastname  :: String
                firstname :: String
                id        :: String
                $(fields.args...)
            end
        end)
end
```

In this scenario, a `LocalUser` might requires only a name and ID (the first two fields for any `User`-derived type).

```
@make_user_type LocalUser begin end
LocalUser("Brown", "Bob", "bbrown")
```

```
LocalUser("Brown", "Bob", "bbrown")
```

For an external user, though, we also want to keep track of an employer.

```
@make_user_type ExternalUser begin
    employer :: String
end
ExternalUser("Smith", "Sam", "ssmith", "NSA")
```

```
ExternalUser("Smith", "Sam", "ssmith", "NSA")
```

Because there is a common set of fields, we can define utility functions that work on these fields for any class of user.

```
fullname(u :: User) = "$(u.firstname) $(u.lastname)"

fullname(LocalUser("Brown", "Bob", "bbrown")),
fullname(ExternalUser("Smith", "Sam", "ssmith", "NSA"))
```

```
("Bob Brown", "Sam Smith")
```

Of course, it is possible to implement this type of functionality without macros as well — it just might involve slightly more typing.

2.11 A matching example

Beyond writing language utilities, Julia macros are useful for writing embedded domain-specific languages (DSLs) for accomplishing particular tasks. In this setting, we are really writing a language interpreter embedded in Julia, using the Julia parse tree as an input and producing Julia code as output.

As an example of both Julia macro design and Julia programming more generally, we will design a small language extension for writing functions that transform code based on structural pattern matching; for example, a rule written as:

```
0-x => x -> -x
```

corresponds to a function something like

```
e ->
    # Check that e is a binary minus and the first argument is zero.
    if (e isa Expr && e.head == :call &&
        e.args[1] == :- && length(e.args) == 3 &&
        e.args[2] == 0)

        # Bind the name "x" to the second argument
        let x = e.args[3]
            result = :(-$x)   # Create a "-x" expression
            true, result      # Yes it matched, return result
        end
    else
        false, nothing        # No, it didn't match
    end
```

That is, we want a function that takes an input

- Is this an expression with a binary - operation at the top?
- Is the second term in the sum a zero?
- If both are true, bind `x` to the second term in the sum, produce a new expression `:(-$x)`, and assign to `result`. Return the pair `(true, result)`.
- Otherwise, return `(false, nothing)`.

The one-line description is already more concise for this simple example; if we wanted to match against a more complicated pattern, the Julia code corresponding to a rule in the syntax

```
pattern => (argument_symbols) -> result_code
```

becomes that much more complex. Our key task is therefore take rules written with the first syntax and to convert them automatically to standard functions.

This is a more ambitious example, and can be safely skipped over. However, it is both a useful example of a variety of features in Julia and introduces concrete tools we will use in later chapters (e.g. when discussing automatic differentiation).

2.11.1 Preprocessing

The parse tree for Julia code includes not only expressions, but also *line numbers*, e.g.

```
:(x->x+1)
```

```
:(x->begin
          #= In[60]:1 =#
          x + 1
      end)
```

In the code shown, the comment line immediately after the `begin` statement corresponds to a `LineNumberNode`. Such `LineNumberNode` nodes are useful for debugging, but are a nuisance for our attempts at pattern matching. Therefore, before doing anything else, we will write a utility to get rid of such nodes. Because `LineNumberNode` is a distinct type, we can use Julia's multiple dispatch as an easy way to accomplish this.

```
filter_line_numbers(e :: Expr) =
    let args = filter(a -> !(a isa LineNumberNode), e.args)
        Expr(e.head, filter_line_numbers.(args)...)
    end
filter_line_numbers(e) = e
```

2.11.2 Matching

At the heart of our matching algorithm will be a function `match_gen` that generates code to check whether an expression matches a pattern. The inputs to the `match_gen` function are

- A dictionary of *bindings*, mapping symbols to values.
- A pattern
- A symbol naming the current expression being matched.

2.11.2.1 Literals

In the simplest case, for non-symbol leaf nodes, we declare a match when the pattern agrees with the expression.

```
match_gen!(bindings, e, pattern) = :($e == $pattern)
```

For example, the following code matches the expression named `expr` to the pattern `0`:

```
match_gen!(Dict(), :expr, 0)
```

```
:(expr == 0)
```

2.11.2.2 Symbols

Things are more complicated when we match a symbol. If the symbol is not in the bindings dictionary, we just check that the expression equals the (quoted) symbol. Otherwise, there are two cases:

- *First use*: we create a new binding for it.
- *Repeat use*: Check if `expr` matches the previous binding.

```
match_gen!(bindings, e, s :: Symbol) =
    if !(s in keys(bindings))
        qs = QuoteNode(s)
        :($e == $qs)
    elseif bindings[s] == nothing
        binding = gensym()
        bindings[s] = binding
        quote $binding = $e; true end
    else
        binding = bindings[s]
```

```
        :($e == $binding)
    end
```

We illustrate all three cases below.

```
let
    bindings = Dict{Symbol,Any}(:x => nothing)
    println( match_gen!(bindings, :expr, :x) )
    println( match_gen!(bindings, :expr, :x) )
    println( match_gen!(bindings, :expr, :y) )
end
```

```
begin
    #= /Users/dbindel/work/class/nmds/nmds/src/matcher.jl:121 =#
    var"##343" = expr
    #= /Users/dbindel/work/class/nmds/nmds/src/matcher.jl:121 =#
    true
end
expr == var"##343"
expr == :y
```

2.11.2.3 Expressions

The most complex case is matching expressions. The basics are not complicated: an item `e` matches an `Expr` pattern if `e` is an expression with the same head as the pattern and if the arguments match.

```
match_gen!(bindings, e, pattern :: Expr) =
    let head = QuoteNode(pattern.head),
        argmatch = match_gen_args!(bindings, e, pattern.args)
        :( $e isa Expr && $e.head == $head && $argmatch )
    end
```

The building block for checking the arguments will be to check that a list of expressions matches a list of patterns.

```
match_gen_lists!(bindings, exprs, patterns) =
    foldr((x,y) -> :($x && $y),
          [match_gen!(bindings, e, p)
           for (e,p) in zip(exprs, patterns)])
```

The more complicated case is ensuring that the arguments match. This is in part because we want to accomodate the possibility that the last argument in the list is "splatted"; that is, a pattern like `f(args...)` should match `f(1, 2, 3)` with `args` bound to the tuple `[1, 2, 3]`. In order to do this, we would first like to make sure that we can sensibly identify a "splatted argument."

```
is_splat_arg(bindings, e) =
    e isa Expr &&
    e.head == :(...) &&
    e.args[1] isa Symbol &&
    e.args[1] in keys(bindings)
```

Note that we only consider something a "splatted argument" if the argument to the splat operator is a symbol in the bindings table.

To implement the check, we create a `let` statement to bind a local name to each argument. If the last pattern is splatted, we make sure the last term in the tuple on the left-hand side of the statement is splatted (and then remove the pattern splat). Finally, we generate checks to make sure each of the locally-named arguments matches with the associated term in the pattern list.

```
match_gen_args!(bindings, e, patterns) =
    if isempty(patterns)
        :(length($e.args) == 0)
    else
        nargs = length(patterns)
        lencheck = :(length($e.args) == $nargs)
        args = Vector{Any}([gensym() for j = 1:length(patterns)])
        argstuple = Expr(:tuple, args...)

        # De-splat pattern / splat arg assignment and
        #   adjust the length check
        if is_splat_arg(bindings, patterns[end])
            patterns = copy(patterns)
            patterns[end] = patterns[end].args[1]
            argstuple.args[end] = Expr(:(...), argstuple.args[end])
            lencheck = :(length($e.args) >= $(nargs-1))
        end

        argchecks = match_gen_lists!(bindings, args, patterns)
        :($lencheck && let $argstuple = $e.args; $argchecks end)
    end
```

We strongly recommend the reader trace through this code for some examples until enlightenment strikes.

2.11.2.4 Compiling the match

Given a list of symbols and a pattern in which they appear, we can produce code to generate a mapping from expressions to either `(true,bindings)` where `bindings` is a list of all the subexpressions bound to the name list; or `(false, nothing)` if there is no match.

```
function compile_matcher(symbols, pattern)
    bindings = Dict{Symbol,Any}(s => nothing for s in symbols)

    # Input expression symbol and matching code
    # (symbol bindings are indicated by bindings table)
    expr = gensym()
    test = match_gen!(bindings, expr, pattern)

    # Get result vals (symbol/nothing) and associated variable names
    result_vals = [bindings[s] for s in symbols]
    declarations = filter(x -> x != nothing, result_vals)

    # Produce the matching code
    results = Expr(:tuple, result_vals...)
    :($expr ->
        let $(declarations...)
            if $test
                (true, $results)
            else
                (false, nothing)
            end
        end)
end
```

It is convenient to compile this into a macro. The macro version also filters line numbers out of the input pattern and out of any exprssion we are trying to match.

```
macro match(symbols, pattern)
    @assert(symbols.head == :tuple &&
            all(isa.(symbols.args, Symbol)),
            "Invalid input symbol list")

    pattern = filter_line_numbers(pattern)
```

```
    matcher = compile_matcher(symbols.args, pattern)
    esc(:($matcher ∘ filter_line_numbers))
end
```

2.11.3 Rules

A rule has the form

```
pattern => args -> code
```

where `args` is the list of names that are bound in the pattern, and these names can be used in the subsequent code. We want to allow the code to potentially use not only the matched subexpression, but also to refer to the symbol as a whole; we use the named argument feature in tuples for that. So, for example, in the rule

```
x - y => (x,y;e) -> process(e)
```

we mean to match any subtraction, bind the operands to `x` and `y` and the expression as a whole to `e`, and call `process(e)` to process the expression.

Now that we have a macro for computing matches, we can use it to help with parsing the rule declarations into argument names, expression name (if present), the pattern, and the code to be called on a match.

```
function parse_rule(rule)
    match_rule = @match (pattern, args, code) pattern => args -> code
    isok, rule_parts = match_rule(rule)
    if !isok
        error("Syntax error in rule $rule")
    end
    pattern, args, code = rule_parts

    match_arg1 = @match (args, expr) (args..., ; expr)
    match_arg2 = @match (args, expr) (args..., )
    begin ismatch, bindings = match_arg1(args); ismatch end ||
    begin ismatch, bindings = match_arg2(args); ismatch end ||
    begin bindings = (Vector{Any}([args]), nothing) end
    symbols, expr_name = bindings

    if !all(isa.(symbols, Symbol))
        error("Arguments should all be symbols in $symbols")
```

```
    end
    if !(expr_name == nothing || expr_name isa Symbol)
        error("Expression parameter should be a symbol")
    end

    symbols, expr_name, pattern, code
end
```

Matching a pattern on an input produces a boolean variable (whether there was a match or not) and a table of bindings from names in the pattern to symbols generated during the match. In order to safely access the right-hand-side symbols that were generated during the match, we need to declare them (in a `let` statement). If there is a match, we bind them to the ordinary input names (things like `:x`) in a second let statement, then call the code in that second let statement and assign the result to the `result` symbol. The resulting code must be used in a context where the `expr` and `result` symbols are already set up.

```
function compile_rule(rule, expr, result)

    symbols, expr_name, pattern, code = parse_rule(rule)
    bindings = Dict{Symbol,Any}(s => nothing for s in symbols)
    test = match_gen!(bindings, expr, pattern)

    # Get list of match symbols and associated declarations
    result_vals = [bindings[s] for s in symbols]
    declarations = filter(x -> x != nothing, result_vals)

    # Set up local bindings of argument names in code
    binding_code = [:($s = $(r == nothing ? (:nothing) : r))
                    for (s,r) in zip(symbols, result_vals)]
    if expr_name != nothing
        push!(binding_code, :($expr_name = $expr))
    end

    # Produce the rule
    ismatch = gensym()
    quote
        let $(declarations...)
            $ismatch = $test
            if $ismatch
                $result = let $(binding_code...); $code end
            end
            $ismatch
```

```
        end
    end
end
```

Now that we are able to compile a rule, we set up a macro for compiling a rule into a standalone function. The input expression symbol (`expr`) is the named argument to this standalone function, and at the end the function returns both the match condition (which is what the compiled code evaluates to) and the result (which the compiled code produces as a side effect).

```
macro rule(r)
    expr, result = gensym(), gensym()
    code = compile_rule(filter_line_numbers(r), expr, result)
    esc(quote
            $expr ->
                let $result = nothing
                    $code, $result
                end
        end)
end
```

Finally, we often want to combine rules, executing the first one that matches a given input. The `@rules` macro takes a set of rules packaged into a block and tries them in sequence, returning the result of whichever rule was executed (or `nothing` if no rule was executed).

```
macro rules(rblock :: Expr)
    rblock = filter_line_numbers(rblock)
    if rblock.head != :block
        error("Rules must be in a begin/end block")
    end

    # Set up input name, output name, and rule list
    expr, result = gensym(), gensym()
    rules = rblock.args

    # Or together all the tests
    rule_calls =
        foldr((x,y) -> :($x || $y),
              [compile_rule(r, expr, result) for r in rules])

    # Call all the rules, return the computed result
    esc(quote
            $expr ->
```

```
                let $result = nothing
                    $rule_calls
                    $result
                end
        end)
end
```

2.11.4 Examples

We conclude our matching example by giving a few examples of the process in practice.

2.11.4.1 Re-associating operations

Julia parses chains of additions and multiplications into one long list. For example, `1+2+3` parses to `(1+2)+3` rather than to `+(+(1,2), 3)`. For some operations, it is simpler to only allow binary addition and multiplication nodes. The following function converts these nodes.

```
function reassoc_addmul(e)

    # Apply folding to an op(args...) node (op = +, *)
    fold_args(args, op) =
        foldl((x,y) -> :($op($x, $y)), args)

    # Rules for processing one expression node
    r = @rules begin
        +(args...) => args -> fold_args(args, :+)
        *(args...) => args -> fold_args(args, :*)
        e => e -> e
    end

    # Recursively process the graph
    process(e :: Expr) = r(Expr(e.head, process.(e.args)...))
    process(e) = e

    # Process the input expression
    process(e)
end
```

For example, the following expression has both long sums and long products that we re-associate into binary operations.

```
let
    e = :(sin(1+2+3*4*f))
    println("Original:    $e")
    println("Transformed: $(reassoc_addmul(e))")
end
```

```
Original:    sin(1 + 2 + 3 * 4 * f)
Transformed: sin((1 + 2) + (3 * 4) * f)
```

If we wanted to, we could also "de-associate" by collapsing sums and multiplications into single nodes.

```
function deassociate(e)
    r = @rules begin
        x * (*(args...)) => (x, args) -> Expr(:call, :*, x, args...)
        (*(args...)) * x => (args, x) -> Expr(:call, :*, args..., x)
        x + (+(args...)) => (x, args) -> Expr(:call, :+, x, args...)
        (+(args...)) + x => (args, x) -> Expr(:call, :+, args..., x)
        e => e -> e
    end
    process(e :: Expr) = r(Expr(e.head, process.(e.args)...))
    process(e) = e
    process(e)
end
```

We take the output of our "reassociate" code as an example input.

```
deassociate(:(sin((1+2) + (3*4)*f)))
```

```
:(sin(1 + 2 + 3 * 4 * f))
```

2.11.4.2 Simplifying expressions

Automatically-generated code of the sort that comes out of naive automatic differentiation often involves operations that can be easily simplified by removing additions with zeros, multiplications with one, and so forth. For example, we write `simplify_sum` and `simplify_mul` routines to handle simplification of sums and products involving zeros and ones.

```
simplify_sum(args) =
    let args = filter(x -> x != 0, args)
        if length(args) == 1
            args[1]
        else
            Expr(:call, :+, args...)
        end
    end

simplify_mul(args) =
    let args = filter(x -> x != 1, args)
        if any(args .== 0)
            0
        elseif length(args) == 1
            args[1]
        else
            Expr(:call, :*, args...)
        end
    end
```

Building on simplifying sums and products, we can recurse up and down an expression tree to apply these and other similar rules.

```
function simplify(e)
    r = @rules begin
        +(args...) => args -> simplify_sum(args)
        *(args...) => args -> simplify_mul(args)
        x - 0 => x -> x
        0 - x => x -> :(-$x)
        -(0)  => (;x) -> 0
        x / 1 => x -> x
        0 / x => x -> 0
        x ^ 1 => x -> x
        x ^ 0 => x -> 1
        e     => e -> e
    end
    process(e :: Expr) = r(Expr(e.head, process.(e.args)...))
    process(e) = e
    process(e)
end
```

We illustrate our simplifier with some operations

```
let
    compare(e) = println("$e simplifies to $(simplify(e))")
    compare(:(0*x + C*dx))
    compare(:(2*x^1*dx))
    compare(:(1*x^0*dx))
end
```

```
0 * x + C * dx simplifies to C * dx
2 * x ^ 1 * dx simplifies to 2 * x * dx
1 * x ^ 0 * dx simplifies to dx
```

2.12 Elements of Julia style

We write programs to be executed by a computer, an unforgivingly literal-minded audience. But the humans who read our code are a more challenging audience by far. We humans get bored. We get sidetracked. We misunderstand the simple things, and declare that we understand complicated things based on only-partly-correct models of how the world works. We read, forget, read again, and forget again. In the face of such an audience, what's a programmer to do?

We seek to write code that accomplishes something worthwhile, and to write in a simple, direct style that can be easily understood by anyone. This is easier said than done. There are some classic books with practical guidance that transcends specific programming languages, and we highly recommend the books of Kernighan and Plauger (1978), Knuth (1984), Kernighan and Pike (1999), and Hunt and Thomas (1999). The style guide for the Julia programming language also provides useful guidance. But since we are here, we will provide our own two cents on what we think are some of the key issues to writing "good" Julia code.

2.12.1 Formatting conventions

Julia code tends to have some common formatting conventions that make it easier to read code written by varying authors:

- Indentation by four spaces
- Functions that modify (mutate) an input end with an exclamation mark
- Module and type names use `CamelCase`
- Function names are `lowercase` and omit separators when feasible (underscores can be used otherwise)

We also recommend limiting globally visible names (modules, structures, or functions) to Latin characters, or providing a Latin alternative when a larger set of Unicode characters is used (for example, `pi` as an alternative to `π`).

There are a number of style guides with more detailed opinions on the proper formatting of Julia code. If you want to adhere tightly to such a guide, we recommend a tool like `JuliaFormatter`.

2.12.2 Correctness first

Correctness is a tricky concept in numerical computing. Approximation is at the heart of most numerical methods, whether for data science or otherwise. Hence, we rarely get to decide whether we have "the right answer"; instead, we have to reason about whether the result of a computation is accurate enough for a downstream task. To make matters even more complicated, we usually face tradeoffs between speed and accuracy, and so need to design simultaneously for "right enough" and "fast enough."

Neither correctness nor performance are easy to reason about. However, it is easier to tell when numerical code is slow than when it is wrong. Our mental models of *why* code is slow may often be wrong, but our observation *that* code is slow are not. Therefore, we usually want to start with simple code together with *analysis and tests* to ensure that it is accurate, and revise the code to tune performance later, informed by profiling (Chapter 3).

2.12.3 Catch errors early

The earlier we catch an error, the easier it is to correct. We would rather:

- Catch conceptual errors before we start writing code.
- Catch implementation errors at compile time than at run time.
- Analyze errors in an individual function than have to perform a root cause analysis.

Tests, type annotations, and careful assertions all help us with finding errors early.

2.12.4 Put it in a (small) function

For a human audience, functions are a natural unit for code. A well-designed function gives name to a concept, and ideally the name, comments, and code are all in obvious agreement – it "does what it says on the cover." Small functions are also relatively easy to test. And assigning appropriate type annotations to the arguments and return values of a function also helps us catch errors in how the function is used.

Apart from readibility, there are good performance reasons for putting code in functions. The Julia compiler works with functions. Code written at a command line or in a script style will not be processed in the same way, and will typically run more slowly. It is also easier to use

Julia's analysis tools to understand the performance implications of design choices made in short functions.

2.12.5 Keep the scope small

Functions and control flow constructs in Julia create new scopes, so Julia variable names often have short lives. This is convenient for human readers, with our limited memories, particularly when we want to save typing and use short names. Variable names with limited scope (and limited opportunity for reassignment) are also easy for the compiler to analyze, and result in more efficient code.

2.12.6 Respect interfaces

Unlike some languages (e.g. Python, Java, and C++), Julia does not have a strong data hiding mechanism, though it is possible to code in a way that hides data structures (e.g. by tucking them inside functions). Nonetheless, if the option exists, it is usually preferable to use getter and setter functions rather than directly accessing the contents of a Julia structure. This makes it easier to monitor access to the data structures for debugging, and it makes it easier to change data structures as the code evolves.

2.12.7 DRY (Don't Repeat Yourself)

A general design principle for any data system is that code and data should ideally have a single, concise, authoritative representation.

Compared to some languages, Julia code is fairly concise, and does not require much boilerplate[6]. Several aspects of the language contribute to this conciseness:

- The ability of the system to dispatch to different methods based on argument types simplifies the process of writing generic code.
- Functions are first-class entities that can be passed around with context, so we can pass around specialized functions for a specific subtask rather than coding it everywhere. For example, we can pass logging routines into our functions as callbacks rather than adding logging infrastructure to all our routines.
- Conversion methods and promotion rules limit the number of variants of methods that we might have to write for different combinations of argument types.
- When other methods fail, we can remove redundancy by writing macros to generate repetitive code for us.

[6]Per Wikipedia, the metal printing plates used for typesetting widely-distributed ads or syndicated columns were called "boilerplate" by analogy with the rolled steel plates used to make water biolers.

2.12.8 KISS and YAGNI

Peer to DRY in our opinions on design style are two other acronyms:

- KISS: Keep It Simple, Stupid
- YAGNI: You Ain't Gonna Need It

That is, we want to start with simple, working code that solves a known problem, and only get fancier if there is a measurable reason to do so. Simple code and simple data structures are easier for people to read, and preferable absent other *real* concerns. For example, any number of Julia constructs that we have discussed (promotion rules, macros, and expansion of tuples into parameter lists) are arguably harder to read than alternative constructions. It is nice to have these features in the language in situations to avoid writing redundant code or to address performance bottlenecks during profiling. But if we have not observed redundancy or performance issues in practice, it is usually best to write the simplest thing possible. In the event we decide that we *do* need a less simple code construct, whether to avoid repetition or to deal with a performance issue, we have the experience of writing the short and simple thing first. And, as with writing English prose, it is usually easier to revise what exists than to start from a blank page.

3 Performance Basics

"Are we there yet?"
— small children everywhere

Numerical code should be right enough and fast enough. Correctness comes first, but we are also impatient, and want our codes to run fast. We do not recommend our readers should become reckless speed demons: improvements in running time should be weighed against the time required to implement and maintain the code, and for most of us it is unwise to get into the business of rewriting our codes every year to eke every jot of speed out of the newest processor. But certain details of how we implement our methods can make order-of-magnitude differences in how fast our codes run on most modern processors, and a little knowledge of those details (along with a few tools) can go a long way toward keeping us happy with both the performance and the tidiness of our codes.

3.1 Time to what?

The key metric in performance of numerical methods is *usually* the wall clock time to a solution. In addition to wall clock time, one might want to understand how much memory, disk space, network bandwidth, or power are used in a given computation. However, all these measures may depend on implementation details, on details of the problem being solved, and on the system used.

We would certainly like code that we have optimized to run fast on problems other than our test problems; and ideally, we would like our performance to be portable (or at least "transportable" with small changes) to new systems. To accomplish this, it is useful to have both performance *models* that we can use to generalize beyond a single instance and performance *experiments* to validate our models and to fit any free parameters. Sometimes we use proxy measures for wall clock time, such as the number of arithmetic operations in a code, with the suggestion that these proxies relate to wall clock time by a simple (usually linear) model. Such proxy measures should be treated with caution unless backed by data. We discuss this in Section 3.2, Section 3.4, and Section 3.7.

The notion of "time to solution" has some additional subtleties. Some algorithms run to completion, then return a result more-or-less instantaneously at the end. In many cases, though, a computation will produce intermediate results. In this case, there are usually two quantities of interest in characterizing the performance:

- The time to first result (response time)
- Rate at which results are subsequently produced (throughput)

For problems involving information retrieval or communication, we usually refer to these as the *latency* and *bandwidth*. But the idea of an initial response time and subsequent throughput rate applies more broadly.

When we think about concepts of latency and throughput, we are measuring performance not by a single number (time to completion), but by a curve of utility versus time. When we think about a *fixed* latency and throughput, we are implicitly defining a piecewise linear model of utility versus time: there's an initial period of zero utility, followed by a period of linearly increasing utility with constant slope (until completion). The piecewise linear model is attractive in its simplicity, but more complex models are sometimes useful. For example, to understand the performance of an iterative solver, the right way to measure performance might be in terms of approximation error versus time.

We are also often interested in situations where we *incrementally recompute* something based on new data. In this case, we might have initial setup costs that need only be run once, and thereafter are not required again (or are only required periodically). In this case, there is a tradeoff: the setup costs may be expensive, but what we care about is the setup cost *together with* the incremental costs. In this case, we say the setup costs are *amortized* over the computation.

While it is not our main topic in this chapter, it is also worthwhile to pay attention to resources that we care about that fall outside our computation. This might include things like input/output costs – reading data in and writing results out can be surprisingly expensive! It can also include things like the number of times that we require human attention, or the amount of experimental data required to adequately fit a model.

3.2 Scaling analysis

For most problem classes, there is some natural measure of the size of the problem. This could be the number of data points we measure, the size of a linear system to be solved, etc. Scaling analysis for algorithms has to do with the way the cost changes as the size parameter n grows, along with other parameters such as the number of processors.

We usually write scaling analyis using order notation (aka "big O" notation), which refers to *sets* of functions with certain growth rates; we say $f(n)$ is

- $O(g(n))$ if f grows no faster than g: there is an integer N and positive constant C such that for all $n \geq N$, $f(n) \leq Cg(n)$.
- $o(g(n))$ if f grows more slowly than g: for any positive C, there is an integer N such that for all $n > N$, $f(n) < Cg(n)$.

- $\Omega(g(n))$ if f grows no more slowly than g: there is an integer N and positive constant C such that for all $n \geq N$, $f(n) \geq Cg(n)$.
- $\omega(g(n))$ if f grows strictly faster than g: for any positive C, there is an integer N such that for all $n > N$, $f(n) < Cg(n)$.
- $\Theta(g(n))$ if f and g grow at the same rate: there is an integer N and positive constants c and C such that for all $n \geq N$, $cg(n) \leq f(n) \leq Cg(n)$.

Even though order notation defines sets of functions, we usually abuse notation and write things like $f(n) = O(g(n))$ rather than the more proper $f(n) \in O(g(n))$. We will see the same notation in a different context (x going to zero rather than n going to infinity) in Chapter 5.

As a concrete example, that will recur frequently, let us consider the complexity of the *Basic Linear Algebra Subroutines* (BLAS), a collection of standard linear algebra operations often used as building blocks for higher-level algorithms. Typical examples are dot products, matrix-vector products, and matrix-matrix products. For the case of square matrices, these cost $O(n^1)$, $O(n^2)$, and $O(n^3)$ time, respectively. In the language of the BLAS standard, we call these level 1, level 2, and level 3 BLAS calls. However, while this captures the correct scaling, it is nowhere near the full story: a fast implementation of the standard matrix-matrix multiply[1] can be orders of magnitude faster than the standard three nested loops.

Order notation is convenient, and we will use it often. But it is convenient in part because it is crude: if we say the runtime of an algorithm is $O(n^3)$, that means there is some N and C such that the runtime is bounded by Cn^3 for all $n \geq N$ – but that says nothing about the size of C or N. Indeed, we expect the tightest version of the constant C to vary depending on details of the implementation and the system we use. As we will see in Section 3.4, we usually will want a scaling analysis as the *start* of understanding performance, but it is not the conclusion.

3.3 Architecture basics

In introductory programming classes, students typically form a mental model of how an idealized computer works. There is a memory, organized into a linear address space. A program is stored in memory, and consists of machine language instructions for things like register read/write, logic, and arithmetic. The computer reads these instructions from memory and executes them in program order. And, we think, all operations take about the same amount of time.

This mental model is incomplete in many respects, though that does not keep it from being useful. The reader who has had an undergraduate computer architecture course (e.g. from Hennessey and Patterson (2017)) will already appreciate the impact on performance of the memory hierarchy and instruction-level parallelism on performance. For the reader who has not had such a course (or has not reviewed the material recently), a brief synopsis may be useful.

[1] Strassen's algorithm for matrix-matrix multiplication has better asymptotic complexity than the standard algorithm, at $O(n^{\log_2 7})$ running time. Despite the better asymptotic complexity, Strassen's algorithm is at best an improvement for rather large matrices, and is not used in most BLAS libraries.

3.3.1 Memory matters

In a crude accounting of the work done by numerical codes, we often count the number of floating point operations (flops) required. On modern machines, though, the cost of a floating point operation pales in comparison to the cost of an access to main memory. On one core of the laptop on which this text is being written[2], more than 2500 64-bit floating point operations can (in theory) be performed in the latency period for a read request to main memory; and the subsequent bandwidth is such that we could execute more than three floating point operations per floating point number read from main memory. Though the details of these numbers vary from machine to machine, the core message remains the same: if we are not careful, the dominant cost of many numerical computations is not arithmetic, but data transfers.

3.3.1.1 Locality

What saves us from being forever limited by main memory is a *memory hierarchy* with small memories with fast access (caches) absorbing some of the traffic that would otherwise go to larger memories with slower access farther away. This hierarchy is engineered to reduce main memory traffic for programs that exhibit two sorts of *locality*:

- *Spatial locality* is the tendency to consecutively access memory locations that are close together in address space.
- *Temporal locality* is the tendency to frequently re-use a (small) "working set" of data in a particular part of a computation.

Some programs are born with data that fits in cache, some achieve good cache utilization through "natural" locality, and some have locality thrust upon them. In many cases, performance tuning of numerical codes falls into the last category: a naively written code will not have good locality and will tend to be limited by memory, but we can impvove matters by rearranging the computations for a more cache-friendly access pattern. But to get the best use from caches, we need a few more details.

3.3.1.2 Hits, misses, and operational intensity

Processors may have a *register file* containing data that can be operated on immediately, two or three levels of *cache* (with L1 the fastest and smallest, then L2, and then L3), and a main

[2]These numbers are for a Firestorm core (performance core) on an Apple M1 Pro, taken from a review on Anandtech. These cores have four floating point pipelines, each of which can execute operations on 128-bit vectors (two double-precision floating point numbers). At 3.2 GHz, this corresponds to a processor bandwidth of 25.6 floating point operations (adds or multiplies) per nanosecond. We estimate the latency to main memory at about 100 ns, so about 2560 floating point operations in the time to get the first byte of a memory transfer from memory. Memory bandwidth to one core seems to be about 60 GB/s, or about 7.5 double-precision floating point numbers per nanosecond.

memory. When a program requests data from a particular address, the processor first consults with the lowest level of cache. If the data is in the cache (a *cache hit*), it can be returned otherwise. In the event of a *cache miss*, the processor consults with higher levels of cache to try to find the data, eventually going to main memory if necessary.

To exploit *spatial locality*, caches are organized into *lines* of several bytes; when data is read into cache, we fetch it a cache line at a time, possibly saving ourselves some subsequent cache misses. To exploit *temporal locality*, the processor tries to keep a line in cache until the space is needed for something else. An *eviction policy* determines when a cache line will be replaced by other data. The eviction policy depends in part on the *associativity* of the cache. In a *direct-mapped* cache, each address can only go in one cache location, usually determined by the low-order bits of the storage address. In an n-way *set associative* cache, each address can go into one of n possible cache locations; in case a line must be evicted, the processor will choose something like the least recently used of the set (an LRU policy). Higher levels of associativity are more expensive in terms of hardware, and so are usually associated with the lowest levels of cache.

Naturally, we would like to minimize the number of cache misses. To design code with few misses, it is useful to think of three distinct types of misses[3]:

- *Compulsory*: when the data is loaded into cache for the first time.
- *Capacity*: when the working set is too big and the cache was filled with other things since it was last used.
- *Conflict*: when there is insufficient associativity for the access pattern.

For codes with low *operational intensity* (or *arithmetic intensity*), compulsory misses are the limiting factor in how well we can use the cache. The operational intensity is defined to be the ratio of operations to memory reads. For example, consider a dot product of two vectors (assumed not to be resident in cache): if each vector is length n, we have $2n$ floating point operations (adds and multiplies) on $2n$ floating point numbers for an operational intensity of one flop per float. We can, at best, hope that we are accessing the numbers in sequential order so that we only have a cache miss every few numbers (depending on the number of floating point numbers that fit into a cache line). Such a routine is inherently *memory-bound*, i.e. it is limited not by the time to do arithmetic but by the time to retrieve data.

In contrast to dot products, square matrix-matrix products involve $2n^3$ floating point operations, but only involve $2n^2$ input numbers — an operational intensity of n operations per float. In this case, we are not so limited by compulsory misses. However, unless the matrices are small enough to fit entirely into cache, a naively-written code may still suffer *capacity* misses. To get around this, we might use *blocking* or *tiling*, reorganizing the matrix-matrix product in terms of products of smaller submatrices. Even with such a reorganization, we might need to

[3] In shared-memory parallelism, we are also concerned about *coherence* misses when multiple processors write to locations in memory that are close to each other, and thus the local cache lines must be invalidated to ensure that all processors are viewing memory in the same ways. This is important, but beyond the scope of the current treatment.

be careful about *conflict* misses, particularly when n is a multiple of a large power of 2 (the worst case scenario for set-associative caches).

3.3.2 Instruction-level parallelism

If we do not use caches well, our speed will be limited by memory traffic. But once we have reduced memory traffic enough, the rate at which we can execute operations becomes the limiting factor; that is, we are *compute bound* rather than *memory bound*. To improve the performance of compute-bound codes, we need to understand a little more about a little about *instruction-level parallelism*.

A typical processor consists of several *cores*. Codes that are not explicitly parallel run on just one core at a time. Each core presents a serial programming *interface*: the core acts like it executes machine instructions in program order. This interface is fine for reasoning about correctness of programs. But the interface hides a much more sophisticated implementation.

In introductory computer architecture classes, we usually start by discussing a *five stage pipeline*, where for each instruction we

- *Fetch* the instruction from memory
- *Decode* the instruction
- *Execute* the instruction
- Perform any *memory* operations
- *Write back* results to the register file

In any given cycle, each pipeline stage can be occupied by a different instruction; while we are writing back the first instruction in a sequence, we might be performing a memory operation associated with the second instruction, doing some arithmetic for the third instruction, and decoding and fetching the fourth and fifth instruction. Hence, though each instruction typically has a *latency* of five cycles to execute (modulo waiting for memory), the processor in principle can execute instructions with a *bandwidth* of one instruction per cycle. We do not always achieve this peak number of instructions per cycle, though. For example, if the second instruction depends on the result of the first instruction, then we cannot execute the second instruction until the first instruction has written its results back to the register file. This results in a "bubble" in the pipeline where some of the stages are idle, and reduces the rate at which the processor can execute instructions.

The picture in most machines now is more complicated than a five stage pipeline. Modern processors are often *wide*: the front-end pipeline can fetch and decode several operations in a single cycle. The fetch-and-decode itself can be rather complex, with one machine language instruction turned into several "micro-ops" internally. Once an instruction is decoded, it is kept in a re-order buffer until the processor is ready to dispatch it to a *functional unit* like a floating point unit or memory unit. These functional units themselves have internal pipelines,

and so can process several instructions concurrently. As the functional units complete their work, the results are written back in an order consistent with the program order.

A wide, pipelined, out-of-order processor can have many instructions in flight at the same time. The extent to which we can keep the processor fully occupied depends on two factors:

- *Limited dependencies*: As with the five-stage pipeline, dependencies between instructions can limit the amount of instruction-level parallelism. These dependencies can take the form of data dependencies (one operation takes as input the result of a previous operation) or control dependencies (we cannot complete instructions after a branch until the branch condition has been computed). The out-of-order buffer and branch prediction techniques can help mitigate the impact of dependencies, but we still expect code with simple control structures and lots of independent operations to run faster than code with complicated control and lots of tight data dependencies.
- *Instruction diversity*: A mix of different types of instructions can keep the different functional units occupied concurrently.

In addition to the *implicit* instruction-level parallelism we have just described, many modern processors provide *explicit* instruction-level parallelism in the form of *vector* or *SIMD* (single-instruction multiple-data) instructions that simultaneously compute the same operation (e.g. addition or multiplication) on short vectors of inputs.

Given code with simple structure and enough evident independent operations, compilers can do a good job at producing code that uses vector instructions, just as they do a good job at reordering instructions for the processor.

3.4 Performance modeling

Runtime can depend on many parameters: the problem size and structure, the algorithms used in the computation, details of the implementation, the number and type of processors used, etc. Performance models predict runtime (and perhaps the use of other resources) as a function of these parameters. There are several reasons to want such a model:

- To decide whether a method is worth the effort of implementing or tuning.
- To decide what algorithm to use (and what parameter settings) for the best runtime on a particular problem and system.
- To decide whether a particular subcomputation is likely to be a bottleneck in a larger computation.
- To determine whether the runtime of a computation is "reasonable" or if there is room for easy improvement in the implementation.

Scaling analysis (Section 3.2) does not usually yield a good performance model on its own. Even if we estimate of the number of floating point operations in a given computation, we know from Section 3.3 that there may be several orders of magnitude depending in runtime depending how memory is used and the amount of instruction-level parallelism. At the same time, a *somewhat* inaccurate model is often fine to guide engineering decisions. Because models reflect our understanding, they will almost always be incomplete; indeed, the most useful models are *necessarily* incomplete, since otherwise they are too cumbersome to reason about!

Experiments reflect what really happens, and are a critical counterpoint to models. The division between performance models and experiments is not sharp. In the extreme case, we can use machine learning and other empirical function fitting methods can be used to estimate runtimes from experimental measurements under weak assumptions. But when strongly empirical models have many parameters, a lot of data is needed to fit them well; otherwise, the models may be *overfit*, and do a poor job of predicting performance except away from the training data. Gathering a lot of data may be appropriate for cases where the model is used as the basis for *auto-tuning* a commonly-used kernel for a particular machine architecture, for example. But performance experiments often aren't cheap – or at least they aren't cheap in the regime where people most care about performance – and so a simple, theory-grounded model is often preferable. There's an art to balancing what should be modeled and what should be treated semi-empirically, in performance analysis as in the rest of science and engineering.

3.4.1 Applications, benchmarks, and kernels

The performance of application codes is usually what we really care about. But application performance is generally complicated. The main computation may involve alternating phases, each complex in its own right, in addition to time to load data, initialize any data structures, and post-process results. Because there are so many moving parts, its also hard to use measurements of the end-to-end performance of a given code on a given machine to infer anything about the speed expected of other codes. Sometimes it's hard even to tell how the same code will run on a different machine!

Benchmark codes serve to provide a uniform way to compare the performance of different machines on "characteristic" workloads. Usually benchmark codes are simplified versions of real applications (or of the computationally expensive parts of real applications); examples include the NAS parallel benchmarks, the Graph 500 benchmarks, and (on a different tack) Sandia's Mantevo package of mini-applications.

Kernels are frequently-used subroutines such as matrix multiply, FFT, breadth-first search, etc. Because they are building blocks for so many higher-level codes, we care about kernel performance a lot; and a kernel typically involves a (relatively) simple computation. A common first project in parallel computing classes is to time and tune a matrix multiplication kernel.

Parallel *patterns* (or "dwarfs") are abstract types of computation (like dense linear algebra or graph analysis) that are higher level than kernels and more abstract than benchmarks. Unlike a

kernel or a benchmark, a pattern is too abstract to benchmark. On the other hand, benchmarks can elucidate the performance issues that occur on a given machine with a particular type of computation.

3.4.2 Model composition

Counting floating-point operations is generally a bad way to estimate performance, because the "cost per flop" is so poorly defined. But the performance of larger building blocks may be much more stable, so that call counts (as opposed to operation counts) are a reasonable basis for performance modeling. For example, we frequently describe the performance of iterative solvers for sparse systems of linear equations in terms of the number of matrix-vector products; and for a particular matrix size, the time for a matrix-vector product will usually be about constant[4]. If all other operations in the solver are cheap compared to the cost of the matrix-vector product, just measuring the number of products and separately building a model for the time t_{matvec} for one product (or measuring it) may be adequate for predicting performance.

If we decide that a particular iterative solver should be faster, we might try to make matrix-vector products run faster, perhaps by rearranging the data structure that represents the matrix. This might be accomplished with some setup cost t_{setup} to analyze the input matrix and rearrange the data. If the rearranging the multiplication speeds it up by a factor of S', then the cost of the original code vs the initial code is

$$\begin{aligned} t_{\text{original}} &= n t_{\text{matvec}} \\ t_{\text{rearranged}} &= t_{\text{setup}} + n t_{\text{matvec}}/S' \end{aligned}$$

The rearranged computation will be faster if

$$t_{\text{setup}} < t_{\text{matvec}} n(S'-1)/S'.$$

Even a fairly large setup cost may be worthwhile if it is *amortized* over enough iterations. Even if rearranging the data structure does not make sense for a single linear solve, it may make sense to rearrange if we are solving many similar systems (in which case we might be able to pay the setup cost just once rather than for every solve).

Alternately, we could try to find an alternative method that requires fewer matrix-vector products, though this might involve more expensive iterations. This can be a net win even if each iteration is "less efficient" in terms of the rate of floating point operations. It is worth remembering that the key measure is not the arithmetic rate, but the time to completion (or time to adequate accuracy). In each case, we can reason about the performance with the same general strategy of reasoning about the performance of smaller building blocks.

[4]For small problems, the cost of the first matrix-vector product in a series of such computations may be slower than subsequent products. This is because the first product might suffer compulsory cache misses, but afterward the cache is "warmed."

3.4.3 Modeling with kernels

When we study dense matrix computations **?@sec-nla-ch**, we will write many of our algorithms in "block" fashion, so that almost all the work is done in level 3 BLAS operations. Because these have high operational intensity, it is possible (though difficult) to write these operations to run at pretty close to the peak arithmetic rate, at least for a single core. Fortunately, these operations are so common that we can usually rely on someone else to have done the hard work of writing a fast implementation in this setting. Hence, in dense matrix computations we often write that something requires a certain number of floating point operations that are "mostly level 3 BLAS," and assume that the reader understands that for this problem, the number of floating point operations times the peak flop rate (or some adjusted version thereof) is a reasonable estimate for the runtime.

More generally, for codes that rely on common well-tuned kernel operations like the level 3 BLAS or tuned FFTs, we can sometimes get away with assuming that the kernel runs at an appropriate "speed of light" for the hardware.

3.4.4 The Roofline model

If we are going to assume kernels that run at an appropriate "speed of light" for hardware, we need to understand what that speed is. The *Roofline model* is a simple model of peak performance for different computational kernels (Williams, Waterman, and Patterson (2009)). The model consists of a log-log plot of the operational intensity I (in floating point operations / byte) vs the operation rate (floating point operations per second). The actual operation rate always sits under a "roofline," the minimum of the limit imposed by the peak memory bandwidth β (operation rate is less than $\beta \times I$, a diagonal line on the plot) and the peak operational rate attainable by the hardware (a horizontal line on the plot). More elaborate versions of the roofline plot can include additional performance ceilings (e.g. the ceiling without the use of vector instructions, or without the use of multiple cores) and bandwidth ceilings (e.g. associated with multiple levels of cache).

3.4.5 Amdahl's law

We measure improvements to performance of a code on a fixed problem size by the *speedup*:

$$S = \frac{T_{\text{baseline}}}{T_{\text{improved}}}.$$

Suppose we have a code involving two types of work: some fraction α is work that we do not know how to speed up, and the remaining fraction $(1-\alpha)$ we think we can improve. Let S' be

the speedup for the part that we know how to improve; then

$$\begin{aligned} S &= \frac{T_{\text{baseline}}}{\alpha T_{\text{baseline}} + (1-\alpha)T_{\text{baseline}}/S'} \\ &= \frac{1}{\alpha + (1-\alpha)/S'} \\ &= \frac{S'}{1 + \alpha(S'-1)} \end{aligned}$$

No matter how big the speedup S' for the part we know how to improve, the overall speedup is bounded by $1/\alpha$. This observation is known as *Amdahl's law*.

Amdahl's law is best known in the context of parallel computing, where we are interested in speedup for a fixed problem size as a function of the number of processors p:

$$S(p) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p)}.$$

Studying the speedup $S(p)$ in this setting (or the parallel efficiency $S(p)/p$) is known as a *strong scaling* study. In this setting, α is the fraction of serial work. If the rest of the computation can be perfectly sped up (i.e. $S' = p$), then

$$S(p) = \frac{p}{1 + \alpha(p-1)} \leq \frac{1}{\alpha}.$$

In practice, this is usually a generous estimate: some overheads usually grow with the number of processors, so that past a certain number of processors the speedup often doesn't just level off, but actually *decreases*.

3.4.6 Gustafson's law

Amdahl's law is an appropriate modeling tool when we are trying to improve the runtime to solve problems of a fixed size, whether by parallel computing or by tuning code. Sometimes, though, we want to improve runtime because we want to solve bigger problems! This leads to a different scaling relationship.

In *weak scaling* studies in parallel computing, we usually consider the *scaled speedup*

$$S(p) = \frac{T_{\text{serial}}(n(p))}{T_{\text{parallel}}(n(p), p)}$$

where $n(p)$ is a family of problem sizes chosen so that the work per processor remains constant. For weak scaling studies, the analog of Amdahl's law is *Gustafson's law*; if a is the amount of serial work and b is the parallelizable work, then

$$S(p) \leq \frac{a + bP}{a + b} = p - \alpha(p-1)$$

where $\alpha = a/(a+b)$ is the fraction of serial work.

3.5 Performance principles

> There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the seed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say 97% of the time: premature optimization is the root of all evil.
>
> Yet, we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified. It is often a mistake to make a priori judgements about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail. …
>
> – Donald Knuth (from Knuth (1974))[5]

We want our codes to run fast, but not at great cost in terms of maintainability or development time. To do this, it is useful to keep some principles in mind.

3.5.1 Think before you write

Back-of-the-envelope performance models are often[6] enough to give us a sense of the "big computations," parts of a code will be critical to performance. We might also have a sense in advance of what the natural algorithmic variants are for this code are. If we think a particular routine might be a performance bottleneck that we will want to play with, it is probably worth thinking about how to code that routine so it is easy to experiment with it (or replace it).

When thinking about performance, it is worthwhile remembering to explicitly account for I/O. Memory accesses may move at a snail's pace compared to arithmetic; but I/O is positively glacial compared to either.

Before writing any code, it is also useful to

- Think through the implications of algorithms and data structures. Sometimes it is possible to just make a code do much less work without too much efort by using the right standard tools.
- Think about data layouts, if only to make sure that the code does not become too wedded to a specific data layout in memory.

[5]The quote about "premature optimization" in Knuth (1974) is probably the best known quote from this paper, but the rest of the paper is worth reading as well.

[6]This assumes that our back-of-the-envelope performance models are at least somewhat correct.

- Decide if an approximation is good enough.

3.5.2 Time before you tune

When codes are slow, it is often because a large amount of time is spent in a few key *bottlenecks*. In this case, our goal is to find and fix those bottleneck computations[7].

For removing bottlenecks – or even for deciding that a code needs to be tuned in the first place – we need to pay attention to some practical points about the design of timing experiments:

- For codes that show any data-dependent performance, it is important to profile on something realistic, as the time breakdown will depend on the use case.
- We also need to be aware that wall-clock time is not the only type of time we can measure – for example, many systems have some facility to monitor "CPU time," in which only the time that a task is using the processor is counted. Wall-clock time is typically what we care about.
- The system wall-clock time resolution is often much coarser than the CPU cycle time. Consequently, we need to make sure that enough work is done in a timing experiment or we might not get good data. A typical approach to this is to put code to be timed into a loop.
- The same routine run repeatedly may run faster after a first iteration that warms up the cache.
- There will probably be other processes running on the system, and cross-interference with other tasks can affect timing.

Profiling involves running a code and measuring how much time (and resources) are used in different parts of the code. One can profile with different levels of detail. The simplest case often involves manually instrumenting a code with timers. There are also tools that *automatically instrument* either the source code or binary objects to record timing information. *Sampling profilers* work differently; they use system facilities to interrupt the program execution periodically and measure where the code is. It is also possible to use *hardware counters* to estimate the number of performance-relevant events (such as cache misses or flops) that have occurred in a given period of time. We will discuss these tools in more detail as we get into the class (and we'll use some of them on our codes).

As with everything else, there are tradeoffs in running a profiler: methods that provide fine-grained information can produce a *lot* of data, enough that storing and processing profiling data can itself be a challenge. There is also an issue that very fine-grained measurement can interfere with the software being measured. It is often helpful to start with a relatively crude,

[7] Some wag once called this process "deslugging."

lightweight profiling technology in order to first find what's interesting, then focus on the interesting bits for more detailed experimentation.

3.5.3 Shoulders of giants

A good computational kernel is a general building block with a simple interface that does a fair amount of work. We want kernels to be general in order to amortize the work of tuning. We also ideally like kernels with high operational intensity, though not all kernels will have this property.

We have already discussed the BLAS as an example of kernel design, and noted that the high arithmetic intensity of level 3 BLAS (e.g. matrix-matrix multiplication) makes it a particularly useful building block for high-performance code. Other common kernel operations include

- Applying a sparse matrix to a vector (or powers of a sparse matrix)
- Computing a discrete Fourier transform
- Sorting a list

Code written using common kernel interfaces is *transportable*: the implementation will differ from plarform to platform depending on architectural details, but the common interface means that code that uses the different kernel implementations will run the same way. It is critical to get properly tuned implementations of these kernels — for example, linear algebra codes run with the reference BLAS library invariably have disappointing performance.

There are some cases when we want to be careful with general-purpose kernels. For example, for some types of structured matrices, the general matrix-matrix multiply routines from the BLAS may have higher operational complexity than a specialized implementation. Whether "higher operational complexity" means "more run time" depends on the size of the problem and the arithmetic rate that a specialized implementation attains — it is sometimes worthwhile to write the more specialized code, but not always! As another example, in some situations (like finite element codes or computer graphics), we may want to do many operations with small matrices (e.g. 3-by-3 or 4-by-4), and the BLAS calls may not be as efficient for such small n.

3.5.4 Help tools help you

Compilers, including the Julia compiler, are often quite effective at local optimizations, particularly those restricted to a "basic block" (straight-line code with no conditionals or loops). Loop optimizations are somewhat harder, and global optimizations that cross function boundaries are much harder. In practice, this means that very local optimizations are usually only effective when the programmer has information that the compiler might not have, but humans can help the compiler much more with figuring out global optimizations.

To take some concrete examples: compilers are better than humans at register allocation and instruction scheduling, and usually at branch joins and jump elimination. Indeed, at this level it is hard for humans to even attempt to interfere with what the compiler is doing! For operations like constant folding and propagation or the elimination of common subexpressions, the compiler might do a good job *if* it can figure out that there are no side effects (like I/O or changing the contents of an array) – but the compiler might not easily be able to establish that there are no side effects without some help[8]. Compilers are often good at simple loop transformations involving loop invariant code motion, unrolling, and vectorization. But compilers are also usually conservative about using algebraic identities to transform code[9], for example.

Apart from using tools for writing our code in performance-friendly Julia (as discussed in Section 3.6), we can help the compiler in two ways. First, we recognize that the compiler mostly looks at a little bit of the code at a time, and does not know the higher-level semantics of that code. Complex algebraic transformations are usually up to the programmer. Second, the compiler is really good at dealing with simple code without too many dependencies. In particular, we want to avoid very complex loops or conditional statements, as well as tricky use of functional programming constructs in performance-sensitive inner loops.

3.5.5 Tune data structures

As we have seen, memory access and communication patterns are critical to performance. The system is designed to make it fast to process small amounts of data that fit in cache, ideally by going through it in memory order (a "unit-stride" access pattern). But this is not the most natural access pattern for all the codes we might want to write! Consequently, tuning often involves looking not at *code* but at the *data* that the code manipulate: rearranging arrays for unit stride, simplifying structures with many levels of indirection,using single precision for high-volume floating point data, etc. With a proper interface abstraction, the fastest way to high performance often involves replacing a low-performance data structure with an equivalent high-performance structure.

While we are mainly concerned with accessing data structures (reads and writes), access is not the only cost. Allocation and deallocation also cost something, as does garbage collection. It is easy to end up paying hidden allocation costs (often followed by hidden copying costs). Fortunately, Julia provides us with diagnostics to identify memory allocations; and with some care, we can write code to pre-allocate key data structures.

[8] A function that always runs the same way with no side effects is called a *pure* function. Though Julia has some support for functional programming, unlike some languages, it does not assume functions are pure by default. There is a `@pure` macro in Julia for declaring functions are pure, but it is easily abused (enough so that the Julia developers have declared that it should only be used in the `Base` packages).

[9] We *want* compilers to be conservative about applying some types of algebraic transformations to floating point code, for reasons we will discuss in Chapter 9.

Matrices are a key data structure in many numerical codes. In Julia, as in MATLAB and Fortran, matrices are stored in a column-major format, with each column layed out consecutively in memory. Consequently, we usually prefer to process data column-by-column (rather than row-by-row). For example, the code

```
for j = 1:n
    for i = 1:m
        y[i] += A[i,j]*x[j]
    end
end
```

will run significantly faster than

```
for i = 1:m
    for j = 1:n
        y[i] += A[i,j]*x[j]
    end
end
```

As another example, consider the layout of many instances of a particular structure type. We could organize this as an array of structures, which is good for locality of access to the data for one item; or we could organize a structure of arrays, which may be more friendly to vectorization. Particularly for read-only access patterns, we might also consider a *copy optimization* where we keep around both data structures[10], and use whichever data structure gives us the best performance in context.

3.6 Performance in Julia

So far, we have focused on general performance issues that are mostly relevant across languages. However, there are some things that are more specific to Julia.

3.6.1 Measurement tools

We repeat the advice of Section 3.5: you should measure your code before tuning it! Fortunately, Julia provides several macros that measure runtime and memory allocation:

- `@time` prints out runtime and allocation information
- `@timev` prints out a more verbose version of the timing and allocation information

[10] Keeping multiple data structures representing the same data is sometimes good for performance, but violates the "don't repeat yourself" advice from Chapter 2.

- `@elapse` prints just the runtime
- `@timed` returns a structure that includes run time, memory allocation, and information about garbage collection
- `@elapsed` returns the elapsed time
- `@allocated` returns the total number of bytes allocated by the expression
- `@allocations` returns the numbre of allocations in the expression

For more elaborate timing with tabular outputs, we can use the `TimerOutputs.jl` package. The `@time` macros and even the more elaborate `TimerOutputs.jl` package time a single run of a function. The `BenchmarkTools.jl` package runs a code multiple times in order to get more accurate timing information.

The Julia `Profile` package implements a statistical profiler. Statistics are gathered by running an expression with the `@profile` macro. There are several different visualization interfaces that can be used to view the profiling results, including a particularly nice visualizer built into VS Code.

3.6.2 Avoiding boxing

In dynamically-typed languages, the types of values are often only known at runtime. Hence, the system keeps "boxes" that contain both type information and the value. The coding for this pair varies from system to system[11], but the practice of boxing in general can cause problems with performance. The run-time system has to switch between different operations depending on the type, which introduces branches in the code for many operations and makes it effectively impossible for the compiler to do code transformations like vectorization. Boxed values also generally take more storage than unboxed values would.

Though Julia is a dynamically-typed language, it can often produce low-level code that avoids boxing. To do this, Julia uses a just-in-time (JIT) compiler that instantiates specialized versions of methods depending on the concrete type signature of the inputs[12]. The just-in-time compilation process is necessary in part because it would be ridiculously expensive to produce specialized implementations of generic methods for every possible compatible type; however, most functions are only invoked with a small number of type signatures in any given program, so most conceivable implementations never need to be generated in practice. For a *type stable* method, the system can determine the concrete type of the return value from the concrete input type signature. Ideally, we would like type stability not just for the return value, but for

[11] A common representation of boxed values is via a tagged union: a structure with an tag (usually stored as an int) and then a union whose bits can be interpreted as an integer, floating point number, pointer to a more complex object, etc. Some language implementations use more elaborate compact representations to pack both data and type information into a 64-bit word (e.g. by using NaN encodings in floating point to signal non-floating-point types, or using the low-order bits of aligned pointers as a type tag).

[12] The JIT compiler operates on functions. The system does not compile script-style code, or code entered at the REPL that is not inside a function. If you want performance, put it in a function!

other intermediate values and variables as well. If the JIT compiler can reliably determine the concrete types of quantities, it can work with that data directly without boxes.

The `@code_warntype` macro in Julia does type inference for a particular method call and colors in red any expressions that represent a performance hazard because the type system can only determine an abstract type. In situations where it is convenient to allow abstract types on input but performance still matters, the Julia performance tips recommends putting the inner part of the function into its own separate kernel function that is type stable (the "function barrier technique").

Several other performance issues in Julia are associated with the potential need for boxing when a concrete type cannot be inferred. For example, the Julia manual recommends avoiding untyped globals, because there is possibility that they might be assigned to different types, so they might be boxed; this also poses is a hazard to the type stability of any method that uses them. We should also avoid containers of abstract types, since the items in the container then have to be boxed; and, for the same reason, we should be careful with structures with abstract types for fields. Finally, while Julia's functional programming features are convenient, when creating a closure it is easy to run into unanticipated boxing of captured variables from the surrounding environment.

3.6.3 Temporary issues

Vector operations in Julia (addition and scalar multiplication) produce temporaries. For example, if `x` and `y` are vector variables, then the function

```
f1(x,y) = x.^2 + 2x.*y + y.^2
```

will generate temporaries for `x.^2`, `2x`, `2x.*y`, and `y.^2` (as well as storage for the final result). If we use dotted operations throughout, then Julia fuses the operations and does not produce temporaries:

```
f2(x,y) = x.^2 .+ 2.0 .* x .* y .+ y.^2
```

An equivalent, but less verbose, option is to use the `@.` macro to make a fused broadcast version of the whole expression:

```
f3(x,y) = @. x^2 + 2x*y + y^2
```

We could also use a broadcast call to a helper function, which again will avoid allocating temporaries:

```
f4scalar(x, y) = x^2 + 2x*y + y^2
f4(x, y) = f4scalar.(x, y)
```

On the machine on which this is being written, the function `f1` takes about five times the memory and six times the run time of any of the other versions for input vectors `x` and `y` of length a million.

If we pre-allocate storage for the result, we do not need *any* allocations:

```
function f5!(result, x, y)
    @. result = x^2 + 2x*y + y^2
end
```

When `x` and `y` are long vectors, there is negligible runtime performance difference between writing to a pre-allocated vector and producing a newly-allocated output vector. If the vectors are short, though, we may start to notice the cost of allocating (and garbage collecting) these temporaries, particularly if the code has a large working set and more vectors leads to more cache pressure. On the other hand, if the input vectors are short *and* their sizes are known at compile time, we may also want to consider declaring them to be static arrays (using the `StaticArrays.jl` package).

Another place where Julia allocates new storage is in slicing operations used to specify a subarray. This happens even if the "slice" is the whole array. For example, if `x` is a vector in Julia, writing

```
z = x
```

assigns the name `z` to the same vector object, while writing

```
z = x[:]
```

creates a new copy of `x`. If we want to refer to the entries of a subarray without creating a copy, we can create a "view" object (a `SubArray`) by either calling the `view` function or using the `@views` macro:

```
z1 = x[1:10]        # Copy of the start of x
z2 = view(x, 1:10)  # View of the start of x
@views z2 = x[1:10] # View of the start of x
z1[1] = 100         # Does not alter x, only z1
z2[1] = 100         # Now x[1] == 100
```

In general, many of the linear algebra functions in Julia provide a mutating version that writes results into user-provided storage. This includes factorization routines (e.g. `cholesky!`), solves (with `ldiv!`), and matrix-vector or matrix-matrix products (with `mul!`). But using these mutating operations is more error-prone, and we do not recommend it until a timing experiment or a performance model suggests that allocating new storage might cause a problem.

3.6.4 Performance annotations

Julia provides several macros to tell the compiler to try certain types of transformations. For example,

- `@inbounds`: tells the compiler that there is no need to check that array accesses are in bounds
- `@fastmath`: tells the compiler that it is OK to apply certain floating point transformations that are not equivalent to the original code (e.g. by allowing the compiler to pretend floating point addition and multiplication are associative)
- `@simd`: tells the compiler to try vectorizing the code through SIMD instructions. `@simd ivdep` promises that loop iterations are independent and references are free from aliasing.

There are a number of packages that provide additional transformations (e.g. `@turbo` from `LoopVectorization.jl`.

We do not recommend using performance annotation macros until and unless a timing experiment or performance model suggests that it addresses a bottleneck. These macros effectively prompting the compiler to do certain transformations it might not otherwise do. If the compiler can deduce that these transformations are equivalent to the original code and will result in a performance improvement, the macros will often be unnecessary. If the compiler cannot deduce that these transformations are equivalent to the original code, but you as the human author think that they are obviously equivalent, you may want to double-check your assumptions of what is "obvious" before proceeding.

3.7 Misconceptions and deceptions

> It ain't ignorance causes so much trouble; it's folks knowing so much that ain't so.
> – Josh Billings

One of the common findings in pedagogy research is that an important part of learning an area is overcoming common *misconceptions* about the area; see, e.g. Leonard, Kalinowski, and Andrews (2014), Sadler et al. (2013), Muller (2008). And there are certainly some common misconceptions about high performance computing! Some misconceptions are exacerbated by bad reporting, which leads to deceptions and delusions about performance.

3.7.1 Incorrect mental models

3.7.1.1 Algorithm = implementation

We can never time algorithms. We only time implementations, and implementations vary in their performance. In some cases, implementations may vary by orders of magnitude in their performance!

3.7.1.2 Asymptotic cost is always what matters

We can't time algorithms, but we can reason about their asymptotic complexity. When it comes to scaling for large n, the asymptotic complexity can matter a lot. But comparing the asymptotic complexity of two algorithms for modest n often doesn't make sense! QuickSort may not always be the fastest algorithm for sorting a list with ten elements…

3.7.1.3 Simple serial execution

Hardware designers go to great length to present us with the *interface* that modern processor cores execute a instructions sequentially. But this *interface* is not the actual *implementation*. Behind the scenes, a simple stream of x86 instructions may be chopped up into micro-instructions, scheduled onto different functional units acting in parallel, and executed out of order. The *effective behavior* is supposed to be consistent with sequential execution – at least, that's what happens on one core – but that illusion of sequential execution does not extend to performance.

3.7.1.4 Flops are all that count

Data transfers from memory to the processor are often more expensive than the computations that are run on that data.

3.7.1.5 Flop rates are what matter

What matters is time to solution. Often, the algorithms that get the best flop rates are not the most asymptotically efficient methods; as a consequence, a code that uses the hardware less efficiently (in terms of flop rate) may still give the quickest time to solution.

3.7.1.6 All speedup is linear

See the comments above about Amdahl's law and Gustafson's law. We rarely achieve linear speedup outside the world of embarrassingly parallel applications.

3.7.1.7 All applications are equivalent

Performance depends on the nature of the computation, the nature of the implementation, and the nature of the hardware. Extrapolating performance from one computational style, implementation, or hardware platform to another is something that must be done very carefully.

3.7.2 Deceptions and self-deceptions

The article "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers" (Bailey (1991)) is a classic in performance analysis. It's still worth reading (as are various follow-up pieces – see the Further Reading section), and highlights issues that we still see now. To summarize slightly, here's my version of the list of common performance deceptions:

3.7.2.1 Unfair comparisons and strawmen

A common sin in scaling studies is to compare the performance of a parallel code on p processors against the performance of the *same code* with $p = 1$. This ignores the fact that the parallel code may have irrelevant overheads, or (worse) that there may be a better organization for a single processor. Consequently, the speedups no longer reflect the reasonable expectation of the reader that this is the type of performance improvement they might see when going to a good parallel implementation from a *good* serial implementation. Of course, it's also possible to see great speedups by comparing a bad serial implementation to a correspondingly bad *parallel* implementation: a lot of unnecessary work can hide overheads.

A similar issue arises when computing with accelerators. Enthusiasts of GPU-accelerated codes often claim order of magnitude (or greater) performance improvements over using a CPU alone. Often, this comes from explicitly tuning the GPU code and not the CPU code (see, e.g., Vuduc et al. (2010)).

3.7.2.2 Using the wrong measures

If what you care about is time to solution, you might not really care so much about watts per GFlop (though you certainly do if you're responsible for supplying power for an HPC installation). More subtlely, you don't necessarily care about scaled speedup if the natural problem size is fixed (e.g. in some graph processing applications).

3.7.2.3 Deceptive plotting

There are so many ways this can happen:

- Use of a log scale when one ought to have a linear scale, and vice-versa;
- Choosing an inappropriately small range to exaggerate performance differences between near-equivalent options;
- Not marking data points clearly, so that there is no visual difference between data falling on a straight line because it closely follows a trend and data falling on a straight line because there are two points.
- Hiding poor scalability by plotting absolute time vs numbers of processors so that nobody can easily see that the time for 100 processors (a small bar relative to the single-processor time) is equivalent to the time for 200 processors.
- And more!

Plots allow readers to absorb trends very quickly, but it also makes it easy to give wrong impressions.

3.7.2.4 Too much faith in models

Any model has limits of validity, and extrapolating outside those limits leads to nonsense. Treat with due skepticism claims that – according to some model – a code will run an order of magnitude faster in an environment where it has not yet been run.

3.7.2.5 Undisclosed tweaks

There are many ways to improve performance. Sometimes, better hardware does it; sometimes, better tuned code; sometimes, algorithmic improvements. Claiming that a jump in performance comes from a new algorithm without acknowledging differences in the level of tuning effort, or acknowledging non-algorithmic changes (e.g. moving from double precision to single precision) is deceptive, but sadly common. Hiding tweaks in the fine print in the hopes that the reader is skimming doesn't make this any less deceptive!

3.7.3 Rules for presenting performance results

In the introduction to Bailey, Lucas, and Williams (2010), David Bailey suggests nine guidelines for presenting performance results without misleading the reader. Paraphrasing only slightly, these are:

1. Follow rules on benchmarks
2. Only present actual performance, not extrapolations
3. Compare based on comparable levels of tuning
4. Compare wall clock times (not flop rates)
5. Compute performance rates from consistent operation counts based on the best serial codes.
6. Speedup should compare to best serial version. Scaled speedup plots should be clearly labeled and explained.
7. Fully disclose information affecting performance: 32/64 bit, use of assembly, timing of a subsystem rather than the full system, etc.
8. Don't deceive skimmers. Take care not to make graphics, figures, and abstracts misleading, even in isolation.
9. Report enough information to allow others to reproduce the results. If possible, this should include

    - The hardware, software and system environment
    - The language, algorithms, data types, and coding techniques used
    - The nature and extent of tuning
    - The basis for timings, flop counts, and speedup computations

4 Linear Algebra

We will not get far in this book without a strong foundation in linear algebra. That means knowing how to compute with matrices, but it also means understanding the abstractions of linear algebra well enough to use them effectively. We assume prior familiarity with linear algebra at the level of an undergraduate course taught from Strang (2023) or Lay, Lay, and McDonald (2015). It will be helpful, but not necessary, to have a more advanced perspective as covered in Axler (2024) or even Lax (2007). Even for readers well-versed with linear algebra, we suggest skimming this chapter, as some ideas (quasimatrices, the framework of canonical forms) are not always standard in first (or even second) linear algebra courses.

While our treatment will be fairly abstract, we try to keep things concrete by regularly giving examples in Julia. Readers unfamiliar with Julia may want to review Chapter 2; it is also possible to simply skim past the code examples.

We do not assume a prior course in functional analysis. This will not prevent us from sneaking a little functional analysis into our discussion, or treating some linear algebra concepts from a functional analysis perspective.

4.1 Vector spaces

A vector space (or linear space) over a field $\mathbb{F}$ is a set with elements (vectors) that can be added or scaled in a sensible way – that is, addition is associative and commutative and scaling (by elements of the field) is distributive. We will always take $\mathbb{F}$ to be $\mathbb{R}$ or $\mathbb{C}$. We generally denote vector spaces by script letters (e.g. $\mathcal{V}, \mathcal{W}$), vectors by lower case Roman letters (e.g. v, w), and scalars by lower case Greek letters (e.g. α, β). But we feel free to violate these conventions according to the dictates of our conscience or in deference to other conflicting conventions.

There are many types of vector spaces. Apart from the ubiquitous concrete vector spaces $\mathbb{R}^n$ and $\mathbb{C}^n$, the most common vector spaces in applied mathematics are different types of function

spaces. These include

$$\begin{aligned}
\mathcal{P}_d &= \{\text{polynomials of degree at most } d\};\\
L(\mathcal{V},\mathcal{W}) &= \{\text{linear maps } \mathcal{V}\to\mathcal{W}\};\\
\mathcal{C}^k(\Omega) &= \{\ k\text{-times differentiable functions on a set } \Omega\};\\
\ell^1 &= \{\text{absolutely summable sequences } x_1, x_2, \ldots\}\\
\ell^2 &= \{\text{absolutely square-summable sequences } x_1, x_2, \ldots\}\\
\ell^\infty &= \{\text{bounded sequences } x_1, x_2, \ldots\}
\end{aligned}$$

and many more. A *finite-dimensional* vector space can be put into 1-1 correspondence with some concrete vector space $\mathbb{F}^n$ (using a *basis*). In this case, n is the *dimension* of the space. Vector spaces that are not finite-dimensional are *infinite-dimensional*.

4.1.1 Polynomials

We compute with vectors in $\mathbb{F}^n$ ($\mathbb{R}^n$ and $\mathbb{C}^n$), which we represent concretely by tuples of numbers in memory, usually stored in sequence. To keep a broader perspective, we will also frequently describe examples involving the polynomial spaces $\mathcal{P}_d$.

The Julia `Polynomials.jl` package provides a variety of representations of and operations on polynomials.

```
using Polynomials
```

For example, we can construct a polynomial in standard form from its coefficients in the monomial basis or from its roots, or we can keep it in factored form internally.

```
Polynomial([2, -3, 1])
```

$2 - 3 \cdot x + x^2$

```
fromroots([1, 2])
```

$2 - 3 \cdot x + x^2$

```
fromroots(FactoredPolynomial, [1, 2])
```

$(x - 1) * (x - 2)$

The fact that there are several natural representations for polynomials is part of what makes them a good example of an abstract vector space.

4.1.2 Dual spaces

A vector space in isolation is a lonely thing[1]. Fortunately, every vector space has an associated *dual* space of linear functionals[2], that is

$$\mathcal{V}^* = \{\text{linear functions } \mathcal{V} \to \mathbb{F}\}.$$

Dual pairs are everywhere in applied mathematics. The relation between random variables and probability densities, the theory of Lagrange multipliers, and the notion of generalized functions are all examples. It is also sometimes convenient to recast equations in terms of linear functionals, e.g. using the fact that $v = 0$ iff $\forall w^* \in \mathcal{V}^*, w^* v = 0$.

In matrix terms, we usually associate vectors with columns and dual vectors with columns (ordinary vectors) and rows (dual vectors). In Julia, we construct a dual vector by taking the conjugate transpose of a column, e.g.

```
typeof([1.0; 2.0; 3.0])   # Column vector in R^3
```

```
Vector{Float64} (alias for Array{Float64, 1})
```

```
typeof([0.0; 1.0; 0.0]')  # Row vector in R^3
```

```
Adjoint{Float64, Vector{Float64}}
```

Applying a linear functional is concretely just "row times column":

```
let
    v = [1.0; 2.0; 3.0]   # Column vector in R^3
    w = [0.0; 1.0; 0.0]'  # Row vector in R^3
    w*v
end
```

```
2.0
```

In order to try to keep the syntax uniform between concrete vectors and operations on abstract polynomial spaces, we will define a `DualVector` type representing a (linear) functional

[1] "Vector spaces are like potato chips: you cannot have just one." - W. Kahan

[2] In infinite-dimensional settings, we will restrict our attention to *continuous* linear functionals (the continuous dual space) rather than all linear functionals (the algebraic dual space).

```
struct DualVector
    f :: Function
end
```

We overload the function evaluation and multiplication syntax so that we can write application of a dual vector w^* to a vector v as `w(v)` or as `w*v`.

```
(w :: DualVector)(v) = w.f(v)
Base.:*(w :: DualVector, v) = w(v)
```

Because dual vectors still make up a vector space, we would like to be able to add and scale them:

```
Base.:+(w1 :: DualVector, w2 :: DualVector) =
    DualVector(v -> w1(v) + w2(v))
Base.:-(w1 :: DualVector, w2 :: DualVector) =
    DualVector(v -> w1(v) - w2(v))
Base.:-(w :: DualVector) =
    DualVector(v -> -w(v))
Base.:*(c :: Number, w :: DualVector) =
    DualVector(v -> c*w(v))
Base.:*(w :: DualVector, c :: Number) = c*w
Base.:/(w :: DualVector, c :: Number) =
    DualVector(v -> w(v)/c)
```

Evaluating a polynomial at a point is one example of a functional in $\mathcal{P}_d^*$; another example is a definite integral operation:

```
let
    p = Polynomial([2.0; -3.0; 1.0])
    w1 = DualVector(p -> p(0.0))
    w2 = DualVector(p -> integrate(p, -1.0, 1.0))
    w1*p, w2*p
end
```

```
(2.0, 4.666666666666666)
```

4.1.3 Subspaces

Given a vector space $\mathcal{V}$, a (nonempty[3]) subset $\mathcal{U} \subset \mathcal{V}$ is a *subspace* if it is closed under the vector space operations (addition and scalar multiplication). We usually take $\mathcal{U}$ to inherit any applicable structure from $\mathcal{V}$, such as a norm or inner product. Many concepts in numerical linear algebra and approximation theory are best thought of in terms of subspaces, whether that is nested subspaces used in approximation theory, subspaces that are invariant under some linear map, or subspaces and their orthogonal complements.

There are a few standard operations involving subspaces:

- The *span* of a subset $S \subset \mathcal{V}$ is the smallest subspace that contains S, i.e. the set of all finite linear combinations $\sum_i c_i s_i$ where each $s_i \in S$ and $c_i \in \mathbb{F}$.
- Given a collection of subspaces $\mathcal{U}_i$, their *sum* is the subspace $\mathcal{U}$ consisting of all sums of elements $\sum_i u_i$ where only finitely many $u_i \in \mathcal{U}_i$ are nonzero. We say the sum is a *direct sum* if the decomposition into vectors $u_i \in \mathcal{U}_i$ is unique.
- The *annihilator* for a subspace $\mathcal{U} \subset \mathcal{V}$ is the set of linear functionals that are zero on $\mathcal{U}$. That is:
$$\mathcal{U}^\perp = \{w^* \in \mathcal{V}^* : \forall u \in \mathcal{U}, w^* u = 0\}.$$
In inner product spaces, the annihilator of a subspace is identified with the *orthogonal complement* (Section 4.1.7).
- Given a subspace $\mathcal{U} \subset \mathcal{V}$, the *quotient space* $\mathcal{V}/\mathcal{U}$ is a vector space whose elements are equivalence classes under the relation $v_1 \sim v_2$ if $v_1 - v_2 \in \mathcal{U}$. We can always find a "complementary" subspace $\mathcal{U}^c \subset \mathcal{V}$ isomorphic to $\mathcal{V}/\mathcal{U}$ such that each equivalence class in $\mathcal{V}/\mathcal{U}$ contains a unique representative from $\mathcal{U}^c$. Indeed, we can generally find many such spaces! For any complementary subspace $\mathcal{U}^c$, $\mathcal{V}$ is a direct sum $\mathcal{U} \oplus \mathcal{U}^c$.
- In the infinite-dimensional setting, the *closure* of a subspace consists of all limit points in the subspace. This only makes sense in *topological* vector spaces where we have enough structure that we can take limits. We will always consider the even stronger structure of *normed* vector spaces.

4.1.4 Quasimatrices

Matrix notation is convenient for expressing linear algebra over concrete vector spaces like $\mathbb{R}^n$ and $\mathbb{C}^n$. To get that same convenience for abstract vector spaces, we introduce the more general notion of *quasimatrices*, matrix-like objects that involve more than just ordinary numbers. This includes letting columns or rows belong to an abstract vector space instead of a concrete space like $\mathbb{R}^m$ or $\mathbb{R}^n$, or using linear maps between subspaces as the "entries" of a matrix.

[3]The set containing only the zero vector is the smallest subspace we ever talk about.

4.1.4.1 Column quasimatrices

A (column) *quasimatrix* is a list of vectors in some space $\mathcal{V}$ interpreted as columns of a matrix-like object, i.e.

$$U = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix}.$$

We write a linear combination of the columns of U as

$$Uc = \sum_{i=1}^{n} u_i c_i$$

for some coefficient vector $c \in \mathbb{F}^n$. The *range space* $\mathcal{R}(U) = \{Uc : c \in \mathbb{F}^n\}$ (or *column space*) is the set of all possible linear combinations of the columns; this is the same as the *span* of the set of column vectors. The key advantages of column quasimatrices over sets of vectors is that we assign indices to name the vectors and we allow for the possibility of duplicates.

When $\mathcal{V}$ is a concrete vector space like $\mathbb{R}^m$ or $\mathbb{C}^m$, the notion of a quasimatrix coincides with the idea of an ordinary matrix, and taking a linear combination of columns corresponds to ordinary matrix-vector multiplication. We will typically use the word "quasimatrix" when the columns correspond to vectors in an abstract vector space like $\mathcal{P}_d$.

We can represent a quasimatrix with columns in $\mathcal{P}_d$ by an ordinary Julia array of `Polynomial`s, e.g.

```
Udemo = let
    p1 = Polynomial([1.0])
    p2 = Polynomial([-1.0, 1.0])
    p3 = Polynomial([2.0, -3.0, 1.0])
    transpose([p1; p2; p3])
end
```

```
1×3 transpose(::Vector{Polynomial{Float64, :x}}) with eltype Polynomial{Float64, :x}:
 Polynomial(1.0)  Polynomial(-1.0 + 1.0*x)  Polynomial(2.0 - 3.0*x + 1.0*x^2)
```

We can use these together with standard matrix operations involving concrete vectors, e.g.

```
Udemo*[1.0 1.0; 1.0 1.0; 1.0 0.0]
```

```
1×2 transpose(::Vector{Polynomial{Float64, :x}}) with eltype Polynomial{Float64, :x}:
 Polynomial(2.0 - 2.0*x + 1.0*x^2)  Polynomial(1.0*x)
```

We also want to be able to multiply a dual vector by a quasimatrix to get a concrete row vector:

```
Base.:*(w :: DualVector, U :: AbstractMatrix) = map(w, U)
```

For example,

```
DualVector(p->p(0.0)) * Udemo
```

```
1×3 transpose(::Vector{Float64}) with eltype Float64:
 1.0  -1.0  2.0
```

We sometimes want to refer to a subset of the columns. In mathematical writing, we will use notation similar to that in Julia or MATLAB:

$$U_{:,p:q} = \begin{bmatrix} u_p & u_{p+1} & \dots & u_q \end{bmatrix}.$$

We will also sometimes (horizontally) *concatenate* two quasimatrices with columns in the same $\mathcal{V}$, e.g.

$$U = \begin{bmatrix} U_{:,1:p} & U_{:,p+1:n} \end{bmatrix}.$$

The range of the horizontal concatenation of two quasimatrices is the same as the sum of the ranges, i.e.

$$\mathcal{R}(U) = \mathcal{R}(U_{:,1:p}) + \mathcal{R}(U_{:,p+1:n}).$$

4.1.4.2 Row quasimatrices

Every vector space $\mathcal{V}$ over a field $\mathbb{F}$ has a natural dual space $\mathcal{V}^*$ of linear functions from $\mathcal{V}$ to $\mathbb{F}$ (aka linear functionals). A *row* quasimatrix is a list of vectors in a dual space $\mathcal{V}^*$ interpreted as the rows of a matrix-like object, i.e.

$$W^* = \begin{bmatrix} w_1^* \\ w_2^* \\ \vdots \\ w_m^* \end{bmatrix}.$$

We write linear combinations of rows as

$$c^* W^* = \sum_{i=1}^{m} \bar{c}_i w_i^*;$$

the set of all such linear combinations is the *row space* of the quasimatrix. We can refer to a subset of rows with colon notation as in Julia or MATLAB:

$$(W^*)_{p:q,:} = \begin{bmatrix} w_p^* \\ w_{p+1}^* \\ \vdots \\ w_q^* \end{bmatrix}.$$

We can also combine row quasimatrices via horizontal concatenation, e.g.

$$W^* = \begin{bmatrix} (W^*)_{1:p,:} \\ (W^*)_{p+1:m,:} \end{bmatrix}.$$

In Julia, analogous to the column quasimatrices over spaces $\mathcal{P}_d$, we can define a structure representing row quasimatrices over spaces $\mathcal{P}_d^*$.

```
Wdemo = DualVector.(
    [p -> p(0.0);
     p -> p(1.0);
     p -> integrate(p, -1.0, 1.0)])
```

```
3-element Vector{DualVector}:
 DualVector(var"#27#30"())
 DualVector(var"#28#31"())
 DualVector(var"#29#32"())
```

4.1.4.3 Duality relations

For a column quasimatrix U with columns in $\mathcal{V}$, the map $c \mapsto Uc$ takes a concrete vector $c \in \mathbb{F}^n$ to an abstract vector $\mathcal{V}$. Alternately, if $w^* \in \mathcal{V}^*$ is a linear functional, we have

$$d^* = w^*U = \begin{bmatrix} w^*u_1 & w^*u_2 & \dots & w^*u_n \end{bmatrix},$$

representing a concrete row vector such that $d^*c = w^*(Uc)$. So U can be associated either with a linear map from a concrete vector space $\mathbb{F}^n$ to an abstract vector space $\mathcal{V}$, or with a linear map from the abstract dual space $\mathcal{V}^*$ to a concrete (row) vector space associated with linear functionals on $\mathbb{F}^n$.

```
# Example: map concrete [1; 1; 1] to a Polynomial
Udemo*[1; 1; 1]
```

$2.0 - 2.0 \cdot x + 1.0 \cdot x^2$

```
# Example: map DualVector to a concrete row
DualVector(p->p(0))*Udemo
```

```
1×3 transpose(::Vector{Float64}) with eltype Float64:
 1.0  -1.0  2.0
```

Similarly, a row quasimatrix W^* can be interpreted as a map from a concrete row vector to an abstract dual vector in $\mathcal{V}^*$, or as a map from an abstract vector in $\mathcal{V}$ to a concrete vector in $\mathbb{F}^m$.

```
# Example: map concrete row to an abstract dual vector
typeof([1.0; 1.0; 1.0]'*Wdemo)
```

```
DualVector
```

```
# Example: map Polynomial to a concrete vector
Wdemo.*Polynomial([2.0, -3.0, 1.0])
```

```
3-element Vector{Float64}:
 2.0
 0.0
 4.666666666666666
```

Composing the actions of a row quasimatrix with a column quasimatrix gives us an ordinary matrix representing a mapping between two abstract vector spaces:

$$W^*U = \begin{bmatrix} w_1^*u_1 & \cdots & w_1^*u_n \\ \vdots & & \vdots \\ w_m^*u_1 & \cdots & w_m^*u_n \end{bmatrix}$$

For example,

```
Wdemo*Udemo
```

```
3×3 Matrix{Float64}:
 1.0  -1.0  2.0
 1.0   0.0  0.0
 2.0  -2.0  4.66667
```

Conversely, composing the actions of a column quasimatrix with a row quasimatrix (assuming $m = n$) gives us an operator mapping $\mathcal{V}$ to $\mathcal{V}$:

$$UW^* = \sum_{k=1}^{n} u_k w_k^*.$$

In Julia, we can write this mapping as

```
UWdemo(p) = Udemo*(Wdemo.*p)
```

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

These two compositions correspond respectively to the *inner product* and *outer product* interpretations of matrix-matrix multiplication.

4.1.4.4 Infinite quasimatrices

We may (rarely) discuss "infinite" quasimatrices with a countable number of columns. We will be analysts about it instead of algebraists. In the case of column quasimatrices:

- We assume the columns lie in some complete vector space.
- When we take a linear combination Uc, we assume the sequence c is restricted to make sense of
$$Uc = \lim_{n\to\infty} U_{:,1:n}c_{1:n}.$$
Usually, this will mean c is bounded (ℓ^∞), absolutely summable (ℓ^1), or absolutely square summable (ℓ^2).
- We take the "range" of such a quasimatrix to be the set of allowed limits of linear combinations.

We can similarly define infinite versions of row-wise quasimatrices.

4.1.5 Bases

A *basis set* for a finite-dimensional vector space $\mathcal{V}$ is a finite set $S \subset \mathcal{V}$ that spans $\mathcal{V}$ and whose elements are linearly independent — that is, no nonzero linear combination of entries of S is equal to zero. We can equivalently define a *basis quasimatrix* U such that $\mathcal{R}(U) = \mathcal{V}$ and the mapping $c \mapsto Uc$ is one-to-one. That is, we think of a basis quasimatrix as an invertible linear mapping from a concrete vector space $\mathbb{F}^n$ to an abstract space $\mathcal{V}$. While we may refer to either the set or the quasimatrix as "a basis" when terminology is unambiguous, by default we use quasimatrices.

Thinking of dual spaces as consisting of rows, we generally represent the basis of $\mathcal{V}^*$ by a row quasimatrix W^*. The map $d^* \mapsto d^*W^*$ is then an invertible linear map from the set of concrete row vectors wo $\mathcal{V}^*$. If U and W^* are bases of $\mathcal{V}$ and $\mathcal{V}^*$, the matrix $X = W^*U$ represents the composition of two invertible maps (from $\mathbb{F}^n$ to $\mathcal{V}$ and back via the two bases), and so is invertible. The map $U^{-1} = X^{-1}W^*$ is the dual basis associated with U, which satisfies that $U^{-1}U$ is the n-by-n identity matrix and UU^{-1} is the identity map on $\mathcal{V}$.

If U and $\hat{U} = UX$ are two bases for $\mathcal{V}$, a vector can be written as $v = Uc$ or $v = \hat{U}\hat{c}$ where $\hat{c} = X^{-1}c$. Because we use X to change from the U basis to the $\hat{U}$ basis and use X^{-1} to change from the c coefficients to the $\hat{c}$ coefficients, we say the coefficient vector is *contravariant* (it changes in contrast to the basis). In contrast, an abstract dual vector $w^* \in \mathcal{V}^*$ can be written in terms of the U and $\hat{U}$ vectors with the concrete row vectors $d^* = w^*U$ or $\hat{d}^* = w^*\hat{U} = d^*X$. Because the transformations from U to $\hat{U}$ and from d^* to $\hat{d}^*$ both involve the same operation with X, we say these coefficient vector is *covariant* (it changes together with the basis).

4.1.5.1 Component conventions

We will generally follow the convention common in many areas of numerical analysis: a vector $v \in \mathcal{V}$ is associated with a column, a dual vector $w^* \in \mathcal{V}^*$ is associated with a row, and writing a dual vector followed by a vector (w^*v) indicates evaluation. However, there are some shortcomings to this convention, such as situations where we want to apply a dual vector to a vector but some other notational convention prevents us from writing the two symbols adjacent to each other. This issue becomes particularly obvious in multilinear algebra (tensor algebra), as we will see in Section 4.5.

An alternate convention that addresses some of these issues is frequently used in areas of physics where tensors are particularly common. In this convention ("Einstein notation"), vectors with respect to a basis

$$V = \begin{bmatrix} \mathbf{v}_1 & \dots & \mathbf{v}_n \end{bmatrix}$$

are written as

$$\mathbf{v} = \mathbf{v}_i c^i$$

where the *summation convention* is in effect, i.e. there is an implicit summation over repeated indices. Dual vectors are written with respect to the basis

$$V^{-1} = \begin{bmatrix} \mathbf{w}^1 \\ \vdots \\ \mathbf{w}^n \end{bmatrix}$$

as

$$\mathbf{w}^* = d_i \mathbf{w}^i.$$

Therefore

$$\mathbf{w}^*\mathbf{v} = d_i c^i.$$

Note that ordering of the symbols in the product makes no difference in this notation: $d_i c^i$ and $c^i d_i$ indicate the same scalar value.

The convention of using subscripts to denote vector coefficients (contravariant coefficients) and superscripts to denote dual coefficients (covariant coefficients) gives us a mechanism for "type checking" expressions: summation over a repeated index should involve one superscript (a covariant index) and one subscript (a contravariant index). At the same time, many authors

who follow the summation convention only use subscript indices in order to save the superscript space for other purposes. *Caveat lector*![4]

Indices are useful for matching functions (dual vectors) and arguments (vectors) without explicit reference to position in an expression. At the same time, indicial notation presumes a basis, with particularly basis-centric authors sometimes writing statements like "a vector is an array that transforms like a vector." While we usually use bases in our computation, we prefer a basis-independent perspective for mathematical arguments when possible. With this said, we will revert to indicial notation when the situation merits it.

4.1.5.2 Polynomial bases

As an example of different choices of bases, we consider the polynomial spaces $\mathcal{P}_d$. A common choice of basis for $\mathcal{P}_d$ is the *power basis*

$$X_{0:d} = \begin{bmatrix} 1 & x & x^2 & \dots & x^d \end{bmatrix}.$$

That is, we write polynomials $p \in \mathcal{P}_d$ in terms of the basis as

$$p = X_{0:d}c = \sum_{j=0}^{d} c_j x^j.$$

A common basis for $\mathcal{P}_d^*$ is a basis of point evaluation functionals, i.e.

$$Y^* p = \begin{bmatrix} p(x_0) \\ p(x_1) \\ \vdots \\ p(x_d) \end{bmatrix}$$

for distinct points $x_0, x_1, \dots, x_d$. The matrix $Y^* X_{0:d}$ is the *Vandermonde* matrix

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^d \\ 1 & x_1 & x_1^2 & \dots & x_1^d \\ 1 & x_2 & x_2^2 & \dots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \dots & x_d^d \end{bmatrix}.$$

The basis $V^{-1}W^*$ is the dual basis associated with the monomial basis. For example, for quartic polynomials with equally spaced polynomials on $[-1, 1]$, we can write

[4] "Caveat lector": let the reader beware!

```
let
    p = Polynomial([1.0; 2.0; 3.0; 4.0; 5.0])
    X = range(-1.0, 1.0, length=5)
    V = [xi^j for xi in X, j in 0:4]
    c = V\p.(X)
end
```

```
5-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

That is, if $p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + c_4 x^4$, and let p_X denote the vector of samples $p(-1), p(-0.5), p(0), p(0.5), p(1)$, then $c = V^{-1} p_X$. This is, alas, not a particularly good numerical approach to recovering a polynomial from point values, a topic that we will return to in Chapter 10.

The power basis is not the only basis for $\mathcal{P}_d$. Other common choices include Newton or Lagrange polynomials with respect to a set of points. Along with Vandermonde matrices, we will discuss these topics in Chapter 10. We will sometimes use the (first kind) Chebyshev[5] polynomial basis $T_{0:d}$ with columns $T_0(x), \ldots, T_d(x)$ given by the recurrence

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{j+1}(x) &= 2xT_j(x) - T_{j-1}(x), \quad j \geq 1. \end{aligned}$$

The Chebyshev polynomials can also be written on $[-1, 1]$ in terms of trigonometric functions:

$$T_m(\cos(\theta)) = \cos(m\theta).$$

This connection between Chebyshev polynomials and trigonometric functions is part of their utility, and allows us to use transform methods for working with Chebyshev polynomials (Chapter 19).

The related second kind Chebyshev polynomial basis $U_{0:d}$ satisfies

$$\begin{aligned} U_0(x) &= 1 \\ U_1(x) &= 2x \\ U_{j+1}(x) &= 2xU_j(x) - U_{j-1}(x), \quad j \geq 1. \end{aligned}$$

[5] Pafnuty Chebyshev was a nineteenth century Russian mathematician, and his name has been transliterated from the Cyrillic alphabet into the Latin alphabet in several different ways. We inherit our usual spelling from one of the French transliterations, but the symbol T for the polynomials comes from the German transliteration Tschebyscheff.

These polynomials satisfy

$$U_m(\cos(\theta))\sin(\theta) = \sin(m\theta).$$

These polynomials are useful in part because of the derivative relationship $T'_m(x) = mU_{m-1}(x)$.

In matrix terms, we can write the relationship between the Chebyshev polynomials and the power basis as $T_{0:d} = X_{0:d}M$ where the matrix M is computed via the three-term recurrence:

```
function chebyshev_power_coeffs(d, kind=1)
    n = max(d+1, 2)
    M = zeros(n, n)
    M[1,1] = 1                          # p0(x) = 1
    M[2,2] = kind                       # p1(x) = kind*x
    for j = 2:d                         # p_{j+1}(x) =
        M[2:end,j+1] .= 2*M[1:end-1,j]  #   2x p_j(x)
        M[:,j+1] .-= M[:,j-1]           #   -p_{j-1}(x)
    end
    UpperTriangular(M[1:d+1,1:d+1])
end
```

```
chebyshev_power_coeffs (generic function with 2 methods)
```

For example, for the quadratic case, we have

```
chebyshev_power_coeffs(2)
```

```
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 1.0  0.0  -1.0
  ·   1.0   0.0
  ·    ·    2.0
```

and the coefficient vectors d and c with respect to the Chebyshev and power bases are such that $p = T_{0:d}d = X_{0:d}Md = X_{0:d}c$, so $Md = c$. For example, to write the polynomial $p(x) = 2 - 3x + x^2$ in the Chebyshev basis, we solve the triangular linear system:

```
chebyshev_power_coeffs(2) \ [2.0; -3.0; 1.0]
```

```
3-element Vector{Float64}:
  2.5
 -3.0
  0.5
```

If we would like to form a representation of the Chebyshev basis, we could use the power basis expansion

```
function chebyshev_basis(d, kind=1)
    M = chebyshev_power_coeffs(d, kind)
    transpose([Polynomial(M[1:j,j]) for j=1:d+1])
end

chebyshev_basis(2)
```

```
1×3 transpose(::Vector{Polynomial{Float64, :x}}) with eltype Polynomial{Float64, :x}:
 Polynomial(1.0)  Polynomial(1.0*x)  Polynomial(-1.0 + 2.0*x^2)
```

However, the Chebyshev polynomials are natively supported in the `Polynomials.jl` package, and we can explicitly write polynomials in terms of Chebyshev polynomials. For example,

```
ChebyshevT([2.5, -3.0, 0.5])
```

$2.5 \cdot T_0(x) - 3.0 \cdot T_1(x) + 0.5 \cdot T_2(x)$

```
convert(Polynomial, ChebyshevT([2.5, -3.0, 0.5]))
```

$2.0 - 3.0 \cdot x + 1.0 \cdot x^2$

The coefficients satisfy exactly the equations from above.

The bases $T_{0:d}$ and $U_{0:d}$ are also both bases for $\mathcal{P}_d$, so there must be a change of basis matrix that converts between the two. We can look up or derive that

$$T_n(x) = \frac{1}{2}\left(U_n(x) - U_{n-2}(x)\right),$$

or, in terms of a matrix representation of the change of basis,

$$T_{0:d} = U_{0:d} B$$

where B is the upper triangular matrix

$$B = \begin{bmatrix} 1 & -\frac{1}{2} & & & \\ & \frac{1}{2} & -\frac{1}{2} & & \\ & & \frac{1}{2} & \ddots & \\ & & & \ddots & -\frac{1}{2} \\ & & & & \frac{1}{2} \end{bmatrix}.$$

Alternately, in code we have

```
function change_U_to_T(d)
    B = Matrix(0.5*I, d+1, d+1)
    B[1,1] = 1.0
    for j = 3:d+1
        B[j-2,j] = -0.5
    end
    B
end
```

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

That is, we should have

```
chebyshev_basis(5) ==
    chebyshev_basis(5,2) * change_U_to_T(5)
```

```
true
```

In addition to the Chebyshev polynomials, we will sometimes want the Legendre polynomial basis $P_{0:d}(x)$ with columns $P_j(x)$ given by the recurrence

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ (j+1)P_{j+1}(x) &= (2j+1)xP_j(x) - jP_{j-1}(x). \end{aligned}$$

As with the Chebyshev polynomials, we can define an upper triangular matrix whose columns are coefficients for the Legendre polynomials in the power basis:

```
function legendre_power_coeffs(d)
    n = max(d+1, 2)
    M = zeros(n, n)
    M[1,1] = 1
    M[2,2] = 1
    for j = 2:d
        M[2:end,j+1] .= (2*j-1)*M[1:end-1,j]
        M[:,j+1] .-= (j-1)*M[:,j-1]
        M[:,j+1] ./= j
    end
    UpperTriangular(M[1:d+1,1:d+1])
end
```

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

If we want a representation in terms of a list of `Polynomial` objects (representing the basis polynomials), we can simply convert the columns:

```
function legendre_basis(d)
    M = legendre_power_coeffs(d)
    transpose([Polynomial(M[1:j,j]) for j=1:d+1])
end

legendre_basis(2)
```

```
1×3 transpose(::Vector{Polynomial{Float64, :x}}) with eltype Polynomial{Float64, :x}:
 Polynomial(1.0)  Polynomial(1.0*x)  Polynomial(-0.5 + 1.5*x^2)
```

Each basis we have discussed has a different use case. Sometimes the "obvious" choice of basis (e.g. the standard basis in $\mathbb{R}^n$ or the power basis in $\mathcal{P}_d$) is not the best choice for numerical computations.

4.1.5.3 Block bases

Consider the direct sum

$$\mathcal{V} = \mathcal{U}_1 \oplus \mathcal{U}_2 \oplus \ldots \oplus \mathcal{U}_p.$$

Given basis quasimatrices U_j for the spaces $\mathcal{U}_j$, there is an associated basis quasimatrix V for $\mathcal{V}$ defined by horizontal concatenation:

$$V = \begin{bmatrix} U_1 & U_2 & \ldots & U_p \end{bmatrix}.$$

Conversely, we can think of partitioning the columns of a basis V for $\mathcal{V}$ into disjoint subsets; then each subset is itself a basis for some subspace, and $\mathcal{V}$ is a direct sum of those subspaces. This sort of "block thinking" will generally be useful in our treatment of numerical linear algebra.

4.1.5.4 Infinite bases

The notion of a basis can be generalized to the case of an infinite-dimensional space $\mathcal{V}$ in two ways. The algebraists version (the *Hamel basis*) is a set of linearly independent vectors such that any vector is a *finite* linear combination of basis vectors. Such a basis always exists if we assume the axiom of choice, but is not particularly useful for our purposes. More useful for topics such as approximation theory and probability is the analyst's notion of a basis: an infinite quasimatrix that represents a map between some sequence space (e.g. ℓ^1, ℓ^2, or ℓ^∞) and an abstract vector space.

4.1.6 Vector norms

A *norm* $\|\cdot\|$ measures vector lengths. It is positive definite, (absolutely) homogeneous, and sub-additive:

$$\begin{aligned}\|v\| \geq 0 \text{ and } \|v\| = 0 \text{ iff } v = 0 \\ \|\alpha v\| = |\alpha|\|v\| \\ \|u+v\| \leq \|u\| + \|v\|.\end{aligned}$$

The three most common vector norms we work with in $\mathbb{R}^n$ are the Euclidean norm (aka the 2-norm), the ∞-norm (or max norm), and the 1-norm:

$$\begin{aligned}\|v\|_2 &= \sqrt{\sum_j |v_j|^2} \\ \|v\|_\infty &= \max_j |v_j| \\ \|v\|_1 &= \sum_j |v_j|\end{aligned}$$

These are all available via Julia's `norm` function:

```
let
    x = [3.0; 4.0]
    norm(x),      # Euclidean norm (also written norm(x,2))
    norm(x,Inf),  # Max norm
    norm(x,1)     # 1-norm
end
```

```
(5.0, 4.0, 7.0)
```

Many other norms can be related to one of these three norms. In particular, a "natural" norm in an abstract vector space will often look strange in the corresponding concrete representation with respect to some basis function. For example, consider the vector space of polynomials with degree at most 2 on $[-1,1]$. This space also has a natural Euclidean norm, max norm, and 1-norm; for a given polynomial $p(x)$ these are

$$\begin{aligned}\|p\|_2 &= \sqrt{\int_{-1}^{1} |p(x)|^2\,dx} \\ \|p\|_\infty &= \max_{x\in[-1,1]} |p(x)| \\ \|p\|_1 &= \int_{-1}^{1} |p(x)|\,dx.\end{aligned}$$

These norms are not built into the `Polynomials` package, but they are not difficult to compute for real polynomials:

```
real_roots(p) = sort(real(filter(isreal, roots(p))))
real_roots(p, a, b) = filter(x -> (a <= x <= b),
                             real_roots(p))

normL2(p) = sqrt(integrate(p*p, -1, 1))

function normL∞(p)
    nodes = [-1.0; real_roots(derivative(p), -1, 1); 1.0]
    maximum(abs.(p.(nodes)))
end

function normL1(p)
    nodes = [-1.0; real_roots(p, -1, 1); 1.0]
    sum(abs(integrate(p, nodes[j], nodes[j+1]))
        for j = 1:length(nodes)-1)
end

let
    p = Polynomial([-0.5, 0.0, 1.0])
    normL2(p),
    normL∞(p),
    normL1(p)
end
```

```
(0.483045891539648, 0.5, 0.60947570824873)
```

The same norms can also be approximated from samples using the midpoint rule (**?@sec-quad-diff-ch**):

```
let
    n = 100
    p = Polynomial([-0.5, 0.0, 1.0])
    X = range(-1.0, 1.0, length=n+1)
    X = (X[1:end-1] + X[2:end])/2
    pX = p.(X)
    norm(pX)*sqrt(2/n),
    norm(pX, Inf),
    norm(pX, 1)*2/n
end
```

```
(0.48297688971626773, 0.4999, 0.6093599999999999)
```

Of course, when we write $p(x)$ in terms of the coefficient vector with respect to the power basis (for example), the max norm of the polynomial is not the same as the max norm of the coefficient vector.

In a finite-dimensional vector space, all norms are *equivalent*: that is, if $\|\cdot\|$ and $|||\cdot|||$ are two norms on the same finite-dimensional vector space, then there exist constants c and C such that for any v in the space,

$$c\|v\| \leq |||v||| \leq C\|v\|.$$

Of course, there is no guarantee that the constants are small! In infinite-dimensional spaces, not all norms are equivalent.

A normed vector space is a metric space, and so we can use it to define things like Cauchy sequences and limits. In a finite-dimensional normed space, any Cauchy sequence $\{v_k\}_{k=1^\infty}$ in a finite-dimensional vector space $\mathcal{V}$ converges to a limit $v_* \in \mathcal{V}$. A normed infinite-dimensional vector space where Cauchy sequences converge (i.e. a space that is complete under the norm) is called a *Banach space*.

For any Banach space $\mathcal{V}$, we can define the *dual norm* for continuous functionals w^* in the dual space $\mathcal{V}^*$ by

$$\|w^*\| = \sup_{v \neq 0} \frac{w^* v}{\|v\|} = \sup_{\|v\|=1} w^* v.$$

In the finite-dimensional case, the supremum is always achieved, i.e. for any w^* there exists a v of unit norm ($\|v\| = 1$) such that $w^* v = \|w^*\|$. Conversely, for any v there exists a w^* of unit norm such that $w^* v = \|v\|$. The ∞-norm and the 1-norm are dual norms of each other, and the Euclidean norm is its own dual.

4.1.7 Inner products

An *inner product* $\langle \cdot, \cdot \rangle$ is a function from two vectors into the real numbers (or complex numbers for an complex vector space). It is positive definite, linear in the first slot, and symmetric (or Hermitian in the case of complex vectors); that is:

$$\begin{aligned} \langle v, v \rangle &\geq 0 \text{ and } \langle v, v \rangle = 0 \text{ iff } v = 0 \\ \langle \alpha u, w \rangle &= \alpha \langle u, w \rangle \text{ and } \langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle \\ \langle u, v \rangle &= \overline{\langle v, u \rangle}, \end{aligned}$$

where the overbar in the latter case corresponds to complex conjugation. The *standard inner product* on $\mathbb{C}^n$ is

$$x \cdot y = y^* x = \sum_{j=1}^{n} \bar{y}_j x_j.$$

A vector space with an inner product is sometimes called an *inner product space* or a *Euclidean space*.

4.1.7.1 The Riesz map

The inner product defines an (anti)linear map (the *Riesz map*) from vectors v to dual vectors v^* via $v^*u = \langle u, v\rangle$. In terms of the Julia code in this chapter,

```
riesz(dot) = v -> DualVector(u -> dot(u, v))
```

According to the *Riesz representation theorem*, in finite-dimensional spaces, this defines a 1-1 correspondence between vectors and dual vectors. For $\mathbb{R}^n$ or $\mathbb{C}^n$ under the standard inner product, the Riesz map is simply the (conjugate) transpose:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \mapsto \begin{bmatrix} \bar{c}_1 & \bar{c}_2 & \dots & \bar{c}_n \end{bmatrix}.$$

Going beyond single vectors at a time, we can think of the Riesz map as inducing a mapping from colum quasimatrices to row quasimatrices.

In infinite-dimensional inner product spaces, we have a 1-1 correspondence between vectors and *continuous* dual vectors (which is all dual vectors for a *complete* inner product space). A (not-necessarily finite-dimensional) Euclidean space that is complete under the Euclidean norm is called a *Hilbert space*.

4.1.7.2 The Gram matrix

Suppose $\mathcal{V}$ is an arbitrary space with an inner product and V is a basis for that space. Then for vectors expressed in that basis, the inner product is

$$\begin{aligned} \langle Vc, Vd\rangle &= \left\langle \sum_j v_j c_j, \sum_i v_i d_i \right\rangle \\ &= \sum_{i,j} \bar{d}_i c_j \langle v_j, v_i\rangle \\ &= d^* M c = \langle c, d\rangle_M, \end{aligned}$$

where $M = V^*V$ is the *Gram matrix* whose entries are inner products of basis elements. In Julia code, we have

```
gram(dot, V) = Symmetric([dot(vj, vi) for vi in V[:], vj in V[:]])
```

The Gram matrix is Hermitian and (because V is a basis) positive definite. We call the expression d^*Mc the M-inner product between the concrete vectors c and d.

We can also write the Riesz map in terms of V and M. Given $w^* \in \mathcal{V}^*$, the Riesz map is $w = VM^{-1}(w^*V)^*$; we can verify this based on the expression of the $\mathcal{V}$-inner product as an M-inner product over coefficient vectors:

$$\langle u, w \rangle = \langle V^{-1}u, M^{-1}(w^*V)^* \rangle = w^*VM^{-*}MV^{-1}u = w^*u.$$

4.1.7.3 Euclidean norms

Every inner product defines a corresponding norm

$$\|v\| = \sqrt{\langle v, v \rangle}.$$

The inner product can also be recovered from the norm. In general, we can expand

$$\begin{aligned} \|u+v\|^2 &= \langle u+v, u+v \rangle \\ &= \langle u, u \rangle + \langle v, u \rangle + \langle u, v \rangle + \langle v, v \rangle \\ &= \|u\|^2 + 2\mathfrak{R}\langle u, v \rangle + \|v\|^2 \end{aligned}$$

Therefore

$$\begin{aligned} \mathfrak{R}\langle u, v \rangle &= \frac{1}{2}\left(\|u+v\|^2 - \|u\|^2 - \|v\|^2\right) \\ \mathfrak{I}\langle u, v \rangle = \mathfrak{R}\langle u, iv \rangle &= \frac{1}{2}\left(\|u+iv\|^2 - \|u\|^2 - \|v\|^2\right). \end{aligned}$$

Of course, for real vector spaces only the first equation is needed.

4.1.7.4 Angles and orthogonality

The inner product and the associated norm satisfy the *Cauchy-Schwarz* inequality

$$|\langle u, v \rangle| \leq \|u\|\|v\|.$$

We define the cosine of the angle between nonzero real vectors u and v to be

$$\cos\theta = \frac{\langle v, w \rangle}{\|v\|\|w\|}.$$

Because of the Cauchy-Schwarz inequality, the cosine is always at most one in absolute value. For a real inner product space, the expansion of $\|u+v\|^2$ and the definition of the cosine gives us the *law of cosines*: for any nonzero u and v,

$$\|u+v\|^2 = \|u\|^2 + 2\|u\|\|v\|\cos(\theta) + \|v\|^2.$$

We say two vectors u and v are *orthogonal* with respect to an inner product if $\langle u, v \rangle = 0$. If u and v are orthogonal, we have the *Pythagorean theorem*:

$$\|u+v\|^2 = \|u\|^2 + \|v\|^2.$$

If $\mathcal{U}$ is any subspace of an inner product space $\mathcal{V}$, we can define the *orthogonal complement* $\mathcal{U}^\perp$:

$$\mathcal{U}^\perp = \{v \in \mathcal{V} : \forall u \in \mathcal{U}, \langle u, v\rangle = 0\}.$$

We can always write $\mathcal{V}$ as the direct sum $\mathcal{U} \oplus \mathcal{U}^\perp$. In other words, the orthogonal complement is isomorphic to the quotient space $\mathcal{V}/\mathcal{U}$, and in inner product spaces these two are often identified with each other.

Vectors that are mutually orthogonal and have unit length in the Euclidean norm are said to be *orthonormal*. A basis V for an n-dimensional space $\mathcal{V}$ is an *orthonormal basis* if all its columns are orthonormal. In this case, V^*V is the n-by-n identity matrix and VV^* is the identity map on $\mathcal{V}$. That is, for an orthonormal basis quasimatrix, the Riesz map and the dual basis are the same, i.e. $U^* = U^{-1}$.

4.1.7.5 Einstein notation

For physicists working with Einstein notation on real vector spaces, the Gram matrix (aka the *metric tensor*) is sometimes written in terms of the components $g_{ij} = \langle \mathbf{v}_j, \mathbf{v}_i\rangle$; the inverse of the Gram matrix is written with components g^{ij}. In this convention, the Riesz map from $\mathcal{V}^*$ to $\mathcal{V}$ is associated with using the metric tensor to "raise an index":

$$v^i = g^{ij} w_j.$$

Conversely, the inverse Riesz map from $\mathcal{V}$ to $\mathcal{V}^*$ is associated with using the metric tensor to "lower an index":

$$w_i = g_{ij} v^j.$$

And the Euclidean norm is written as

$$\|\mathbf{v}\|^2 = g_{ij} v^i v^j$$

When an orthonormal basis is used, the Gram matrix is just the identity, and raising or lowering indices appears to be a purely formal matter.

4.1.7.6 The L^2 inner product

Returning to our example of a vector space of polynomials, the standard $L^2([-1,1])$ inner product is

$$\langle p, q\rangle_{L^2([-1,1])} = \int_{-1}^{1} p(x)\bar{q}(x)\,dx.$$

In Julia code, we can write this as

```
dotL2(p, q) = integrate(p*conj(q), -1, 1)
```

The Gram matrix for this inner product with the power basis (using zero-based indexing) is

$$a_{ij} = \int_{-1}^{1} x^{i-1} x^{j-1}\, dx = \begin{cases} 2/(i+j+1), & i+j \text{ even} \\ 0, & \text{otherwise} \end{cases}$$

which we can implement in Julia as

```
gram_L2_power(d) = [(i+j)%2 == 0 ? 2.0/(i+j+1) : 0.0
                    for i=0:d, j=0:d]
```

For example, for quadratics we have

```
gram_L2_power(2)
```

```
3×3 Matrix{Float64}:
 2.0       0.0       0.666667
 0.0       0.666667  0.0
 0.666667  0.0       0.4
```

Hence we can compute $\langle 1 + x + x^2, 2 - 3x + x^2 \rangle$ either via

```
let
    p = Polynomial([1.0; 1.0; 1.0])
    q = Polynomial([2.0; -3.0; 1.0])
    dotL2(p, q)
end
```

```
4.4
```

or via

```
[1.0; 1.0; 1.0]'*gram_L2_power(2)*[2.0; -3.0; 1.0]
```

```
4.3999999999999995
```

For the Chebyshev basis, the Gram matrix has entries

$$\int_{-1}^{1} T_i(x) T_j(x)\, dx.$$

This can also be computed using the relations

$$T_m(x) T_n(x) = \frac{1}{2}\left(T_{m+n}(x) + T_{|m-n|}(x)\right)$$
$$\int_{-1}^{1} T_n(x)\, dx = \begin{cases} \frac{2}{1-n^2}, & n \text{ even} \\ 0, & n \text{ odd.} \end{cases}$$

```
function gram_L2_chebyshev(d)
    tint(j) = j%2 == 0 ? 2.0/(1-j^2) : 0.0
    m(i,j) = (tint(i+j) + tint(abs(i-j)))/2.0
    [m(i,j) for i=0:d, j=0:d]
end
```

```
gram_L2_chebyshev (generic function with 1 method)
```

For example, the Gram matrix for $T_{0:4}$ is

```
gram_L2_chebyshev(4)
```

```
5×5 Matrix{Float64}:
  2.0        0.0       -0.666667   0.0       -0.133333
  0.0        0.666667   0.0       -0.4        0.0
 -0.666667   0.0        0.933333   0.0       -0.361905
  0.0       -0.4        0.0        0.971429   0.0
 -0.133333   0.0       -0.361905   0.0        0.984127
```

We can sanity check our integration by comparing to another routine:

```
gram_L2_chebyshev(4) ≈ gram(dotL2, chebyshev_basis(4)[:])
```

```
true
```

For the Legendre basis, the Gram matrix is

```
gram(dotL2, legendre_basis(4)[:])
```

```
5×5 Symmetric{Float64, Matrix{Float64}}:
 2.0  0.0       0.0  0.0       0.0
 0.0  0.666667  0.0  0.0       0.0
 0.0  0.0       0.4  0.0       0.0
 0.0  0.0       0.0  0.285714  0.0
 0.0  0.0       0.0  0.0       0.222222
```

That is, the Gram matrix for the Legendre basis is diagonal, with diagonal entries $2/1, 2/3, 2/5, 2/7, 2/9, \ldots$. In other words, the Legendre basis vectors are orthogonal with respect to the $L^2([-1,1])$ inner product. The *normalized* Legendre polynomials are

$$\sqrt{\frac{2j+1}{2}} P_j,$$

or, in code

```
normalized_legendre_basis(d) =
    transpose([sqrt((2j+1)/2)*Pj
               for (Pj,j) = zip(legendre_basis(d), 0:d)])
```

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

and these form an orthonormal basis for the $\mathcal{P}_d$ spaces under the $L^2([-1,1])$ inner product. Hence, the Gram matrix for a normalized Legendre basis (using the $L^2([-1,1])$ inner product) is just the identity:

```
let
    P̄ = normalized_legendre_basis(3)
    gram(dotL2, P̄)
end
```

```
4×4 Symmetric{Float64, Matrix{Float64}}:
 1.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0
```

Using an orthonormal basis U makes it straightforward to evaluate the Riesz map. For a given functional $w^* \in \mathcal{V}^*$ the associated $w \in \mathcal{V}$ under the Riesz map is

$$w = U(w^*U)^*,$$

where $(w^*U)^*$ represents the ordinary conjugate transpose of the concrete row vector w^*U. Then taking the inner produce with w is equivalent to evaluating w^*, since

$$\langle Uc, w\rangle = \langle Uc, U(w^*U)^*\rangle = (w^*U)c = w^*(Uc).$$

In the case of the normalized Legendre polynomials, for example, this gives us a way of computing a vector δ_x associated with the evaluation functional δ_x^* such that $\delta_x^* p = p(x)$:

```
function dualL2_δx(x, d)
    P̄ = normalized_legendre_basis(d)
    c = [P̄j(x) for P̄j in P̄]'
    P̄*c[:]
end
```

```
dualL2_δx (generic function with 1 method)
```

For example, evaluating $2 - 3x + x^2$ at $x = 0.5$ gives $p(0.5) = \langle p, \delta_{0.5}\rangle = 0.75$; in code, we have

```
let
    p = Polynomial([2.0; -3.0; 1.0])
    δhalf = dualL2_δx(0.5, 2)
    p(0.5) ≈ dotL2(p, δhalf)
end
```

```
true
```

4.2 Maps and forms

A central object in linear algebra is a linear map between two spaces. If $\mathcal{V}$ and $\mathcal{U}$ are two vector spaces, we denote the vector space of linear maps[6] between them by $L(\mathcal{V}, \mathcal{U})$.

Closely associated with linear maps *between* two vector spaces are certain maps *from* two vector spaces. A *bilinear form* is a map $a : \mathcal{U} \times \mathcal{V} \to \mathbb{F}$ which is linear in both arguments. Such a

[6] Some authors write $L(\mathcal{V}, \mathcal{U})$ for the set of *bounded* linear maps between two spaces. This distinction matters when we are dealing with infinite-dimensional normed vector spaces — since we are *mostly* concerned with the finite-dimensional case, we will not worry about it.

form can be identified with a linear map from $\mathcal{V}$ to $\mathcal{U}^*$ by partial evaluation: $v \mapsto a(\cdot, v)$. A *sesquilinear form* is linear in one argument and antilinear (conjugate linear) in the second argument, and can be identified with an antilinear map from $\mathcal{V}$ to $\mathcal{U}^*$. We are generally interested in bilinear forms over real spaces and sesquilinear forms over complex spaces. When we want to talk about real bilinear forms and complex sesquilinear forms at the same time, we will usually just say "sesquilinear forms."

On inner product spaces, each sesquilinear form $a : \mathcal{V} \times \mathcal{W} \to \mathbb{F}$ is associated with a unique linear map $L \in L(\mathcal{V}, \mathcal{W})$ such that

$$a(v, w) = \langle Lv, w \rangle_{\mathcal{U}}.$$

That is, sesquilinear forms can be identified with linear maps and vice-versa, and so we will address them together.

A linear map from a space to itself is called a linear operator. Linear operators have enough additional structure that we will deal with them separately.

4.2.1 Dual and adjoint maps

Any linear map $L \in L(\mathcal{V}, \mathcal{U})$ comes with an associated dual map in $L(\mathcal{U}^*, \mathcal{V}^*)$ with the action $u^* \mapsto u^* L$. If both spaces have inner products, then we compose the dual map with the Riesz map on both sides to get the *adjoint* $L^* \in L(\mathcal{U}, \mathcal{V})$ with the action $u \mapsto (u^* L)^*$. Equivalently, the adjoint map has the property that

$$\langle Lv, u \rangle_{\mathcal{U}} = \langle v, L^* u \rangle_{\mathcal{V}}.$$

In concrete spaces with the standard inner product, the adjoint map is represented by the conjugate transpose of the matrix of the original map.

4.2.2 Matrices

4.2.2.1 Matrices for maps

The usual representation of a linear map $L \in L(\mathbb{F}^n, \mathbb{F}^m)$ is via an m-by-n matrix $A \in \mathbb{F}^{m \times n}$. Applying the function $y = Lx$ is equivalent to the matrix multiplication $y = Ax$:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j.$$

More generally, if $L \in L(\mathcal{V}, \mathcal{U})$ and V and U are bases for $\mathcal{V}$ and $\mathcal{U}$, then there is an associated matrix representation

$$A = U^{-1} L V.$$

The action of the matrix A comes from converting from $\mathbb{F}^n$ to $\mathcal{V}$ (via V), applying L to get a vector in $\mathcal{U}$, and mapping from $\mathcal{U}$ back to the concrete space $\mathbb{F}^m$. Equivalently, we can write the abstract operator as

$$L = UAV^{-1},$$

i.e. we map from $\mathcal{V}$ to a concrete vector in $\mathbb{F}^n$, map that to a concrete vector in $\mathbb{F}^m$ by matrix multiplication with A, and map $\mathbb{F}^m$ to $\mathcal{U}$ via the U basis.

If we consider the case where V' and U' are alternate bases for $\mathcal{V}$ and $\mathcal{U}$, there is an associated matrix representation

$$A' = U'^{-1}LV' = (U'^{-1}U)A(V^{-1}V')$$

Here the (invertible) matrix $V^{-1}V'$ represents a *change of basis*: if $v = V'c$ is a representation in the V' basis and $v = Vd$ is a representation in the V basis, then $d = V^{-1}V'c$. Similarly, the matrix $U'^{-1}U$ converts the coefficients in the U basis to coefficients in the U' basis.

4.2.2.2 Matrices for sesquilinear forms

Suppose $a : \mathcal{V} \times \mathcal{W} \to \mathbb{F}$ is a sesquilinear form and V and W are bases for $\mathcal{V}$ and $\mathcal{W}$. Then

$$a(Vx, Wy) = \sum_{j,i} x_j \bar{y}_i a(v_j, w_i) = y^* A x$$

where A is the matrix with entries $a_{ij} = a(v_j, w_i)$. In an inner product space where we can write a in terms of a linear operator L, we have

$$a(Vx, Wy) = \langle LVx, Wy \rangle_{\mathcal{W}} = y^* W^* LVx,$$

i.e. we can also think of A as W^*LV.

We note that the matrix representation of the sesquilinear form is *not* identical to the matrix representation of L as a linear map, except in the special case where we constrain V and W to be orthonormal bases. For more general matrices, the expression W^*LV involves both the matrix for the linear map ($W^{-1}LV$) and the Gram matrix ($M = W^*W$); that is, we can also write

$$y^* W^* LVx = y^*(W^*W)W^{-1}LVx = \langle W^{-1}LVx, y \rangle_M.$$

4.2.2.3 Representing derivatives

As a concrete example of matrix representations, we consider the derivative operator $D : \mathcal{P}_d \to \mathcal{P}_{d-1}$. With respect to the power basis for both $\mathcal{P}_d$ and $\mathcal{P}_{d-1}$, we write the relation

$$\frac{d}{dx} x^n = n x^{n-1}$$

as

$$DX_{0:d} = X_{0:d-1}A$$

where

$$A = \begin{bmatrix} 0 & 1 & & & \\ & & 2 & & \\ & & & 3 & \\ & & & & \ddots \\ & & & & & d \end{bmatrix}.$$

In code, we have

```
function derivative_power(d)
    A = zeros(d,d+1)
    for i=1:d
        A[i,i+1] = i
    end
    A
end
```

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

Alternately, we can write $D = X_{0:d-1}AX_{0:d}^{-1}$ or $A = X_{0:d-1}^{-1}DX_{0:d}$.

A similar relationship holds for the first and second kind Chebyshev polynomials:

$$\frac{d}{dx}T_n(x) = nU_{n-1}(x),$$

and so if we use the first-kind polynomial basis $T_{0:d}$ for $\mathcal{P}_d$ and the second-kind polynomial basis $U_{0:d-1}$ for $\mathcal{P}_{d-1}$, we have

$$DT_{0:d} = U_{0:d-1}A$$

where A is the same matrix as before. Using the change of basis, we have

$$DT_{0:d} = T_{0:d-1}B^{-1}A.$$

In code, the matrix representation $T_{0:d-1}^{-1}DT_{0:d} = B^{-1}A$ can be computed by

```
derivative_chebyshevT(d) =
    change_U_to_T(d-1)\derivative_power(d)
```

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

It is always useful to compare against another implementation of the same computation, e.g.

```
let
    Dmatrix = zeros(5,6)
    DT = derivative.(chebyshev_basis(5))
    for j = 1:length(DT)
        for ci = collect(pairs(convert(ChebyshevT, DT[j])))
            Dmatrix[ci[1]+1,j] = ci[2]
        end
    end
    Dmatrix ≈ derivative_chebyshevT(5)
end
```

```
true
```

Now consider a sesquilinear form on $\mathcal{P}_d \times \mathcal{P}_{d-1}$ given by

$$a(u, w) = \int_{-1}^{1} \bar{w}(x) u'(x)\, dx.$$

If we use the Chebyshev basis for both arguments, we have

$$\begin{aligned} a(T_{0:d}x, T_{0:d-1}y) &= y^* T_{0:d-1}^* D T_{0:d} x \\ &= y^* T_{0:d-1}^* T_{0:d-1} B^{-1} A x, \end{aligned}$$

that is, the matrix is $T_{0:d-1}^* T_{0:d-1} B^{-1} A$. In terms of codes we have written, we have (for $d = 5$)

```
derivative_chebyshevT_bilinear(d) =
    gram_L2_chebyshev(d-1)*derivative_chebyshevT(d)
```

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

As before, we can sanity check against an alternate computation

```
let
    T = chebyshev_basis(5)
    a(p, q) = dotL2(derivative(p), q)

    derivative_chebyshevT_bilinear(5) ≈
        [a(T[j+1], T[i+1]) for i=0:4, j=0:5]
end
```

```
true
```

With respect to the Legendre basis, one can differentiate the Bonnet recurrence relation for P_n and rearrange to get a recurrence relation for the derivatives:

$$P'_{n+1}(x) = (2n+1)P_n(x) + P'_{n-1}(x).$$

Together with the initial conditions $P'_0(x) = 0$ and $P'_1(x) = P_0(x)$, we have the following expression of D with respect to the Legendre bases for $\mathcal{P}_d$ and $\mathcal{P}_{d-1}$, i.e. $DP_{0:d} = P_{0:d-1}C$:

```
function derivative_legendre(d)
    DP = zeros(d,d+1)
    if d > 0
        DP[1,2] = 1
    end
    for j = 1:d-1
        DP[:,  j+2] = DP[:,j]
        DP[j+1,j+2] = 2j+1
    end
    DP
end
```

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

As usual, we sanity check our computation:

```
derivative.(legendre_basis(3)) ==
    legendre_basis(2)*derivative_legendre(3)
```

```
true
```

If we want the derivative for the normalized Legendre polynomials, it is convenient to write them as $P_{0:d}S_d$ where S_d is the diagonal matrix with elements $\sqrt{(2j+1)/2}$ for $j = 0$ to d. Then

$$DP_{0:d}S_d = P_{0:d-1}S_{d-1}(S_{d-1}^{-1}CS_d).$$

In code, we have

```
derivative_nlegendre(d) =
    sqrt.(2.0./(2*(0:d-1).+1)) .*
    derivative_legendre(d) .*
    sqrt.((2*(0:d).+1)/2)'
```

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

Unlike some of our earlier computations, there is a little difference between different approaches to computing the normalized Legendre polynomials, so we check whether the results are close rather than exactly the same:

```
let
    err = normalized_legendre_basis(2)*derivative_nlegendre(3) -
        derivative.(normalized_legendre_basis(3))
    all(iszero.(truncate.(err, atol=1e-12)))
end
```

```
true
```

4.2.3 Block matrices

Sometimes we are interested in $L \in L(\mathcal{V}, \mathcal{U})$ where the two spaces are direct sums:

$$\mathcal{V} = \mathcal{V}_1 \oplus \mathcal{V}_2 \oplus ... \oplus \mathcal{V}_n$$
$$\mathcal{U} = \mathcal{U}_1 \oplus \mathcal{U}_2 \oplus ... \oplus \mathcal{U}_m.$$

In this case, we can write $u = Lv$ with the subspace decompositions $u = u_1 + ... + u_m$ and $v = v_1 + ... + v_n$ as

$$u_i = \sum_{j=1}^{n} L_{ij} v_j$$

where $L_{ij} \in L(\mathcal{V}_j, \mathcal{U}_i)$. We can write this in *block quasimatrix* notation as

$$\begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix} = \begin{bmatrix} L_{11} & ... & L_{1n} \\ \vdots & & \vdots \\ L_{m1} & ... & L_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

Now suppose we have associated partitioned *bases*

$$V = \begin{bmatrix} V_1 & V_2 & ... & V_n \end{bmatrix}$$
$$U = \begin{bmatrix} U_1 & U_2 & ... & U_m \end{bmatrix}.$$

Then the matrix $A = U^{-1}LV$ can be thought of as a *block matrix* with blocks $A_{ij} = U_i^{-1} L_{ij} V_j$ associated with each of the L_{ij}. Concretely, if $w_i = U_i d_i$ and $v_j = V_j c_j$ for appropriately-sized concrete vectors d_i and c_i, we have

$$\begin{bmatrix} d_1 \\ \vdots \\ d_m \end{bmatrix} = \begin{bmatrix} A_{11} & ... & A_{1n} \\ \vdots & & \vdots \\ A_{m1} & ... & A_{mn} \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

4.2.4 Canonical forms

A *canonical form* for a map $L \in L(\mathcal{V}, \mathcal{U})$ is a "simple as possible" matrix representation associated with a particular choice of bases. This also makes sense when $\mathcal{V}$ and $\mathcal{U}$ are concrete vector spaces, in which case the canonical form is a sort of matrix decomposition.

In general, we will consider two types of canonical forms: those associated with *general* choices of bases and those associated with *orthonormal* bases (when $\mathcal{V}$ and $\mathcal{U}$ are inner product spaces). The canonical forms are essentially the same whether we are working with linear maps or sesquilinear forms; we will focus on the linear map case.

4.2.4.1 Block decomposition

The first step to the canonical forms for a general linear map is to decompose $\mathcal{V}$ and $\mathcal{U}$ into special direct sums. For $\mathcal{V}$, we decompose into the kernel (or null space) of L, i.e. $\mathcal{N}(L) = \{v \in \mathcal{V} : Lv = 0\}$, along with any complementary space:

$$\mathcal{V} = \mathcal{V}_1 \oplus \mathcal{V}_2, \quad \mathcal{V}_2 = \mathcal{N}(L).$$

For $\mathcal{U}$, we decompose into the range space $\mathcal{R}(L) = \{Lv : v \in \mathcal{V}\}$ and any complementary space:

$$\mathcal{U} = \mathcal{U}_1 \oplus \mathcal{U}_2, \quad \mathcal{U}_1 = \mathcal{R}(L).$$

The null space $\mathcal{N}(L)$ and the range $\mathcal{R}(L)$ are sometimes also known as the kernel and the image of L. The space $\mathcal{U}_2$ is isomorphic to $\mathcal{U}/\mathcal{R}(L)$, sometimes called the *cokernel*, while the space $\mathcal{V}_1$ is isomorphic to $\mathcal{V}/\mathcal{N}(L)$, sometimes called the *coimage*. These four spaces (the kernel, cokernel, range, and corange) play a key role in linear algebra. The rank r is thus the both the dimension of the range and the dimension of the coimage.

With respect to such a decomposition, we have the block quasimatrix

$$L = \begin{bmatrix} L_{11} & 0_{r\times(n-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{bmatrix}.$$

We have indicated the dimensions of the zero blocks here for clarity. The structure of this quasimatrix comes directly from the definition of the null space (which guarantees that the second block column should be zero) and the range space (which guarantees that the second block row should be zero). The block $L_{11} \in L(\mathcal{V}_1, \mathcal{U}_1)$ is one-to-one (because all null vectors of L are in the $\mathcal{V}_2$ space) and onto (because $\mathcal{U}_1$ is the range space). Therefore, L_{11} is invertible.

The block decomposition of a map is closely connected to the *Fredholm alternative* theorem. Consider the equations $Lx = b$, and suppose we decompose our spaces in terms of kernel, cokernel, range, and corange as above to get the quasimatrix equation

$$\begin{bmatrix} L_{11} & 0_{r\times(n-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}.$$

Then one of two conditions must hold; the system is either

- *Consistent* ($b_2 = 0$): There is a solution to the system. In fact, if x_* is any solution (a *particular* solution), then the set of all possible solutions is $\{x_* + v : Lv = 0\}$. more generally, an $(n-r)$-dimensional set of solutions
- *Inconsistent* ($b_2 \neq 0$): There is not a solution to the system ($b_2 \neq 0$), but there does exist $y^* \in \mathcal{U}^*$ such that $y^*L = 0$ and $y^*b \neq 0$.

Without the decomposition of $\mathcal{U}$ and $\mathcal{V}$, the statement of the Fredholm alternative (for finite dimensional spaces) is a little mysterious. With the decomposition, it is almost a matter of inspection.

The decomposition of a map based on range and cokernel (for $\mathcal{U}$) and kernel and coimage (for $\mathcal{V}$) works equally well in the infinite-dimensional setting. Of course, in this setting some of the block dimensions will be infinite! But in some cases when the kernel and cokernel are finite-dimensional, we can define the *index* of L to be the difference between the dimension of the kernel and the dimension of the cokernel (or the codimension of the range space, which is the same thing). In the finite-dimensional picture, this is

$$\dim(\mathcal{V}_2) - \dim(\mathcal{U}_2) = (m-r) - (n-r) = m - n.$$

In other words, the index generalizes the notion of "how many more equations than variables" there are.

4.2.4.2 General bases

Let $\mathcal{V} = \mathcal{V}_1 \oplus \mathcal{V}_2$ and let $\mathcal{U}_1 \oplus \mathcal{U}_2$ as above (i.e. $\mathcal{U}_1$ is the range space and $\mathcal{V}_2$ is the null space). For any basis V_1 for $\mathcal{V}_1$, the quasimatrix $U_1 = LV_1$ is a basis for $\mathcal{U}_1$. Concatenating any bases V_2 and U_2 for $\mathcal{V}_2$ and $\mathcal{U}_2$, we have bases $V = \begin{bmatrix} V_1 & V_2 \end{bmatrix}$ for $\mathcal{V}$ and $U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}$ for $\mathcal{U}$. With respect to these bases, we have the matrix representation

$$U^{-1}LV = \begin{bmatrix} I_{r\times r} & 0 \\ 0 & 0 \end{bmatrix}.$$

This matrix representation is completely characterized by the dimensions of $\mathcal{V}$ and $\mathcal{U}$ and the rank r.

4.2.4.3 Orthonormal bases

In a general vector space, we can use any complementary subspaces to the range and null space of L. But in an inner product space, we prefer to use the *orthogonal* complements:

$$\begin{aligned} \mathcal{V}_1 &= \mathcal{N}(L)^\perp && \text{coimage} \\ \mathcal{V}_2 &= \mathcal{N}(L) && \text{kernel / null space} \\ \mathcal{U}_1 &= \mathcal{R}(L) && \text{range / image} \\ \mathcal{U}_2 &= \mathcal{R}(L)^\perp && \text{cokernel} \end{aligned}$$

These are sometimes referred to as the "four fundamental subspaces" for a linear map[7] (see Strang (2023)).

Our canonical form in this setting involves a special choice for the bases V_1 and U_1. Specifically, we choose unit-length vectors $v_1, \dots, v_r$ by maximizing $\|Lv_j\|$ over all vectors in $\mathcal{V}$ orthogonal to $v_1, \dots, v_{j-1}$. As we will discuss in Chapter 18, the vectors $Lv_1, \dots, Lv_r$ themselves are then (nonzero) orthogonal vectors. The vectors v_j are the *right* singular vectors, the norms $\sigma_j = \|Lv_j\|$ are the *singular values*, and the vectors $u_j = Lv_j/\sigma_j$ are the *left* singular vectors. Concatenating the orthonormal bases $V_1 = \begin{bmatrix} v_1 & \dots & v_r \end{bmatrix}$ and $U_1 = \begin{bmatrix} u_1 & \dots & u_r \end{bmatrix}$ with orthonormal bases V_2 and U_2 for the orthogonal complements, we have the canonical form

$$U^* L V = \Sigma = \begin{bmatrix} \Sigma_{11} & 0 \\ 0 & 0 \end{bmatrix}$$

where Σ_{11} is the diagonal matrix with the singular values $\sigma_1, \dots, \sigma_r$ on the diagonal. By construction, these singular values are nonzero and are listed in descending order. Frequently, we write this canonical form as the *singular value decomposition* (SVD) of L:

$$L = U \Sigma V^* = U_1 \Sigma_{11} V_1^*.$$

4.2.4.4 Decomposed derivatives

We return again to our example of the derivative mapping from $\mathcal{P}_d$ to $\mathcal{P}_{d-1}$. For this example, the range space is all of $\mathcal{P}_{d-1}$ and the null space is the constant vectors. Decomposing with respect to the $L^2([-1,1])$ inner product, the cokernel and coimage are the empty set and the mean zero vectors. Therefore, we can find general bases in which the matrix is

$$\begin{bmatrix} I_{d \times d} & 0_{d \times 1} \end{bmatrix}.$$

More interesting is the singular value decomposition in the $L^2([-1,1])$ inner product. Here it is easiest to compute using the matrix representation of the derivative with respect to orthonormal

[7] In the absence of an inner product, the coimage is most naturally thought of as a subspace of $\mathcal{V}^*$ and the cokernel as a subspace of $\mathcal{W}^*$. The Riesz map allows us to move these back to subspaces of $\mathcal{V}$ and $\mathcal{W}$.

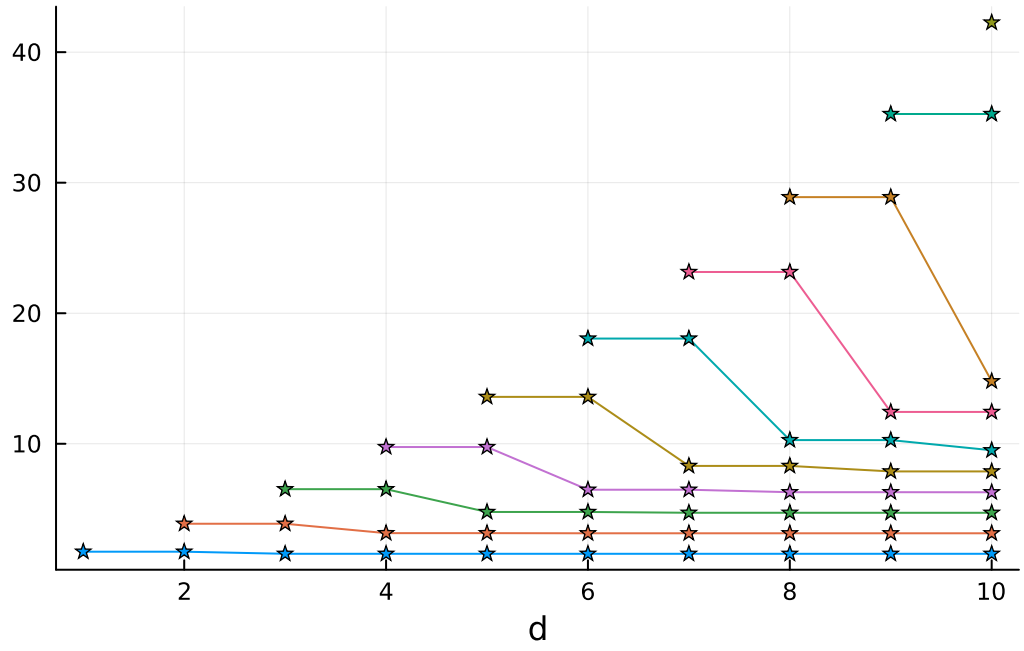
bases like the normalized Legendre polynomials $\bar{P}_{0:d}$. Then taking $C = U\Sigma V^T$ as the singular value decomposition of the matrix representing the derivative, the singular value decomposition of the derivative map is

$$D = (\bar{P}_{0:d-1}U)\Sigma(\bar{P}_{0:d}V)^T.$$

That is, the orthonormal basis vectors that diagonalize D are given by $\hat{U} = \bar{P}_{0:d-1}U$ and $\bar{P}_{0:d}V$.

The singular values of the derivative mapping from $\mathcal{P}_d$ to $\mathcal{P}_{d-1}$ depend on the dimension d of the space. It is illuminating to look at the behavior of these values in ascending order:

```
let
    dmax = 10

    # Compute the singular values for each degree d
    σs = zeros(dmax,dmax)
    for d = 1:dmax
        F = svd(derivative_nlegendre(d))
        σs[1:d,d] .= F.S[end:-1:1]
    end

    # Draw one line for the ith smallest singular value as a function of d
    p = plot(1:dmax, σs[1,:], legend=:false, marker=:star, xlabel="d")
    for i = 2:dmax
        plot!(i:dmax, σs[i,i:dmax], marker=:star)
    end
    p
end
```



As we increase d, we get more singular values; but each singular value (ordered from smallest to biggest) appears to fairly quickly converge from above to a limiting value. Indeed, we can make sense of these limits.

As an example, consider the smallest singular value and the associated basis vectors for the derivative acting on $\mathcal{P}_{10}$ with the $L^2([-1,1])$ inner product. We expect that regardless of the degree, we should have $Dv_{\min} = u_{\min}\sigma_{\min}$ where $\sigma_{\min}$ is the smallest (nonzero) singular value and $u_{\min}$ and $v_{\min}$ are the associated singular vectors (columns in the U and V bases). By $d = 10$, we also expect to be close to the limiting behavior: the smallest singular value should converge to $\pi/2$, with associated singular vectors $v_{\min}(x) \to \cos(x\pi/2)$ and $u_{\min}(x) \to \sin(x\pi/2)$. This anticipated behavior is borne out in numerical experiment:

```
let
    # Compute the SVD of D acting on P_10
    d = 10
    F = svd(derivative_nlegendre(d))
    U = normalized_legendre_basis(d-1)*F.U
    V = normalized_legendre_basis(d)*F.V
    σs = F.S

    # Get the singular triple associated with
    # the smallest (nonzero) singular value
    σ, u, v = σs[end], U[end], V[end]
```

```
    # Check various relations (use a mesh of 100 points)
    xx = range(-1,1,length=100)
    s = sign(u(0))
    println("""
- |D*u-v*σ| = $(norm((derivative(v)-u*σ).(xx), Inf))
- |sin(π/2*x)-v| = $(norm(s*sin.(π/2*xx)-v.(xx), Inf))
- |cos(π/2*x)-u| = $(norm(s*cos.(π/2*xx)-u.(xx), Inf))
- |σmin-π/2| = $(abs(σ-π/2))
""")
end
```

```
- |D*u-v*σ| = 5.076326047387471e-14
- |sin(π/2*x)-v| = 4.919919749379886e-9
- |cos(π/2*x)-u| = 1.3086657329447118e-7
- |σmin-π/2| = 3.552713678800501e-15
```

In general, the kth smallest singular value converges to $k\pi/2$, and the singular vectors converge to sines and cosines.

4.2.5 (Pseudo)inverses

Suppose $L \in L(\mathcal{V}, \mathcal{U})$ has a canonical form with respect to a general basis of

$$L = U \begin{bmatrix} I_{r\times r} & 0_{r\times(n-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(n-r)} \end{bmatrix} V^{-1}.$$

If $r \neq m$ and $r \neq n$, then L does not have an inverse. However, we can define an associated *pseudoinverse* $M \in L(\mathcal{V}, \mathcal{U})$ by

$$M = V \begin{bmatrix} I_{r\times r} & 0_{r\times(m-r)} \\ 0_{(n-r)\times r} & 0_{(n-r)\times(m-r)} \end{bmatrix} U^{-1}.$$

Let $P_V = ML$ and $P_U = LM$. In terms of the bases, we have

$$P_V = V \begin{bmatrix} I_{r\times r} & 0_{r\times(n-r)} \\ 0_{(n-r)\times r} & 0_{(n-r)\times(n-r)} \end{bmatrix} V^{-1}$$

and

$$P_U = U \begin{bmatrix} I_{r\times r} & 0_{r\times(m-r)} \\ 0_{(m-r)\times r} & 0_{(m-r)\times(m-r)} \end{bmatrix} U^{-1}.$$

Both P_V and P_U are projectors, i.e. $P_V^2 = P_V$ and $P_U^2 = P_U$. The P_V projector is the identity on the subspace $\mathcal{V}_1$ that is the complementary to the kernel, and the null space is $\mathcal{V}_1$. The P_U

projector is the identity on the range space $\mathcal{U}_1$, and the null space is $\mathcal{U}_2$. If L is one-to-one, P_V is simply the identity; and if L is onto, P_U is the identity.

The pseudoinverse depends on the decompositions $\mathcal{V} = \mathcal{V}_1 \oplus \mathcal{N}(L)$ and $\mathcal{U} = \mathcal{R}(L) \oplus \mathcal{U}_2$, though it is independent of the details of the bases chosen for the four subspaces. In an inner product space, we most often choose to decompose $\mathcal{V}$ and $\mathcal{U}$ into orthogonal complements. This gives the *Moore-Penrose pseudoinverse* (often just called "the pseudoinverse" when the context does not specify some other pseudoinverse). The Moore-Penrose pseudoinverse plays a central role in least squares and minimal norm problems, as we discuss in Chapter 17. In terms of the singular value decomposition, the Moore-Penrose pseudoinverse is

$$L^\dagger = V \begin{bmatrix} \Sigma_{11}^{-1} & 0 \\ 0 & 0 \end{bmatrix} U^*.$$

For the Moore-Penrose pseudoinverse, the associated projectors are $P_V = V_1 V_1^*$ and $P_U = U_1 U_1^*$. These are *orthogonal* projectors, since their range spaces are orthogonal complements of their null spaces. Projectors that are not orthogonal are said to be *oblique* projectors.

As an example, the Moore-Penrose pseudoinverse of the derivative operator from $\mathcal{P}_d$ to $\mathcal{P}_{d-1}$ is the indefinite integral operator $\mathcal{P}_{d-1}$ to $\mathcal{P}_d$ where the constant is chosen to make the result have zero mean. The associated projector on $\mathcal{P}_d$ is $q(x) \mapsto q(x) - \bar{q}$ where $\bar{q} = \frac{1}{2}\int_{-1}^{1} q(x)\,dx$.

4.2.6 Norms

The space $L(\mathcal{V}, \mathcal{U})$ is a vector space; and, as with other vector spaces, it makes sense to talk about norms. In particular, we frequently want norms that are *consistent* with vector norms on the range and domain spaces; that is, for any w and v, we want

$$u = Lv \implies \|u\| \leq \|L\|\|v\|.$$

Particularly useful are the norms *induced* by vector norms on $\mathcal{V}$ and $\mathcal{U}$:

$$\|A\| = \sup_{v \neq 0} \frac{\|Av\|}{\|v\|} = \sup_{\|v\|=1} \|Av\|.$$

In a finite-dimensional space[8], the supremum is achieved, i.e. there exists a maximizing v.

Suppose $S \in L(\mathcal{U}, \mathcal{V})$ and $T \in L(\mathcal{V}, \mathcal{W})$, and we have consistent norms on the three vectors spaces $\mathcal{U}, \mathcal{V}, \mathcal{W}$ and the spaces of linear maps $L(\mathcal{U}, \mathcal{V})$ and $L(\mathcal{V}, \mathcal{W})$. Then for all $u \in \mathcal{U}$, consistency implies

$$\|TSu\| \leq \|T\|\|Su\| \leq \|T\|\|S\|\|u\|.$$

[8]The notion of an induced norm makes sense in infinite-dimensional spaces as well, but there the supremum may not be achieved.

Taking the supremum over $\|u\| = 1$ implies that the induced norm on $L(\mathcal{U}, \mathcal{W})$, we have

$$\|TS\| \leq \|T\|\|S\|.$$

This fact is most often used when the norms on $L(\mathcal{U}, \mathcal{V})$ and $L(\mathcal{V}, \mathcal{W})$ are also induced norms.

For inner product spaces, we can also define norms in terms of the singular value decomposition. Some examples include

- The spectral norm (the largest of the singular values)
- The Frobenius norm (the two-norm of the singular values)
- The nuclear norm (the sum of the singular values)

All of these are also examples of *Ky Fan* norms (the Ky Fan k-norm is the sum of the k largest singular values) and *Schatten* norms (the Schatten norm is the p-norm of the vector of singular values). The spectral norm is the norm induced by the Euclidean structure on the two spaces. All the other Ky Fan and Schatten norms are consistent, though they are not induced norms.

For concrete vector spaces, we can write the *Frobenius norm* as

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}.$$

The induced norms corresponding to the vector 1-norm and ∞-norm are

$$\|A\|_1 = \max_j \sum_i |a_{ij}| \quad \text{(max absolute column sum)}$$
$$\|A\|_\infty = \max_i \sum_j |a_{ij}| \quad \text{(max absolute row sum)}$$

The norm induced by the standard Euclidean norm (the spectral norm) is harder to compute without appealing to the singular value decomposition.

4.2.7 Isometries

A linear map that preserves norms is called an isometry. Isometries are always one-to-one, since any null vector must have zero norm. If an isometry is also onto, it is sometimes called a *global* isometry. Global isometries are invertible, and the inverse is also an isometry.

For general norms, there are not always many interesting isometries. For example, the only isometries from $\mathbb{R}^n$ to itself preserving the 1-norm or the ∞-norm are (signed) permutations. For inner product spaces, though, there is a much richer group of isometries, and we will focus on this case.

Isometries between inner product spaces necessarily preserve inner products, since the inner product can be recovered from the associated norm. For global isometries between inner product spaces, the adjoint and the inverse are the same. A global isometry from a space to

itself is called an *orthogonal operator* in the real case, or a *unitary operator* in the complex case.

A unitary operator maps an orthonormal basis to another orthonormal basis. Therefore, if $L \in L(\mathcal{V}, \mathcal{U})$ and W and Z are unitary operator operators on $\mathcal{U}$ and $\mathcal{V}$, respectively, then L and WLZ have the same singular values. This also means that any norms that can be defined in terms of singular values (including the spectral norm, the Frobenius norm, and the nuclear norm) are the same for L and WLZ. This properties is known as *unitary invariance* (or *orthogonal invariance* in the real case).

4.2.8 Volume scaling

In an inner product space, we usually say that a paralleliped associated with an orthonormal basis (i.e. a unit cube) has unit volume. A unitary operator between two spaces maps a unit cube in one space to a unit cube in another space, i.e. it preserves volume. More generally, if $L \in L(\mathcal{V}, \mathcal{U})$ and V and U are orthonormal bases associated with the singular value canonical form, then we can see $L = U\Sigma V^*$ as

- A volume-preserving transformation V^* from $\mathcal{V}$ into $\mathbb{R}^n$ (or $\mathbb{C}^n$).
- A diagonal scaling operation on $\mathbb{R}^n$ that change each side length from 1 to σ_j. The volume of the scaled paralleliped in $\mathbb{R}^n$ is then $\prod_{j=1}^{n} \sigma_j$.
- A volume-preserving transformation U from $\mathbb{R}^n$ into $\mathcal{V}$.

Therefore, if $\Omega \subset \mathcal{V}$ is a region with volume $\text{vol}(\Omega)$, then

$$\text{vol}(L\Omega) = \left(\prod_{j=1}^{n} \sigma_j \right) \text{vol}(\Omega).$$

In other words, the product of the singular values gives the magnitude of the determinant of L. We will discuss determinants and signed volumes in Section 4.5.5.

4.3 Operators

An operator is a map $L \in L(\mathcal{V}, \mathcal{V})$. Everything we know about general maps also works for operators, but we have a lot of additional structure coming from the fact that the space is mapped to itself.

Most of our discussion of operators focuses on three connected ideas:

- A key feature of an operator L is its *invariant subspaces*, i.e subspaces $\mathcal{U} \subset \mathcal{V}$ such that $L\mathcal{U} = \{Lu : u \in \mathcal{U}\}$ lies within $\mathcal{U}$. Given an invariant subspace, we can analyze the associated *restriction* $L|_{\mathcal{U}}$ of L to $\mathcal{U}$. One-dimensional invariant subspaces play a particular notable role, as they contain the *eigenvectors* of L; the restrictions are the associated eigenvalues.
- We can sensibly talk about polynomials in L: if $p(z) = \sum_{j=0}^{d} c_j z^j$, then $p(L) = \sum_{j=0}^{d} c_j L^j$. Such polynomials are useful for designing and analyzing numerical methods (e.g. Krylov subspace methods for linear systems and eigenvalue problems). They are also useful as a more purely theoretical tool.

In Chapter 5, we will also discuss an alternative approach to analyzing operators by applying the tools of complex analysis to the *resolvent* $R(z) = (zI - L)^{-1}$, a rational function on $\mathbb{C}$ whose poles correspond to eigenvalues. This perspective is useful both for computation and theory. It is particularly useful if we want to generalize from the finite-dimensional to the infinite-dimensional setting.

4.3.1 Minimal polynomial

If $\mathcal{V}$ is an n-dimensional space, then $L(\mathcal{V}, \mathcal{V})$ is an n^2-dimensional space, and so there must be a linear dependency between the set of $n^2 + 1$ operators $\{L^j\}_{j=0}^{n^2}$. Consequently, there must be an *annihilating polynomial* of degree at most n^2, i.e. a polynomial p such that $p(L) = 0$. Among the annihilating polynomials for L, there must be some monic[9] annihilating polynomial $p_{\min}$ of minimal degree m. We call $p_{\min}$ the *minimal polynomial*.

If p is any annihilating polynomial, polynomial division lets us write

$$p(z) = q(z)p_{\min}(z) + r(z), \quad \deg(r) < m.$$

Because $p_{\min}(L) = 0$, we have

$$p(L) = q(L)p_{\min}(L) + r(L) = r(L),$$

and therefore $r = 0$; otherwise r would be an annihilating polynomial of lower degree than m, contradicting the minimality. Therefore any annihilating polynomial can be written as $q(z)p_{\min}(z)$ for some other polynomial $q(z)$. As a direct consequence, the minimal polynomial is unique. Also as a consequence, the minimal polynomial of $L|_{\mathcal{U}}$ for any subspace $\mathcal{U}$ divides $p_{\min}$ for L, since $p_{\min}$ for L is an annihilating polynomial for $L|_{\mathcal{U}}$.

We can write the minimal polynomial in factored form (over $\mathbb{C}$) as

$$p_{\min}(z) = \prod_{j=1}^{s} (z - \lambda_j)^{d_j}.$$

[9] A monic polynomial is one in which the coefficient in front of the highest-degree term is 1.

Each term $L - \lambda_j I$ must be singular; otherwise, we would have $(L - \lambda_j I)^{-1} p_{\min}(L) = 0$, contradicting minimality.

The λ_j are the eigenvalues of L, and the null spaces of each $L - \lambda_j I$ are comprised of eigenvectors. If $d_j = 1$ for each eigenvalue, then there is a basis of eigenvectors; otherwise, we have to consider generalized eigenvectors as well to get a basis. In either case, the space $\mathcal{V}$ can be decomposed into a direct sum of invariant subspaces $\mathcal{N}\left((L - \lambda_j I)^{d_j}\right)$; these are the *maximal invariant subspaces* associated with the eigenvalues.

If we let $m_j = \dim \mathcal{N}\left((L - \lambda_j I)^{d_j}\right)$, we can also define the *characteristic polynomial*

$$p_{\text{char}}(z) = \prod_{j=1}^{s} (z - \lambda_j)^{m_j}.$$

The dimension m_j is the *algebraic multiplicity* of the eigenvalue λ_j. The *geometric multiplicity* is the dimension of $\mathcal{N}(L - \lambda_j I)$, i.e. the number of independent eigenvectors for λ_j.

The characteristic polynomial is often also written as

$$p_{\text{char}}(z) = \det(zI - L),$$

but writing the characteristic polynomial in terms of determinants has the disadvantage that we need an appropriate definition of determinants first!

4.3.2 Canonical forms

As in the case of maps between different spaces, we are interested in canonical matrix representations for operators $L \in L(\mathcal{V}, \mathcal{V})$, i.e. choices of bases that make the matrix representation "as simple as possible." We consider two such forms: one involving an arbitrary basis, and the other involving an orthonormal basis.

4.3.2.1 Jordan form

As we noted above, the space $\mathcal{V}$ can be written as

$$\mathcal{V} = \mathcal{V}_1 \oplus \mathcal{V}_2 \oplus ... \oplus \mathcal{V}_s,$$

where $\mathcal{V}_s$ is the maximal invariant subspace associated with the eigenvalue λ_s, i.e.

$$\mathcal{V}_j = \mathcal{N}((L - \lambda_j I)^{d_j}).$$

With respect to this partitioning of $\mathcal{V}$, we have the block diagonal quasimatrix representation

$$\begin{bmatrix} L_{11} & & & \\ & L_{22} & & \\ & & \ddots & \\ & & & L_{ss} \end{bmatrix}$$

where L_{jj} represents the restriction $L|_{\mathcal{V}_j}$. With a choice of bases for each of the $\mathcal{V}_j$, this leads to a block diagonal matrix representation.

When the minimal polynomial exponent d_j is 1, we have that $\mathcal{V}_j = \mathcal{N}(L - \lambda_j I)$, and so

$$L|_{\mathcal{V}_j} = \lambda_j I.$$

Hence, if there are no repeated zeros of the minimal polynomial, we can choose bases V_j for each subspace $\mathcal{V}_j$, and each will satisfy

$$LV_j = V_j(\lambda_j I).$$

Putting these together, we have the concatenated basis

$$V = \begin{bmatrix} V_1 & V_2 & \dots & V_s \end{bmatrix}$$

satisfying the relationships

$$LV = V\Lambda$$

where Λ is a diagonal matrix in which each eigenvalue λ_j appears $\dim \mathcal{V}_j$ times in a row. With respect to the eigenvector basis V, we therefore have the canonical matrix representation

$$\Lambda = V^{-1}LV.$$

Hence, such operators are *diagonalizable*.

When the minimal polynomial has zeros with multiplicity greater than one, the matrix is not diagonalizable. In this case, we cannot form a basis of eigenvectors, but we can form a basis if we include *generalized* eigenvectors. In the *Jordan canonical form*, we find bases consisting of *Jordan chains*, i.e. vectors satisfying

$$\begin{aligned} (L - \lambda_j I)u_1 &= 0 \\ (L - \lambda_j I)u_k &= u_{k-1}, \quad 1 < k \leq b. \end{aligned}$$

Rewriting the second expression as $Lu_k = u_{k-1} + \lambda_j u_k$ and defining the quasimatrix $U = \begin{bmatrix} u_1 & \dots & u_b \end{bmatrix}$, we can summarize with the matrix equation

$$LU = UJ_{b\times b}(\lambda)$$

where

$$J_{b\times b}(\lambda) \begin{bmatrix} \lambda_j & 1 & & & \\ & \lambda_j & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda_j & 1 \\ & & & & \lambda_j \end{bmatrix}$$

is a b-dimensional *Jordan block*. The maximum size of any Jordan block is equal to the multiplicity d_j of λ_j in the minimal polynomial; the number of Jordan blocks for λ_j is equal to

the geometric multiplicity $\dim \mathcal{N}(L-\lambda_j I)$; and the sum of the sizes of the Jordan blocks is equal to the algebraic multiplicity $\dim \mathcal{N}((L-\lambda_j I)^{d_j})$. Taking a basis formed from concatenating such Jordan chains for each space, we have the *Jordan canonical form*, a block diagonal matrix in which each diagonal block is a Jordan block.

The Jordan form is fragile: almost all operators are diagonalizable and have distinct eigenvalues, and infinitesimally small changes will usually completely destroy any nontrival Jordan blocks. Moreover, diagonalizable operators that are *close* to operators with nontrivial Jordan blocks may have eigenvector bases, but those eigenvector bases are not all that nice. We therefore would like an alternate canonical form that gives up some of the goal of being "as diagonal as possible."

4.3.2.2 Schur form

In our discussion of the Jordan form, we started with the decomposition of $\mathcal{V}$ into maximal invariant subspaces associated with each eigenvalue. For the Schur form, we decompose $\mathcal{V}$ in terms of *nested* invariant subspaces:

$$\mathcal{W}_k = \oplus_{j=1}^{k} \mathcal{V}_j.$$

Assuming $\mathcal{V}$ is an inner product space, we can also write

$$\mathcal{W}_k = \oplus_{j=1}^{k} \mathcal{U}_j$$

where $\mathcal{U}_1 = \mathcal{V}_1$ and otherwise $\mathcal{U}_j$ is the orthogonal complement of $\mathcal{W}_{j-1}$ in $\mathcal{W}_j$. With respect to the decomposition $\mathcal{V} = \oplus_{j=1}^{s} \mathcal{U}_j$, we have the block upper triangular quasimatrix representation

$$\begin{bmatrix} L_{11} & L_{12} & \dots & L_{1s} \\ & L_{22} & \dots & L_{2s} \\ & & \ddots & \vdots \\ & & & L_{ss} \end{bmatrix}$$

With an appropriate choice of orthonormal basis U_j for each $\mathcal{U}_j$ and setting $U = \begin{bmatrix} U_1 & U_2 & \dots & U_s \end{bmatrix}$, we have

$$U^* L U = T$$

where T is an upper triangular matrix for which the eigenvalues are on the diagonal. This is a (complex) *Schur canonical form.*

The complex Schur form may in general involve complex choices of the basis U and a complex matrix representation T, even if $\mathcal{V}$ is most naturally treated as a vector space over the reals. However, there is a *real* Schur form which involves a real orthonormal basis and a block upper triangular matrix T where the diagonal blocks are 1-by-1 (for real eigenvalues) or 2-by-2 (for conjugate pairs of eigenvalues).

4.3.3 Similar matrices

So far, we have focused on matrices as representations of operators on abstract vector spaces, where we choose one basis V for the space $\mathcal{V}$:

$$A = V^{-1}LV.$$

If we consider an alternate basis $V' = VX$ for an invertible matrix X, we have the associated matrix representation

$$A' = V'^{-1}LV' = (V^{-1}V')^{-1}A(V^{-1}V') = X^{-1}AX.$$

The transformation $A \mapsto X^{-1}AX$ is a *similarity transformation*, and the matrices A and A' are said to be *similar*.

The properties we have discussed so far (eigenvalues, the minimal and characteristic polynomial, the Jordan form) are fundamentally properties of operators, and do not depend on the basis in which those operators are expressed. As much as possible, we try to make this clear by giving basis-free explanations when possible. But it is also possible to give matrix-based arguments in which we argue for invariance under similarity transformations. For example, if $\widehat{A} = X^{-1}AX$, then

$$\widehat{A}^j = (X^{-1}AX)^j = X^{-1}A^jX$$

and, more generally, $p(\widehat{A}) = X^{-1}p(A)X$. Therefore, any annihilating polynomial for $p(A)$ is also an annihilating polynomial for $\widehat{A}$; and the minimal and characteristic polynomials for A will similarly be the minimal and characteristic polynomials for $\widehat{A}$.

4.3.4 Trace

We typically describe the *trace* in terms of a matrix representation: the trace of a matrix is the sum of its diagonal entries, i.e.

$$\operatorname{tr}(A) = \sum_{i=1}^{n} a_{ii}.$$

The trace satisfies the *cyclic* property that for any $B \in \mathbb{F}^{m\times n}$ and $C \in \mathbb{F}^{n\times m}$ we have

$$\begin{aligned}\operatorname{tr}(AB) &= \sum_{i=1}^{m}\sum_{j=1}^{n} b_{ij}c_{ji} \\ &= \sum_{j=1}^{n}\sum_{i=1}^{m} c_{ji}b_{ij} \\ &= \operatorname{tr}(BA).\end{aligned}$$

Using this property, for any invertible $X \in \mathbb{F}^{n\times n}$ we have

$$\operatorname{tr}(X^{-1}AX) = \operatorname{tr}(AXX^{-1}) = \operatorname{tr}(A).$$

That is, the trace is *invariant under similarity*, and is thus a property of the underlyling linear operator and not just of a particular matrix representation. In particular, if $X^{-1}AX = J$ is a Jordan canonical form, we can see that the trace is the sum of the eigenvalues (counting geometric multiplicity).

4.3.5 Frobenius inner product

The trace is also useful for defining other structures on spaces of linear maps in a basis-free way. Let $L \in L(\mathcal{V}, \mathcal{U})$ be a linear map between two inner product spaces, and let $L = U\Sigma V^*$ be the singular value decomposition. Then

$$\operatorname{tr}(L^*L) = \operatorname{tr}(V\Sigma^2 V^*) = \operatorname{tr}(\Sigma^2 V^* V) = \operatorname{tr}(\Sigma^2) = \sum_{j=1}^{n} \sigma_j^2 = \|L\|_F^2.$$

where $\|L\|_F^2$ is the squared Frobenius norm. Also, if B is any matrix representation of L with respect to orthonormal bases, then we have

$$\|L\|_F^2 = \|B\|_F^2 = \operatorname{tr}(B^*B) = \sum_{i,j} |b_{ij}|^2,$$

so it is very simple to compute the Frobenius norm of L given any matrix representation with respect to orthonormal bases.

More generally, the trace can be used to define the *Frobenius inner product* on the space $L(\mathcal{V}, \mathcal{W})$ via

$$\langle L, M \rangle_F = \operatorname{tr}(M^*L).$$

The Frobenius norm is the standard Euclidean norm associated with the Frobenius inner product.

4.3.6 Hermitian and skew

If $L \in L(\mathcal{V}, \mathcal{V})$, then

$$L = H + S, \quad H = \frac{1}{2}(L + L^*), \quad S = \frac{1}{2}(L - L^*).$$

The operators H and S are known as the Hermitian and skew-Hermitian parts of L. We note that and $\langle H, S \rangle_F = 0$, so by the Pythagorean theorem,

$$\|L\|_F^2 = \|H\|_F^2 + \|S\|_F^2.$$

More generally, $L(\mathcal{V}, \mathcal{V})$ is a direct sum of Hermitian and skew-Hermitian subspaces, and these spaces are orthogonal complements of each other.

Any operator L has a Schur canonical form

$$U^*LU = T$$

where T is an upper triangular matrix. For a Hermitian matrix H, we have

$$T = U^*HU = (U^*HU)^* = T^*,$$

which implies that T must be diagonal (since the subdiagonal elements of T are zero and the superdiagonal elements of T^* are zero), and that the diagonal part of T must be real (since $t_{ii} = t_{ii}^*$). Similarly, for a skew-Hermitian matrix S, we have

$$T = U^*SU = -(U^*SU)^* = -T^*,$$

which simplies that in this case T is diagonal and the diagonal part is imaginary. Hence, the decomposition of a matrix into Hermitian and skew-Hermitian parts in many ways mirrors the decomposition of a complex number into real and imaginary parts.

There is an even stronger number between the Hermitian / skew-Hermitian decomposition and the complex numbers for the case of $\mathbb{R}^{2\times 2}$. Let $g : \mathbb{C} \to \mathbb{R}^{2\times 2}$ be the map

$$g(a + bi) = aI + bJ, \quad J = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

Then

$$\begin{aligned} g(0) &= 0 \\ g(1) &= I \\ g(z + w) &= g(z) + g(w) \\ g(zw) &= g(z)g(w) \\ g(z^{-1}) &= g(z)^{-1}. \end{aligned}$$

That is, the complex matrices are isomorphic to the subset of the 2-by-2 real matrix where the symmetric part is proportional to the identity.

4.4 Hermitian and quadratic forms

A sesquilinear form $a : \mathcal{V} \times \mathcal{V} \to \mathbb{F}$ is *Hermitian* if $a(v, w) = \overline{a(w, v)}$.
If $a(v, w) = -\overline{a(w, v)}$, the form is called *skew-Hermitian*. Any sesquilinear form on $\mathcal{V} \times \mathcal{V}$ can be decomposed into Hermitian and skew-Hermitian parts:

$$\begin{aligned} a(v, w) &= a^H(v, w) + a^S(v, w) \\ a^H(v, w) &= \frac{1}{2}\left(a(v, w) + \overline{a(w, v)}\right) \\ a^S(v, w) &= \frac{1}{2}\left(a(v, w) - \overline{a(w, v)}\right). \end{aligned}$$

We use the terms Hermitian and skew-Hermitian generically to also refer to symmetric and skew-symmetric forms on real vector spaces.

If $\mathcal{V}$ is an inner product space, then for any sesquilinear form $a : \mathcal{V} \times \mathcal{V} \to \mathbb{F}$, there is an associated operator $L : \mathcal{V} \to \mathcal{V}$ such that

$$a(v, w) = \langle Lv, w \rangle.$$

When a is Hermitian, we have

$$\langle Lv, w \rangle = a(v, w) = \overline{a(w, v)} = \langle v, Lw \rangle,$$

i.e. $L = L^*$ is a self-adjoint operator. A similar argument says that if a is skew-Hermitian, the corresponding operator will satisfy $L = -L^*$. Hence, the study of Hermitian forms is closely linked to the study of self-adjoint operators.

A *quadratic form* is a function $\phi : \mathcal{V} \to \mathbb{F}$ that is homogeneous of degree 2 (i.e. $\phi(\alpha v) = \alpha^2 \phi(v)$) and such that $(u, v) \mapsto \phi(u+v) - \phi(u) - \phi(v)$ is bilinear. The associated bilinear form is always symmetric. Hence, on real vector spaces the study of Hermitian forms is also closely linked to the study of quadratic forms.

4.4.1 Matrices

Given a basis V for $\mathcal{V}$, the standard matrix representation of a sesquilinear form on $\mathcal{V} \times \mathcal{V}$ is a square matrix with entries $a_{ij} = a(v_j, v_i)$ so that

$$a(Vc, Vd) = d^* A c.$$

Decomposing the form into Hermitian and skew-Hermitian parts corresponds to decomposing $A = H + S$ where $H = H^*$ and $S = -S^*$ are the Hermitian and skew-Hermitian parts of the matrix.

A quadratic form ϕ has an associated symmetric bilinear form

$$a(u, v) = \frac{1}{2} \left(\phi(u + v) - \phi(u) - \phi(v) \right).$$

The matrix representation for a quadratic form is

$$\phi(Vc) = c^T A c$$

where the entries of A are $a_{ij} = a(v_i, v_j)$. Conversely, for any real bilinear form, a, we have that $\phi(v) := a(v, v)$ is a quadratic form, and $\phi(Vc) = c^T H c$ where H is the matrix for the symmetric part of a.

Let A be a matrix representing a sesquilinear form $a : \mathcal{V} \times \mathcal{V} \to \mathcal{F}$ with respect to a basis V, i.e.

$$a(Vc, Vd) = d^* A v.$$

We can write any other basis as $\hat{V} = VX$ for some nonsingular X; with respect to the $\hat{V}$ basis, we have

$$a(\hat{V}c\hat{V}d) = a(VXc, VXd) = d^*(X^*AX)c.$$

For any nonsingular X, we say A and $\hat{A} = X^*AX$ are *congruent*. Just as similar matrices represent the same operator under different bases, congruent matrices represent the same sesquilinear form under different bases. When we restrict our attention to orthonormal bases, the matrix X will be unitary; we note that $X^* = X^{-1}$ for unitary X, so a unitary congruence and a unitary similarity are the same thing.

4.4.2 Canonical forms

In the case of a general basis, there exists is a basis such that the matrix representation of a Hermitian form is

$$\begin{bmatrix} I_{\nu_+} & & \\ & 0_{\nu_0} & \\ & & -I_{\nu_-} \end{bmatrix}$$

The triple (ν_+, ν_0, ν_-) is the *inertia* of the form. Inertia, like rank, is a property of the linear algebraic object (in this case the Hermitian form) rather than its representation in any specific basis. Let $n = \dim(\mathcal{V})$; then we classify the form based on its inertia as:

- Positive definite if $\nu = (n, 0, 0)$
- Positive semidefinite if $\nu = (r, n-r, 0)$
- Negative definite if $\nu = (0, 0, n)$
- Negative semidefinite if $\nu(0, n-r, r)$
- Strongly indefinite if $\nu_+ > 0$ and $\nu_- > 0$.

A Hermitian positive definite form is an inner product.

If $\mathcal{V}$ is an inner product space and we restrict ourselves to orthonormal bases, we have a diagonal matrix Λ with ν_+ positive values, ν_0 zeros, and ν_- negative values. If we identify $a(v, w)$ with $\langle Lv, w\rangle$, then this canonical form gives that $a(v_i, v_j) = \langle Lv_i, v_j\rangle = \lambda_i \delta_{ij}$. The Hermitian property implies that $a(v_i, v_i) = \overline{a(v_i, v_i)} = \lambda$, so λ must be real even when the form is over $\mathbb{C}$. Given the orthonormality of the basis, this means

$$Lv_j = v_j \lambda_j,$$

i.e. the canonical decomposition of the Hermitian form is really the same as the eigenvalue decomposition of the associated self-adjoint operator L.

The Hermitian eigenvalue problem has an enormous amount of structure that comes from the fact that it can be interpreted *either* in terms of a Hermitian form or the associated quadratic *or* in terms of an operator.

4.4.3 A derivative example

As before, it is useful to consider a specific example that does not involve a concrete vector space. We will use the real symmetric form on $\mathcal{P}_d$ given by

$$a(p,q) = \left\langle \frac{dp}{dx}, \frac{dq}{dx} \right\rangle$$

using the $L^2([-1,1])$ inner product between polynomials. Integration by parts implies that

$$\begin{aligned} a(p,q) &= \int_{-1}^{1} \frac{dp}{dx}\frac{dq}{dx}\,dx \\ &= \frac{dp}{dx} q \Big|_{-1}^{1} - \int_{-1}^{1} \frac{d^2p}{dx^2} q\,dx \\ &= \left\langle \delta_1\delta_1^* \frac{dp}{dx} - \delta_{-1}\delta_{-1}^* \frac{dp}{dx} - \frac{d^2p}{dx^2}, q \right\rangle, \end{aligned}$$

where δ_1^* and δ_{-1}^* are the evaluation functionals at ± 1 and δ_1 and δ_{-1} are the associated vectors under the Riesz map. Hence, the self-adjoint linear map associated with the form is

$$L = \delta_1\delta_1^* \frac{d}{dx} - \delta_{-1}\delta_{-1}^* \frac{d}{dx} - \frac{d^2}{dx^2}.$$

We can check ourselves by implementing this relation in code

```
let
    # Implement the L map
    d = 2
    δp1 = dualL2_δx( 1.0, d)
    δn1 = dualL2_δx(-1.0, d)
    function L(p)
        Dp = derivative(p)
        D2p = derivative(Dp)
        δp1*Dp(1) - δn1*Dp(-1) - D2p
    end

    # Test polynomials p and q
    p = Polynomial([2.0; -3.0; 1.0])
    q = Polynomial([1.0; 1.0; 1.0])

    # Test agreement with the form and self-adjointness
    dotL2(derivative(p), derivative(q)) ≈ dotL2(L(p), q),
    dotL2(L(p), q) ≈ dotL2(p, L(q))
end
```

```
(true, true)
```

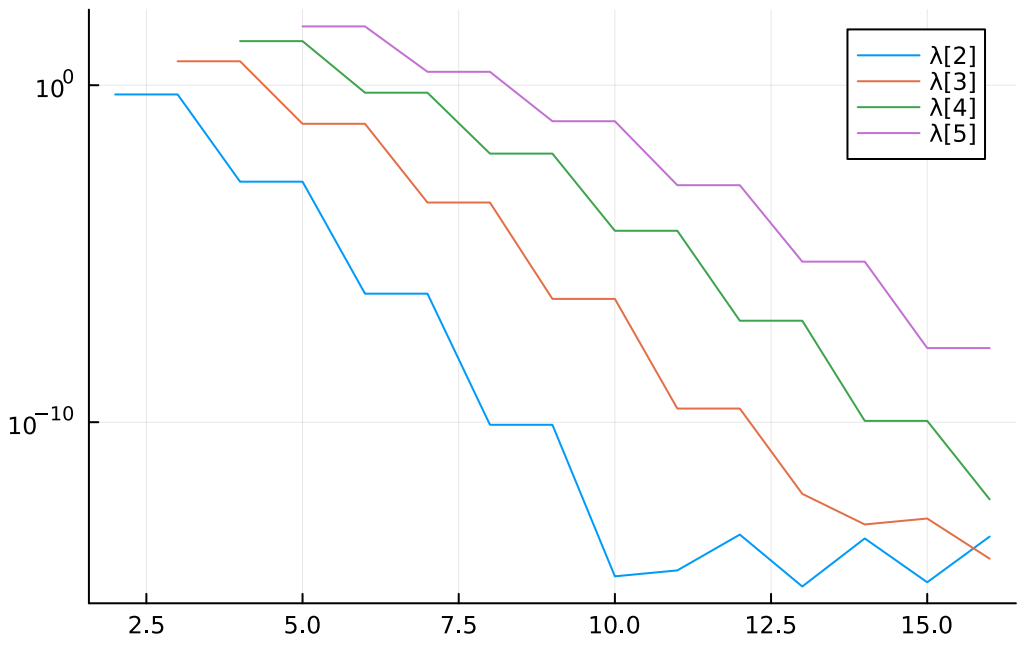
The form is positive semidefinite, since $a(1,1) = 0$ but a cannot ever by negative (by positive definiteness of the inner product); the inertia is $(d, 1, 0)$. While we can compute the eigenvalues in terms of a matrix derived from L, it is simpler to compute the matrix directly from the form:

```
let
    d = 3

    # Compute the matrix in terms of the normalized Legendre polynomials
    P̄ = normalized_legendre_basis(d)
    AP̄ = gram(dotL2, derivative.(P̄))

    # Compute the eigendecomposition
    F = eigen(AP̄)
    U = P̄*F.vectors

    # Check orthonormality of the U basis,
    # and that the associated matrix is Λ
    gram(dotL2, U) ≈ I,
    gram(dotL2, derivative.(U)) ≈ Diagonal(F.values)
end
```

```
(true, true)
```

Similar to what we saw in our discussion of the singular value decomposition, the kth nonzero eigenvalue converges as a function of d to a limiting value of $(k\pi/2)^2$. We plot the convergence of the first few eigenvalues below:

```
let
    d = 15
    P̄ = normalized_legendre_basis(d)
    AP̄ = gram(dotL2, derivative.(P̄))

    p = plot()
    for j=2:5
        λs = [eigvals(AP̄[1:n,1:n])[j] for n=j:d+1]
        plot!(j:d+1, abs.(λs.-((j-1)*π/2)^2),
              yscale=:log10, label="λ[$j]")
    end
    p
end
```



4.5 Tensors and determinants

So far, we have dealt with linear, bilinear, sesquilinear, and quadratic functions on vector spaces. Going beyond this to functions that are linear in several arguments takes us into the land of *tensors*.

4.5.1 Takes on tensors

There are several ways to approach tensors. In much of numerical analysis, only tensors of concrete spaces are considered, and tensors are treated as arrays of multi-dimensional arrays of numbers (with matrices as a special case where there are two dimensions used for indexing). For example, a tensor with three indices is represented by an element a_{ijk} of $\mathbb{F}^{m\times n\times p}$. We will see this perspective in detail later, both in our discussion of numerical linear algebra and in our discussion of tensors in latent factor models.

Here, though, we will focus on tensors as basic objects of *multilinear* algebra. There are still several ways to approach this:

- We can consider tensors purely in terms of bases, along with change-of-basis rules. This view, common in some corners of physics, says that "a tensor is an object that transforms like a tensor," a perspective similar to "a vector is an object that transforms like a vector"

that we mentioned earlier. We will prefer a discussion that gives changes of basis as a consequence of a more fundamental definition, rather than as the definition itself.

- We can consider tensors from an algebraic perspective as a quotient space: the *tensor product* space $\mathcal{V} \otimes \mathcal{W}$ is a vector space spanned by elements of the Cartesian product $\mathcal{V} \times \mathcal{W}$, modulo the equivalence relations

$$\begin{aligned} \alpha(v, w) &\sim (\alpha v, w) \sim (v, \alpha w) \\ (u + v, w) &\sim (u, w) + (v, w) \\ (u, v + w) &\sim (u, v) + (u, w) \end{aligned}$$

  We write $v \otimes w$ to denote the equivalence class associated with (v, w). While this definition has the advantage of being very general (it can generalize to structures other than vector spaces) and it does not depending on a basis, it can be a bit overly abstract.

- We will mostly consider tensors as *multilinear forms*, generalizing the notion of bilinear forms discussed above. This has the advantage of extending our discussion of bilinear and sesquilinear forms from earlier, remaining abstracted away from a specific basis while not too abstract.

4.5.2 Multilinear forms

Let $\mathcal{V}$ and $\mathcal{U}$ be two vector spaces. For any $u \in \mathcal{U}$ and $w^* \in \mathcal{V}^*$, the *outer product* uw^* is associated with a rank 1 map in $L(\mathcal{V}, \mathcal{U})$ given by $v \mapsto uw^*v$. Alternately, we can see uw^* as a bilinear map from $\mathcal{U}^* \times \mathcal{V} \to \mathbb{F}$ given by $(f^*, v) \mapsto f^*uw^*v$; or we can see uw^* as a dual map in $L(\mathcal{U}^*, \mathcal{V}^*)$ given by $f^* \mapsto f^*uw^*$. We will use the uniform notation $u \otimes w^*$ to cover all of these possible interpretations of the outer product.

More generally, let $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_d$ be vector spaces. Then for any vectors $v_1, v_2, \dots, v_d$ in these spaces, the tensor product $v_1 \otimes v_2 \otimes \dots \otimes v_d$ can be interpreted as the multilinear map from $\mathcal{V}_1^* \times \mathcal{V}_2^* \times \dots \mathcal{V}_d^* \to \mathbb{F}$ given by

$$(v_1 \otimes v_2 \otimes \dots \otimes v_d)(w_1^*, w_2^*, \dots, w_d^*) = (w_1^* v_1)(w_2^* v_d) \dots (w_d^* v_d).$$

As in the case of just two vector spaces, we can define other maps by partial evaluation. For example, we can consider this as a map from $\mathcal{V}_2^* \times \dots \times \mathcal{V}_d^* \to \mathcal{V}_1$ by

$$(w_d^*, \dots, w_d^*) \mapsto a(\cdot, w_2^*, \dots, w_d^*)$$

where "filling in" all but the first argument gives an element of $\mathcal{V}_1$ (or technically of $\mathcal{V}_1^{**}$, but this is isomorphic[10] to $\mathcal{V}_1$).

[10] There is a canonical injection $\mathcal{V} \to \mathcal{V}^{**}$ via $v \mapsto (w^* \mapsto w^*v)$. In finite-dimensional spaces, this is alway also surjective; in infinite-dimensional spaces, it is only surjective when the space is *reflexive*.

The vector space of all multilinear forms $\mathcal{V}_1^* \times ... \mathcal{V}_d^* \to \mathbb{F}$ is written as $\mathcal{V}_1 \otimes ... \otimes \mathcal{V}_d$. If $\mathcal{V}_j$ has a basis $V^{(j)}$ and associated dual basis $(W^{(j)})^*$, then an arbitrary element $a \in \mathcal{V}_1 \otimes ... \otimes \mathcal{V}_d$ can be written uniquely as

$$a = \sum_{i_1,...,i_d} a_{i_1 ... i_d} \, v_{i_1}^{(1)} \otimes ... \otimes v_{i_d}^{(d)}.$$

where

$$a_{i_1 ... i_d} = a(w_{i_1}^{(1)*}, ... \, w_{i_d}^{(d)*}).$$

That is, the set of all tensor products of basis vectors for the spaces $\mathcal{V}_j$ form a basis for the tensor product space $\mathcal{V}_1 \otimes ... \otimes \mathcal{V}_d$.

In terms of the partial evaluation operation discussed above, we might consider a as a map from $\mathcal{V}_2^* \otimes ... \otimes \mathcal{V}_d^*$ to $\mathcal{V}_1$ by taking

$$\begin{aligned} b &= \sum_{i_2,...,i_d} b_{i_2 ... i_d} \, w_{i_2}^{(2)*} \otimes w_{i_d}^{(d)*} \\ &\mapsto \sum_{i_2,...,i_d} b_{i_2 ... i_d} \, a(\cdot, w_{i_2}^{(2)*}, ... \, w_{i_d}^{(d)*}) \\ &= \sum_{i_1,i_2,...,i_d} v_{i_1}^{(1)} \, a_{i_1 ... i_d} b_{i_2 ... i_d}. \end{aligned}$$

This generalized partial evaluation operation is known as a *tensor contraction*. In general, we can contract two tensors on any pairs of "slots" in the form that take vectors from a dual pair of spaces.

When a real inner product space $\mathcal{V}$ appears in a tensor, we can construct a new tensor in which $\mathcal{V}^*$ appears in the same slot by composing that slot with the inverse Riesz map ("lowering the index" in the physics parlance). Similarly, we can convert a $\mathcal{V}^*$ slot to a $\mathcal{V}$ slot via the Riesz map ("raising the index"). When a tensor is represented with respect to orthonormal bases, raising or lowering operations do not change any of the coefficients.

It is tempting to ask if the coefficients might be made simple by a special choices of the bases for the component spaces, analogous to the anonical forms we discussed before. Unfortunately, tensors beyond bilinear forms lack any nice canonical representation.

4.5.3 Polynomial products

As an example of tensor product spaces in action, we consider the space of polynomials in three variables with maximum degree per variable of d. In the power basis, we would write this as

$$\mathcal{P}_d \otimes \mathcal{P}_d \otimes \mathcal{P}_d = \left\{ \sum_{i,j,k} c_{ijk} x^i y^j z^k : c_{ijk} \in \mathbb{R}^{d \times d \times d} \right\}.$$

Evaluating a polynomial $p \in \mathcal{P}_d \otimes \mathcal{P}_d \otimes \mathcal{P}_d$ at a particular point (α, β, γ) is associated with evaluation of a trilinear form, or equivalently with contraction against a tensor in $\mathcal{P}_d^* \otimes \mathcal{P}_d^* \otimes \mathcal{P}_d^*$; i.e.

$$p \cdot (\delta_\alpha^* \otimes \delta_\beta^* \otimes \delta_\gamma^*)$$

We can similarly think about contracting in some slots but not others, or contracting against other linear functionals (like integration).

For concrete tensors, contractions can also be used to express operations like changing basis. For example, if $T_{0:d}A = X_{0:d}$, we can write

$$\begin{aligned} p(x,y,z) &= \sum_{i,j,k} c_{ijk}\, x^i y^j z^k \\ &= \sum_{l,m,n} d_{lmn}\, T_l(x) T_m(y) T_n(z) \\ d_{lmn} &= \sum_{i,j,k} c_{ijk} a_{li} a_{jm} a_{kn} \end{aligned}$$

That is, we change bases by applying A on each "fiber" of the coefficient tensor (i.e. a vector associated with holding two indices fixed and letting the other vary). Indeed, the transformation from the c_{ijk} to the d_{lmn} coefficients is most easily coded in terms of such a matrix operation:

```
let
    d = 2
    C = rand(d+1,d+1,d+1)  # Power coeffs

    # Transform to the Chebyshev coefficients
    D = copy(C)
    M = chebyshev_power_coeffs(d)  # A = inv(M)

    # Contract with a_li
    for j=1:d+1
        for k = 1:d+1
            D[:,j,k] = M\D[:,j,k]
        end
    end

    # Contract with a_mj
    for i=1:d+1
        for k=1:d+1
            D[i,:,k] = M\D[i,:,k]
        end
    end
```

```
    # Contract with a_nk
    for i=1:d+1
        for j=1:d+1
            D[i,j,:] = M\D[i,j,:]
        end
    end

    # Check correctness by comparing evaluations a random point
    T = chebyshev_basis(d)
    x,y,z = rand(3)
    p1 = 0.0
    p2 = 0.0
    for i=1:d+1
        for j=1:d+1
            for k=1:d+1
                p1 += C[i,j,k] * x^(i-1) * y^(j-1) * z^(k-1)
                p2 += D[i,j,k] * T[i](x) * T[j](y) * T[k](z)
            end
        end
    end
    p1 ≈ p2
end
```

```
true
```

4.5.4 Alternating forms

An *alternating form* is a multilinear form on several copies of the same vector space $\mathcal{V}$ with the property that it is zero if any input is repeated. This implies that swapping the order of any pair of arguments reverses the sign; for example, if $a : \mathcal{V} \times \mathcal{V} \times \mathcal{V} \to \mathbb{F}$ is a multilinear form, then

$$\begin{aligned} a(u,v,w) + a(w,v,u) &= a(u,v,w) + a(u,v,u) + a(w,v,w) + a(w,v,u) \\ &= a(u,v,w+u) + a(w,v,w+u) \\ &= a(u+w,v,w+u) \\ &= 0. \end{aligned}$$

The set of all alternating forms on k copies of $\mathcal{V}$ is written as $\mathcal{V}^* \wedge ... \wedge \mathcal{V}^*$, a subspace of $\mathcal{V}^* \otimes ... \otimes \mathcal{V}^*$.

The *wedge product* of two tensors over copies of the same space is

$$f \wedge g = f \otimes g - g \otimes f.$$

Alternating forms of a given number of arguments also form a vector space. For example, $\mathcal{V}^* \wedge \mathcal{V}^* \wedge \mathcal{V}^*$ has a basis of vectors $w_i^* \wedge w_j^* \wedge w_k^*$ for each possible subset $\{i, j, k\}$ of three distinct indices from the range $1, \ldots, n$. Hence, it has dimension n choose 3.

If $\mathcal{V}$ is n-dimensional, the space of n copies of $\mathcal{V}^*$ wedged together has n choose n dimensions — that is, there is only one alternating form on n inputs, up to scaling. We arbitrarily choose one such form by picking a basis V and letting $a(v_1, v_2, \ldots, v_n) = 1$ and declaring this to be the *signed volume* form. In most situations, $\mathcal{V}$ is an inner product space and we choose V to be an orthonormal basis.

4.5.5 Determinants

Let $f \in \mathcal{V}^* \wedge \ldots \wedge \mathcal{V}^*$ be the signed volume form for a space $\mathcal{V}$. Rather than writing evaluation of f as $f(v_1, v_2, \ldots, v_n)$, we will collect the arguments into a quasimatrix and write $f(V)$ for the volume associated with a parallelipiped whose edges are vectors V.

If U is a basis such that $f(U) = 1$, then we can write any other set of n vectors as UA for some matrix A. The *determinant* of A is the function such that $f(UA) = f(U)\det(A) = \det(A)$. Put differently, the determinant represents the change of volume in a mapping $L = UAU^{-1}$ represented with respect to a basis quasimatrix U corresponding to a parallelipiped of unit (signed) volume. In order to satisfy this definition, the determinant must be

- An alternating form on the concrete space $\mathbb{F}^n \otimes \ldots \otimes \mathbb{F}^n$ (written in terms of matrices $\mathbb{F}^{n \times n}$)
- Equal to one for the identity matrix.

The interpretation of the determinant as representing a scaling of (signed) volumes explains various other properties, such as the important fact that $\det(AB) = \det(A)\det(B)$. It also explains why determinants arise in the change-of-variables formula for integration, which is one of the main places where we will see them.

The determinant of a singular matrix must be zero, which explains why we can write the characteristic polynomial for a matrix as $\det(zI - A)$. However, we prefer to treat both determinants and characteristic polynomials as interesting objects in their own right, and the connection through the determinant as a derived fact.

5 Calculus and analysis

We assume the reader has standard introductory courses in single-variable and multivariable calculus. It will also be helpful to have seen some elements of analysis, including the $\epsilon - \delta$ definition of continuity. Otherwise, it will be helpful to be familiar with the Julia programming language (Chapter 2) and with the contents of the previous chapter (Chapter 4).

5.1 Formulas and foundations

The modern conception of "calculus" grew out of the 17th century work of Newton and Liebnitz. Many elements of calculus preceded Newton and Liebnitz; indeed, some arguments with a flavor of calculus were known to Archimedes and other Greek geometers. But Newton and Liebnitz created something new by producing "the *Calculus*, a general symbolic and systematic method of analytic operations, to be performed by strictly formal rules, independent of geometric meaning" (p. 84, Rosenthal 1951). The mechanical nature of calculus means that students and computers can routinely manipulate derivatives and integrals without constantly (or ever) having to think deeply about the geometry of the problem at hand. Indeed, the system of calculus has been generalized in many ways to situations that bear little resemblance to geometric problems at all, but still follow the formal patterns that make the machinery work.

Formal calculations are power tools: easily dispatching intractable problems in the right hands, but dangerous when used incautiously. This problem is not unique to calculus: the road to "proofs" that $1 = 0$ is lined with use of the identity $xy/x = y$ without checking if $x = 0$. Such cautionary examples aside, many formal calculations produce valid results even when they have no right to do so, leaving mathematicians the challenge of figuring out what the "right" set of hypotheses really are.

The development of the calculus was a product of the seventeenth century, but the early formal system was somewhat unsound: the rules were missing important hypotheses, and could potentially lead to incorrect conclusions. Though concerns about the solidity of the foundations of calculus go back to the time of Newton and Liebnitz, many mathematicians were perfectly happy to restrict their attention to functions "nice enough" so that the rules of the calculus worked and get on with it[1]. The project to develop definitions and hypotheses to make the formal rules of calculus *sound* did not really get fully underway until the nineteenth century,

[1] "I turn away with fright and horror from this lamentable evil of functions which do not have derivatives." – Hermite (according to Kline 1990, 973)

with the work of Cauchy and even more of Weierstrass, the "father of modern analysis" and originator of the $\epsilon - \delta$ definitions and arguments familiar to modern students of calculus and analysis.

The $\epsilon - \delta$ world of Weierstrass has an asymptotic flavor: for a given $\epsilon > 0$, we *eventually* get within ϵ of a target, for small enough δ for for large enough N. Such arguments and definitions are often very convenient for establishing that a solution to some problem exists, or that it has certain properties. For numerical computation, though, we often want something stronger. Saying a sequence converges to a limit in the long run is all well and good — but fast as computers are, in the long run we are all dead[2].

In our review of calculus and analysis, we will give a great deal of attention to the formal machinery of the calculus. But we also want some reassurance that our computations make sense, and so we will take some care to give a proper account of what regularity hypotheses are needed for our calculations to work. We will frequently be somewhat more conservative with these hypotheses than is strictly necessary. This is partly for convenience, but there is more to it. Numerical methods are inherently approximate and finite things, and so we would like strong enough hypotheses to ensure that our calculations become "right enough, fast enough" – preferably with quantifiable meanings for both "right enough" and "fast enough."

5.2 Metric spaces

We will mostly be concerned with normed vector spaces, but sometimes we will want something a little more general. For the present section, we consider *metric spaces* and their structure, as this is the minimal structure that we really need for Weierstrass-style $\epsilon - \delta$ arguments to make sense. A metric space is simply a set X with a distance function (a metric) $d : X \times X \to \mathbb{R}$ with three properties:

- *Symmetry*: $d(x, y) = d(y, x)$;
- *Positive definiteness*: $d(x, y) \geq 0$, and $d(x, y) = 0$ iff $x = y$;
- *Triangle inequality*: $d(x, z) \leq d(x, y) + d(y, z)$.

A normed vector space is a metric space where the distance is just the norm of the difference, i.e. $d(u, v) = \|u - v\|$. But there are many other examples that are relevant, including the shortest distance between points in a network or on a manifold, and we will see these in our discussions of nonlinear dimensionality reduction and network analysis.

[2] "But this long run is a misleading guide to current affairs. In the long run we are all dead. Economists set themselves too easy, too useless a task if in tempestuous seasons they can only tell us that when the storm is long past the ocean will be flat again." — John Maynard Keynes, (from p.80, Keynes 1929)

5.2.1 Metric topology

A metric space automatically comes with a metric *topology*[3] where the open sets can be generated from unions of open balls

$$B_\delta(x) = \{y \in X : d(x, y) < \delta\}.$$

A set containing such a ball is sometimes called a "neighborhood" of x. For any subset $\Omega \subset X$, we say $x \in \Omega$ is an *interior point* if Ω contains a neighborhood of x; otherwise, it is a *boundary point*. The complements of open sets are *closed sets*. A closed set contains all its limit points; that is, if $C \subset X$ is closed, and for every $\delta > 0$ there is a $y \in C$ with $d(x, y) < \delta$, then x must also be in C.

The idea of continuity of a function makes sense for any topological spaces, using a definition in terms of open sets: a function $f : X \to Y$ is continuous if for any open $U \subset Y$, the preimage $f^{-1}(U) = \{x \in X : f(x) \in U\}$ is also open. For metric spaces, this coincides with the more usual notion of continuity due to Weierstrass, i.e. that if $f(x) = y$ then there is some neighborhood $B_\delta(x)$ in the preimage $f^{-1}(B_\epsilon(y))$.

5.2.2 Convergence

When we want to numerically find the minimum or maximum of a function, or solve some system of equations, we generally have to deal with two questions:

- Does the thing we want even exist?
- Can we find it (at least approximately), ideally in a reasonable amount of time?

One standard way to tackle both these questions is to construct a sequence of approximate solutions $x_1, x_2, ... \in X$ in an appropriate metric space X. The sequence converges to a limit x_* if for every neighborhood U of x_*, the sequence eventually enters and stays in U. Taking balls of radius ϵ as neighborhoods gives us the usual Weierstrass-style definition of convergence:

$$\forall \epsilon > 0, \exists N : \forall n \geq N, d(x_n, x_*) < \epsilon.$$

If a sequence of approximate solutions x_j converges to a limit x_* and we have some property that guarantees that a limit of approximate solutions is an exact solution for the problem in hand, then this is a constructive way to get a handle on the thing we want.

A convergent sequence of approximations gives us a way to compute arbitrarily good approximations to the limit x_* — eventually. But the word "eventually" is rarely a satisfactory answer to the question "when will we get there?" When we are focused on computation (as opposed to existence proofs), we typically seek a more quantitative answer, e.g. $d(x_n, x_*) \leq \gamma_n$ where

[3] A *topological space* is a set X and a collection of sets (called the open sets) that include the empty set and all of X and are closed under unions and finite intersections.

γ_n is some sequence with well-defined convergence properties. Alternately, we might give some function $\nu(\epsilon)$ such that $d(x_n, x_*) \leq \epsilon$ for $n \geq \nu(\epsilon)$.

For example, we say x_n converges *geometrically* (or sometimes *linearly*) to x_* if $d(x_n, x_*) \leq \alpha^n d(x_0, x_*)$ for some $0 \leq \alpha < 1$ (often called the rate constant). If x_n is geometrically convergent with rate $\alpha > 0$, then we can guarantee $d(x_n, x_*) < \epsilon$ for $n > \nu(\epsilon) = (\log(\epsilon) - \log(d(x_0, x_*))) / \log(\alpha)$.

5.2.3 Completeness

A *Cauchy sequence* $x_1, x_2, ...$ in a metric space X is a sequence such that for any $\epsilon > 0$ there is an N such that for all $i, j \geq N$, $d(x_i, x_j) < \epsilon$. We say Cauchy sequences $x_1, x_2, ...$ and $y_1, y_2, ...$ are equivalent if $d(x_i, y_i) \to 0$. We might believe that if there is any justice in the world, Cauchy sequences should converge, and equivalent Cauchy sequences should converge to the same thing. Unsurprisingly, though, sometimes we need to create our own justice.

A *complete* metric space is one in which Cauchy sequences converge. Alas, not all metric spaces are complete. However, for any metric space X we can define a new metric space $\bar{X}$ whose elements are equivalence classes of Cauchy sequences in X, with the metric given by $d(\{x_i\}, \{y_i\}) = \lim_{i \to \infty} d(x_i, y_i)$. There is also a canonical embedding of X into $\bar{X}$ which preserves the metric, given by taking $x \in X$ to the equivalence class associated with the constant sequence $x, x, ...$ The space $\bar{X}$ is the *completion* of X — and, by construction, it is complete, and every element of $\bar{X}$ can be expressed as a limit point for the embedding of X into $\bar{X}$.

Every finite-dimensional vector space over $\mathbb{R}$ or $\mathbb{C}$ is complete. There are also many complete infinite-dimensional vector spaces: for example, the space $C([a, b], \mathbb{R})$ of continuous real-valued functions on a closed interval $[a, b]$ is complete under the sup norm. However, there are also many examples of function spaces that are *not* complete, and in order to make analysis more tractable, we usually work with the completions of these spaces. A complete normed vector space is called a *Banach space*, and a complete inner product space is called a *Hilbert space*.

5.2.4 Compactness

In $\mathbb{R}^n$ (or any finite-dimensional normed vector space), there is a special role for sets that are *closed* and *bounded*. We generalize this to the idea of *compact* sets. The standard definition of a set K being compact is that any open cover C (a collection of open sets such that every point in Y belongs to some $U \in C$) has a *finite subcover*, i.e. $Y \subset \bigcup_{i=1}^{n} U_i$ where each U_i belongs to the original cover C. One sometimes considers covers consisting of open balls, in which case compactness of K is sometimes colloquiually rendered as "K can be guarded by a finite number of arbitrarily nearsighted watchmen."

Compact metric spaces have a number of useful properties. For example, the continuous image of a compact set is compact; this means, for example, that if $f : K \to \mathbb{R}$ is continuous and K is

compact, then $f(K)$ is also compact — so closed and bounded (for $\mathbb{R}$). Therefore, a continuous function on a compact set K always has a finite maximum and minimum that are achieved somewhere in K. Compact metric spaces are also *sequentially compact*[4], which means that if K is compact and $x_1, x_2, ...$ is a sequence in K, then there is some subsequence $x_{j_1}, x_{j_2}, ...$ that converges to a point in K.

In a finite dimensional normed vector space, the closed unit ball $\{v \in \mathcal{V} : \|v\| \leq 1\}$ is always compact (since it is a continuous map of a closed and bounded set in $\mathbb{R}^n$ via a basis). One can use the fact that norms are continuous to show that this implies *norm equivalence* among finite-dimensional spaces, i.e. for any norms $\|\cdot\|$ and $\|\cdot\|'$ on a finite dimensional $\mathcal{V}$, there exist constants $0 < m \leq M < \infty$ such that

$$\forall v \in \mathcal{V}, m\|v\|' \leq \|v\| \leq M\|v\|'.$$

The constants may be very big or small in general, but they exist! In contrast, for infinite dimensional spaces, the closed unit ball is generally *not* compact, and different norms give different topologies.

5.2.5 Contractions

One of the more frequently-used theorems in numerical analysis is the *Banach fixed point theorem*, also known as the *contraction mapping theorem*.

Theorem 5.1 (Banach fixed point theorem)**.** *Suppose X is a (non-empty) complete metric space and $f : X \to X$ is a* contraction*, i.e. there is some $0 \leq \alpha < 1$ such that $d(f(x), f(y)) \leq \alpha d(x, y)$. Then there exists a unique point $x_* \in X$ such that $f(x_*) = x_*$.*

Proof. Choose any $x_0 \in X$; there must be at least one since X is non-empty. Using x_0 as a starting point, define the sequence $x_0, x_1, x_2, ...$ by the iteration $x_{k+1} = f(x_k)$. By contractivity, $d(x_k, x_{k+1}) \leq \alpha^k d(x_0, x_1)$. By the triangle inequality and a geometric series bound, for any $j > i$,

$$d(x_i, x_j) \leq \sum_{k=i}^{j-1} d(x_k, x_{k+1}) \leq \sum_{k=i}^{j-1} \alpha^k d(x_0, x_1) \leq \alpha^i \frac{d(x_0, x_1)}{1-\alpha}.$$

Therefore, for any $\epsilon > 0$ we can choose an N such that for all $i, j > N$, $d(x_i, x_j) \leq \alpha^n d(x_0, x_1)/(1-\alpha) < \epsilon$, i.e. the iterates x_j form a Cauchy sequence, which must converge to some $x_* \in X$ by completeness.

Now suppose $x'_* \in X$ satisfies $x'_* = f(x'_*)$. Then by contractivity and the fixed point condition,

$$d(x_*, x'_*) = d(f(x_*), f(x'_*)) < \alpha d(x_*, x'_*),$$

which can only happen if $d(x_*, x'_*) = 0$. Therefore x_* and x'_* must be the same point. □

[4] In general topological spaces, compactness and sequential compactness are not the same thing. Fortunately, it's all the same in metric space (assuming the axiom of choice).

The Banach fixed point theorem is useful because it gives us not only the existence of a fixed point, but also a method to compute arbitrarily good approximations to that fixed point *with a rate of convergence*. Consequently, this theorem is a mainstay in the convergence analysis of many iterative numerical methods.

5.3 Continuity and beyond

The study of real analysis often devolves into a study of counterexamples: we start with something that seems like it should be true about a function, give an example where it is not, and then come up with a hypothesis under which it is true. While we make frequently resort to assuming that everything is "as nice as possible" (several times continuously differentiable, at least), it is worth a page or two to talk about more modest hypotheses and what they give us. In particular:

- *Continuity* gives us the intermediate value theorem.
- *Uniform continuity* gives us a set of functions that is closed under (uniform) limits.
- *Absolute continuity* gives us the fundamental theorem of calculus.
- *Bounded variation* gives us a dual space to continuous functions.

For stronger control that can be used to prove approximation bounds, we might turn *Lipschitz continuity* (or a more general *modulus of continuity*).

We will focus on functions between normed vector spaces. But almost everything in this section, and indeed in this chapter, generalizes beyond this setting.

5.3.1 Continuity

When we compute, we almost always presume functions that are continuous almost everywhere. Floating point cannot exactly represent every real number, and so when we evaluate a function from $\mathbb{R}$ to $\mathbb{R}$ on the computer, it helps if the small changes to the input (and the output) can be forgiven.

Let $\mathcal{V}$ and $\mathcal{W}$ are normed vector spaces, and suppose $\Omega \subset \mathcal{V}$. We say $f : \Omega \subset \mathcal{V} \to \mathcal{W}$ is continuous at $x \in \Omega$ if

$$\forall \epsilon > 0, \exists \delta > 0 : \forall y \in \Omega, \|x - y\| \leq \delta \implies \|f(y) - f(x)\| \leq \epsilon$$

In words: for any tolerance ϵ there is a corresponding tolerance δ so that getting y within δ of x means we will get $f(y)$ within ϵ of $f(x)$. Composition of continuous functions is continuous: if g is continuous at $f(x)$ and f is continuous at x, then $g \circ f$ is continuous at x.

A function is continuous on Ω if it is continuous at each point in Ω. The set of such continuous functions from $\Omega \subset \mathcal{V}$ to $\mathcal{W}$ is called $C(\Omega, \mathcal{W})$. The continuous functions $C(\Omega, \mathcal{W})$ form a vector

space: sums of continuous functions are continuous, and so are scalar multiples of continuous functions.

For functions from $[a, b] \subset \mathbb{R}$ to $\mathbb{R}$, the *intermediate value theorem* is a fundamental result: if $x, y \in [a, b]$ take on values $f(x)$ and $f(y)$, then for every target value $f_* \in (f(x), f(y))$, there is a $z \in (x, y)$ such that $f(z) = f_*$. The intermediate value theorem is fundamentally one-dimensional, but still tells us useful things about functions between vector spaces through composition: if $f : \Omega \subset \mathcal{V} \to \mathcal{W}$ and $w^* \in \mathcal{W}^*$ is any (bounded) linear functional, then, the function $s \in [0, 1] \mapsto w^* f((1 - s)x + sy)$ is a continuous real-valued function on the interval $[0, 1]$ for which the intermediate value theorem applies.

5.3.2 Uniform continuity

A function is *uniformly* continuous if the same ϵ-δ relation works for *every* x in Ω. In a finite-dimensional space, any continuous function on a closed and bounded Ω is automatically also uniformly continuous. A sequence of functions is *uniformly equicontinuous* if the same ϵ-δ relation works for every $x \in \Omega$ *and* for every function in the sequence.

We say that $f_k \to f$ uniformly if for all $\epsilon > 0$ there is an N such that for $k > N$ we must have $\|f_k - f\|_\infty \leq \epsilon$, where $\|f_k - f\|_\infty = \sup_{x \in \Omega} \|f_k(x) - f(x)\|$. This matters because while a pointwise limit of continuous functions does not have to be continuous, a uniform limit of uniformly continuous functions is uniformly continuous.

If $f : [a, b] \subset \mathbb{R} \to \mathbb{R}$ is continuous (and therefore uniformly continuous), the Weierstrass approximation theorem says that f can be written as the uniform limit of a sequence of polynomials.

5.3.3 Absolute continuity

We say $f : \Omega \subset \mathbb{R} \to \mathbb{R}$ is *absolutely continuous* on an interval if

$$\forall \epsilon > 0, \exists \delta > 0 : \forall \{(x_i, y_i) \subset \Omega\}_{i=1}^n \text{ disjoint}, \\ \sum_{i=1}^n |x_i - y_i| < \delta \implies \sum_{i=1}^n |f(x_i) - f(y_i)| < \epsilon.$$

Absolutely continuous functions are those that are nice enough that the (Lebesgue) fundamental theorem of calculus applies; that is, the function f has derivatives almost everywhere, and

$$f(b) = f(a) + \int_a^b f'(x)\, dx.$$

Most (perhaps all) continuous functions that we will encounter in this class will be absolutely continuous.

5.3.4 Bounded variation

The *total variation* of a function $f : [a, b] \to \mathbb{R}$ is a supremum over partitions $a = x_0 < x_1 < ... < x_{n_\mathcal{P}} = b$ of how much the function value jumps:

$$V_a^b = \sup_{\mathcal{P}} \sum_{i=0}^{n_\mathcal{P}-1} |f(x_{i+1}) - f(x_i)|.$$

For continuously differentiable functions, the total variation is the integral of the absolute value of the derivative. The notion generalizes to functions on more general vector spaces, though it is somewhat more technical and involves a little measure theory.

Bounded variation functions (those with finite total variation) are those that are nice enough that they can be integrated against continuous functions (in the Riemann sense). Indeed, bounded variation functions correspond to the dual space to $C[a, b]$: every linear functional $w^* \in (C[a, b])^*$ can be written as $w^* f = \int_a^b f(t)\, dv(t)$ where v is a bounded variation function and $dv(t)$ is its (distributional) derivative.

5.3.5 Lipschitz continuity

A function $f : \Omega \subset \mathcal{V} \to \mathcal{W}$ is Lipschitz continuous with constant C if $\forall x, y \in \Omega$,

$$\|f(x) - f(y)\| \leq C\|x - y\|.$$

Unlike other notions of continuity we have considered, Lipschitz continuity involves no *explicit* $\epsilon - \delta$ relationship. Also unlike other notions of continuity, Lipschitz continuity gives us control on the relationship between function values even at points that may be far from each other.

5.3.6 Modulus of continuity

We have already mentioned the classic theorem of Weierstrass says that a continuous real-valued function f on a finite closed interval $[a, b]$ can be approximated arbitrarily well in the L^∞ sense, i.e.

$$\forall \epsilon > 0, \exists p \in \mathcal{P}_d : \|f - p\|_\infty \leq \epsilon,$$

where $\|f - p\|_\infty = \max_{x \in [a,b]} |f(x) - p(x)|$. But while this is a useful theorem, it lacks the level of detail we would like if we are trying to compute. What degree polynomial is needed for a particular ϵ? How do we construct such a polynomial? We will return to some of these points later in the book, but for now we simply want to make the point that we usually want more information about our functions than simply that they are continuous. In particular, we would like to be able to say something more specific about a relationship between ϵ and δ, either close to a particular point or more globally, e.g. making a statement of the form

$$\|f(x) - f(y)\| \leq g(x - y)$$

for appropriate x and close enough y, where g is some well-understood real-valued *gauge function*. One example of this is Lipschitz continuity, but it useful to generalize to other gauge functions.

The *modulus of continuity* for a function f on some domain Ω is a non-decreasing function $\omega(\delta)$ for $\delta > 0$ given by

$$\omega(\delta) = \sup_{x,y \in \Omega : \|x-y\| \leq \delta} \|f(x) - f(y)\|.$$

Put slightly differently, the modulus of continuity is the minimal function such that

$$f(x+d) = f(x) + r(x,d), \quad \|r(x,d)\| \leq \omega(\|d\|).$$

for any x and $x+d$ in Ω. A function f is uniformly continuous iff $\omega(\delta) \to 0$ as $\delta \to 0$.

Several standard rules for differentiation have analogous inequalities involving the modulus of continuity. For example, if $f : \Omega \to \mathcal{V}$ and $g : \Omega \to \mathcal{V}$ are functions with moduli of continuity ω_f and ω_g, then

- $f + g$ has modulus of continuity $\omega_{f+g}(\delta) \leq \omega_f(\delta) + \omega_g(\delta)$.
- αf has modulus of continuity $\omega_{\alpha f}(\delta) \leq |\alpha| \omega_f(\delta)$
- If $w^* \in \mathcal{V}^*$, then $\omega_{w^* f}(\delta) \leq \|w^*\| \omega_f(\delta)$
- If $\mathcal{V}$ is an inner product space, $x \mapsto \langle f(x), g(x) \rangle$ has modulus of continuity $\omega_{fg}(\delta) \leq \omega_f(\delta)\|g\|_\infty + \|f\|_\infty \omega_g(\delta)$ where $\|g\|_\infty = \sup_{x \in \Omega} \|g(x)\|$ and similarly with $\|f\|_\infty$.

If $f : \Omega \to \mathbb{R}$, and $\beta = \inf_{x \in \Omega} |f(x)|$, we also have $\omega_{1/f}(\delta) \leq \omega_f(\delta)/\beta^2$. Finally, if $f : \Omega \to \mathcal{V}$ and $g : f(\Omega) \to \mathcal{W}$, we have

$$\omega_{g \circ f}(\delta) \leq \omega_g(\omega_f(\delta)).$$

The proof of these facts is left as an exercise for the reader.

A function f is *Lipschitz continuous* on Ω if the modulus of continuity on Ω satisfies $\omega_f(\delta) \leq C\delta$ for some C. Put differently, a Lipschitz function satisfies

$$\forall x, y \in \Omega, \|f(x) - f(y)\| \leq C\|x - y\|.$$

The constant C is a *Lipschitz constant* for f.

A more general notion than Lipschitz continuity is α-Hölder continuity, which is the condition that $\omega_f(\delta) \leq C\delta^\alpha$ This condition is interesting for $0 < \alpha \leq 1$, with $\alpha = 1$ corresponding to Lipschitz continuity. The only α-Hölder continuous functions for $\alpha > 1$ are the constant functions.

Returning to our discussion of approximating continuous real-valued functions on $[a, b]$ by polynomials, Jackson's theorem provides a quantitative bound on the optimal error in terms of

the modulus of continuity. If f is a continuous function on $[a, b]$, Jackson's theorem gives that there exists a polynomial of degree n with

$$\|f - p\|_\infty \leq 6\omega\left(\frac{b-a}{n}\right).$$

We can directly substitute the bounds from the definitions of Lipschitz or α-Hölder continuous functions to get bounds for those cases.

5.3.7 Order notation

The modulus of continuity provides global control over functions on some domain. However, sometimes we are satisfied with much more local notions of control. To express these, it is useful to again turn to order notation, which we visited briefly in Section 3.2. The same notation can be used to compare functions $f(n)$ and $g(n)$ in a limit as $n \to \infty$ or to compare functions $f(\epsilon)$ and $g(\epsilon)$ in the limit as $\epsilon \to 0$. We are typically interested in upper bounds when dealing with calculus; that is, we care about

- $f(\epsilon) = O(g(\epsilon))$, i.e. $\exists C > 0, \rho > 0$ s.t. $\forall \epsilon < \rho$, $|f(\epsilon)| < Cg(\epsilon)$.
- $f(\epsilon) = o(g(\epsilon))$, i.e. $\forall C > 0, \exists \rho > 0$ s.t. $\forall \epsilon < \rho$, $|f(\epsilon)| < Cg(\epsilon)$.

From a less notationally messy perspective, we have

$$\limsup_{\epsilon \to 0} |f(\epsilon)|/g(\epsilon) = \begin{cases} C \geq 0, & \text{for } f(\epsilon) = O(g(\epsilon)) \\ 0, & \text{for } f(\epsilon) = o(g(\epsilon)), \end{cases}$$

and when $\lim_{\epsilon \to 0} f(\epsilon)/g(\epsilon) = C \neq 0$, we say $f(\epsilon) = \Theta(g(\epsilon))$. When $f : \mathbb{R} \to \mathcal{V}$ for some normed space $\mathcal{V}$, we abuse order notation slightly to write

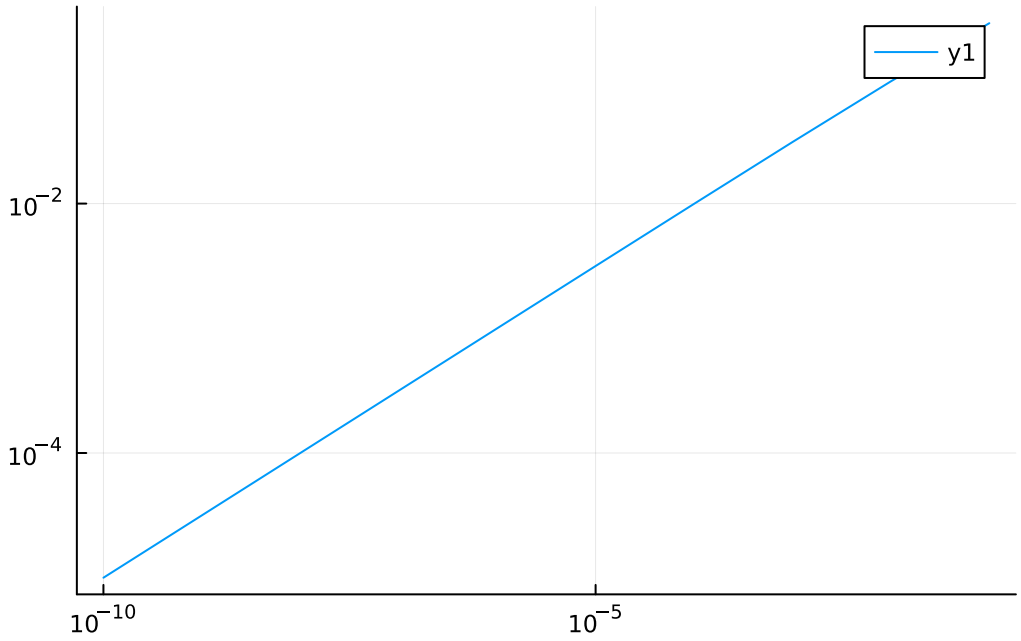
$$\limsup_{\epsilon \to 0} \|f(\epsilon)\|/g(\epsilon) = \begin{cases} C \geq 0, & \text{for } f(\epsilon) = O(g(\epsilon)) \\ 0, & \text{for } f(\epsilon) = o(g(\epsilon)), \end{cases}$$

We are frequently interested in the case of functions that are $O(\epsilon^r)$ for some integer or rational r. If $f(\epsilon) = \Theta(\epsilon^r)$ is positive, then near zero we expect $\log(f(\epsilon)) \approx \log(C\epsilon^r) = r\log(\epsilon) + \log(C)$. We can check this condition for small ϵ graphically with a log-log plot:

```
let
    # Should be O(ϵ^{1/2})
    f(ϵ) = maximum(roots(Polynomial([-ϵ, 0, 1, 1])))

    ϵs = 10.0.^(-10:-1)
    fs = f.(ϵs)
    println("Approximate slope: ",
```

```
            (log(fs[2])-log(fs[1]))/
            (log(εs[2])-log(εs[1])) )
    plot(εs, fs, xscale=:log10, yscale=:log10)
end
```

```
Approximate slope: 0.4999953048691424
```



This type of plot is, of course, somewhat heuristic: the asymptotic behavior as $\epsilon \to 0$ should manifest for small enough ϵ, but nothing says that 10^{-10} (for example) is "small enough." We can be more rigorous by including additional information, as we will discuss in later chapters.

5.4 Derivatives

We assume the reader is thoroughly familiar with differentiation in one dimension. Our focus in the rest of this section is on differentiation on functions between vector spaces.

5.4.1 Gateaux and Frechet

5.4.1.1 Functions from $\mathbb{R}$ to $\mathbb{R}$

The standard definition for the derivative of $f : \mathbb{R} \to \mathbb{R}$ is

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

Alternately, we can write f as

$$f(x+h) = f(x) + f'(x)h + o(|h|).$$

Of course, not all functions are differentiable. When f is differentiable at x, we will sometimes also write $Df(x)$ to denote the derivative.

When the derivative exists and is continuous on some interval (a, b) containing x, we say f is C^1 on (a, b). In this case, the fundamental theorem of calculus gives that for any $x + h \in (a, b)$,

$$\begin{aligned} f(x+h) &= f(x) + \int_0^h f'(x+\xi)\, d\xi \\ &= f(x) + f'(x)h + r(h) \\ r(h) &= \int_0^h \left(f'(x+\xi) - f'(x)\right)\, d\xi. \end{aligned}$$

Here $r(h)$ is the *remainder term* for the first-order approximation $f(x) + f(x)h$. If $\omega_{f'}$ is the modulus of continuity for f' on (a, b), then

$$\|f'(x+\xi) - f'(x)\| \leq \omega_{f'}(|\xi|),$$

and therefore

$$|r(h)| \leq \int_0^{|h|} \omega_{f'}(\xi)\, d\xi.$$

When f' is Lipschitz with constant C, we have

$$|r(h)| \leq \int_0^{|h|} \omega_{f'}(|\xi|)\, d\xi \leq \int_0^{|h|} C\xi\, d\xi = \frac{1}{2} C h^2.$$

Hence, in regions where f' is not only continuous but Lipschitz, the error $r(h)$ in the linearized approximation is not only $o(|h|)$, but it is even $O(h^2)$.

When a differentiable function f is only available by evaluation, we can approximate the derivative at x by finite differences, e.g. taking

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

where h is "small enough." More precisely, if f is C^1 on an interval containing (a, b), we have

$$\frac{f(x) + f'(x)h + r(h) - f(x)}{h} = f'(x) + \frac{r(h)}{h},$$

where r is the remainder for the linear approximation about x. When f' is Lipschitz with constant C, the error in this finite difference approximation is therefore

$$\frac{|r(h)|}{|h|} \leq \frac{1}{2}C|h| = O(|h|).$$

We further analyze how the size of h affects accuracy, as well as other methods for approximating derivatives, in Chapter 12. For this chapter, though, it will still be useful to sanity check derivatives via finite differences:

```
finite_diff(f, x; h=1e-8) = (f(x+h)-f(x))/h
```

5.4.1.2 Functions from $\mathbb{R}$ to $\mathcal{W}$

Suppose $g : \mathbb{R} \to \mathcal{W}$ for some normed linear space $\mathcal{W}$. We can define the derivative of g almost identically to the derivative for a function from $\mathbb{R}$ to $\mathbb{R}$, i.e.

$$g(x + h) = g(x) + g'(x)h + r(h), \quad \|r(h)\| = o(|h|).$$

The only thing that differs is that we are controlling the norm of $r(h)$ rather than the absolute value. Also similarly to the real case, if g' is Lipschitz with constant C, then

$$\|r(h)\| \leq \frac{1}{2}Ch^2,$$

and we can bound the error in simple finite difference estimates of the derivative as we do in the case of functions from $\mathbb{R}$ to $\mathbb{R}$.

There are a handful of theorems that are specific to real-valued functions, like the mean value theorem (Section 5.4.2). These theorems usually still tell us interesting things about vector valued functions by considering functions $x \mapsto w^* g(x)$ where $w^* \in \mathcal{W}^*$ is some dual vector.

5.4.1.3 Functions from $\mathcal{V}$ to $\mathbb{R}$

Suppose $f : \mathcal{V} \to \mathbb{R}$ for some normed linear space $\mathcal{V}$. Then for any point $x \in \mathcal{V}$ and nonzero vector $u \in \mathcal{V}$, we can define a function $f \circ c$ from $\mathbb{R}$ to $\mathbb{R}$ that evaluates f along a ray $c_{x,u}(s) = x + su$. Assuming it exists, the derivative $(f \circ c_{x,u})'$ is the *Gateaux derivative* (directional derivative) of f at a point x and in a direction u:

$$D[f(x); u] = (f \circ c_{x,u})'(0).$$

Alternately, the Gateaux derivative is the number such that

$$f(x + su) = f(x) + sD[f(x); u] + o(s).$$

We can estimate the Gateaux derivative by finite differencing in the given direction

```
finite_diff(f, x, u; h=1e-8) = (f(x+h*u)-f(x))/h
```

When the Gateaux derivative of f at x is defined for all $u \in \mathcal{V}$, we say f is *Gateaux differentiable* at x.

When all the Gateaux derivatives are continuously defined in a neighborhood of v, we say f is C^1 on some open set containing v. In this case, Gateaux derivatives are related by the *Frechet derivative*, a functional $f'(x) \in \mathcal{V}^*$ (also written $Df(x)$) such that

$$D[f(x); u] = f'(x)u$$

or, equivalently, for any direction u,

$$f(x + su) = f(x) + sf'(x)u + o(s).$$

When the Frechet derivative is Lipschitz with some constant C in a consistent norm, we can bound the remainder term by $Cs^2/2$, as in the one-dimensional case. Frechet differentiability is a stronger condition than Gateaux differentiability. When we say a function is "differentiable at x" without qualifications, we mean it is Frechet differentiable.

The Frechet derivative (or just "the derivative") $f'(x)$ is an element in $\mathcal{V}^*$. In a (real) inner product space, the *gradient* $\nabla f(x) \in \mathcal{V}$ is the dual of $f'(x)$ given by the Riesz map, i.e.

$$f'(x)u = \langle \nabla f(x), u \rangle.$$

In $\mathbb{R}^n$ with the standard inner product, the gradient (a column vector) is the transpose of the derivative (a row vector).

5.4.1.4 A polynomial example

Following our pattern from the previous chapter, we consider an example in a polynomial space. Let $f : \mathcal{P}_d \to \mathbb{R}$ be given by $f(p) = \frac{1}{2}\int_{-1}^{1} p(x)^2\,dx$. The Frechet derivative of f (which is a function of the polynomial $p \in \mathcal{P}_d$, not the indeterminate x) is then the functional $f'(p)$ such that the action on a vector $q \in \mathcal{P}_d$ is

$$f'(p)q = \int_{-1}^{1} p(x)q(x)\,dx.$$

With respect to the power basis, we can write $f'(q)$ in terms of the row vector

$$d^T = \begin{bmatrix} f'(p)x^0 & f'(p)x^1 & \dots & f'(p)x^d \end{bmatrix}.$$

The gradient with respect to the $L^2([-1,1])$ inner product is then

$$\nabla f(p) = X_{0:d} M^{-1} d$$

where M is the Gram matrix for the power basis.

We can compute with $f'(p)$ by calling the `integrate` command or through the power basis. We can also compute with $f'(p)$ by writing it in terms of a basis of $\mathcal{P}_d^*$ in terms of point evaluation functionals, i.e.

$$f'(p) = c^T W^*, \quad W^* = \begin{bmatrix} \delta_{x_0}^* \\ \vdots \\ \delta_{x_d}^* \end{bmatrix}.$$

We can also compute the c coefficients by solving

$$c^T A = d^T$$

where A is the Vandermonde matrix

$$A = W^* X_{0:d} = \begin{bmatrix} 1 & x_0^1 & \dots & x_0^d \\ 1 & x_1^1 & \dots & x_1^d \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_d^1 & \dots & x_d^d \end{bmatrix}.$$

We can sanity check this against finite differences for a random pair of polynomials:

```
let
    # Functions for f and the Frechet derivative f'(p)
    f(p) = integrate(0.5*p*p, -1, 1)
    Df(p) = q -> integrate(p*q, -1, 1)

    # Express f'(p) in the power basis
    d(p) = [Df(p).(Polynomial(ej)) for ej in eachcol(Matrix(I,4,4))]'

    # Gram matrix for the L2 inner product
    m(k) = k % 2 == 1 ? 0.0 : 2/(k+1)
    M = [m(i+j) for i=0:3, j=0:3]

    # Compute the gradient
    ∇f(p) = Polynomial(M\d(p)')
```

```
    # Vandermonde matrix A for four equispaced points
    x = range(-1, 1, length=4)
    A = [xi^j for xi=x, j=0:3]

    # Compute c in terms of point evaluations
    c(p) = d(p)/A

    # Test everything
    ptest = Polynomial(rand(4))
    qtest = Polynomial(rand(4))
    Dfpq_fd = finite_diff(f, ptest, qtest)     # Finite diff
    Dfpq1 = Df(ptest)(qtest)                   # Analytic form
    Dfpq2 = c(ptest)*qtest.(x)                 # Via evaluation fnls
    Dfpq3 = integrate(∇f(ptest)*qtest, -1, 1)  # Via gradient

    # Compare -- everything should be equal (some approx for fd)
    isapprox(Dfpq1, Dfpq_fd, rtol=1e-6) &&
    Dfpq1 ≈ Dfpq2 &&
    Dfpq1 ≈ Dfpq3

end
```

```
true
```

5.4.1.5 General mappings

Now consider $f : \mathcal{V} \to \mathcal{W}$ where $\mathcal{V}$ and $\mathcal{W}$ are normed linear spaces. In this case, the Gateaux derivative in a direction $u \in \mathcal{V}$ is a vector in $\mathcal{W}$, still given as

$$D[f(v); u] = (f \circ c_{v,u})'(0) \text{ for } c(s) = x + su$$

and when there is a Frechet derivative $Df(v) = f'(v) \in L(\mathcal{V}, \mathcal{W})$ we have

$$f(v + u) = f(v) + f'(v)u + o(\|u\|),$$

and the Gateaux derivatives satisfy

$$D[f(v); u] = f'(v)u.$$

When the Frechet derivative is Lipschitz with constant C (with respect to the induced norm), then the remainder for the linear approximation is bounded by $C\|u\|^2/2$. The representation of the Frechet derivative with respect to a particular basis is the *Jacobian matrix* $J = W^{-1}f'(p)V$.

5.4.2 Mean values

For one-dimensional continuously differentiable functions $f : \Omega \subset \mathbb{R} \to \mathbb{R}$ for some open connected Ω, the fundamental theorem of calculus tells us that for $a, b \in \Omega$ we have

$$f(b) - f(a) = \int_0^1 f'(zb + (1-z)a)(b-a)\,dz.$$

The *mean value theorem* tells us that for some intermediate $c \in [a, b]$ ($c = \xi b + (1-\xi)a$ for $\xi \in [0, 1]$), we have

$$f'(c)(b-a) = \int_0^1 f'(zb + (1-z)a)(b-a)\,dz.$$

The fundamental theorem of calculus holds even when f has discontinuities in the derivative at some points, but the second equation relies on continuity of the derivative. This useful theorem generalizes to the vector space case in various ways.

For continuously differentiable functions $f : \Omega \subset \mathcal{V} \to \mathbb{R}$ where Ω is open and convex (i.e. the line segment connecting a and b lies within Ω), we have

$$\begin{aligned} f(b) - f(a) &= \int_0^1 f'(zb + (1-z)a)(b-a)\,dz \\ &= f'(\xi b + (1-\xi)a)(b-a). \end{aligned}$$

for some $\xi \in [0, 1]$. And for continuously differentiable functions $g : \Omega \subset \mathcal{V} \to \mathcal{W}$ where Ω is open and convex, we have that for any $w^* \in \mathcal{V}^*$ we can apply the mean value theorem to the real-valued $f(v) = w^* g(v)$; that is, there is a $\xi \in [0, 1]$ such that

$$\begin{aligned} w^*(g(b) - g(a)) &= \int_0^1 w^* g(zb + (1-z)a)(b-a)\,dz \\ &= w^* g'(\xi b + (1-\xi)a)(b-a). \end{aligned}$$

However, the choice of ξ depends on w^*, so it is *not* true that we can fully generalize the mean value theorem to maps between vector spaces. On the other hand, we can get something that is almost as good, at least for the purpose of proving bounds used in numerical computing, by choosing w^* to be a vector such that $\|w^*\| = 1$ (using the dual norm to whatever norm we are using for $\mathcal{W}$) and $w^*(g(b) - g(a)) = \|g(b) - g(a)\|$. With this choice, and consistent choices of norms, we have that for

$$\begin{aligned} \|g(b) - g(a)\| &= \int_0^1 w^* g'(zb + (1-z)a)(b-a)\,dz \\ &\leq \int_0^1 \|w^*\| \|g'(zb + (1-z)a)\| \|b-a\|\,dz \\ &\leq \left(\int_0^1 \|g'(zb + (1-z)a)\|\,dz \right) \|b-a\|. \end{aligned}$$

We can also bound the average norm of the derivative on the segment from a to b by

$$\int_0^1 \|g'(zb + (1-z)a)\|\, dz \leq \max_{z\in[0,1]} \|g'(zb + (1-z)a)\|.$$

When $\|g'(x)\| \leq C$ for all $x \in \Omega$, the function g is Lipschitz with constant C on Ω.

5.4.3 Chain rule

Suppose $f : \mathcal{V} \to \mathcal{W}$ and $g : \mathcal{U} \to \mathcal{V}$ are *affine* functions, i.e.

$$\begin{aligned} g(x) &= g_0 + J_g x \\ f(y) &= f_0 + J_f y. \end{aligned}$$

Then $h = f \circ g$ is another affine map

$$h(x) = f(g(x)) = f(g(0)) + J_f J_g x.$$

The chain rule is just a generalization from affine maps to functions well approximated by affine maps (i.e. differentiable functions).

Now let $f : \mathcal{V} \to \mathcal{W}$ and $g : \mathcal{U} \to \mathcal{V}$ be functions where g is differentiable at u and f is differentiable at $v = g(u)$. Then

$$\begin{aligned} g(u+x) &= g(u) + g'(u)x + r_g(x), \quad r_g(x) = o(\|x\|), \\ f(v+y) &= f(v) + f'(u)y + r_f(y), \quad r_f(y) = o(\|y\|). \end{aligned}$$

Let $h = f \circ g$; then setting $y = g'(u)x + r_g(x) = O(\|x\|)$, we have

$$\begin{aligned} h(u+x) &= f(v) + f'(v)y + r_f(y) \\ &= f(v) + f'(v)g'(u)x + r_h(x), \\ r_h(x) &= g'(u)r_g(x) + r_f(g'(u)x + r_g(x)) = o(\|x\|). \end{aligned}$$

That is, h is differentiable at u with $h'(u) = f'(v)g'(u)$.

5.4.3.1 A polynomial example

Now consider

$$h(p) = \frac{1}{2}\int_{-1}^{1} \left(\frac{dp}{dx} - \phi(x)\right)^2 dx,$$

which we can see as the composition of the functional

$$f(r) = \frac{1}{2}\int_{-1}^{1} r(x)^2\, dx$$

which we analyzed in the previous section, together with

$$r(p) = \frac{dp}{dx} - \phi(x).$$

The function $r(p)$ is affine in p, and the derivative $r'(p)$ is simply the differentiation operator $\frac{d}{dx}$. We already saw that

$$f'(r)q = \int_{-1}^{1} r(x)q(x)\,dx.$$

Therefore, by the chain rule we have

$$h'(p)q = f'(r)r'(p)q = \int_{-1}^{1} r(x)\frac{dq}{dx}(x)\,dx.$$

As is often the case, working code is a useful check to make sure that we understand what is happening with the mathematics:

```
let
    ϕ = Polynomial(rand(4))
    r(p) = derivative(p) - ϕ
    f(r) = 0.5*integrate(r*r, -1, 1)
    h(p) = f(r(p))
    Dh(p) = q -> integrate(r(p)*derivative(q), -1, 1)

    ptest = Polynomial(rand(5))
    qtest = Polynomial(rand(5))
    isapprox(Dh(ptest)(qtest),
             finite_diff(h, ptest, qtest), rtol=1e-6)
end
```

```
true
```

5.4.4 Partial derivatives

Consider a Frechet-differentiable function $f : \mathcal{V} \oplus \mathcal{U} \to \mathcal{W}$. For $x \in \mathcal{V} \oplus \mathcal{U}$, we can write the Frechet derivative as quasimatrix with respect to the two components of the domain space:

$$Df(x) = \begin{bmatrix} (Df(x))_1 & (Df(x))_2 \end{bmatrix},$$

where $(Df(x))_1 \in L(\mathcal{V}, \mathcal{W})$ and $(Df(x))_2 \in L(\mathcal{U}, \mathcal{W})$. However, it is often more convenient to "unpack" the input to the function into different components, i.e. thinking of $f : \mathcal{V} \times \mathcal{U} \to \mathcal{W}$. We similarly unpack y into a $\mathcal{V}$ component y_1 and a $\mathcal{U}$ component y_2. Then we write

$$Df(x_1, x_2)(y_1, y_2) = D_1 f(x_1, x_2)y_1 + D_2 f(x_1, x_2)y_2,$$

where the maps $D_1 f(x_1, x_2) \in L(\mathcal{V}, \mathcal{W})$ and $D_2 f(x_1, x_2) \in L(\mathcal{U}, \mathcal{W})$ are the same as $(Df(x))_1$ and $(Df(x))_2$ above. We refer to $D_1 f$ and $D_2 f$ as the *partial derivatives* of f with respect to the first and second argument.

More generally, if f is a Frechet-differentiable function with m arguments, we write the quasimatrix expression

$$\begin{aligned} & Df(x_1, \ldots, x_m)(y_1, \ldots, y_m) \\ &= \begin{bmatrix} D_1 f(x_1, \ldots, x_m) & \ldots & D_m f(x_1, \ldots, x_m) \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \\ &= \sum_j D_j f(x_1, \ldots, x_m) y_j. \end{aligned}$$

Though this may look like a regular matrix expression, we are deliberately *not* assuming that the arguments have the same type. Indeed, the arguments may belong to wildly different spaces: the first component might belong to $\mathcal{P}_d$, the second to $\mathbb{R}^n$, and so on. Nonetheless, we can formally put a (normed) vector space structure on the whole list taken as a direct sum of the argument spaces, and then treat the partial derivatives $D_j f$ as pieces of the overall derivative Df.

A common source of confusion in partial differentiation comes from implicit function composition. For example, consider $f : \mathcal{V} \to \mathbb{R}$ where $\mathcal{V}$ is a two dimensional space with bases $V = \begin{bmatrix} v_1 & v_2 \end{bmatrix}$ and $U = \begin{bmatrix} u_1 & u_e \end{bmatrix}$. Then we can write

$$\begin{aligned} h_V(\alpha, \beta) &= (f \circ g_V)(\alpha, \beta), \quad g_V(\alpha, \beta) = v_1 \alpha + v_2 \beta, \\ h_U(a, b) &= (f \circ g_U)(a, b), \quad g_U(a, b) = u_1 a + u_2 b. \end{aligned}$$

In this notation, the partials are clearly defined:

$$\begin{aligned} D_k h_V(\alpha, \beta) &= f'(g_V(\alpha, \beta)) v_k, \\ D_k h_U(\alpha, \beta) &= f'(g_U(\alpha, \beta)) v_k. \end{aligned}$$

In contrast, classical notation often conflates f, $f \circ g_V$, and $f \circ g_U$:

Classical notation	Our notation
$\partial f / \partial \alpha$	$D_1(f \circ g_V)$
$\partial f / \partial \beta$	$D_2(f \circ g_V)$
$\partial f / \partial a$	$D_1(f \circ g_U)$
$\partial f / \partial b$	$D_2(f \circ g_U)$

We will sometimes use symbolic labels rather than indices to refer to arguments to a function. For example, we might write $D_u f(u, v)$ instead of $D_1 f(u, v)$. This approach is *only* sensible when the symbolic label unambiguously refers to an argument of f, and we must treat it carefully. When there is a danger of ambiguity, we will bend our notation to make clear that we are labeling a slot. For example, we can write $D_u f(u = a, v = b)$ to refer to $D_1 f(a, b)$.

5.4.5 Implicit functions

Now suppose $\mathcal{V}$ and $\mathcal{W}$ are spaces of equal dimension, and consider a function

$$f : \mathcal{V} \times \mathcal{U} \to \mathcal{W}$$

that is continuously differentiable on some open set including a point (v, u) for which $f(v, u) = 0$. We will write the Frechet derivative of f in quasimatrix form as

$$f'(v, u) = \begin{bmatrix} D_1 f(v, u) & D_2 f(v, u) \end{bmatrix},$$

where $D_1 f$ is the piece of the map associated with v and $D_2 f$ is the piece associated with u. If $D_1 f(v, u))$ is invertible, the the *implicit function theorem* says that there exists a function

$$g : \Omega \subset \mathcal{U} \to \mathcal{V}$$

defined on an open set containing u and such that $g(u) = v$ and $f(g(z), z) = 0$ for z in an open neighborhood about u. By the chain rule, we must satisfy

$$D_1 f(g(z), z)g'(z) + D_2 f(g(z), z) = 0.$$

Therefore, $g'(z)$ can be computed as

$$g'(z) = -\left(D_1 f(g, z)\right)^{-1} D_2 f(g, z),$$

where we have suppressed z as an argument to g to simplify notation.

Often we are interested not in $g(z)$, but some $h(g(z))$ that might lie in a much lower dimensional space. If everything is differentiable, then by the chain rule we have

$$(h \circ g)'(z) = -h'(g(z)) \left[\left(D_1 f(g, z)\right)^{-1} D_2 f(g, z)\right].$$

We can use associativity to rewrite $(h \circ g)'(z)$ as

$$\begin{aligned} \bar{g}^* &= -h'(g(z)) \left(D_1 f(g, z)\right)^{-1} \\ (h \circ g)'(z) &= \bar{g}^* D_2 f(g, z), \end{aligned}$$

where $\bar{g}^* \in \mathcal{V}^*$ is a dual variable. Computing $\bar{g}^*$ is sometimes called an *adjoint solve*, since in an inner product space this is equivalent to solving a linear system with the adjint of the derivative operator:

$$\bar{g} = \left(D_1 f(g, z)\right)^{-*} \nabla h(g(z)).$$

5.4.6 Variational notation

We will often use a concise notation for differential calculus sometimes called *variational notation* (as in "calculus of variations"). If f is differentiable at x and

$$y = f(x)$$

then in variational notation we would write

$$\delta y = f'(x)\,\delta x.$$

Here δx and δy are read as "variation in x" and "variation in y," and we describe the process of differentiating as "taking variations of $y = f(x)$." If we think of differentiable functions $\tilde{x}(s)$ and $\tilde{y}(s)$ with $x = \tilde{x}(0)$ and $y = \tilde{y}(0)$, then $\delta x = \tilde{x}'(0)$ and $\delta y = \tilde{y}'(0)$. Put differently,

$$\begin{aligned}\tilde{x}(\epsilon) &= x + \epsilon\,\delta x + o(\epsilon),\\ \tilde{y}(\epsilon) &= y + \epsilon\,\delta y + o(\epsilon),\end{aligned}$$

and similarly for other related quantities. If x and y have types that can be added, scaled, and multiplied together, we also have the standard rules

$$\begin{aligned}\delta(x+y) &= \delta x + \delta y\\ \delta(\alpha x) &= \alpha\,\delta x\\ \delta(xy) &= \delta x\, y + x\,\delta y.\end{aligned}$$

This last is derived in the way that we usually do:

$$(x + \epsilon\,\delta x + o(\epsilon))(y + \epsilon\,\delta y + o(\epsilon)) = xy + \epsilon(\delta x\, y + x\,\delta y) + o(\epsilon).$$

We note that this is true even for things like linear maps of the right types, which can be multiplied but do not commute.

Variational notation is sometimes tidier than other ways of writing derivative relationships. For example, suppose $A \in L(\mathcal{V}, \mathcal{V})$ is an invertible linear map and we are interested in the variation in $B = A^{-1}$ with respect to the variation in A. We can see B as a differentiable function of A by the implicit function theorem on the equation

$$AB - I = 0.$$

Taking variations of this relationship, we have

$$(\delta A)B + A(\delta B) = 0,$$

which we can rearrange to

$$\delta B = -A^{-1}(\delta A)A^{-1}.$$

The formula for taking variations of A^{-1} generalizes the 1D formula $(x^{-1})' = -x^{-2}$, which is often used as a first example of implicit differentiation in introductory calculus courses. A short computation verifies that our formula agrees with a finite difference estimate for a small example.

```
let
    A =  [ 94.0  29.0  26.0 ;
           65.0  25.0  66.0 ;
           85.0  92.0  72.0 ]
    δA = [ 57.0  15.0  46.0 ;
           56.0  44.0  79.0 ;
           42.0  45.0  48.0 ]
    δB = -A\δA/A
    δB_fd = finite_diff(inv, A, δA)
    isapprox(δB, δB_fd, rtol=1e-6)
end
```

```
true
```

We *could* work out the same formula with the notation introduced in the previous sections, but the version involving variational notation is arguably easier to read and certainly shorter to write.

5.4.7 Adjoints

Consider a map μ taking an vector input x to an scalar output y via a vector intermediates u and v, written in terms of the relations

$$\begin{aligned} u &= f(x) \\ v &= g(x,u) \\ y &= h(x,u,v) \end{aligned}$$

Then we have the derivative relationships

$$\begin{aligned} \delta u &= D_1 f \, \delta x \\ \delta v &= D_1 g \, \delta x + D_2 g \, \delta u \\ \delta y &= D_1 h \, \delta y + D_2 g \, \delta u + D_3 g \, \delta v \end{aligned}$$

We can rearrange this into the (quasi)matrix form

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -D_1 f & 1 & 0 & 0 \\ -D_1 g & -D_2 g & 1 & 0 \\ -D_1 h & -D_2 h & -D_3 h & 1 \end{bmatrix} \begin{bmatrix} \delta x \\ \delta u \\ \delta v \\ \delta y \end{bmatrix} = \begin{bmatrix} I \\ 0 \\ 0 \\ 0 \end{bmatrix} \delta x.$$

That is, we the computation of δy and then δz from δx as an example of *forward substitution* for a 4-by-4 linear system.

If we truly only care about the relationship between x and y, there is no reason why we should explicitly compute the intermediate δy. An equally good way to solve the problem is to solve the *adjoint equation* discussed in Section 5.4.5:

$$\begin{bmatrix} \bar{x}^* & \bar{u}^* & \bar{v}^* & \bar{y}^* \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -D_1 f & 1 & 0 & 0 \\ -D_1 g & -D_2 g & 1 & 0 \\ -D_1 h & -D_2 h & -D_3 h & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}.$$

via *backward substitution* on the 4-by-4 linear system:

$$\begin{aligned} \bar{y}^* &= 1 \\ \bar{v}^* &= \bar{y}^* D_3 h \\ \bar{u}^* &= \bar{v}^* D_2 g + \bar{y}^* D_2 h \\ \bar{x}^* &= \bar{u}^* D_1 h + \bar{v}^* D_1 g + \bar{y}^* D_1 h. \end{aligned}$$

We observe that

$$\delta y = \bar{x}^* \, \delta x;$$

that is, $\bar{x}^*$ is the derivative of the map from x to z.

While the variations δy and δz lie in the same space as y and z, the dual variables $\bar{u}^*, \bar{v}^*, \bar{y}^*$ (also called adjoint variables) live in the associated dual spaces. The dual variables can also be interpreted as the sensitivity of the output y to the associated primary variable, *assuming earlier variables were held constant.*

The idea of using dual variables to compute derivatives of a function with many intermediates (rather using variations as intermediate quantities) is the basis of *adjoint mode* or *reverse mode* automatic differentiation, which we will discuss further in Chapter 11.

5.4.8 Higher derivatives

5.4.8.1 Maps from $\mathbb{R}$ to $\mathbb{R}$

A main reason — if not *the* main reason — why we care about higher derivatives is because of their role in approximation.

As a particular starting point, consider $g : \Omega \subset \mathbb{R} \to \mathbb{R}$ on an interval Ω. Assuming g has at least $k+1$ continuous derivatives (i.e. $g \in C^{k+1}$), then Taylor's theorem with integral remainder gives

$$g(t) = \sum_{j=0}^{k} \frac{1}{j!} g^{(j)}(t) + r_{k+1}(t),$$

where

$$r_{k+1}(t) = \int_0^t \frac{1}{k!} (t-s)^k g^{(k+1)}(s) \, ds.$$

This formula can be verified with integration by parts. By the mean value theorem, there is some $\xi \in (0, t)$ such that

$$r_{k+1}(t) = \frac{t^{k+1}}{(k+1)!} g^{(k+1)}(\xi).$$

This is the mean value form of the remainder.

We will sometimes work with slightly less regular functions, e.g. $g : \mathbb{R} \to \mathbb{R}$ with k continuous derivatives and $g^{(k)}$ Lipschitz with constant C (which implies absolute continuity of $g^{(k)}$). In this case, we do not have the mean value form of the remainder, but can still use the integral form to get the bound

$$|r_{k+1}(t)| \leq \frac{C|t|^{k+1}}{(k+1)!}.$$

Even for C^{k+1} functions, these types of bounds are sometimes easier to work with than the mean value form of the remainder.

5.4.8.2 Maps from $\mathcal{V}$ to $\mathbb{R}$

If $f : \mathcal{V} \to \mathbb{R}$ is differentiable in an open set Ω containing x, the Frechet derivative is a function

$$f' : \mathcal{V} \to \mathcal{V}^* = L(\mathcal{V}, \mathbb{R})$$

where $f'(x) \in \mathcal{V}^*$ maps directions to directional derivatives. If f' is differentiable, then

$$f'' : \mathcal{V} \to L(\mathcal{V}, \mathcal{V}^*).$$

The mappings $f''(x)$ from $\mathcal{V}$ to $\mathcal{V}^*$ can also be thought of as a bilinear form from two vectors in $\mathcal{V}$ to $\mathbb{R}$, i.e. $(u, v) \mapsto (f''(x)u)v$. Hence, we write

$$f'' : \mathcal{V} \to L(\mathcal{V} \otimes \mathcal{V}, \mathbb{R}).$$

When f'' is continuously defined in some neighborhood of x, the bilinear form is also guaranteed to be symmetric, i.e.

$$f''(x)\,(u \otimes v) = f''(x)(v \otimes u).$$

As discussed in Chapter 4, there is a 1-1 correspondence between symmetric bilinear forms and quadratic forms, and we will mostly be interested in $f''(x)$ as representing the quadratic $u \mapsto f''(x)\,(u \otimes u)$.

The matrix representation of $f''(x)$ with respect to a particular basis V is called the *Hessian* matrix, and is sometimes written $H_f(x)$. The entries of this matrix are $(H_f(x))_{ij} = f''(x)(v_i \otimes v_j)$, so that we write evaluation of the Hessian on a pair of vectors as

$$f''(x)(Vc \otimes Vd) = d^* H_f(x) c.$$

In most cases, we are interested in evaluating the quadratic form associated with $f''(x)$, i.e.

$$f''(x)(Vc \otimes Vc) = c^* H_f(x) c.$$

Hessian matrices will play a central role in our discussion of numerical optimization methods.

In thinking about first derivatives, we started by considering differentiation along a straight ray. We will do the same thing when thinking about using second derivatives. Suppose $c_{x,u}(s) = x + su$ and $c_{x,u}([0,t]) \in \Omega$, and let $g = f \circ c_{x,u}$. Suppose $f \in C^2(\Omega)$ where f'' is Lipschitz with constant M (with respect to an induced norm); that is:

$$\begin{aligned} \|f''(x)\| &= \max_{\|w\|=1} |f'(x)(w \otimes w)| \\ \|f''(x) - f''(y)\| &\leq M\|x - y\|. \end{aligned}$$

Then Taylor's theorem with remainder gives

$$g(t) = g(0) + g'(0)t + \frac{1}{2}g''(0)t^2 + r(t), \quad |r(t)| \leq \frac{M\|u\|^3}{6}t^3.$$

where

$$\begin{aligned} g'(0) &= f'(0)u \\ g''(0) &= f''(0)\,(u \otimes u). \end{aligned}$$

We usually skip the intermediate function g, and write that when f'' is Lipschitz and Ω is convex (so that the line segment connecting $x, u \in \Omega$ will lie entirely within Ω) then

$$f(x+u) = f(x) + f'(x)u + \frac{1}{2}f''(x)(u \otimes u) + O(\|u\|^3),$$

where we are only dealing with the asymptotics of the third-order term. When dealing with a concrete space $\mathbb{R}^n$ (or when working in terms of a basis for $\mathcal{V}$), we usually write

$$f(x+u) = f(x) + f'(x)u + \frac{1}{2}u^* H_f(x) u + O(\|u\|^3).$$

One can continue to take higher derivatives in a similar manner — e.g., if f'' is differentiable, then $f'''(x)$ is a trilinear form in $L(\mathcal{V} \otimes \mathcal{V} \otimes \mathcal{V}, \mathbb{R})$, and we represent it with respect to a basis V as a concrete tensor with entries that we usually write as $f_{,ijk} = f'''(x)\,(v_i \otimes v_j \otimes v_k)$. In the rare cases when we use more than second derivatives, we often revert to indicial notation for computing with functions on concrete spaces; with the summation convention in effect, the Taylor series becomes

$$f(x+u) = f(x) + f_{,i}(x)u_i + \frac{1}{2}f_{,ij}(x)u_i u_j + \frac{1}{6}f_{,ijk}(x)u_i u_j u_k + \dots.$$

The size and notational complexity goes up significantly as we move beyond second derivatives, enough so that we avoid higher derivatives if we can.

5.4.8.3 General maps

We now consider the case of $f : \mathcal{V} \to \mathcal{W}$. If f is C^2 on Ω, then the second derivative is

$$f'' : \mathcal{V} \to L(\mathcal{V}, L(\mathcal{V}, \mathcal{W})) \equiv L(\mathcal{V} \otimes \mathcal{V}, \mathcal{W}),$$

and the bilinear map $f''(x)$ is symmetric in the arguments, i.e.

$$f''(x)\,(u \otimes v) = f''(x)\,(v \otimes u).$$

Expanding to second order with remainder again looks like

$$f(x+u) = f(x) + f'(x)u + \frac{1}{2}f''(x)(u \otimes u) + O(\|u^3\|).$$

But when we want to compute with concrete quantities, we need three indices even to keep track of the second derivative term: with respect to bases V and W, we write the concrete tensor

$$f_{i,jk} = \left[W^{-1} f(x)(v_j \otimes v_k)\right]_i$$

or, equivalently (using indices and the summation convention),

$$f(x)(Vc \otimes Vd) = \mathbf{w}_i f_{i,jk} c_j d_k.$$

5.4.9 Analyticity

So far, we have considered functions on real vector spaces. But for some of what we have to say about matrix calculus later, it will be useful to also consider functions on the complex plane.

We begin with an algebra fact. Suppose that $z = x + iy$ and $c = a + ib$ are complex numbers written in rectangular form; then

$$cz = \begin{bmatrix} 1 & i \end{bmatrix} \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

That is, the matrix mapping the real and imaginary components of z to the real and imaginary components of cz has the form $aI + bJ$ where I is the identity and J is the 2-by-2 skew matrix

$$J = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

Hence, we can interpret complex multiplication as a 2-by-2 matrix on the rectangular coordinate parameterization of the reals — but it is a matrix of a very specific type.

Now let $f : \mathbb{C} \to \mathbb{C}$. We would like to say that f is differentiable at z if

$$f(z+w) = f(z) + f'(z)w + o(|w|).$$

Phrased in terms of rectangular coordinates ($f = g + ih$, $z = x + iy$, $w = u + iv$), we have that

$$\begin{bmatrix} D_1 g(x,y) & D_2 g(x,y) \\ D_1 h(x,y) & D_2 h(x,y) \end{bmatrix} = \begin{bmatrix} a(x,y) & -b(x,y) \\ b(x,y) & a(x,y) \end{bmatrix}$$

i.e. the partial derivatives of the real and imaginary components of f must satisfy the *Cauchy-Riemann* equations

$$D_1 g = D_2 h, \quad D_2 g = -D_1 h.$$

A function that is complex differentiable on an open set Ω is called *holomorphic* (or *(complex) analytic*) on that set.

Holomorphic functions are about as nice as functions can be. Among other nice properties:

- They are automatically infinitely (complex) differentiable,
- They have convergent Taylor series about any point in the domain Ω,
- The real and imaginary components are harmonic functions (i.e. $D_1^2 g + D_2^2 g = 0$, and similarly for h), and
- Contour integration around any loop Γ whose interior is wholly within Ω gives $\int_\Gamma f(z)\, dz = 0$.

We will heavily rely on this last fact (and related facts) when dealing with the contour integral view of functions of matrices in Section 5.7.

5.5 Series

A *series* is an infinite sequence of terms $v_1, v_2, \ldots$ that are (formally) added together. When students first learn about series in introductory calculus courses, the terms v_j are typically real or complex numbers. We will consider the more general case where the v_j are drawn from some normed vector space $\mathcal{V}$, which we will often assume is complete (a Banach space). The partial sums are

$$S_n = \sum_{j=1}^{n} v_j;$$

and when $\mathcal{V}$ is a Banach space and the partial sums form a Cauchy sequence, we say the series *converges* to

$$S = \sum_{j=1}^{\infty} v_j = \lim_{n \to \infty} S_n.$$

Series that do not converge (divergent series) are still useful in many situations; we simply need to understand that they are to be treated formally, and we might not be able to make sense of the limiting value.

5.5.1 Convergence tests

Many of the standard convergence tests for series in $\mathbb{R}$ or $\mathbb{C}$ generalize to series in a Banach space. The *comparison test* is of particular use:

Theorem 5.2 (Comparison test)**.** *Suppose $\{v_j\}_{j=1}^{\infty}$ is a sequence in a Banach space $\mathcal{V}$, where the terms are bounded by $\|v_j\| \leq a_j$. Suppose $\sum_{j=1}^{\infty} a_j$ converges. Then $\sum_{j=1}^{\infty} v_j$ also converges absolutely; moreover,*

$$\left\| \sum_{j=1}^{\infty} v_j \right\| \leq \sum_{j=1}^{\infty} a_j.$$

Proof. Let A_n and A denote the nth partial sum and the limit for the series $\sum_{j=1}^{\infty} a_j$, and let S_n denote the nth partial sum for the series $\sum_{j=1}^{\infty} v_j$. By the triangle inequality, for any $i \geq j$, we have

$$\|S_j - S_i\| = \left\| \sum_{k=i+1}^{j} v_k \right\| \leq \sum_{k=i+1}^{j} \|v_k\| \leq A_j - A_i.$$

The fact that the partial sums A_n form a Cauchy sequence means that for any $\epsilon > 0$, there exists N such that if $i \geq j \geq N$,

$$|A_j - A_i| < \epsilon;$$

and because $\|S_j - S_i\| \leq |A_j - A_i|$, the partial sums S_n also form a Cauchy sequence, and therefore converge to a limit S.

The final bound comes from applying the triangle inequality to show

$$\|S_n\| = \left\| \sum_{j=1}^{n} v_j \right\| \leq \sum_{j=1}^{n} \|v_j\| \leq A_n,$$

and from there arguing that the limit $\|S\|$ is bounded by the limit A. □

Many other convergence tests similarly generalize to series in Banach spaces, often using the comparison test as a building block. An example is the ratio test:

Theorem 5.3 (Ratio test)**.** *Suppose $\{v_j\}_{j=1}^{\infty}$ is a sequence in a Banach space $\mathcal{V}$, where $\lim_{n\to\infty} \|v_{n+1}\|/\|v_n\| = r < 1$. Then $\sum_{j=1}^{\infty} v_j$ converges absolutely.*

Proof. Let $r < r' < 1$ (e.g. choose $r' = (r+1)/2$). Then by the hypothesis, there exists some N such that for all $n \geq N$, $\|v_{n+1}\|/\|v_n\| \leq r'$. Hence, for $n \geq N$ we have $\|v_n\| \leq (r')^{n-N}\|v_N\|$,

and by the comparison test and convergence of the geometric series, the series $\sum_j v_j$ converges, and moreover we have the bound

$$\begin{aligned}\left\|\sum_{n=1}^{\infty} v_n\right\| &\leq \left\|\sum_{n=1}^{N-1} v_n\right\| + \sum_{n=N}^{\infty} (r')^{n-N}\|v_N\| \\ &= \left\|\sum_{n=1}^{N-1} v_n\right\| + \frac{\|v_N\|}{1-r'}.\end{aligned}$$

□

5.5.2 Function series

Frequently, we are interested in series where the terms are functions of some variable, e.g. $v_j : \Omega \subset \mathbb{C} \to \mathcal{V}$ for some Banach space $\mathcal{V}$. Often the functional dependence of the terms on v_j is fairly simple, e.g. $v_j(z) = \bar{v}_j z^{p_j}$ where the exponents p_j may be just positive integers (a power series), positive and negative integers (a Laurent series), or fractions with a fixed denominator (a Puiseaux series). We also often see terms of the form $v_n(z) = \bar{v}_n \exp(inz)$ (a Fourier series).

When the terms themselves belong to some Banach space, we can analyze the convergence or divergence of the series as a whole just as we would analyze the convergence or divergence of any other series over a Banach space. For example, if Ω is compact, then the set of continuous functions $C(\Omega, \mathcal{V})$ (which are automatically uniformly continuous) is a Banach space. Convergence in this Banach space corresponds to uniform convergence of the functions, and uniform convergence of uniformly continuous functions gives a uniformly continuous limit, as noted before.

However, more generally sometimes the terms may not belong to a Banach space; or if they do belong to a Banach space, they might diverge. In this case, we might care about the convergence or divergence of the series $\sum_{j=1}^{\infty} v_j(z)$ for *specific* values of z; and, if we have pointwise convergence over some subset $\Omega' \subset \Omega$, we care what properties the limiting function $S : \Omega' \to \mathcal{V}$ might inherit from the terms. This issue can be subtle: for example, a Fourier series can easily converge at almost all points in some interval, but converge to a discontinuous function despite the fact that all the terms are infinitely differentiable.

Function series can also be extremely useful even where they diverge, both for formal manipulation and for actual calculations where the approximation properties of finite sums are more important than the limiting behavior of the series.

5.5.3 Formal series

A *formal power series* is a series of the form

$$S(z) = \sum_{j=0}^{\infty} a_j z^j$$

where the coefficients a_j belong to some appropriate vector space $\mathcal{V}$. In some circumstances, it also makes sense to include negative powers, in which case we sometimes refer to this as a formal Laurent series. Formal power series appear in a variety of applications. We will mostly see them in signal processing, where such a formal power series is also called the z-transform of the sequence of coefficients a_j. In that setting, is it particularly useful not only to add and scale formal power series, but also to multiply them together using the rule

$$\left(\sum_{j=0}^{\infty} a_j z^j\right)\left(\sum_{j=0}^{\infty} b_j z^j\right) = \sum_{k=0}^{\infty}\left(\sum_{i+j=k} a_i b_j\right) z^k;$$

that is, the coefficients in the product are the *convolution* of the coefficient sequences of the multiplicand series. Hence, the power series can be seen as a convenient way of dealing with the book-keeping of tracking sequences and their convolutions, independent of whether things converge. Formal power series (along with other formal function series) also play a key role in combinatorics and statistics, often under the name of "generating functions" (see Wilf (2006)).

If $\mathcal{V}$ is a Banach space and $\log \|a_n\| = O(n)$, the ordinary formal power series associated with the coefficient sequence a_j will converge for values of $z \in \mathbb{C}$ close enough to zero. Under such assumptions (which often hold), we can treat the formal power series as defining a function, and can bring the machinery of analytic function theory to bear. Nonetheless, it is helpful to distinguish the formal power series — essentially a trick for indexing elements of a series — from the associated function to which it converges. Only the former is needed for tracking convolutions and the like.

5.5.4 Asymptotic series

An *asymptotic series* for a function f is a formal function series with terms $v_n(z)$ such that

$$f(z) - \sum_{n=0}^{N-1} v_n(z) = o(\|\phi_{N-1}(z)\|)$$

for some limiting behavior in z (usually $z \to 0$ or $z \to \infty$). Asymptotic series often do not converge, but are nonetheless useful for approximation.

A standard example of an asymptotic series is Stirling's approximation for the gamma function (generalized factorial function):

$$\Gamma(z+1) \sim \sqrt{2\pi z}\left(\frac{z}{e}\right)^z \left(1 + \frac{1}{12}z^{-1} + \frac{1}{288}z^{-2} - \frac{139}{51840}z^{-3} - O(z^{-4})\right).$$

Here the symbol $\sim$ is used to denote that the series is asymptotic to the function, but does not necessarily converge to the function in the limit of an infinite number of terms. Another standard example is the asymptotic expansion for the tail of the standard normal cdf:

$$1 - \Phi(z) = \phi(z)z^{-1}\left(1 - z^{-2} + 3z^{-4} + O(z^{-6})\right).$$

Both these methods are derived via Laplace's method, which is also very useful for approximating integrals arising in Bayesian inference. These types of asymptotic approximations are typically complementary to numerical methods, providing very accurate estimate precisely in the regions where more conventional numerical approaches have the most difficulty.

5.5.5 Analytic functions

While formal series and asymptotic approximation are nice, even nicer things can happen for power series that converge. When a series

$$f(z) = \sum_{j=0}^{\infty} c_j (z - z_0)^j$$

converges in a neighborhood of z_0 (in $\mathbb{R}$ or $\mathbb{C}$), we say the function f is *analytic* at z_0. Such a function is automatically infinitely differentiable in a neighborhood of z_0. However, there are (useful) examples of non-analytic functions on $\mathbb{R}$ that are infinitely differentiable, e.g.

$$g(x) = \begin{cases} \exp(-1/x), & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Complex analyticity around a point $z_0 \in \mathbb{C}$ (existence of a convergent Taylor series) and the property of being *holomorphic* around a point (having complex derivatives that satisfy the Cauchy-Riemann equations) are equivalent to each other. Functions that are real analytic in the neighborhood of a point in $\mathbb{R}$ are also complex analytic in a neighborhood. However, a function that is real analytic over all of $\mathbb{R}$ may not extend to a function that is complex analytic over all of $\mathbb{C}$.

The property of analyticity at a point z_0 is closed under addition, scaling, multiplication of functions, integration, differentiation, and composition. It is also closed under division, assuming that the denominator function is nonzero at z_0.

5.5.6 Operator functions

Just as we can compute powers of $z \in \mathbb{C}$, we can compute powers of $A \in L(\mathcal{V}, \mathcal{V})$. Indeed, as we noted in **?@sec-tbd**, polynomials of an operator are enormously useful both as a theoretical tool (e.g. for studying eigenvalues) and as a numerical tool (e.g. in the design of Krylov subspace methods, which we will see later in this book). Unsurprisingly, taking limits of sequences of polynomials is also useful, and power series expressions for analytic functions on $\mathbb{C}$ work equally well for describing analytic functions on operator spaces.

For our purposes, we will mostly consider $\mathcal{V}$ to be a finite-dimensional normed space, but the concept works equally well for general Banach spaces. In either case, the space of bounded operators $L(\mathcal{V}, \mathcal{V})$ is itself a Banach space, and our earlier comments about convergent series in general Banach spaces holds. For example, using a consistent family of norms (e.g. the operator norm), consider

$$\exp(A) = \sum_{k=0}^{\infty} \frac{1}{k!} A^k;$$

by comparison to the series for $\exp(\|A\|)$, this series converges absolutely, and

$$\|\exp(A)\| \leq \exp(\|A\|).$$

We observe that if we have an eigenpair $Av = v\lambda$, then

$$\exp(A)v = \sum_{k=0}^{\infty} \frac{1}{k!} A^k v = \sum_{k=0}^{\infty} \frac{1}{k!} v\lambda^k = v \exp(\lambda).$$

When A is diagonalizable (i.e. $A = V\Lambda V^{-1}$), then we can completely characterize $\exp(A)$ as

$$\exp(A) = V \exp(\Lambda) V^{-1},$$

where $\exp(\Lambda)$ is the diagonal matrix of exponentials of eigenvalues.

What works for the exponential tends to work for more general analytic functions f. Things get a little more complicated for operators with nontrivial Jordan blocks (and certainly for the more elaborate spectra that can occur in the infinite-dimensional case), but the fundamental picture remains the same: an analytic function f makes sense as a function of an operator when the spectrum is inside the domain of analyticity; and $f(A)$ will have the same eigenvector structure as A while eigenvalues are transformed by the mapping $\lambda \mapsto f(\lambda)$.

5.5.7 Neumann series

While the exponential provides a nice example of an operator power series, we will see far more of the *Neumann series*, which is the generalization of the geometric series. When $\|A\| < 1$ under any consistent norm, we have that the Neumann series

$$(I - A)^{-1} = \sum_{k=0}^{\infty} A^k$$

converges by comparison with a geometric series in $\|A\|$; and (courtesy that same comparison), that

$$\|(I-A)^{-1}\| \leq \sum_{k=0}^{\infty} \|A\|^k = \frac{1}{1-\|A\|}.$$

With a little supplementary algebra, this Neumann series bound is the basis for many inequalities that will prove useful later. As an example left as an exercise for the student, we observe that when A is invertible and $\|A^{-1}E\| < 1$, we have

$$\|(A+E)^{-1}\| \leq \frac{\|A^{-1}\|}{1-\|A^{-1}E\|}.$$

5.6 Integration

5.6.1 Measure and integration

5.6.2 Standard inequalities

5.6.3 Change of variables

5.7 Contour integrals

5.7.1 Poles and residues

5.7.2 Resolvent calculus

5.8 Function spaces

6 Optimization theory

6.1 Optimality conditions

We will frequently phrase model fitting problems in terms of optimization, which means we need to remember how optimality conditions work. This includes not only remembering first-order and second-order optimality conditions for derivatives, but also knowing a little about optimality conditions for non-differentiable functions. The key concept behind optimality conditions is to locally approximate a function in terms of a local polynomial approximation, where it usually suffices to go out to first or second order.

6.1.1 Minima and maxima

Let $f : \Omega \subset \mathcal{V} \to \mathbb{R}$ be a continuous function. The set Ω is the *feasible* set. We say the problem of minimizing or maximizing f is an *unconstrained* optimization problem when $\Omega = \mathcal{V}$ (all points in the space are feasible). Otherwise, it is a *constrained* optimization problem.

We say $x \in \Omega$ is a *local minimizer* if any $y \in \Omega$ close enough to x satisfies $f(y) \geq f(x)$ (more precisely: there is some $\epsilon > 0$ so that for all y within ϵ of x, $f(y) \geq f(x)$). The value $f(x)$ is the local minimum value. For a *strong local minimizer*, the inequality is strict – any $y \neq x$ in a neighborhood of x satisfies $f(y) > f(x)$. We say x is a *global* minimizer if $f(x) \leq f(y)$ for all $y \in \Omega$, and a *strong* global minimizer if $f(x) < f(y)$ for all $y \in \Omega$ other than x.

We can define a local maximizer or a strong local maximizer analogously, but we usually focus on the case of minimization rather than maximization (and in any event, maximizing f is the same as minimizing $-f$).

When the feasible set Ω is *compact* (for finite-dimensional vector spaces, this means closed and bounded), we are guaranteed that there is a global minimizer in Ω. In other cases, f may not be bounded below, or it may be bounded below but with no point where the greatest lower bound (the *infimum*) is achieved.

6.1.2 Derivative and gradient

If $f : \mathcal{V} \to \mathbb{R}$ is differentiable, a *stationary* point is a point x such that $f'(x) = 0$. At a non-stationary point, there is guaranteed to be some direction u such that $f'(x)u > 0$ (and $f'(x)(-u) < 0$), so that f can neither attain an (unconstrained) minimum nor maximum at that point. Stationary points can be minima, maxima, or saddles; we usually classify them as such by the *second derivative test*.

In an inner product space, the gradient $\nabla f(x)$ (the Riesz map of $f'(x)$) gives us the "direction of steepest ascent," and the negative gradient gives us the direction of steepest descent. It is important to realize that this direction depends on the inner product used! Moreover, the concept of steepest ascent or descent generalizes to other normed linear spaces where we do not assume an inner product: all we need is the notion of the change in the function value relative to the distance from a starting point. For example, if $\mathcal{V}$ is $\mathbb{R}^n$ with the ℓ^∞ norm, then the direction of steepest descent (or *a* direction, if $f'(x)$ has some zero components) is given by a vector of s where $s_j = \text{sign}(f'(x)_j)$; here s is the vector of unit l^∞ norm such that $f'(x)s$ is maximal.

6.1.3 Second derivative test

When $f : \mathcal{V} \to \mathbb{R}$ is twice differentiable, we can characterize the function near a stationary point x by the second derivative:

$$f(x+u) = f(x)\frac{1}{2}f''(x)(u \otimes u) + o(\|u\|^2).$$

The point x is a local minimizer if the quadratic form given by $f''(x)$ is positive definite and a local maximizer if $f''(x)$ is negative definite. If the Hessian is semidefinite, we need to look at higher derivatives to classify x. If $f''(x)$ is strongly indefinite (i.e. the inertia has nonzero positive and negative components), then x is a *saddle point*.

For computation, of course, we usually express $f''(x)$ with respect to a basis. In this case, we describe the second derivative test in terms of the inertia of the *Hessian matrix* $H_f(x)$.

6.1.4 Constraints and cones

We have described the first and second derivative tests in the context of unconstrained optimization, but the general approach is equally sensible for constrained optimization problems. We generally define the feasible domain Ω in terms of a set of *constraints*, equations and inequalities involving functions $c_i : \mathcal{V} \to \mathbb{R}$ that (for our purposes) are at least continuously differentiable:

$$\begin{aligned} c_i(x) &= 0, \quad i \in \mathcal{E}, \\ c_i(x) &\leq 0, \quad i \in \mathcal{I}. \end{aligned}$$

We say the ith inequality constraint is *active* at x if $c_i(x) = 0$. We write the active set at x as

$$\mathcal{A}(x) = \{i \in \mathcal{I} : c_i(x) = 0\}.$$

We do not worry about trying to classify the equality constraints, though in a sense they are always active.

6.1.4.1 Tangent cone

The first derivative test tells us that if f is locally minimized at x, then a local linear model should not predict a nearby point with smaller function values. That is, there should not be $f'(x)u < 0$ and a (differentiable) feasible path $\gamma : [0, \tau) \to \mathcal{V}$ with $\gamma(0) = x$ and $\gamma'(0) = u$, or we will have

$$f(\gamma(\epsilon)) = f(x) + \epsilon f'(x)u + o(\epsilon) < f(x).$$

for all sufficiently small $\epsilon > 0$. The set of directions u that are tangent to some feasible path at x is known as the *tangent cone* at x. A function f satisfies a (first order) *geometric optimality condition* at x if

$$\forall u \in T(x), f'(x)u \geq 0.$$

where $T(x)$ denotes the tangent cone. Equivalently,

$$-f'(x) \in T(x)^o,$$

where $T(x)^o$ is the *polar cone* of $T(x)$, i.e.

$$T(x)^o = \{w^* \in \mathcal{V}^* : \forall u \in T(x), w^* u \leq 0\}.$$

A local minimizer must satisfy this condition, though not all points that satisfy this condition need be local minimizers.

6.1.4.2 Linearized cone

The trouble with the geometric optimality condition is that we somehow need to characterize the tangent cone at each feasible point. Because we are characterizing the feasible set in terms of differentiable constraint functions, it is tempting to try to characterize feasible directions via the derivatives of the constraints by looking at the *linearized cone*

$$\begin{aligned} L(x) = \{u \in \mathcal{V} : \quad & (\forall i \in \mathcal{E}, c_i'(x)u = 0) \wedge \\ & (\forall i \in \mathcal{A}(x), c_i'(x)u \leq 0)\}. \end{aligned}$$

It is true that $T(x) \subseteq L(x)$, and the polars satisfy $L(x)^o \subseteq T(x)^o$. Unfortunately, the linearized cone can be strictly bigger than the tangent cone. This is true even in 1D. For example, consider the domain $x \geq 0$ written as $c(x) = -x^3 \leq 0$. The tangent cone at $x = 0$ is $T(0) = \{u \in \mathbb{R} : u \geq 0\}$, but the linearized cone is $L(0) = \mathbb{R}$. Hence, if we seek to minimize $f(x) = x$ subject to $-x^3 \leq 0$, the point $x = 0$ satisfies the geometric optimality condition, but the condition is not satisfied if we replace the tangent cone with the linearized cone.

6.1.4.3 Constraint qualification

A *constraint qualification condition* is a hypothesis on the constraints that guarantees that $L^o(x) = T^o(x)$, so that we can use the linearized cone in lieu of the tangent cone in the geometric optimality condition. Two common examples are the linearly independent constraint qualification (LICQ), which holds at x if

$$\{c_i'(x) : i \in \mathcal{A}(x) \cup \mathcal{E}\} \text{ is linearly independent.}$$

and the Mangasarian-Fromovitz constraint qualification (MFCQ), which holds at x if

$$\begin{aligned}&\{c_i'(x) : i \in \mathcal{E}\} \text{ is linearly independent and}\\ &\exists u \in \mathcal{V} : c_i'(x)u < 0 \text{ for } i \in \mathcal{A}(x) \text{ and } c_i'(x)u = 0 \text{ for } i \in \mathcal{E}.\end{aligned}$$

Note that our example of the constraint $-x^3 \leq 0$ satisfies neither LICQ nor MFCQ at 0.

6.1.5 Multipliers and KKT

We have already seen one theorem of alternatives in our discussion of linear algebra — the Fredholm alternative theorem, which deals with solvability of linear equations. There are many other theorems of alternatives for dealing with inequalities, associated with mathematicians like Motzkin, Kuhn, and Farkas. One such theorem is *Farkas' lemma*: if $A \in \mathbb{R}^{m\times n}$ and $b \in \mathbb{R}^m$, then exactly one of the two statements is true:

- There exists $x \in \mathbb{R}^n$ such that $Ax = b$ and $x \geq 0$
- There exists $y \in \mathbb{R}^m$ such that $y^*A \geq 0$ and $y^*b < 0$

Here the inequalities are taken elementwise.

Using Farkas's lemma, one can rewrite the polar of the linearized cone as

$$L(x)^o = \left\{ w^* \in \mathcal{V}^* : w^* = \sum_{i\in\mathcal{E}} \lambda_i c_i'(x) + \sum_{i\in\mathcal{A}(x)} \mu_i c_i'(x), \mu_i \geq 0 \right\}.$$

It is usually convenient to define $\mu_i = 0$ for inequality constraints that are inactive; in this case, we can rewrite

$$\begin{aligned}L(x)^o = \{w^* \in \mathcal{V}^* : \quad & w^* = \lambda^* c_{\mathcal{E}}'(x) + \mu^* c_{\mathcal{I}}'(x),\\ & \mu^* c_{\mathcal{I}}(x) = 0, \mu^* \geq 0\},\end{aligned}$$

where $c_{\mathcal{E}}(x)$ and $c_{\mathcal{I}}(x)$ are (column) vectors of equality and inequality constraint functions and λ^* and μ^* are concrete row vectors. The statement that $\mu^* c_{\mathcal{I}}(x) = 0$, typically called a *complementary slackness condition* is equivalent to saying that inactive inequality constraints (where $c_i(x) < 0$) must be associated with zero multipliers.

With this machinery in place, we can now rewrite the condition $-f'(x) \in L(x)^o$ in the form that we usually use. We define the *Lagrangian* function

$$\mathcal{L}(x, \mu^*, \lambda^*) = f(x) + \lambda^* c_{\mathcal{E}}(x) + \mu^* c_{\mathcal{I}}(x)$$

The variables λ^* and μ^* that multiply the constraints are known as *Lagrange multipliers*. The *Karush-Kuhn-Tucker* (KKT) conditions at x_*, equivalent to $-f'(x_*) \in L(x^*)^o$ are

$$\begin{aligned} D_1\mathcal{L}(x_*, \lambda, \mu) = 0 & \qquad \text{constrained stationarity} \\ c_{\mathcal{E}}(x_*) = 0, & \qquad \text{primal feasibility (equality)} \\ c_{\mathcal{I}}(x_*) \leq 0, & \qquad \text{primal feasibility (inequality)} \\ \mu \geq 0, & \qquad \text{dual feasibility} \\ \mu^* c_{\mathcal{I}}(x_*) = 0, & \qquad \text{complementary slackness.} \end{aligned}$$

When a constraint qualification condition holds, the KKT conditions are necessary first-order conditions for optimality at x_*.

6.1.6 Multipliers and adjoints

Now suppose that we wanted to find a minimum (or maximum or other stationary point) of y subject to the equations we saw in Section 5.4.7

$$\begin{aligned} u - f(x) &= 0 \\ v - g(x, u) &= 0 \\ y - h(x, u, v) &= 0 \end{aligned}$$

The Lagrangian for this system is

$$\mathcal{L} = y - \bar{u}^*(u - f(x)) - \bar{v}^*(v - g(x, u)) - \bar{y}^*(y - h(x, u, v))$$

where x, u, v, y are the primal variables and $\bar{u}^*, \bar{v}^*, \bar{y}^*$ are multipliers[1]. Stationarity gives

$$\begin{bmatrix} \bar{u}^* & \bar{v}^* & \bar{y}^* \end{bmatrix} \begin{bmatrix} -D_1 f & 1 & 0 & 0 \\ -D_1 g & -D_2 g & 1 & 0 \\ -D_1 h & -D_2 h & -D_3 h & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}.$$

We can write this equivalently as $\bar{x}^* = 0$ where

$$\begin{bmatrix} \bar{x}^* & \bar{u}^* & \bar{v}^* & \bar{y}^* \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -D_1 f & 1 & 0 & 0 \\ -D_1 g & -D_2 g & 1 & 0 \\ -D_1 h & -D_2 h & -D_3 h & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}.$$

This is precisely the set of equations that we saw in Section 5.4.7, except that where we said "dual variables" in Section 5.4.7, here we are saying (negative) "Lagrange multipliers."

[1] For equality constraints, the signs of the Lagrange multipliers are unimportant.

6.1.7 Mechanical analogies

6.1.8 Constrained second derivatives

The second-order geometric optimality condition for a twice-differentiable f at x is a straightforward extension of the first-order geometric optimality condition. If x satisfies the first-order condition

$$\forall u \in T(x), f'(x)u \geq 0,$$

and we satisfy that

$$\forall 0 \neq u \in T(x) \text{ s.t. } f'(x)u = 0, f''(x)(u \otimes u) > 0,$$

then x is a constrained local minimum. If there are nonzero directions $u \in T(x)$ where $f'(x)u = 0$ and $f'(x)(u \otimes u) < 0$, then x is *not* a constrained local minimum. Otherwise, we cannot determine minimality without looking at higher derivatives.

As with the geometric first-order condition, the geometric second-order condition is complicated by the need to get a handle on the tangent cone. Assuming that we satisfy the linearly independent constraint qualification condition at x, we have a more straightforward version of the second derivative test. Let

$$\mathcal{J} = \mathcal{E} \cup \{i \in \mathcal{I} : \mu_i > 0\}.$$

Then a sufficient condition for x to be a strong constrained local minimizer is if

$$\forall \text{ nonzero } u \in \mathcal{N}(c'_{\mathcal{J}}(x)), \quad f''(x)(u \otimes u) > 0.$$

That is, the Hessian should be positive definite in all directions that are not already "uphill" at first order. Such a condition is known as *conditional* positive definiteness, since $f''(x)$ is positive definite conditioned on some constraints on the directions considered. If there are such directions for which $f''(x)(u \otimes u) < 0$, then we do not have a local minimizer; otherwise, we may need to consider higher derivatives to make a diagnosis.

6.2 Vector optimization

6.3 Convexity

A *set* $\Omega \subset \mathcal{V}$ is convex if every line segment between points in Ω also lies in Ω, i.e.

$$\forall x, y \in \Omega, \forall s \in [0, 1], (1 - s)x + sy \in \Omega.$$

A *function* $f : \Omega \subset \mathcal{V} \to \mathbb{R}$ is convex if the graph of f on a line segment in Ω always lies below the secant connecting the endpoint values:

$$\forall x, y \in \Omega, \forall s \in [0,1], f((1-s)x + sy) \leq (1-s)f(x) + sf(y).$$

We say f is *strictly* convex if the inequality is strict on the interior of the segment connecting x and y:

$$\forall x \neq y \in \Omega, \forall s \in (0,1), f((1-s)x + sy) < (1-s)f(x) + sf(y).$$

If f is twice differentiable, convexity corresponds to positive semi-definiteness of the Hessian, and strict convexity corresponds to positive definiteness of the Hessian. However, functions can be convex even if they are not twice differentiable.

A convex function on a closed, bounded set in finite dimensions is guaranteed to take on its minimum value $f_{\min}$ at some point $x \in \Omega$. There are no local minimizers, only global minimizers. The set of global minimizers $\{x \in \Omega : f(x) = f_{\min}\}$ is itself a convex set. If f is *strictly* convex, then the global minimizer is unique.

6.3.1 Subderivatives

A *subderivative* of a convex f at $x \in \Omega$ is the set of functionals $w^* \in \mathcal{V}$ such that $f(x+u) \geq f(x) + w^*u$ for all $x + u \in \Omega$; that is, the subderivative corresponds to the set of "supporting hyperplanes" that agree with $f(x)$ and lie below f elsewhere on Ω. When $\mathcal{V}$ is an inner product space, we say $w \in \mathcal{V}$ is in the *subgradient* at $x \in \Omega$ if $f(x+u) \geq f(x) + \langle u, w \rangle$ whenever $x + u \in \Omega$.

When f is differentiable at x, the only element of the subderivative is the derivative (and the only element of the subgradient is the gradient). However, the concept of a subderivative continues to make sense even for *nonsmooth* functions. For example, the absolute value function $x \mapsto |x|$ on $\mathbb{R}$ is not differentiable at 0, but has a subgradient $[-1, 1]$.

The notion of a subderivative allows us to generalize the usual notion of stationary points for differentiable functions: a point $x \in \Omega$ is a stationary point for a convex f if the subderivative of f at x contains the $0 \in \mathcal{V}^*$, and a minimizer must be a stationary point.

7 Probability

I assume that you have seen some probability theory before, and that this is just a reminder. If you need a more thorough refresher, the book by Ross (Ross 2014) is a popular introductory text that covers discrete and continuous problems, but not more general probability measures. Another good undergraduate text by Chung and AitSahlia (Chung and AitSahlia 2003) includes a little bit of measure theory. Good graduate texts include the books by Billingsley (Billingsley 1995) and by Breiman (Breiman 1992). If you want a reminder that is more thorough than the one we give here, but less than a full textbook, the treatment in (Deisenroth, Faisal, and Ong 2020) is a good starting point.

- Axiomatic probability, counting, and measure
- Conditional and marginal probabilities
- Bayesian and frequentist perspectives
- Random variables
- The wonder of Gaussians (and not all RVs are Gaussian!)
- Moments (+ linearity of expectations, variance and precision,heteroscedasticity, etc) and tails
- Independence, conditional independence, factorization of probabilities, graphical models
- Standard statistics (incl variance and precision)
- Central limit theorem / LLN / sums
- Parameter estimation
- Bayes and updating
- Conjugate priors
- Uninformative priors
- Markov chains
- Martingales
- Stochastic processes
- Standard inequalities and concentration of measure

Part II

Fundamentals in 1D

8 Notions of Error

- Error, uncertainty, etc - known unknowns vs unknown unknowns
- Sources of error
- Models of error (worst case, statistical, etc)
- Relative and absolute errors
- Mixed error
- Propagation of error
- Sensitivity and conditioning
- Forward, backward, and residual error
- Applications in data analysis

"Explanatory, not exculpatory" Floating point is not everything!

Testing and debugging numerical codes - Testing vs debugging (draw a picture) - Support for approx, testing framework in the standard library - Random tests and hard tests - Testing from the ground up - It's not all floating point! - Saving random seeds

9 Floating Point

- Fundamentals of floating point (and fixed point!)
- Floating point formats
- Basic arithmetic operations, rules for IEEE
- Exceptional conditions
- Error analysis
- Design principles
- Multiple precision computations
- Simulated extra precision

10 Approximation

- Inputs: function values, sampling oracle (values, derivs, integrals, etc)
- Output: Approximating function, often used for other tasks. Plus some notion of error!
- Q to ask: what data is available? Noise? What is assumed about fn (smoothness, derivatives, etc)?
- What is needed from approximator (derivatives?). How fast is fitting? Evaluation? Convergence?
- Discuss general theory and influence of noise later. Focus here on piecewise polynomials and interpolation thereof.
- Polynomial interpolation and regression
- Different forms of the interpolation problem
- Choice of basis and numerical issues, computational speed
- Error analysis
- Piecewise polynomials: splines and more
- Kernel-based approximation
- Which space to use?
- Rational approximation
- Approximation of densities

11 Automatic Differentiation

- Automatic differentiation: forward, backward, adjoint

11.1 Dual numbers

Computing with variations is not only useful for pen-and-paper computations; it is also the basis for automatic differentiation with so-called *dual numbers*. We describe some of the fundamentals of working with dual numbers below. A more full-featured implementation of the dual numbers is given in the `ForwardDiff.jl` package.

11.1.1 Scalar computations

A *dual number* is a pair $(x, \delta x)$ consisting of a value x and a variation δx. As above, above, we can think of this as a function $\tilde{x}(\epsilon) = x + \epsilon\,\delta x + o(\epsilon)$. For ordinary scalar numbers, we represent this with a Julia structure.

```
struct Dual{T<:Number} <: Number
    val :: T  # Value
    δ :: T    # Variation
end

value(x :: Dual) = x.val
variation(x :: Dual) = x.δ
```

We want to allow constants, which are dual numbers with a zero variation. We can construct these directly or convert them from other numbers.

```
Dual{T}(x :: T) where {T} = Dual{T}(x, zero(T))
Base.convert(::Type{Dual{T}}, x :: S) where {T,S<:Number} =
    Dual{T}(x, zero(T))
```

We also want to be able to convert between types of Julia dual numbers, and we want promotion rules for doing arithmetic that involves dual numbers together with other types of numbers.

```
Base.convert(::Type{Dual{T}}, x :: Dual{S}) where {T,S} =
    Dual{T}(x.val, x.δ)
Base.promote_rule(::Type{Dual{T}}, ::Type{Dual{S}}) where {T,S} =
    Dual{promote_type(T,S)}
Base.promote_rule(::Type{Dual{T}}, ::Type{S}) where {T,S<:Number} =
    Dual{promote_type(T,S)}
```

We would like to define a variety of standard operations (unary and binary) for dual numbers. Rather than writing the same boilerplate code for every function, we write macros `@dual_unary` and `@dual_binary` that take the name of an operation and the formula for the derivative in terms of $x, \delta x, y, \delta y$, where x and y are the first and second arguments, respectively.

```
macro dual_unary(op, formula)
    :(function $op(x :: Dual)
          x, δx = value(x), variation(x)
          Dual($op(x), $formula)
      end)
end

macro dual_binary(op, formula)
    :(function $op(x :: Dual, y :: Dual)
          x, δx = value(x), variation(x)
          y, δy = value(y), variation(y)
          Dual($op(x, y), $formula)
      end)
end
```

We overload the `+`, `-`, and `*` operators to work with dual numbers, using the usual rules of differentiation. We also overload both the left and right division operations and the exponentiation operation.

```
@dual_binary(Base.:+, δx + δy)
@dual_binary(Base.:-, δx - δy)
@dual_unary(Base.:-, -δx)
@dual_binary(Base.:*, δx*y + x*δy)
@dual_binary(Base.:/, (δx*y - x*δy)/y^2)
@dual_binary(Base.:\, (δy*x - y*δx)/x^2)
@dual_binary(Base.:^, x^y*(y*δx/x + log(x)*δy))
```

We provide a second version of the power function for when the exponent is a constant integer (as is often the case); this allows us to deal with negative values of x gracefully.

```
Base.:^(x :: Dual, n :: Integer) = Dual(x.val^n, n*x.val^(n-1)*x.δ)
```

For comparisons, we will only consider the value and ignore the variation.

```
Base.:(==)(x :: Dual, y :: Dual)  = x.val == y.val
Base.isless(x :: Dual, y :: Dual) = x.val < y.val
```

For convenience, we also write a handful of the standard functions that one learns to differentiate in a first calculus course.

```
@dual_unary(Base.abs,  δx*sign(x))
@dual_unary(Base.sqrt, δx/sqrt(x))
@dual_unary(Base.exp,  exp(x)*δx)
@dual_unary(Base.log,  δx/x)
@dual_unary(Base.sin,  cos(x)*δx)
@dual_unary(Base.cos, -sin(x)*δx)
@dual_unary(Base.asin, δx/sqrt(1-x^2))
@dual_unary(Base.acos,-δx/sqrt(1-x^2))
```

With these definitions in place, we can automatically differentiate through a variety of functions without writing any special code. For example, the following code differentiates the Haaland approximation to the Darcy-Weisbach friction factor in pipe flow and compares to a finite difference approximation:

```
let
    fhaaland(ϵ, D, Re) = 1.0/(-1.8*log( (ϵ/D/3.7)^1.11 + 6.9/Re ))^2
    δy = fhaaland(0.01, 1.0, Dual(3000, 1)).δ
    δy_fd = finite_diff(Re->fhaaland(0.01, 1.0, Re), 3000, h=1e-3)
    isapprox(δy, δy_fd, rtol=1e-6)
 end
```

```
true
```

11.1.2 Matrix computations

We can also use dual numbers inside of some of the linear algebra routines provided by Julia. For example, consider $x = A^{-1}b$ where b is treated as constant; the following code automatically computes both $A^{-1}b$ and $-A^{-1}(\delta A)A^{-1}b$.

```
function test_dual_solve()
    Aval = [1.0 2.0; 3.0 4.0]
    δA   = [1.0 0.0; 0.0 0.0]
    b = [3.0; 4.0]
    A = Dual.(Aval, δA)
    x = A\b

    # Compare ordinary and variational parts to formulas
    value.(x) ≈ Aval\b &&
    variation.(x) ≈ -Aval\(δA*(Aval\b))
end

test_dual_solve()
```

```
true
```

While this type of automatic differentiation through a matrix operation works, it is relatively inefficient. We can also define operations that act at a matrix level for relatively expensive operations like matrix multiplication and linear solves. We give as an example code in Julia for more efficient linear solves with matrices of dual numbers, using LU factorization (Chapter 16) as the basis for applying A^{-1} to a matrix or vector:

```
function Base.:\(AA :: AbstractMatrix{Dual{T}},
                 BB :: AbstractVecOrMat{Dual{S}}) where {T,S}
    A, δA = value.(AA), variation.(AA)
    B, δB = value.(BB), variation.(BB)
    F = lu(A)
    X = F\B
    Dual.(X, F\(δB-δA*X))
end

function Base.:\(AA :: AbstractMatrix{Dual{T}},
                 B :: AbstractVecOrMat{S}) where {T,S}
    A, δA = value.(AA), variation.(AA)
    F = lu(A)
    X = F\B
    Dual.(X, F\(-δA*X))
end

function Base.:\(A :: AbstractMatrix{T},
                 BB :: AbstractVecOrMat{Dual{S}}) where {T,S}
```

```
    B, δB = value.(BB), variation.(BB)
    F = lu(A)
    Dual.(F\B, F\δB)
end

test_dual_solve()
```

```
true
```

11.1.3 Special cases

There are other cases as well where automatic differentiation using dual numbers needs a little help. For example, consider the thin-plate spline function, which has a removable singularity at zero:

```
ϕtps(r) = r == 0.0 ? 0.0 : r^2*log(abs(r))
```

If we treat r as a dual number, the output for $r = 0$ will be an ordinary floating point number, while the output for every other value of r will be a dual number. However, we can deal with this by writing a specialized version of the function for dual numbers.

```
ϕtps(r :: Dual) = r == 0.0 ? Dual(0.0, 0.0) : r^2*log(r)
```

With this version, the function works correctly at both zero and nonzero dual number arguments:

```
ϕtps(Dual(0.0, 1.0)), ϕtps(Dual(1.0, 1.0))
```

```
(Dual{Float64}(0.0, 0.0), Dual{Float64}(0.0, 1.0))
```

In addition to difficulties with removable singularities, automatic differentiation systems may lose accuracy do to floating point effects even for functions that are well-behaved. We return to this in Chapter 9.

11.2 Forward and backward

In the previous section, we described automatic differentiation by tracking variations (dual numbers) through a computation, often known as *forward mode* differentiation. An alternate approach known as *backward mode* (or *adjoint mode*) differentiation involves tracking a different set of variables (dual variables)

11.2.1 An example

We consider again the sample function given in the previous section:

$$f(x_1, x_2) = \left(-1.8 \log(x_1/3.7)^{1.11} + 6.9/x_2\right)^{-2}.$$

We would usually write this concisely as

```
fhaaland0(x1, x2) = (-1.8*log( (x1/3.7)^1.11 + 6.9/x2 ))^-2
```

When we compute or manipulate such somewhat messy expressions (e.g. for differentiation), it is useful to split them into simpler subexpressions (as in the single static assignment (SSA) style introduced in Chapter 2). For example, we can rewrite $f(x_1, y_1)$ in terms of seven intermediate expressions, and compute variations of each intermediate to get a variation of the final result. In code, this is

```
function fhaaland1(x1, x2)
    y1 = x1/3.7
    y2 = y1^1.11
    y3 = 6.9/x2
    y4 = y2+y3
    y5 = log(y4)
    y6 = -1.8*y5
    y7 = y6^-2
    function Df(δx1, δx2)
        δy1 = δx1/3.7
        δy2 = 1.11*y1^0.11 * δy1
        δy3 = -6.9/x2^2 * δx2
        δy4 = δy2+δy3
        δy5 = δy4/y4
        δy6 = -1.8*δy5
        δy7 = -2*y6^(-3)*δy6
    end
    y7, Df
end
```

The function `Df` inside `fhaaland1` computes derivatives similarly to using dual numbers as in the previous section.

Another way to think of this computation is that we have solved for the intermediate y variables as a function of x from the relationship

$$G(x, y) = h(x, y) - y = 0,$$

where the rows of h are the seven equations above, e.g. $h_1(x,y) = x_1/3.7$. Differentiating this relation gives us

$$D_1 h(x,y)\,\delta x + (D_2 h(x,y) - I)\;\delta y = 0.$$

The formula for the variations in the y variables can be thought of as coming from using forward substitution to solve the linear system

$$(D_2 h(x,y) - I)\delta y = -D_1 h(x,y)\delta x.$$

That is, we compute the components of δy in order by observing that $h_i(x,y)$ depends only on $y_1, \ldots, y_{i-1}$ and writing

$$\delta y_i = D_1 h_i(x,y)\,\delta x + \sum_{j=1}^{i=1} \left(D_2 h_i(x,y)\right)_j\;\delta y_j.$$

A key observation is that we do *not* then use all of δy; we only care about

$$f'(x)\delta x = w^* \delta y,$$

where w^* is a functional that selects the desired output (here $w^* = e_7^*$). Putting the steps of the previous computation together, we have

$$f'(x)\,\delta x = w^*\left[\left(D_2 h(x,y) - I\right)^{-1}\left(-D_1 h(x_y)\,\delta x\right)\right].$$

But associativity, we could also write this as

$$f'(x)\,\delta x = \left[(-w^*(D_2 h(x,y) - I)^{-1})\,D_1 h(x,y)\right]\;\delta x.$$

Giving names to the parenthesized pieces of this latter equation, we have

$$\begin{aligned} \bar{y}^* &= -w^*(D_2 h(x,y) - I)^{-1} \\ f'(x) &= \bar{y}^* D_1 h(x,y). \end{aligned}$$

Where we solved the system for the variations δy by *forward* substitution, we solve the system dual variables $\bar{y}^*$ by *backward substitution*, applying the formula

$$\bar{y}_j^* = w_j^* + \sum_{i=j+1}^{n} \bar{y}_i^* (D_2 h_i(x,y))_j$$

for $j = m, m-1, \ldots, 1$. Because this computation can be written in terms of a solve with the adjoint matrix $(D_2 h(x,y) - I)^*$ using backward substitution, this method of computing derivatives is sometimes called *adjoint mode* or *backward mode* differentiation.

Translating these ideas into code for our example function, we have

```
function fhaaland2(x1, x2)
    y1 = x1/3.7
    y2 = y1^1.11
    y3 = 6.9/x2
    y4 = y2+y3
    y5 = log(y4)
    y6 = -1.8*y5
    y7 = y6^-2

    ȳ7 = 1
    ȳ6 = ȳ7 * (-2*y6^-3)
    ȳ5 = ȳ6 * (-1.8)
    ȳ4 = ȳ5/y4
    ȳ3 = ȳ4
    ȳ2 = ȳ4
    ȳ1 = ȳ2 * (1.11*y1^0.11)

    Df = [ȳ1/3.7  ȳ3*(-6.9/x2^2)]
    y7, Df
end
```

We now have three different methods for computing the derivative of f, which we illustrate below (as well as checking that our computations agree with each other).

```
let
    # Compute f and the derivative using dual numbers
    f0a = fhaaland0(Dual(0.01, 1.0), 3000)
    f0b = fhaaland0(0.01, Dual(3000, 1))
    Df0 = [variation(f0a) variation(f0b)]

    # Compute using the SSA-based forward mode code
    f1, Df1fun = fhaaland1(0.01, 3000)
    Df1 = [Df1fun(1,0) Df1fun(0,1)]

    # Compute using the SSA-based backward mode code
    f2, Df2 = fhaaland2(0.01, 3000)

    # Compare to see that all give the same results
    f1 == f2 && f1 ≈ value(f0a) && f1 ≈ value(f0b) &&
    Df1[1] ≈ Df2[1] && Df0[1] ≈ Df2[1] &&
    Df1[2] ≈ Df2[2] && Df0[2] ≈ Df2[2]
end
```

```
true
```

We note that we only need to compute the $\bar{y}$ variables *once* to get the derivative vector, where for forward mode we need to compute the δy variables *twice*.

The cost of computing the derivative in one direction $(\delta x_1, \delta y_2)$ is only about as great as the cost of evaluating f. But if we wanted the derivative with respect to both x_1 and x_2, we would need to run at least the code in `Df` twice (and with the dual number implementation from the previous section, we would also run the evaluation of f twice).

11.3 A derivative example

Beyond writing language utilities, Julia macros are useful for writing embedded domain-specific languages (DSLs) for accomplishing particular tasks. In this setting, we are really writing a language interpreter embedded in Julia, using the Julia parse tree as an input and producing Julia code as output.

One example has to do with *automatic differentiation* of simple code expressions, where we are giving another interpretation to simple Julia expressions. This is a more ambitious example, and can be safely skipped over. However, it is also a useful example of a variety of features in Julia. Some types of automatic differentiation can be done in arguably simpler and more powerful ways without writing macros, and we will return to this in later chapters.

11.3.1 Normalization

We define a "simple" expression as one that only involves only:

- Literal nodes (including symbols),
- `Expr` nodes of `call` type, including arithmetic and function calls,
- Assignment statements,
- Tuple constructors,
- `begin`/`end` blocks.

We can test for simple expressions by recursing through an expression tree

```
# Simple expression nodes
is_simple(e :: Expr) =
    e.head in [:call, :(=), :tuple, :block] &&
    all(is_simple.(e.args))

# Other nodes are literal, always simple
is_simple(e) = true
```

In addition to insisting that expressions are simple, we also want to simplify some peculiarities in Julia's parsing of addition and multiplication. In many languages, an expression like `1 + 2 + 3` is parsed as two operations: `(1 + 2) + 3`. In Julia, this results in a single node. It will simplify our life to convert these types of nodes to binary nodes, assuming left associativity of the addition operation. We do this by recursively rewriting the expression tree to replace a particular type of operation node (calling `op`) with a left-associated version of that same node:

```
leftassoc(op :: Symbol, e) = e
function leftassoc(op :: Symbol, e :: Expr)
    args = leftassoc.(op, e.args)
    if e.head == :call && args[1] == op
        foldl((x,y) -> Expr(:call, op, x, y), args[2:end])
    else
        Expr(e.head, args...)
    end
end
```

We are particularly interested in left associativity of addition and multiplication, so we write a single function for those operations

```
leftassoc(e) = leftassoc(:+, leftassoc(:*, e))
```

We can see the effect by printing the parse of a particular example:

```
let
    e = :(a*b*c + 2 + 3)
    println("Before: $e")
    println("After:  $(leftassoc(e))")
end
```

```
Before: a * b * c + 2 + 3
After:  ((a * b) * c + 2) + 3
```

Finally, the Julia parser adds `LineNumberNodes` to blocks in order to aid with debugging. We will dispose of those here so that we can focus solely on processing expressions.

```
filter_line(e) = e
filter_line(e :: Expr) =
    Expr(e.head,
         filter(x->!(x isa LineNumberNode),
                filter_line.(e.args))...)
```

Putting everything together, we will normalize simple expressions by eliminating line numbers and re-associating addition and multiplication.

```
normalize_expr(e) = e
function normalize_expr(e :: Expr)
    @assert is_simple(e) "Expression must be simple"
    leftassoc(filter_line(e))
end
```

11.3.2 SSA

Simple expressions of the type that we have described can be converted to *static single assignment* (SSA) form, where each intermediate subexpression is assigned a unique local name (which we produce with a call to `gensym`). We represent a program in SSA form as a vector of `(symbol, expression)` pairs to be interpreted as "evaluate the expression and assign it to the symbol." We *emit* SSA terms by appending them to the end of a list.

```
function ssa_generator()
    result = []

    # Add a single term to the SSA result
    function emit!(s, e)
        push!(result, (s,e))
        s
    end

    # Add several terms to the SSA result;
    #  s is the designated "final result"
    function emit!(s, l :: Vector)
        append!(result, l)
        s
    end

    # Emit a term with a new local name
    emit!(e) = emit!(gensym(), e)

    result, emit!
end
```

The `to_ssa` function returns a final result in SSA form Each symbol is only assigned once. Each arithmetic, function call, and tuple constructor results in a new symbol. For assignments, we record a binding of the left hand side symbol to the result computed for the right hand side.

```
function to_ssa(e :: Expr)

    # Check and normalize the expression
    e = normalize_expr(e)

    # Set up results list and symbol table
    elist, emit! = ssa_generator()
    symbol_table = Dict{Symbol,Any}()

    # Functions for recursive processing
    process(e) = e
    process(e :: Symbol) = get(symbol_table, e, e)
    function process(e :: Expr)
        args = process.(e.args)
        if e.head == :block
            args[end]
        elseif e.head == :(=)
            symbol_table[e.args[1]] = args[2]
            args[2]
        else
            emit!(Expr(e.head, args...))
        end
    end

    process(e), elist
end
```

In case we want to convert a single leaf to SSA form, we define a second method:

```
function to_ssa(e)
    s = gensym()
    s, [(s,e)]
end
```

We also want a utility to compute from SSA back to ordinary Julia code

```
from_ssa(sym, elist) =
    Expr(:block,
         [Expr(:(=), s, e) for (s,e) in elist]...,
         sym)
```

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

As is often the case, an example is useful to illustrate the code.

```
let
    s, elist = to_ssa(
        quote
            x = 10
            x = 10+x+1
            y = x*x
        end)
    from_ssa(s, elist)
end
```

```
quote
    var"##230" = 10 + 10
    var"##231" = var"##230" + 1
    var"##232" = var"##231" * var"##231"
    var"##232"
end
```

11.3.3 Derivative function

To set up our differentiation function, we will need some differentiation rules. We store this in a dictionary where the keys are different types of operations and functions, and the arguments are the arguments to those functions and their derivatives. Each rule takes expressions for the value of the function, the arguments, and the derivatives of the arguments, and then returns the derivative of the function. In some cases, we may want different methods associated with the same function, e.g. to distinguish between negation and subtraction or to provide a special case of the power function where the exponent is an integer constant.

```
deriv_minus(f,x,dx) = :(-$dx)
deriv_minus(f,x,y,dx,dy) = :($dx-$dy)
deriv_pow(f,x,n :: Int, dx, dn) = :($n*$x^$(n-1)*$dx)
deriv_pow(f,x,y,dx,dy) = :($f*($dy/$x*$dx + $dy*log($x)))

deriv_rules = Dict(
    :+   => (f,x,y,dx,dy) -> :($dx+$dy),
    :*   => (f,x,y,dx,dy) -> :($dx*$y + $x*$dy),
    :/   => (f,x,y,dx,dy) -> :(($dx - $f*$dy)/$y),
    :-   => deriv_minus,
    :^   => deriv_pow,
    :log => (f,x,dx)    -> :($dx/$x),
```

```
    :exp => (f,x,dx)    -> :($f*$dx)
)
```

Now we are in a position to write code to simultaneously evaluate the expression and the derivative (with respect to some symbol `s`). To do this, it is helpful to give a (locally defined) name to each subexpression and its derivative, and to produce code that is a sequence of assignments to those names. We accumulate those assignments into a list (`elist`). An internal `process` function with different methods for different types of objects is used to generate the assignments for the subexpression values and the associated derivatives.

```
function derivative(s :: Symbol, e :: Expr)

    # Convert input to SSA form, set up generator
    sym, elist = to_ssa(e)
    eresult, emit! = ssa_generator()

    # Derivatives of leaves, init with ds/ds = 1.
    deriv_table = Dict{Symbol,Any}()
    deriv_table[s] = 1
    deriv(e :: Symbol) = get(deriv_table, e, 0)
    deriv(e) = 0

    # Rules to generate differentiation code
    function derivcode(s, e :: Expr)
        if e.head == :call
            rule = deriv_rules[e.args[1]]
            dexpr = rule(s, e.args[2:end]...,
                         deriv.(e.args[2:end])...)
            emit!(to_ssa(dexpr)...)
        elseif e.head == :tuple
            emit!(Expr(:tuple, deriv.(e.args)...))
        else
            error("Unexpected expression of type $(e.head)")
        end
    end

    derivcode(s, e) = emit!(gensym(), deriv(e))

    # Produce code for results and derivatives
    for (s,e) in elist
        emit!(s,e)
        deriv_table[s] = derivcode(s, e)
```

```
    end

    # Add a tuple for return at the end (function + deriv)
    emit!(Expr(:tuple, sym, deriv_table[sym])), eresult
end
```

As an example, consider computing the derivative of $mx + b$ with respect to x

```
from_ssa(derivative(:x, :(m*x+b))...)
```

```
quote
    var"##233" = m * x
    var"##235" = 0 * x
    var"##236" = m * 1
    var"##237" = var"##235" + var"##236"
    var"##234" = var"##233" + b
    var"##238" = var"##237" + 0
    var"##239" = (var"##234", var"##238")
    var"##239"
end
```

Giving more comprehensible variable names, this is equivalent to

```
y1 = m * x
dy1a = 0 * x
dy1b = m * 1
dy1 = dy1a + dy1b
y2 = y1 + b
dy2 = dy1 + 0
result = (y2, dy2)
result
```

This is correct, but it is also a very complicated-looking way to compute m! We therefore consider putting in some standard simplifications.

11.3.4 Simplification

There are many possible algebraic relationships that we can use to simplify derivatives. Some of these are relationships, like multiplication by zero, are things that we believe are safe but the

compiler cannot safely do on our behalf[1]. Others are things that the compiler could probably do on our behalf, but we might want to do on our own to understand how these things work.

We will again use the matching rules on multiple dispatch to write our simplification rules as different methods under a common name. For example, the simplification rules for $x + y$ (where the prior expression is saved as e) are:

```
# Rules to simplify e = x+y
simplify_add(e, x :: Number, y :: Number) = x+y
simplify_add(e, x :: Number, y :: Symbol) = x == 0 ? y : e
simplify_add(e, x :: Symbol, y :: Number) = y == 0 ? x : e
simplify_add(e, x, y) = e
```

The rules for minus (unary and binary) are similar in flavor. We will give the unary and binary versions both the same name, and distinguish between the two cases based on the number of arguments that we see.

```
# Rules to simplify e = -x
simplify_sub(e, x :: Number) = -x
simplify_sub(e, x) = e

# Rules to simplify e = x-y
simplify_sub(e, x :: Number, y :: Symbol) = x == 0 ? :(-$y) : e
simplify_sub(e, x :: Symbol, y :: Number) = y == 0 ? y : e
simplify_sub(e, x, y) = e
```

With multiplication, we will use the ordinary observation that anything times zero should be zero, ignoring the peculiarities of floating point.

```
# Rules to simplify e = x*y
simplify_mul(e, x :: Number, y :: Number) = x*y
simplify_mul(e, x :: Number, y :: Symbol) =
    if     x == 0  0
    elseif x == 1  y
    else           e
    end
simplify_mul(e, x :: Symbol, y :: Number) =
    if     y == 0  0
    elseif y == 1  x
    else           e
```

[1] In IEEE floating point, `0 * Inf` is a `NaN`, not a zero. The compiler does not know that we don't expect infinities in the code, and so cannot safely eliminate products with 0.

```
    end
simplify_mul(e, x, y) = e
```

Finally, we define rules for simplifying quotients and powers.

```
# Rules to simplify e = x/y
simplify_div(e, x :: Number, y :: Number) = x/y
simplify_div(e, x :: Symbol, y :: Number) = y == 1 ? x : e
simplify_div(e, x :: Number, y :: Symbol) = x == 0 ? 0 : e
simplify_div(e, x, y) = e

# Simplify powers
simplify_pow(e, x :: Symbol, n :: Number) =
    if     n == 1  x
    elseif n == 0  1
    else           e
    end
simplify_pow(e, x, n) = e
```

To simplify a line in an SSA assignment, we look up the appropriate rule function in a table (as we did with differentiation rules) and apply it.

```
simplify_rules = Dict(
    :+ => simplify_add,
    :- => simplify_sub,
    :* => simplify_mul,
    :/ => simplify_div,
    :^ => simplify_pow)

simplify_null(e, args...) = e
simplify(e) = e
simplify(e :: Expr) =
    if e.head == :call
        rule = get(simplify_rules, e.args[1], simplify_null)
        rule(e, e.args[2:end]...)
    else
        e
    end
```

```
simplify (generic function with 2 methods)
```

Putting everything together, our rule for simplifying code in SSA form is:

- Look up whether a previous simplification replaced any arguments with constants or with other symbols. If a replacement was also replaced earlier, reply the rule recursively.
- Apply the appropriate simplification rule.
- If the expression now looks like `(symbol, x)` where `x` is a leaf (a constant or another symbol), make a record in the lookup table to replace future references to `symbol` with references to `x`.

The following code carries out this task.

```
function simplify_ssa(sym, elist)

    # Set up and add to replacement table
    replacements = Dict{Symbol,Any}()
    function replacement(s,e)
        replacements[s] = e
    end

    # Apply a replacement policy
    replace(e) = e
    replace(s :: Symbol) =
        if s in keys(replacements)
            replace(replacements[s])
        else
            s
        end

    # Simplify each term
    eresult, emit! = ssa_generator()
    for (s,e) in elist
        e = e isa Expr ? Expr(e.head, replace.(e.args)...) : e
        enew = simplify(e)
        if enew isa Number || enew isa Symbol
            replacement(s, enew)
        end
        emit!(s, enew)
    end
    replace(sym), eresult
end
```

Finally, we do *dead code elimination*, keeping only "live" computations on which the final result depends. Liveness is computed recursively: the final result is live, and anything that a live

variable depends on is live. Because in SSA each expression depends only on previous results, we can compute liveness by traversing the list from back to front and updating what we know to be live as we go.

```
function prune_ssa(sym, elist)

    # The end symbol is live
    live = Set([sym])

    # Propogate liveness
    mark_live(s :: Symbol) = push!(live, s)
    mark_live(e :: Expr) = mark_live.(e.args)
    mark_live(e) = nothing

    # If the RHS is live, so is anything it depends on
    for (s,e) in reverse(elist)
        if s in live
            mark_live(e)
        end
    end

    # Return only the live expressions
    sym, filter!(p -> p[1] in live, elist)
end
```

Repeating our experiment with differentiating $mx + b$ but including simplification and pruning, we get a much terser generated code.

```
from_ssa(
    prune_ssa(
        simplify_ssa(
            derivative(:x, :(m*x+b))...)...)...)
```

```
quote
    var"##240" = m * x
    var"##241" = var"##240" + b
    var"##246" = (var"##241", m)
    var"##246"
end
```

11.3.5 The macro

Finally, we package the derivatives into a macro. The macro gets the derivative information from the `derivative` function and packages it into a block that at the end returns the final expression value and derivative as a tuple. We note that the namespace for macros is different from the namespace for functions, so it is fine to use the same name (`derivative`) for both our main function and for the macro that calls it. Note that we escape the full output in this case – we want to be able to use the external names, and we only write to unique local symbols.

```
macro derivative(s :: Symbol, e :: Expr)
    sym, elist = derivative(s, e)
    sym, elist = simplify_ssa(sym, elist)
    sym, elist = prune_ssa(sym, elist)
    esc(from_ssa(sym, elist))
end
```

We will test out the macro on a moderately messy expression that is slightly tedious to compute by hand, and compare our result to the (slightly tedious) hand computation.

```
let
    # An expression and a hand-computed derivative
    f(x) = -log(x^2 + 2*exp(x) + (x+1)/x)
    df(x) =
        -(2*x + 2*exp(x) -1/x^2)/
        (x^2 + 2*exp(x) + (x+1)/x)

    # Autodiff of the same expression
    g(x) = @derivative(x, -log(x^2 + 2*exp(x) + (x+1)/x))

    # Check that the results agree up to roundoff effects
    xtest = 2.3
    gx, dgx = g(xtest)
    gx ≈ f(xtest) && dgx ≈ df(xtest)
end
```

```
true
```

12 Numerical Differentiation

- Finite differencing (real and complex)
- Differentiation via models
- Differentiation matrices

13 Quadrature

- Numerical indefinite and definite integration
- Low-dimensional quadrature rules
- Gaussian quadrature (Gauss-Legendre, Gauss-Hermite, etc)
- Specialized quadrature, transforms, etc
- Matrices, moments, and quadrature
- Some example statistical computations (being Bayesian, for example)
- Bayesian quadrature (a pointer ahead)
- Monte Carlo quadrature (a pointer ahead)

14 Root Finding and Optimization

Global and local search Hard and easy problems, initial guesses, bounding intervals Bisection Fixed point iteration and analysis Newton's method and analysis Beyond Newton's method – polynomial proxies, etc (fwd pointer) Secant method Safeguards Comments re multiple parameters, implicit differentiation (look ahead) Connections: parameter estimation

15 Computing with Randomness

NB: Maybe this is just section 4

Randomized methods - Monte Carlo and Las Vegas Variational inference (??) Probabilistic numerics (including BQ) Uncertainty quantification

Part III

Numerical Linear Algebra

16 Linear Systems

Many applications of linear systems (including optimization)

The usual intro to LU Role of pivoting Cholesky (+ other areas where pivoting is not needed) Block factorization, role of level 3 BLAS

General meaning of Schur complements SMW and auxiliary variable formulations Schur complements, covariance, precision, and conditioning

Condition estimation

Iterative refinement

Sparsity and dependencies Data sparse matrices Direct sparse solver ideas When do direct sparse solvers make sense? Fast and superfast solvers (fwd ref: signals)

Examples with various Markov chain computations, PageRank?

17 Least Squares

Underdetermined and overdetermined systems Standard approaches (QR, NE, SVD) Augmented system formulation Alternate norms, continuous least squares and associated factorizations Regularization (Tikhonov, truncated SVD, implicit regularization via iteration) Sketching Leverage scores Sparsity vs incoherence Bayesian least squares Cross-validation, GCV, etc

18 Eigenvalue Problems and the SVD

Reminder of the canonical forms; difference between the SEP, NEP, SVD Reminder: application of eigenvalue problems. What is actually needed? Power method (+ PageRank as an example)

Subspace iteration Randomized range finder

From subspace iteration to QR Hessenberg matrices (+ control?)

Symmetric eigenvalue problem and Rayleigh quotients Tridiagonalization

19 Signals and Transforms

Linear ODEs, linear difference equations, etc Convolutions for differential and difference equations Convolutions in probability Auto-regressive models Fast Fourier transforms Toeplitz and Hankel matrices Displacement rank and superfast solvers Chebyshev-Fourier connection

20 Stationary Iterations

Standard splittings and such PageRank, Markov chains, etc Stochastic iterations? (look ahead)
Regularization via iteration

21 Krylov Subspaces

Part IV

Nonlinear Equations and Optimization

22 Calculus Revisited

Finite differences and sparsity Use and abuse of automatic differentiation Backpropagation and neural networks (SIREV, Neidinger 2010)

23 Nonlinear Equations and Unconstrained Optimization

Global vs local search Zeroth order methods

24 Continuation and Bifurcation

25 Constrained Optimization

Equality vs inequality constraints Simple non-negativity constraints Penalties and barriers Projected gradients and SPG Interior point methods Augmented Lagrangian methods

26 Nonlinear Least Squares

Part V

Computing with Randomness

27 Sampling

28 Quadrature and Monte Carlo

29 Solvers from Monte Carlo to Las Vegas

30 Uncertainty Quantification

Part VI

Dimension Reduction and Latent Factor Models

31 Latent Factors and Matrix Factorization

Matrices as data vs matrices in linear algebra - stacked/multimodal vs multiway data SVD, PCA, etc ID and CUR NMF and parts decompositions

32 From Matrices to Tensors

Basic vocabulary Why are tensors harder CP Tucker Tensor train

33 Nonlinear Dimensionality Reduction

Part VII

Function Approximation

34 Fundamental Concepts

Space of functions / parameterization (NB: “nonparametric”) Consistency, stability, bias/variance decomposition, curse of dimensionality Interpolation or other fitting process, stability in map from data to fns (noise and numerics)

35 Low-Dimensional Structure

On input: active subspace, sloppy models, random embed for optimization On output: POD, DEIM, etc; approximation from a subspace

36 Kernels and RBFs

Interpretations Error analysis Computing for modest n Computing for large n

37 Neural Networks

Part VIII

Network Analysis

38 Graphs and Matrices

39 Functions on Graphs

40 Clustering and Partitioning

41 Centrality Measures

Part IX

Learning Dynamics

42 Fundamentals

43 Model Reduction

44 Learning Linear Dynamics

45 Extrapolation and Acceleration

46 From Markov to Koopman

47 Learning Nonlinear Dynamics

References

Axler, Sheldon. 2024. *Linear Algebra Done Right*. Fourth. Springer.

Bailey, David H. 1991. "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Comptuers." *Supercomputing Review*, August. https://www.davidhbailey.com/dhbpapers/twelve-ways.pdf.

Bailey, David H., Robert F. Lucas, and Samuel Williams, eds. 2010. *Performance Tuning of Scientific Applications*. CRC Press. https://doi.org/10.1201/b10509.

Billingsley, Patrick. 1995. *Probability and Measure*. Third. Wiley Series in Probability and Mathematical Statistics. Wiley.

Breiman, Leo. 1992. *Probability*. SIAM. https://doi.org/10.1137/1.9781611971286.

Chung, Kai Lai, and Farid AitSahlia. 2003. *Elementary Probability Theory*. Undergraduate Texts in Mathematics. Springer. https://doi.org/10.1007/978-0-387-21548-8.

Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. 2020. *Mathematics for Machine Learning*. Cambridge University Press. https://mml-book.com.

Graham, Paul. 1993. *On Lisp*. Prentice Hall. https://paulgraham.com/onlisp.html.

Hennessey, John L., and David A. Patterson. 2017. *Computer Architecture: A Quantitative Approach*. Sixth. Elsevier.

Hunt, Andy, and David Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley.

Kernighan, Brian W., and Rob Pike. 1999. *The Practice of Programming*. Addison Wesley.

Kernighan, Brian W., and P. J. Plauger. 1978. *The Elements of Programming Style*. Second. McGraw-Hill.

Keynes, John Maynard. 1929. *A Tract on Monetary Reform*. The Macmillan Company. https://archive.org/details/tractonmonetaryr0000keyn/mode/2up.

Kline, M. 1990. *Mathematical Thought from Ancient to Modern Times*. Oxford University Press.

Knuth, Donald E. 1974. "Structured Programming with go to Statements." *Computing Surveys* 6 (4).

———. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Lax, Peter D. 2007. *Linear Algebra and Its Applications*. Second. Wiley.

Lay, David C., Steven R. Lay, and Judi J. McDonald. 2015. *Linear Algebra and Its Applications*. Fifth. Pearson.

Leonard, Mary J., Steven T. Kalinowski, and Tessa C. Andrews. 2014. "Misconceptions Yesterday, Today, and Tomorrow." *CBE – Life Sciences Education* 13 (2). https://doi.org/10.1187/cbe.13-12-0244.

Muller, Derek A. 2008. “Designing Effective Multimedia for Physics Education.” PhD thesis, University of Sydney.

Rosenthal, Arthur. 1951. “The History of Calculus.” *The American Mathematical Monthly* 58 (2): 75–86. https://doi.org/10.2307.2308368.

Ross, Sheldon. 2014. *A First Course in Probability*. Ninth. Pearson.

Sadler, Philip M., Gerhard Sonnert, Harold P. Coyle, Nancy Cook-Smith, and Jaimie L. Miller. 2013. “The Influence of Teachers’ Knowledge on Student Learning in Middle School Physical Science Classrooms.” *American Educational Research Journal* 50 (5). https://doi.org/10.3102/0002831213477680.

Strang, Gilbert. 2023. *Introduction to Linear Algebra*. Sixth. Wellesley-Cambridge Press.

Vuduc, Richard, Aparna Chandramowlishwaran, Jee Choi, Kenneth Czechowski, Marat Guney, Logan Moon, and Aashay Shringarpure. 2010. “On the Limits of GPU Acceleration.” In *HotPar’10: Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*. Berkeley, CA. https://doi.org/10.5555/1863086.1863099.

Wilf, Herbert S. 2006. *Generatingfunctionology*. Third. AK Peters, Ltd. https://www2.math.upenn.edu/~wilf/DownldGF.html.

Williams, Samuel, Andrew Waterman, and David Patterson. 2009. “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” *Communications of the ACM* 52 (4). https://doi.org/10.1145/1498765.1498785.