Julia Programming

From Fundamentals to Scientific Computing

A 30-hour practical course

Yaé Ulrich Gaba

AIRINA Labs

2026

Contents

Preface

Julia is a programming language designed for people who need both high-level expressiveness and high performance. It was created at MIT in 2012 by researchers who wanted "the speed of C, the dynamism of Ruby, the mathematical notation of Matlab, and the generality of Python." A decade later, Julia has become a serious tool for scientific computing, machine learning, and data science.

This course is for anyone who already knows basic programming concepts in some language — Python, R, Matlab, C — and wants to learn Julia. You do not need to be an expert programmer. You need to be comfortable with the idea of variables, loops, functions, and arrays.

We cover Julia from the ground up: types, multiple dispatch, arrays, data wrangling, visualization, performance tuning, differential equations, optimization, and machine learning. Every chapter uses real datasets and real Julia code that you can run in Pluto notebooks or the Julia REPL.

What this course is not. This is not a computer science theory course. It is not a numerical analysis textbook (though we do numerical computing). It is a *practical programming course* for scientists, engineers, and data practitioners who want to use Julia effectively.

Prerequisites. Basic programming experience in any language. Familiarity with linear algebra and calculus is helpful for Chapters 8–10 but not required for Chapters 1–7.

Software. Julia 1.10 or later, VS Code with the Julia extension, and Pluto.jl for reactive notebooks. All packages are free and open source.

Yaé Ulrich Gaba
Cotonou, 2026

Chapter 1

Getting started with Julia

> *"We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want homoiconicity, true macros, and a language that is as usable for general programming as Python."* — Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman (2012)

1.1 Why Julia?

Julia was created to solve the *two-language problem*: scientists prototype in a high-level language (Python, Matlab, R) and then rewrite performance-critical code in C or Fortran. Julia eliminates this step. Code that reads like Python runs at speeds comparable to C.

Key features that make Julia distinctive:

1. **Just-in-time (JIT) compilation.** Julia compiles functions to native machine code using LLVM the first time they are called. Subsequent calls are fast.
2. **Multiple dispatch.** Functions can have many methods, selected based on the types of *all* arguments. This is Julia's central design principle.
3. **Rich type system.** Abstract types, parametric types, and union types let you write generic, reusable code.
4. **Metaprogramming.** Julia code is represented as Julia data structures (expressions), enabling powerful macros.
5. **Package ecosystem.** Over 10,000 registered packages for scientific computing, data science, optimization, and machine learning.

Julia tip

Julia uses 1-based indexing, like Matlab and R, not 0-based like Python and C. The first element of an array `v` is `v[1]`.

1.2 Installing Julia

1.2.1 Using juliaup (recommended)

```
# macOS / Linux
```

```
curl -fsSL https://install.julialang.org | sh

# Windows (PowerShell)
winget install julia -s msstore
```

Then verify:

```
julia --version
# julia version 1.10.x
```

1.2.2 VS Code integration

Install the Julia extension in VS Code. It provides:

- Syntax highlighting and autocompletion
- Integrated REPL (`Alt+J`, `Alt+O` to open)
- Plot pane for inline visualization
- Debugger and profiler integration

1.3 The Julia REPL

Launch Julia by typing `julia` in your terminal. The REPL (Read-Eval-Print Loop) has four modes:

```
# Julian mode (default) - execute Julia code
julia> 2 + 3
5

julia> sqrt(144)
12.0
```

```
# Package mode - press ]
(@v1.10) pkg> add DataFrames

# Help mode - press ?
help?> sqrt
  sqrt(x)
  Return the square root of x.

# Shell mode - press ;
shell> ls
```

Julia tip

Press `Backspace` on an empty line to return to Julian mode from any special mode. Press `Tab` for autocompletion — it works on function names, file paths, and even Unicode characters.

1.4 Package management with Pkg

Julia's built-in package manager is powerful and reproducible:

```
using Pkg

# Install a package
Pkg.add("DataFrames")

# Install multiple packages at once
Pkg.add(["CSV", "Plots", "BenchmarkTools"])

# Update all packages
Pkg.update()

# Check installed packages
Pkg.status()

# Create a project environment (recommended for reproducibility)
Pkg.activate("MyProject")
Pkg.add("DataFrames")
# This creates Project.toml and Manifest.toml
```

Caution

Always use project environments for serious work. The default global environment (`@v1.10`) can develop dependency conflicts if you install too many packages. Run `Pkg.activate(".")` in your project directory to create a local environment.

1.5 Your first Julia program

```
# hello.jl - Your first Julia script

# Variables
name = "Julia"
year = 2012
age = 2026 - year

println("Hello from $name!")
println("Julia is $age years old and faster than ever.")
```

```julia
# A simple computation
function fibonacci(n)
    n <= 1 && return n
    a, b = 0, 1
    for _ in 2:n
        a, b = b, a + b
    end
    return b
end

# Print the first 15 Fibonacci numbers
for i in 0:14
    println("F($i) = $(fibonacci(i))")
end
```

Run this script from the terminal:

```
julia hello.jl
```

Or from the REPL:

```
julia> include("hello.jl")
```

1.6 Pluto notebooks

Pluto is Julia's reactive notebook environment. Unlike Jupyter, Pluto notebooks are:

- **Reactive:** changing one cell automatically updates all cells that depend on it.
- **Reproducible:** cells execute in dependency order, not in the order you click.
- **Pure Julia files:** saved as `.jl` files, not JSON. Easy to version-control with Git.

```julia
# Install and launch Pluto
using Pkg
Pkg.add("Pluto")

using Pluto
Pluto.run()   # opens a browser tab at localhost:1234
```

Inside a Pluto notebook, each cell contains one expression or `begin...end` block:

```julia
# Cell 1
x = 42

# Cell 2 - automatically re-runs when x changes
```

```
y = x^2 + 1

# Cell 3
md"The value of y is **$(y)**"
```

Julia tip

Pluto enforces one assignment per variable per notebook. This prevents hidden state bugs. If you need multiple statements in one cell, wrap them in a `begin...end` block.

1.7 IJulia and Jupyter

If you prefer Jupyter notebooks (e.g., for sharing with Python colleagues):

```
using Pkg
Pkg.add("IJulia")

using IJulia
notebook()   # opens Jupyter in the browser with Julia kernel
```

1.8 Unicode in Julia

Julia supports Unicode variable names and operators. Type them using LaTeX-like tab completion:

```
# Type \alpha then press Tab to get
 = 0.05
 = 1 -

# Type \sqrt then Tab to get √
√2 = √(2)    # equivalent to sqrt(2)

# Greek letters in functions
function Δ(a, b, c)
    return b^2 - 4a*c
end

# Matrix operations with Unicode
A = [1 2; 3 4]
x = [5, 6]           # \vec then Tab after x
result = A * x
```

Exercise

1. Install Julia 1.10 on your machine using `juliaup`. Verify by running `julia --version`.
2. In the REPL, compute $\sqrt{2^{10} + 3^{10}}$ and $\sum_{k=1}^{100} \frac{1}{k^2}$.
3. Install Pluto.jl and create a notebook with three cells: (a) define $n = 20$, (b) compute $n!$ using `factorial(n)`, (c) display the result in a Markdown cell. Change n and observe reactivity.
4. Create a project environment called `JuliaCourse`, add the packages `DataFrames`, `CSV`, and `Plots`. Check the `Project.toml` file.
5. Write a function `collatz(n)` that returns the number of steps to reach 1 in the Collatz sequence. Test it on $n = 27$.

1.9 Chapter summary

- Julia solves the two-language problem: high-level syntax with compiled speed.
- The REPL has four modes: Julian, Package (`]`), Help (`?`), and Shell (`;`).
- Use `Pkg.activate(".")` to create reproducible project environments.
- Pluto.jl provides reactive notebooks saved as plain Julia files.
- IJulia integrates Julia with the Jupyter ecosystem.
- Julia supports Unicode natively — use Greek letters and math symbols freely.

Chapter 2

Types, variables, and control flow

> *"In Julia, types are not classes. They are tags that the compiler uses to generate fast code."*

2.1 Primitive types

Julia has a rich hierarchy of numeric types:

```
# Integers
x = 42              # Int64 (on 64-bit systems)
y = Int32(42)       # explicitly 32-bit
big = Int128(10)^30 # 128-bit integer
huge = big"99999999999999999999999999999999"  # BigInt (arbitrary precision)

typeof(x)           # Int64
sizeof(x)           # 8 bytes

# Floating-point
a = 3.14            # Float64 (double precision)
b = Float32(3.14)   # single precision
c = BigFloat( )     # arbitrary precision

# Boolean
flag = true         # Bool (subtype of Integer)
flag + 1            # 2 - true is 1, false is 0

# Characters and strings
ch = ' '            # Char (Unicode character, 4 bytes)
s = "Hello, Julia"  # String (UTF-8 encoded)
```

Julia tip

Use `typeof(x)` to check any value's type. Use `supertypes(Int64)` to see the full type hierarchy: `Int64 <: Signed <: Integer <: Real <: Number <: Any`.

2.2 Variables and assignment

Variables in Julia are bindings to values, not typed boxes:

```
x = 10          # x refers to an Int64
x = "hello"     # now x refers to a String - no error

# Multiple assignment
a, b, c = 1, 2.0, "three"

# Swap without a temporary variable
a, b = b, a

# Constants - the compiler optimizes these
const SPEED_OF_LIGHT = 299_792_458  # underscores for readability
const  _approx = 355 // 113         # Rational{Int64}
```

> **Caution**
>
> Redefining a `const` raises a warning (not an error) if the new value has a different type. Use `const` for values that truly should not change.

2.3 The type hierarchy

Julia's type system is a tree rooted at `Any`:

```
# Abstract types - cannot be instantiated, used for dispatch
abstract type Shape end

# Concrete types - can be instantiated
struct Circle <: Shape
    radius::Float64
end

struct Rectangle <: Shape
    width::Float64
    height::Float64
end

# Check type relationships
Circle <: Shape       # true
Circle <: Any         # true
Int64 <: Number       # true
String <: Number      # false

# Useful introspection
subtypes(Integer)     # [Bool, Signed, Unsigned]
supertype(Float64)    # AbstractFloat
```

2.4 Strings

Strings in Julia are immutable, UTF-8 encoded sequences:

```
# String basics
s = "Julia is fast"
length(s)              # 13 (number of characters)
sizeof(s)              # 13 (bytes - same here, but differs for Unicode)
s[1]                   # 'J' (Char)
s[end]                 # 't'
s[1:5]                 # "Julia" (substring)

# String interpolation
name = "World"
greeting = "Hello, $name!"
computation = "2 + 3 = $(2 + 3)"

# Multi-line strings
poem = """
    Roses are red,
    Julia is fast,
    Python is nice,
    but C won't last.
    """

# Common operations
uppercase("julia")           # "JULIA"
replace("Hello", "l" => "r")  # "Herro"
split("a,b,c,d", ",")       # ["a", "b", "c", "d"]
join(["one", "two", "three"], " + ")  # "one + two + three"
occursin("fast", "Julia is fast")     # true

# Regular expressions
m = match(r"(\d+)\s*kg", "Weight: 82 kg")
m.captures[1]                # "82"
```

Julia tip

Indexing into UTF-8 strings by byte position can fail for multi-byte characters. Use `eachindex(s)` or `collect(s)` to iterate safely over characters.

2.5 Control flow: conditionals

```
# if-elseif-else
temperature = 38.5

if temperature >= 39.0
    status = "high fever"
elseif temperature >= 37.5
```

```
    status = "low-grade fever"
else
    status = "normal"
end
println("Status: $status")

# Ternary operator
status = temperature >= 37.5 ? "fever" : "normal"

# Short-circuit evaluation
x = -5
x > 0 || error("x must be positive")  # throws an error
x > 0 && println("x is positive")     # does nothing (no error)
```

2.6 Control flow: loops

```
# for loops with ranges
for i in 1:5
    println("Iteration $i")
end

# Iterating over collections
fruits = ["apple", "banana", "cherry"]
for fruit in fruits
    println(fruit)
end

# Enumerate (index + value)
for (i, fruit) in enumerate(fruits)
    println("$i: $fruit")
end

# Nested loops with a single for
for i in 1:3, j in 1:3
    i == j && println("($i, $j) is on the diagonal")
end

# while loop
n = 1
while n <= 1000
    n *= 2
end
println("First power of 2 above 1000: $n")  # 1024

# break and continue
for i in 1:100
    i % 7 != 0 && continue   # skip non-multiples of 7
    i > 50 && break           # stop after 50
    println(i)
```

```
end
```

2.7 Comprehensions

Comprehensions are the Julian way to build collections concisely:

```
# Array comprehension
squares = [x^2 for x in 1:10]
# [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# With a filter
even_squares = [x^2 for x in 1:10 if iseven(x)]
# [4, 16, 36, 64, 100]

# 2D comprehension → Matrix
multiplication_table = [i * j for i in 1:5, j in 1:5]

# Generator expression (lazy - no memory allocation)
total = sum(1/k^2 for k in 1:1_000_000)
#    ²/6

# Dictionary comprehension
word_lengths = Dict(w => length(w) for w in ["Julia", "Python", "R"])
# Dict("Julia" => 5, "Python" => 6, "R" => 1)

# Set comprehension
unique_remainders = Set(x % 7 for x in 1:100)
```

Performance tip

Generator expressions (without square brackets) are lazy: `sum(x^2 for x in 1:10^8)` uses almost no memory because it never allocates the array. Always prefer generators when you only need to aggregate a result.

2.8 Tuples and named tuples

```
# Tuples - immutable, ordered collections
point = (3.0, 4.0)
point[1]                # 3.0
x, y = point            # destructuring

# Named tuples - tuples with field names
patient = (name="Alice", age=34, bp=120)
patient.name            # "Alice"
patient.age             # 34

# Useful for returning multiple values from functions
```

```julia
function divrem_custom(a, b)
    return (quotient=a ÷ b, remainder=a % b)
end
result = divrem_custom(17, 5)
result.quotient         # 3
result.remainder        # 2
```

2.9 Nothing, Missing, and error handling

```julia
# Nothing - the absence of a value (like None in Python)
function maybe_find(collection, target)
    for item in collection
        item == target && return item
    end
    return nothing
end

result = maybe_find([1, 2, 3], 5)
isnothing(result)       # true

# Missing - represents missing data (like NA in R)
data = [1.0, 2.0, missing, 4.0]
sum(data)               # missing (propagates!)
sum(skipmissing(data))  # 7.0

# Exception handling
try
    x = parse(Int, "not_a_number")
catch e
    println("Error: $(typeof(e)) - $(e.msg)")
finally
    println("This always runs")
end
```

Exercise

1. Explore the type hierarchy: starting from `Float64`, use `supertype()` repeatedly to climb to `Any`. How many levels are there?
2. Write a function `classify_bmi(weight, height)` that returns a named tuple `(bmi=..., category=...)` using WHO BMI categories.
3. Using a comprehension, generate all Pythagorean triples (a, b, c) with $a < b < c \leq 50$ where $a^2 + b^2 = c^2$.
4. Create a vector with `missing` values: `v = [10, missing, 30, missing, 50]`. Compute the mean of the non-missing values.
5. Write a program that reads a string of DNA nucleotides (e.g., `"ATCGGATCCA"`) and returns a dictionary counting each nucleotide.

6. Use string interpolation and a loop to print a formatted multiplication table for numbers 1 through 10.

2.10 Chapter summary

- Julia has a rich numeric type hierarchy: `Int64`, `Float64`, `BigInt`, `Rational`, etc.
- Variables are untyped bindings; values carry the type.
- Strings are UTF-8, immutable, and support interpolation with `$`.
- Control flow uses `if/elseif/else`, `for`, `while`, and short-circuit operators.
- Comprehensions and generators build collections concisely and efficiently.
- `missing` propagates through computations; use `skipmissing()` to handle it.
- Named tuples are lightweight, immutable structs for returning structured data.

Chapter 3

Functions and multiple dispatch

> *"Multiple dispatch is to Julia what objects are to Java: the organizing principle around which everything else revolves."*

3.1 Defining functions

Julia offers three syntactic forms for functions:

```
# Standard form
function greet(name)
    return "Hello, $name!"
end

# Short form (for one-liners)
square(x) = x^2

# Anonymous (lambda) function
double = x -> 2x
```

Functions are first-class objects:

```
# Pass functions as arguments
map(square, [1, 2, 3, 4])    # [1, 4, 9, 16]
filter(isodd, 1:10)          # [1, 3, 5, 7, 9]

# Apply with do-block syntax
map(1:5) do x
    x^2 + 1
end
# [2, 5, 10, 17, 26]
```

3.2 Arguments and return values

```
# Positional arguments
```

```julia
function power(base, exponent)
    return base^exponent
end
power(2, 10)    # 1024

# Default values
function power(base, exponent=2)
    return base^exponent
end
power(5)        # 25
power(5, 3)     # 125

# Keyword arguments (after semicolon)
function create_grid(; rows=10, cols=10, fill_value=0.0)
    return fill(fill_value, rows, cols)
end
create_grid(rows=3, cols=4, fill_value=1.0)

# Varargs (variable number of arguments)
function mysum(args...)
    total = 0
    for a in args
        total += a
    end
    return total
end
mysum(1, 2, 3, 4, 5)    # 15

# Multiple return values (as tuple)
function minmax(v)
    return minimum(v), maximum(v)
end
lo, hi = minmax([3, 1, 4, 1, 5, 9])
```

Julia tip

Julia convention: functions that modify their arguments end with `!`. For example, `sort(v)` returns a sorted copy, while `sort!(v)` sorts in place. Always name your mutating functions with `!`.

3.3 Type annotations

Type annotations constrain what a function accepts:

```julia
# Annotate argument types
function add(x::Number, y::Number)
    return x + y
end
```

```
add(3, 4.5)         # 7.5 - works (Int64 <: Number, Float64 <: Number)
add("a", "b")       # MethodError - String is not a Number

# Annotate return type (mainly for documentation)
function safe_sqrt(x::Real)::Float64
    x < 0 && error("Cannot take sqrt of negative number")
    return sqrt(x)
end

# Parametric type annotations
function first_element(v::Vector{T}) where T
    return v[1]::T
end
```

> **Caution**
>
> Do not over-annotate types. Writing `f(x::Float64)` prevents the function from working with `Int64`, `Float32`, or `BigFloat`. Prefer abstract types like `Real`, `Number`, or `AbstractVector` unless you need to restrict dispatch.

3.4 Multiple dispatch

Multiple dispatch is Julia's defining feature. A single function can have many *methods*, and Julia selects the most specific one based on the types of *all* arguments:

```
# Define an abstract type and concrete subtypes
abstract type Shape end

struct Circle <: Shape
    radius::Float64
end

struct Rectangle <: Shape
    width::Float64
    height::Float64
end

struct Triangle <: Shape
    base::Float64
    height::Float64
end

# Define area() with multiple methods
area(c::Circle) =   * c.radius^2
area(r::Rectangle) = r.width * r.height
area(t::Triangle) = 0.5 * t.base * t.height

# Julia dispatches based on the argument type
area(Circle(5.0))           # 78.539...
```

```
area(Rectangle(3.0, 4.0))  # 12.0
area(Triangle(6.0, 3.0))   # 9.0

# Check all methods of a function
methods(area)   # 3 methods
```

Dispatch on multiple arguments:

```
# Collision detection between shapes
collides(a::Circle, b::Circle) =
    sqrt((a.x - b.x)^2 + (a.y - b.y)^2) < a.radius + b.radius

collides(a::Circle, b::Rectangle) = # ... different algorithm
collides(a::Rectangle, b::Rectangle) = # ... yet another algorithm

# This is why it's called MULTIPLE dispatch:
# the method is chosen based on BOTH argument types
```

Julia tip

Compare with single dispatch (Python, Java): in `a.collides(b)`, only the type of `a` determines which method runs. In Julia, `collides(a, b)` considers the types of both `a` and `b`. This is far more expressive for mathematical and scientific code.

3.5 Closures

A closure captures variables from its enclosing scope:

```
function make_counter(start=0)
    count = start
    increment() = (count += 1; count)
    reset() = (count = start; count)
    return (increment=increment, reset=reset)
end

c = make_counter(10)
c.increment()    # 11
c.increment()    # 12
c.increment()    # 13
c.reset()        # 10

# Closures are common with map/filter
threshold = 0.5
data = rand(10)
filtered = filter(x -> x > threshold, data)

# Adder factory
make_adder(n) = x -> x + n
```

```
add5 = make_adder(5)
add5(10)         # 15
```

3.6 Function composition and piping

```
# Composition operator   (\circ + Tab)
f = sqrt ∘ abs
f(-16)           # 4.0

# Pipe operator |>
-16 |> abs |> sqrt    # 4.0

# Chain operations on data
result = [3, 1, 4, 1, 5, 9, 2, 6] |>
    sort |>
    unique |>
    reverse
# [9, 6, 5, 4, 3, 2, 1]

# With anonymous functions in the pipe
[1, 2, 3, 4, 5] |>
    xs -> filter(isodd, xs) |>
    xs -> map(x -> x^2, xs) |>
    sum
# 35 (= 1 + 9 + 25)
```

3.7 Metaprogramming basics

Julia code is represented as data structures called *expressions*:

```
# An expression is a Julia data structure
ex = :(2 + 3 * x)
typeof(ex)         # Expr
dump(ex)           # shows the AST

# Evaluate an expression
x = 10
eval(ex)           # 32

# Macros transform code at parse time
macro sayhello(name)
    return :(println("Hello, ", $name, "!"))
end

@sayhello "Julia"  # prints: Hello, Julia!

# Useful built-in macros
```

```
@show 2 + 3        # prints: 2 + 3 = 5, returns 5
@time sum(rand(10^7))  # prints elapsed time and allocations
@assert 1 + 1 == 2 "Math is broken!"
```

```
# String macros
r"pattern"    # Regex
v"1.10"       # VersionNumber
raw"no \n escaping"  # raw string

# A practical macro: @elapsed
t = @elapsed begin
    A = rand(1000, 1000)
    B = A * A'
end
println("Matrix multiply took $t seconds")
```

> **Caution**
>
> Metaprogramming is powerful but can make code hard to read. Use macros sparingly and only when normal functions cannot achieve the same result. Most Julia code should be plain functions with multiple dispatch.

3.8 Scope rules

```
# Global scope
x = 10

# Local scope (functions, for loops, let blocks)
function demo()
    y = 20        # local to demo
    println(x)    # can read global x
    # x += 1      # ERROR in Julia 1.10 - use 'global x' first
    return y
end

# let blocks create a fresh scope
let
    temp = 42
    println(temp)
end
# temp is not defined here

# for loops have local scope
for i in 1:3
    local_var = i^2
end
# local_var is not defined here
```

```
# But in the REPL (soft scope), for loops CAN access globals:
# julia> total = 0
# julia> for i in 1:10; total += i; end  # works in REPL
```

Exercise

1. Define a struct `Point2D` with fields `x::Float64` and `y::Float64`. Write a multi-dispatch `distance` function that computes: (a) distance from origin for one point, (b) distance between two points.
2. Implement a simple linked list using parametric types: `struct Cons{T}` with `head::T` and `tail`. Write `mymap(f, list)` using recursion.
3. Write a function `make_polynomial(coeffs)` that returns a closure evaluating the polynomial. For example, `make_polynomial([1, 2, 3])` returns $f(x) = 1 + 2x + 3x^2$.
4. Use the pipe operator to take the vector `1:100`, filter multiples of 3, square them, and compute their sum.
5. Write a macro `@repeat n expr` that expands to a loop executing `expr` exactly `n` times.
6. Define an abstract type `Animal` with subtypes `Dog` and `Cat`. Write a multi-dispatch `interact(a, b)` function with four methods (Dog-Dog, Dog-Cat, Cat-Dog, Cat-Cat) that each return a different string.

3.9 Chapter summary

- Julia has three function forms: standard, short, and anonymous.
- Functions support default values, keyword arguments, and varargs.
- Multiple dispatch selects the most specific method based on all argument types.
- Closures capture variables from enclosing scopes.
- The pipe operator `|>` and composition $\circ$ enable functional-style code.
- Metaprogramming (expressions, macros) transforms code at parse time.
- Convention: mutating functions end with `!`.

Chapter 4

Arrays and linear algebra

> *"In Julia, arrays are not a library feature. They are a language feature, and everything is built to make them fast."*

4.1 Creating arrays

```
# 1D arrays (vectors)
v = [1, 2, 3, 4, 5]          # Vector{Int64}
w = [1.0, 2.0, 3.0]          # Vector{Float64}
mixed = Any[1, "two", 3.0]   # Vector{Any} - avoid for performance

# Ranges (lazy, no memory allocation)
r = 1:10                     # UnitRange
r2 = 0.0:0.1:1.0             # StepRangeLen
collect(1:5)                  # [1, 2, 3, 4, 5]

# Constructor functions
zeros(5)                      # [0.0, 0.0, 0.0, 0.0, 0.0]
ones(3)                       # [1.0, 1.0, 1.0]
fill(42, 5)                   # [42, 42, 42, 42, 42]
rand(5)                       # 5 uniform random numbers in [0,1)
randn(5)                      # 5 standard normal random numbers
collect(range(0, 1, length=11))  # 11 equally spaced points
```

4.2 2D arrays (matrices)

```
# Row elements separated by spaces, rows by semicolons
A = [1 2 3; 4 5 6; 7 8 9]    # 3×3 Matrix{Int64}

# Or with newlines
B = [1.0 2.0
     3.0 4.0]                 # 2×2 Matrix{Float64}

# Constructor functions
zeros(3, 4)                   # 3×4 zero matrix
```

```
ones(2, 3)                      # 2×3 ones matrix
rand(3, 3)                      # 3×3 random matrix
Matrix{Float64}(I, 3, 3)       # 3×3 identity (need: using LinearAlgebra)

# Useful constructors
diagm([1, 2, 3])                # 3×3 diagonal matrix
reshape(1:12, 3, 4)             # 3×4 matrix filled column-major

# Properties
size(A)                         # (3, 3)
size(A, 1)                      # 3 (rows)
ndims(A)                        # 2
length(A)                       # 9 (total elements)
eltype(A)                       # Int64
```

4.3 Indexing and slicing

```
A = [10 20 30; 40 50 60; 70 80 90]

# Single element
A[2, 3]              # 60 (row 2, column 3)
A[1]                 # 10 (linear index, column-major order)

# Slices (views into the array)
A[1, :]              # [10, 20, 30]   - first row
A[:, 2]              # [20, 50, 80]   - second column
A[1:2, 2:3]          # [20 30; 50 60] - submatrix

# Logical indexing
v = [3, 1, 4, 1, 5, 9, 2, 6]
v[v .> 3]            # [4, 5, 9, 6]

# Views (no copy - alias to original memory)
v_sub = @view A[1:2, :]
v_sub[1, 1] = 999    # modifies A!

# findall, findfirst
findall(x -> x > 50, A)   # CartesianIndex array
findfirst(x -> x > 50, A) # CartesianIndex(3, 1) → value 70
```

Performance tip

Slicing with `A[1:2, :]` allocates a new array. Use `@view A[1:2, :]` or the macro `@views` before a block of code to avoid allocations. In hot loops, views can dramatically reduce memory pressure.

4.4 Broadcasting: the dot syntax

Broadcasting applies an operation element-wise, automatically expanding dimensions:

```
# Element-wise operations with dot
a = [1, 2, 3]
b = [10, 20, 30]

a .+ b              # [11, 22, 33]
a .* b              # [10, 40, 90]
a .^ 2              # [1, 4, 9]
sin.(a)             # [sin(1), sin(2), sin(3)]

# Broadcasting a scalar
a .+ 10             # [11, 12, 13]

# Fused broadcasting with @. (one allocation for the whole expression)
@. result = sin(a) + cos(b) * a^2

# Broadcasting with matrices
A = [1 2; 3 4]
v = [10, 20]

A .+ v              # adds v to each column
A .+ v'             # adds v' to each row (v' is a 1×2 matrix)

# Comparison broadcasting
A .> 2              # [false false; true true]
```

Julia tip

The `@.` macro adds dots to every function call and operator in the expression. So `@. y = sin(x) + cos(x)^2` becomes `y .= sin.(x) .+ cos.(x).^2`, which fuses into a single loop with one allocation.

4.5 Common array operations

```
v = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

# Aggregation
sum(v)                  # 39
prod(v)                 # 32400
minimum(v), maximum(v)  # (1, 9)
extrema(v)              # (1, 9)
mean(v)                 # 3.9 (need: using Statistics)
std(v)                  # standard deviation

# Sorting
```

```
sort(v)                 # sorted copy
sort!(v)                # in-place sort
sortperm(v)             # indices that would sort v

# Searching
in(5, v)                # true (or: 5   v)
count(isodd, v)         # number of odd elements
any(x -> x > 8, v)     # true
all(x -> x > 0, v)     # true

# Mutation
push!(v, 99)            # append
pop!(v)                 # remove last
pushfirst!(v, 0)        # prepend
deleteat!(v, 3)         # remove element at index 3
append!(v, [10, 20])    # concatenate

# Concatenation
vcat([1, 2], [3, 4])    # [1, 2, 3, 4]
hcat([1, 2], [3, 4])    # [1 3; 2 4]
```

4.6 Linear algebra

```
using LinearAlgebra

A = [4.0 1.0; 2.0 3.0]
b = [1.0, 2.0]

# Basic operations
A'                      # conjugate transpose (adjoint)
transpose(A)            # transpose (no conjugation)
det(A)                  # determinant: 10.0
tr(A)                   # trace: 7.0
inv(A)                  # inverse (avoid - use \ instead)
rank(A)                 # 2

# Solving linear systems: Ax = b
x = A \ b               # much faster and more stable than inv(A) * b
A * x   b              # true (  is \approx + Tab)

# Eigenvalues and eigenvectors
vals, vecs = eigen(A)
vals                     # eigenvalues
vecs                     # eigenvectors (columns)

# SVD
U, S, V = svd(A)

# Norms
norm(b)                  # Euclidean norm (L2)
```

```
norm(b, 1)               # L1 norm
norm(b, Inf)             # infinity norm
norm(A)                  # operator norm (largest singular value)

# Dot product and cross product
dot([1, 2, 3], [4, 5, 6])    # 32
cross([1, 0, 0], [0, 1, 0])  # [0, 0, 1]

# Special matrix types (Julia optimizes operations for these)
D = Diagonal([1, 2, 3])      # Diagonal matrix
S = Symmetric(A)              # Symmetric wrapper
U = UpperTriangular(A)        # Upper triangular
```

Caution

Never compute `inv(A) * b`. Always use `A \ b`. The backslash operator automatically selects the most efficient and numerically stable algorithm (LU, Cholesky, QR, etc.) based on the matrix structure.

4.7 Matrix factorizations

```
using LinearAlgebra

A = rand(5, 5)
A = A + A'  # make symmetric positive definite (approximately)
A += 5I     # add 5 to diagonal to ensure positive definiteness

# LU factorization
F = lu(A)
F.L          # lower triangular
F.U          # upper triangular
F.p          # permutation vector
F.L * F.U   A[F.p, :]   # true

# Cholesky factorization (for symmetric positive definite)
C = cholesky(A)
C.L          # lower triangular factor
C.L * C.L'   A   # true

# QR factorization
Q, R = qr(A)

# Solve with factorizations (efficient for multiple right-hand sides)
F = lu(A)
x1 = F \ b1    # reuses the factorization
x2 = F \ b2
```

4.8 Sparse arrays

```julia
using SparseArrays

# Create sparse matrices
S = sparse([1, 2, 3, 1], [1, 2, 3, 3], [10, 20, 30, 5], 3, 3)
# 3×3 SparseMatrixCSC with 4 stored entries:
#  [1, 1]  =  10
#  [2, 2]  =  20
#  [1, 3]  =  5
#  [3, 3]  =  30

# Convert dense to sparse
A_dense = [1 0 0; 0 2 0; 0 0 3]
A_sparse = sparse(A_dense)

# Sparse identity
I_sparse = spdiagm(0 => ones(1000))

# Memory comparison
A_dense = zeros(1000, 1000)
A_dense[1,1] = 1.0; A_dense[500,500] = 2.0; A_dense[1000,1000] = 3.0
A_sparse = sparse(A_dense)

Base.summarysize(A_dense)    # ~8 MB
Base.summarysize(A_sparse)   # ~80 bytes (only nonzeros stored)

# Operations work the same
b = rand(1000)
x_dense = A_dense \ b
x_sparse = A_sparse \ b     # much faster for large sparse systems
```

Performance tip

For large systems with few nonzero entries per row (finite elements, graphs, PDEs), sparse matrices can be thousands of times faster than dense ones. Always check the sparsity pattern of your problem.

Exercise

1. Create a 10-element vector of random integers between 1 and 100. Compute the mean, median, and standard deviation.
2. Generate a 5×5 random matrix A, a vector b of length 5, and solve $Ax = b$. Verify that $\|Ax - b\| < 10^{-10}$.
3. Use broadcasting to compute $\sin^2(x) + \cos^2(x)$ for $x \in \{0, 0.1, 0.2, \ldots, 2\pi\}$ and verify the result is always 1 (up to floating-point precision).
4. Create a 1000×1000 tridiagonal matrix using `spdiagm` and solve a linear system with it. Compare the time with a dense solve.

5. Implement the power iteration algorithm to find the dominant eigenvalue of a matrix. Compare with `eigen()`.

6. Use comprehensions to build the $n \times n$ Hilbert matrix $H_{ij} = \frac{1}{i+j-1}$. Compute its condition number with `cond(H)` for $n = 5, 10, 15, 20$. Why does it grow?

4.9 Chapter summary

- Julia arrays are column-major and 1-indexed.
- Broadcasting (dot syntax) fuses element-wise operations into efficient loops.
- Use `@view` to avoid allocations when slicing.
- LinearAlgebra provides `eigen`, `svd`, `lu`, `cholesky`, `qr`, and the backslash solver.
- Always use `A \ b` instead of `inv(A) * b`.
- SparseArrays store only nonzero entries, enabling efficient operations on large sparse systems.
- Special matrix types (Diagonal, Symmetric, Triangular) trigger optimized algorithms automatically.

Chapter 5

DataFrames and data wrangling

"Data wrangling is the process of transforming messy, real-world data into a clean, structured format suitable for analysis."

5.1 The DataFrames.jl package

```
using DataFrames

# Create a DataFrame from columns
df = DataFrame(
    name   = ["Alice", "Bob", "Charlie", "Diana", "Eve"],
    age    = [28, 35, 42, 31, 27],
    salary = [55000, 72000, 88000, 63000, 51000],
    dept   = ["Engineering", "Marketing", "Engineering", "HR", "Marketing"]
)

# Basic inspection
size(df)              # (5, 4)
nrow(df), ncol(df)    # (5, 4)
names(df)             # ["name", "age", "salary", "dept"]
describe(df)          # summary statistics
first(df, 3)          # first 3 rows
last(df, 2)           # last 2 rows
eltype.(eachcol(df))  # column types
```

5.2 Reading and writing data

```
using CSV, DataFrames

# Read CSV files
df = CSV.read("data.csv", DataFrame)

# Read with options
df = CSV.read("data.csv", DataFrame;
    delim=',',
```

```
    header=true,
    missingstring="NA",
    types=Dict(:age => Int, :salary => Float64),
    select=[:name, :age, :salary]   # read only these columns
)

# Read directly from a URL
using Downloads
url = "https://raw.githubusercontent.com/owid/covid-19-data/" *
      "master/public/data/owid-covid-data.csv"
Downloads.download(url, "covid.csv")
covid = CSV.read("covid.csv", DataFrame)

# Write to CSV
CSV.write("output.csv", df)

# Read from RDatasets (built-in classic datasets)
using RDatasets
iris = dataset("datasets", "iris")
mtcars = dataset("datasets", "mtcars")
```

Julia tip

`CSV.read` is very fast — often faster than pandas for large files. For even better performance on huge files, use `CSV.read("big.csv", DataFrame; ntasks=8)` to parallelize parsing across threads.

5.3 Selecting and filtering

```
using DataFrames, RDatasets

iris = dataset("datasets", "iris")

# Select columns
select(iris, :SepalLength, :Species)
select(iris, Not(:PetalWidth))
select(iris, Between(:SepalLength, :PetalLength))
select(iris, r"Sepal")          # regex column selection

# Create new columns with select
select(iris, :SepalLength, :SepalWidth,
    [:SepalLength, :SepalWidth] => ByRow((l, w) -> l * w) => :SepalArea
)

# Filter rows
filter(:Species => ==("setosa"), iris)
filter(row -> row.SepalLength > 5.0 && row.Species == "setosa", iris)
subset(iris, :SepalLength => x -> x .> 5.0)
```

```
# Combine selection and filtering
iris |>
    x -> filter(:Species => ==("virginica"), x) |>
    x -> select(x, :SepalLength, :PetalLength)
```

5.4 Transforming data

```
using DataFrames, Statistics

df = DataFrame(
    city = ["Paris", "London", "Berlin", "Madrid", "Rome"],
    pop_million = [2.16, 8.98, 3.64, 3.22, 2.87],
    area_km2 = [105, 1572, 892, 604, 1285]
)

# Add a new column
transform(df, [:pop_million, :area_km2] =>
    ByRow((p, a) -> p * 1e6 / a) => :density)

# In-place transformation
transform!(df, :city => ByRow(uppercase) => :city_upper)

# Multiple transformations
transform(df,
    :pop_million => (x -> x .* 1e6) => :population,
    :area_km2 => (x -> x ./ 2.59) => :area_sq_miles
)

# Replace values
df.city = replace.(df.city, "Paris" => "Paris (France)")

# Handle missing data
df_missing = DataFrame(
    x = [1, 2, missing, 4, missing],
    y = [missing, 2, 3, missing, 5]
)
dropmissing(df_missing)          # drop rows with any missing
dropmissing(df_missing, :x)      # drop rows where x is missing
coalesce.(df_missing.x, 0)      # replace missing with 0
```

5.5 Sorting

```
using DataFrames, RDatasets

mtcars = dataset("datasets", "mtcars")

# Sort by one column
```

```
sort(mtcars, :MPG)                    # ascending
sort(mtcars, :MPG, rev=true)          # descending

# Sort by multiple columns
sort(mtcars, [:Cyl, :MPG])            # Cyl ascending, then MPG ascending
sort(mtcars, [:Cyl, order(:MPG, rev=true)])

# In-place sort
sort!(mtcars, :HP, rev=true)
```

5.6 Grouping and split-apply-combine

```
using DataFrames, Statistics, RDatasets

iris = dataset("datasets", "iris")

# Group by species
gdf = groupby(iris, :Species)
length(gdf)    # 3 groups

# Apply aggregation
combine(gdf,
    :SepalLength => mean => :mean_sepal_length,
    :SepalLength => std  => :std_sepal_length,
    :PetalLength => mean => :mean_petal_length,
    nrow => :count
)
# 3×5 DataFrame: one row per species

# Multiple grouping columns
mtcars = dataset("datasets", "mtcars")
combine(groupby(mtcars, [:Cyl, :Gear]),
    :MPG => mean => :avg_mpg,
    :HP  => mean => :avg_hp,
    nrow => :count
)

# Transform within groups (keeps all rows)
transform(groupby(iris, :Species),
    :SepalLength => (x -> (x .- mean(x)) ./ std(x)) => :SepalLength_zscore
)

# Filter groups
subset(groupby(mtcars, :Cyl),
    :MPG => x -> x .> median(x)   # keep above-median MPG within each Cyl group
)
```

Julia tip

The split-apply-combine pattern in DataFrames.jl is: (1) `groupby` to split, (2) `combine`/`transform`/`subset` to apply and combine. This is analogous to pandas `groupby().agg()` or dplyr `group_by() %>% summarise()`.

5.7 Joins

```
using DataFrames

# Two related tables
employees = DataFrame(
    id = [1, 2, 3, 4, 5],
    name = ["Alice", "Bob", "Charlie", "Diana", "Eve"],
    dept_id = [10, 20, 10, 30, 20]
)

departments = DataFrame(
    dept_id = [10, 20, 30, 40],
    dept_name = ["Engineering", "Marketing", "HR", "Finance"]
)

# Inner join (only matching rows)
innerjoin(employees, departments, on=:dept_id)

# Left join (keep all employees)
leftjoin(employees, departments, on=:dept_id)

# Right join (keep all departments)
rightjoin(employees, departments, on=:dept_id)

# Outer join (keep everything)
outerjoin(employees, departments, on=:dept_id)

# Join on different column names
sales = DataFrame(emp_id=[1,2,3], revenue=[1000, 2000, 1500])
innerjoin(employees, sales, on=:id => :emp_id)

# Anti-join (employees NOT in sales)
antijoin(employees, sales, on=:id => :emp_id)

# Semi-join (employees who ARE in sales, but only employee columns)
semijoin(employees, sales, on=:id => :emp_id)
```

5.8 Reshaping: stack and unstack

```
using DataFrames
```

```julia
# Wide format
wide = DataFrame(
    country = ["France", "Germany", "Italy"],
    pop_2020 = [67.39, 83.24, 59.55],
    pop_2021 = [67.75, 83.16, 59.24],
    pop_2022 = [67.97, 84.08, 58.86]
)

# Wide → Long (stack / melt)
long = stack(wide, r"pop_", :country;
    variable_name=:year, value_name=:population)

# Clean the year column
transform!(long, :year => ByRow(y -> parse(Int, last(string(y), 4))) => :year)

# Long → Wide (unstack)
unstack(long, :country, :year, :population)
```

Exercise

1. Load the iris dataset from RDatasets. Compute the mean and standard deviation of all four measurements, grouped by species.
2. Load the mtcars dataset. Filter for cars with more than 100 HP and 6 or more cylinders. Sort by MPG descending.
3. Download the Our World in Data COVID-19 CSV. Filter for a country of your choice, select date and new cases, and compute the 7-day rolling average.
4. Create two DataFrames: `students` (id, name, major) and `grades` (student_id, course, grade). Perform an inner join and compute the average grade per major.
5. Using the mtcars dataset, demonstrate the split-apply-combine pattern: group by number of cylinders, compute mean MPG, mean HP, and count.
6. Create a wide-format DataFrame with monthly sales for 4 products over 12 months. Reshape to long format, then compute total annual sales per product.

5.9 Chapter summary

- DataFrames.jl is Julia's tabular data package, similar to pandas and dplyr.
- CSV.jl reads and writes CSV files efficiently, with parallel parsing.
- `select`, `filter`/`subset`, and `transform` are the core manipulation verbs.
- `groupby` + `combine` implements the split-apply-combine pattern.
- Six join types are available: inner, left, right, outer, semi, anti.

- `stack` (wide to long) and `unstack` (long to wide) reshape data.
- RDatasets.jl provides hundreds of classic datasets for practice.

Chapter 6

Visualization

"The purpose of visualization is insight, not pictures." — Ben Shneiderman

6.1 Plots.jl basics

Plots.jl provides a unified API across multiple rendering backends:

```
using Plots

x = range(0, 2 , length=100)
plot(x, sin.(x), label="sin(x)", linewidth=2)
plot!(x, cos.(x), label="cos(x)", linewidth=2)  # add to existing plot
xlabel!("x"); ylabel!("y"); title!("Trigonometric functions")
```

```
# Scatter plot
scatter(randn(50), 2 .* randn(50), label="Data",
    xlabel="x", ylabel="y", markersize=5, alpha=0.7)

# Histogram
histogram(randn(10_000), bins=50, label="N(0,1)", alpha=0.7)

# Heatmap
Z = [sin(x)*cos(y) for x in range(- , ,length=50),
                       y in range(- , ,length=50)]
heatmap(Z, color=:viridis, title="sin(x)cos(y)")
```

Julia tip

The `!` versions (`plot!`, `scatter!`, `xlabel!`) mutate the current plot instead of creating a new one. This is the standard Julian convention for in-place modification.

6.2 More plot types

```
# Bar, contour, surface, pie
```

```
bar(["A","B","C","D"], [23,45,12,38], label="Counts", color=:steelblue)

f(x, y) = x^2 + y^2 - x*y
x = y = range(-3, 3, length=100)
contour(x, y, f, levels=15, fill=true, color=:turbo)
surface(x, y, f, color=:viridis, camera=(30, 60))
```

6.3 Backends

```
gr()            # GR (default) - fast, good for interactive use
plotly()        # Plotly - interactive HTML plots (hover, zoom)
pgfplotsx()     # PGFPlotsX - LaTeX-quality output for papers
gr()            # switch back
```

Julia tip

Use `gr()` for everyday work. Use `pgfplotsx()` for publication figures that integrate with LaTeX. Use `plotly()` for interactive presentations.

6.4 Customization and layouts

```
x = range(0, 4 , length=200)
plot(x, [sin.(x) sin.(x).*exp.(-x/8)],
    label=["sin(x)" "damped"], linewidth=[1 2],
    linestyle=[:dash :solid], color=[:blue :red],
    title="Damped oscillation", xlabel="Time (s)", ylabel="Amplitude",
    legend=:topright, size=(800,400), dpi=300)

# Themes
theme(:ggplot2)     # ggplot2 style
theme(:default)     # reset

# Subplots
p1 = plot(x, sin.(x), title="sin", legend=false)
p2 = plot(x, cos.(x), title="cos", legend=false)
p3 = plot(x, tan.(x), title="tan", ylims=(-5,5), legend=false)
p4 = plot(x, sin.(x).*cos.(x), title="sin*cos", legend=false)
plot(p1, p2, p3, p4, layout=(2,2), size=(800,600))
```

6.5 CairoMakie for publication-quality figures

```
using CairoMakie

fig = Figure(size=(800, 500), fontsize=14)
```

```
ax = Axis(fig[1,1], title="Damped oscillation",
    xlabel="Time (s)", ylabel="Amplitude")
x = range(0, 4 , length=300)
lines!(ax, x, sin.(x), label="sin(x)", linewidth=2)
lines!(ax, x, sin.(x).*exp.(-x/8), label="Damped",
    linewidth=2, linestyle=:dash)
axislegend(ax, position=:rt)
fig
```

```
# Multi-panel figure
fig = Figure(size=(1000, 400), fontsize=12)
ax1 = Axis(fig[1,1], title="(a) Scatter")
scatter!(ax1, randn(100), randn(100), markersize=8)
ax2 = Axis(fig[1,2], title="(b) Histogram")
hist!(ax2, randn(5000), bins=40, color=:orange)
ax3 = Axis(fig[1,3], title="(c) Heatmap")
heatmap!(ax3, randn(30,30), colormap=:inferno)
fig
```

Performance tip

Use CairoMakie for static publication figures (PDF, SVG, PNG). Use GLMakie for interactive 3D visualization. Both share the same API — just change the import.

6.6 AlgebraOfGraphics.jl

AlgebraOfGraphics brings a grammar-of-graphics approach (like R's ggplot2):

```
using AlgebraOfGraphics, CairoMakie, RDatasets
iris = dataset("datasets", "iris")

# Scatter with color mapping + linear fit per group
plt = data(iris) *
    mapping(:SepalLength, :PetalLength, color=:Species) *
    (visual(Scatter) + linear())
draw(plt, axis=(; title="Iris dataset"))

# Faceted plot
plt = data(iris) *
    mapping(:SepalLength, :PetalLength, layout=:Species) *
    visual(Scatter)
draw(plt)
```

Julia tip

AlgebraOfGraphics uses `*` to combine data, mappings, and visuals, and `+` to layer multiple visuals. Think of `*` as "apply to" and `+` as "overlay with."

6.7 Saving figures

```
# Plots.jl
savefig(p, "figure.pdf")   # PDF (vector, best for LaTeX)
savefig(p, "figure.svg")   # SVG (vector)
savefig(p, "figure.png")   # PNG (raster)

# CairoMakie
save("figure.pdf", fig)                    # PDF for papers
save("figure.png", fig, px_per_unit=3)     # high-resolution PNG
```

Caution

For LaTeX documents, always save figures as PDF or SVG. PNG images look blurry when scaled. Set `dpi=300` or `px_per_unit=3` if you must use PNG.

6.8 Practical: Gapminder-style animated scatter plot

```
using Plots, CSV, DataFrames, Downloads

url = "https://raw.githubusercontent.com/plotly/datasets/" *
      "master/gapminderDataFiveYear.csv"
Downloads.download(url, "gapminder.csv")
gm = CSV.read("gapminder.csv", DataFrame)

anim = @animate for yr in sort(unique(gm.year))
    df_yr = filter(:year => ==(yr), gm)
    scatter(df_yr.gdpPercap, df_yr.lifeExp,
        markersize = sqrt.(df_yr.pop) ./ 5000,
        group=df_yr.continent, xlabel="GDP per capita (USD)",
        ylabel="Life expectancy (years)", title="Year: $yr",
        xlims=(0,50_000), ylims=(20,90), xscale=:log10,
        legend=:bottomright, alpha=0.7, size=(900,550))
end
gif(anim, "gapminder.gif", fps=2)
```

```
# Static CairoMakie version for a single year
using CairoMakie
df2007 = filter(:year => ==(2007), gm)
fig = Figure(size=(900,550), fontsize=13)
ax = Axis(fig[1,1], xlabel="GDP per capita (log)", ylabel="Life exp.",
    title="Gapminder 2007", xscale=log10)
for cont in unique(df2007.continent)
    sub = filter(:continent => ==(cont), df2007)
    scatter!(ax, sub.gdpPercap, sub.lifeExp,
        markersize=sqrt.(sub.pop)./4000, label=cont, alpha=0.7)
end
```

```
axislegend(ax, position=:rb)
save("gapminder_2007.pdf", fig)
```

Exercise

1. Create a plot of $\sin(x)$, $\sin(2x)$, $\sin(3x)$ on $[0, 2\pi]$ with proper labels and legend.
2. Generate 1000 bivariate normal points and create a scatter plot with marginal histograms using subplots.
3. Reproduce a scatter matrix of the iris dataset colored by species using AlgebraOfGraphics.jl.
4. Create a 2×2 CairoMakie subplot grid: line, scatter, histogram, heatmap.
5. Save the same figure as PNG (300 DPI), SVG, and PDF. Compare file sizes and quality.
6. Extend the Gapminder animation to highlight a specific country with a larger marker and label.

6.9 Chapter summary

- Plots.jl provides a unified API with backends: GR (fast), Plotly (interactive), PGFPlotsX (LaTeX).
- Common types: `plot`, `scatter`, `histogram`, `heatmap`, `bar`, `contour`, `surface`.
- CairoMakie offers fine-grained control for publication-quality static figures.
- AlgebraOfGraphics.jl uses a composable grammar of graphics with `*` and `+`.
- Save figures as PDF or SVG for LaTeX; use PNG only with high DPI.
- Animations are built with `@animate` and saved with `gif()`.

Chapter 7

Performance and optimization

> *"Premature optimization is the root of all evil — but so is premature pessimization."* — after Donald Knuth

7.1 Why Julia is fast

Julia achieves C-like speed through three pillars:

1. **JIT compilation.** Julia compiles functions to native machine code via LLVM the first time they are called with specific argument types.
2. **Type specialization.** `f(1, 2)` and `f(1.0, 2.0)` compile to different, optimized machine code.
3. **Type inference.** The compiler traces types through function bodies, eliminating runtime type checks when all types are known.

```
function mysum(v)
    s = zero(eltype(v))
    for x in v; s += x; end
    return s
end
mysum([1, 2, 3])        # compiles mysum(::Vector{Int64})
mysum([1.0, 2.0, 3.0])  # compiles mysum(::Vector{Float64})
```

Julia tip

The first call to a function is slow (compilation latency). Julia 1.9+ uses package precompilation to reduce this. Use `PrecompileTools.jl` in your own packages.

7.2 Measuring performance

```
# @time - run twice! First call includes compilation time.
@time sum(rand(10^7))   # compilation
@time sum(rand(10^7))   # true timing
```

```
# BenchmarkTools.jl - accurate benchmarks for fast functions
using BenchmarkTools
v = rand(10_000)

@btime sum($v)          # runs many times, reports minimum
#   1.256  s (0 allocations: 0 bytes)

@benchmark sum($v)      # full statistical report

# IMPORTANT: interpolate variables with $ to avoid global access cost
A = rand(100, 100)
@btime $A * $A          # correct
@btime A * A            # wrong - includes global lookup overhead
```

> **Caution**
>
> Always interpolate input variables with `$` in `@btime` and `@benchmark`. Without interpolation, the benchmark measures global variable access overhead, which skews results.

7.3 Type stability

A function is *type-stable* if the return type depends only on argument types, not values:

```
# Type-UNSTABLE: return type depends on the VALUE of x
function unstable_sqrt(x)
    x >= 0 ? sqrt(x) : "negative"  # returns Float64 OR String!
end

# Type-STABLE: return type always matches input type
stable_sqrt(x) = x >= 0 ? sqrt(x) : NaN

# Common instability: changing variable type in a loop
function bad_sum(n)
    s = 0            # Int, becomes Float64 after s += i/2
    for i in 1:n; s += i / 2; end
    return s
end

function good_sum(n)
    s = 0.0          # Float64 from the start
    for i in 1:n; s += i / 2; end
    return s
end
```

Performance tip

Type stability is the single most important factor for Julia performance. When a function is type-stable, the compiler generates code with no boxing, no dynamic dispatch, and no heap allocations for local variables.

7.4 Inspecting compiled code

```
f(x, y) = x^2 + y^2

@code_warntype f(1.0, 2.0)   # MOST USEFUL: red=instability, blue=stable
@code_typed f(1.0, 2.0)      # typed intermediate representation
@code_llvm f(1.0, 2.0)       # LLVM IR
@code_native f(1.0, 2.0)     # machine assembly
```

```
# Detecting and fixing instability
unstable(x) = x > 0 ? x : 0        # returns Union{Float64, Int64}
stable(x)   = x > 0 ? x : zero(x)  # always returns typeof(x)

@code_warntype unstable(1.0)  # Body::Union{Float64, Int64} - RED
@code_warntype stable(1.0)    # Body::Float64 - BLUE
```

7.5 Common performance pitfalls

7.5.1 Pitfall 1: Global variables

```
x_global = 1.0   # BAD: type could change at any time

function bad_global()
    s = 0.0
    for i in 1:1000; s += x_global * i; end
    return s
end

good_local(x) = (s = 0.0; for i in 1:1000; s += x * i; end; s)

const X_CONST = 1.0   # GOOD: const lets the compiler optimize
good_const() = (s = 0.0; for i in 1:1000; s += X_CONST * i; end; s)

using BenchmarkTools
@btime bad_global()     # ~10  s (slow, allocations)
@btime good_local(1.0)  # ~0.5  s (20x faster)
```

7.5.2 Pitfall 2: Abstract containers and untyped structs

```
v_any   = Any[1.0, 2.0, 3.0]       # BAD: boxed elements
v_typed = Float64[1.0, 2.0, 3.0]   # GOOD: contiguous memory

struct BadPoint; x; y; end                    # BAD: fields are ::Any
struct GoodPoint; x::Float64; y::Float64; end # GOOD: concrete types
struct FlexPoint{T<:Real}; x::T; y::T; end   # BEST: generic AND fast
```

7.5.3 Pitfall 3: Growing arrays in a loop

```
# BAD: push! causes repeated reallocations
bad_build(n) = (v = Float64[]; for i in 1:n; push!(v, sin(i)); end; v)

# GOOD: preallocate
function good_build(n)
    v = Vector{Float64}(undef, n)
    for i in 1:n; v[i] = sin(i); end
    return v
end

# ALSO GOOD: comprehension (Julia preallocates internally)
good_build2(n) = [sin(i) for i in 1:n]
```

7.6 Memory allocation and profiling

```
# @allocated - returns bytes allocated
bytes = @allocated rand(1000, 1000)
println("Allocated $(bytes / 1024^2) MiB")

# In-place operations to reduce allocations
A = rand(1000, 1000)
A .= rand.(Float64)   # reuse memory, near-zero new allocations
```

```
# Statistical profiler
using Profile

function heavy_work()
    A = rand(1000, 1000); B = A * A'; return sum(eigen(B).values)
end

heavy_work()               # warm up
@profile heavy_work()
Profile.print(format=:flat)
Profile.clear()
```

```
# Visual flame graph
using ProfileView
@profview heavy_work()      # wide bars = bottlenecks
```

7.7 Practical: optimize a slow function step by step

```
using BenchmarkTools, LinearAlgebra

# v1: naive - Vector{Any}, type instability
data = Any[rand() for _ in 1:100_000]
function slow_norm(v)
    s = 0  # Int!
    for i in 1:length(v); s = s + v[i]^2; end
    return sqrt(s)
end
@btime slow_norm($data)   # ~2.5 ms

# v2: fix container type
data_typed = rand(100_000)
@btime slow_norm($data_typed)   # ~1.2 ms (2x)

# v3: fix type instability
function better_norm(v)
    s = 0.0
    for i in eachindex(v); s += v[i]^2; end
    return sqrt(s)
end
@btime better_norm($data_typed)  # ~50  s (50x)

# v4: @simd + @inbounds
function fast_norm(v)
    s = 0.0
    @inbounds @simd for i in eachindex(v); s += v[i]^2; end
    return sqrt(s)
end
@btime fast_norm($data_typed)    # ~15  s (170x)

# v5: built-in BLAS
@btime norm($data_typed)         # ~12  s (200x vs v1)
```

Performance tip

The optimization checklist: (1) avoid globals, (2) concrete container types, (3) type stability, (4) preallocate, (5) `@views`, (6) `@inbounds @simd`, (7) `@threads` for parallelism.

Exercise

1. Write `my_dot(a, b)` computing the dot product. Benchmark against `LinearAlgebra.dot` and optimize until within 2x.
2. Use `@code_warntype` to find and fix the instability in: `mystery(x) = (x > 0 ? x : 0) * 2.0`.
3. Benchmark matrix-vector multiplication with `Vector{Any}` vs. `Vector{Float64}` for $n = 1000$.
4. Profile `heavy_work()` and identify which sub-operation takes the most time.
5. Compute $\sum_{k=1}^{n} \sin(k)/k$ via (a) loop, (b) generator, (c) broadcasting. Benchmark all three.
6. Take a slow function from a previous exercise and apply the optimization checklist. Document each step.

7.8 Chapter summary

- Julia achieves C-like speed through JIT compilation, type specialization, and type inference via LLVM.
- Use `@btime` from BenchmarkTools.jl for accurate benchmarks; always interpolate inputs with `$`.
- Type stability is the most important performance factor: use `@code_warntype` to detect instabilities.
- Avoid global variables, abstract containers (`Vector{Any}`), and untyped struct fields.
- Preallocate arrays and use `@views` to reduce memory allocations.
- Profile with `@profile` and visualize with ProfileView.jl to find bottlenecks.

Chapter 8

Scientific computing

"The purpose of computing is insight, not numbers." — Richard Hamming

8.1 The SciML ecosystem

Julia's Scientific Machine Learning ecosystem provides state-of-the-art solvers:

- **DifferentialEquations.jl** — ODE, SDE, PDE solvers
- **Optimization.jl** — unified optimization interface
- **ModelingToolkit.jl** — symbolic modeling and simplification

8.2 Ordinary differential equations

8.2.1 The Lorenz attractor

$$\dot{x} = \sigma(y - x), \quad \dot{y} = x(\rho - z) - y, \quad \dot{z} = xy - \beta z$$

```
using DifferentialEquations, Plots

function lorenz!(du, u, p, t)
     ,  ,   = p
    du[1] =   * (u[2] - u[1])
    du[2] = u[1] * (  - u[3]) - u[2]
    du[3] = u[1] * u[2] -   * u[3]
end

prob = ODEProblem(lorenz!, [1.0, 0.0, 0.0], (0.0, 50.0), (10.0, 28.0, 8/3))
sol  = solve(prob, Tsit5(); saveat=0.01)

plot(sol, idxs=(1,2,3), title="Lorenz attractor",
    legend=false, linewidth=0.5, color=:viridis, linez=sol.t)
```

8.2.2 The SIR epidemic model

$$\dot{S} = -\beta SI/N, \quad \dot{I} = \beta SI/N - \gamma I, \quad \dot{R} = \gamma I$$

```
function sir!(du, u, p, t)
    S, I, R = u;  ,   = p; N = S + I + R
    du[1] = -  * S * I / N
    du[2] =    * S * I / N -   * I
    du[3] =    * I
end

prob = ODEProblem(sir!, [0.99, 0.01, 0.0], (0.0, 200.0), (0.3, 0.1))
sol  = solve(prob, Tsit5())

plot(sol, label=["S" "I" "R"], xlabel="Time (days)",
    ylabel="Fraction", title="SIR model (R  = 3.0)", linewidth=2)
```

Julia tip

The `!` in `lorenz!` means the function modifies `du` in place. In-place ODE functions avoid memory allocation at every time step, which is critical for performance.

8.3 Problem types and solvers

```
# SDE: du = f(u,p,t)dt + g(u,p,t)dW
function noise!(du, u, p, t)
    du[1] = 0.1 * u[1]
end
prob_sde = SDEProblem(sir!, noise!, u0, tspan, p)

# Solver selection:
sol = solve(prob, Tsit5())    # explicit RK 5(4), non-stiff (default)
sol = solve(prob, Vern7())    # 7th order, high accuracy
sol = solve(prob, Rodas5())   # implicit, for stiff problems
sol = solve(prob, TRBDF2())   # L-stable, stiff-aware

# Controlling accuracy
sol = solve(prob, Tsit5(); abstol=1e-10, reltol=1e-10, saveat=0.1)
```

Performance tip

For stiff ODEs (chemical kinetics, circuits, reaction-diffusion), implicit solvers like `Rodas5()` can be thousands of times faster than explicit ones. If `Tsit5()` is very slow, switch to an implicit solver.

8.4 Working with solutions

```
sol.t              # time points
sol.u              # solution vectors at each time point
sol(10.5)          # interpolate at any time (continuous output)
sol[2, :]          # second component over all time steps
Array(sol)         # convert to matrix (n_vars × n_times)
```

8.5 Optimization

8.5.1 Minimizing the Rosenbrock function

$f(x, y) = (a - x)^2 + b(y - x^2)^2$, minimum at (a, a^2).

```
using Optimization, OptimizationOptimJL

rosenbrock(u, p) = (p[1] - u[1])^2 + p[2] * (u[2] - u[1]^2)^2

# Derivative-free
prob = OptimizationProblem(rosenbrock, [-1.0, 1.0], (1.0, 100.0))
sol  = solve(prob, NelderMead())
println("Minimum at: $(sol.u)")    #   [1.0, 1.0]
```

```
# Gradient-based with automatic differentiation
optf = OptimizationFunction(rosenbrock, Optimization.AutoForwardDiff())
prob = OptimizationProblem(optf, [-1.0, 1.0], (1.0, 100.0);
    lb=[-2.0,-2.0], ub=[2.0,2.0])
sol  = solve(prob, LBFGS())
println("LBFGS result: $(sol.u)")  #   [1.0, 1.0]
```

8.6 ModelingToolkit.jl

```
using ModelingToolkit, DifferentialEquations
using ModelingToolkit: t_nounits as t, D_nounits as D

@variables S(t) I(t) R(t)
@parameters

N = S + I + R
eqs = [D(S) ~ - *S*I/N, D(I) ~  *S*I/N -  *I, D(R) ~  *I]
@named sir_sys = ODESystem(eqs, t)
sir_simple = structural_simplify(sir_sys)

prob = ODEProblem(sir_simple,
    [S=>0.99, I=>0.01, R=>0.0], (0.0, 200.0), [ =>0.3,  =>0.1])
```

```
sol = solve(prob, Tsit5())
```

Julia tip

ModelingToolkit automatically detects conserved quantities ($S + I + R = 1$), eliminates redundant equations, and generates optimized code. For large systems this significantly reduces problem size.

8.7 Linear algebra deep dive

```
using LinearAlgebra, SparseArrays, BenchmarkTools

# Factorize once, solve many times
n = 2000; A = rand(n,n); A = A + A' + n*I
F = cholesky(A)
x1 = F \ rand(n)    # reuses factorization
x2 = F \ rand(n)

# Sparse systems (e.g., 1D Laplacian)
function laplacian_1d(n)
    e = ones(n-1)
    return spdiagm(-1 => -e, 0 => 2ones(n), 1 => -e)
end

A_sp = laplacian_1d(10_000); b = rand(10_000)
@btime $A_sp \ $b     # ~1 ms (sparse)
@btime Matrix($A_sp) \ $b  # ~500 ms (dense, 500x slower)
```

```
# Iterative solvers for very large systems
using IterativeSolvers
A = laplacian_1d(100_000); b = rand(100_000)
x, info = cg(A, b; log=true, tol=1e-10)
println("Converged in $(info.iters) iterations")
```

8.8 Practical: SIR model with parameter fitting

```
using DifferentialEquations, Optimization, OptimizationOptimJL, Plots

# Generate synthetic "observed" data with noise
true_p = (0.35, 0.1)
prob_true = ODEProblem(sir!, [0.99,0.01,0.0], (0.0,100.0), true_p)
sol_true  = solve(prob_true, Tsit5(); saveat=1.0)
I_obs = max.(sol_true[2,:] .+ 0.005 .* randn(length(sol_true.t)), 0.0)

# Loss function: sum of squared errors on infected curve
```

```
function loss(params, data)
    t_obs, I_obs, u0 = data
    prob = ODEProblem(sir!, u0, (t_obs[1], t_obs[end]), params)
    sol  = solve(prob, Tsit5(); saveat=t_obs)
    sol.retcode != :Success && return Inf
    return sum((sol[2,:] .- I_obs).^2)
end

# Optimize from a wrong initial guess
data = (sol_true.t, I_obs, [0.99, 0.01, 0.0])
optf = OptimizationFunction(loss, Optimization.AutoForwardDiff())
prob = OptimizationProblem(optf, [0.5, 0.2], data; lb=[0.01,0.01],
↪  ub=[1.0,1.0])
result = solve(prob, LBFGS())
println("True:  =0.35,  =0.1")
println("Fitted:  =$(round(result.u[1],digits=3)),  =$(round(result.u[2],digits=3))")
```

```
# Plot the fit vs observed data
sol_fit = solve(ODEProblem(sir!, [0.99,0.01,0.0], (0.0,100.0), result.u),
↪  Tsit5())
plot(sol_fit, idxs=2, label="Fitted I(t)", linewidth=2)
scatter!(sol_true.t, I_obs, label="Observed", markersize=2, alpha=0.5)
xlabel!("Time (days)"); ylabel!("Fraction infected")
title!("SIR parameter fitting")
```

Caution

Parameter fitting for ODE models can have multiple local minima. Try several initial guesses and verify that $R_0 = \beta/\gamma$ is reasonable for the disease being modeled.

Exercise

1. Solve the Lotka–Volterra system $\dot{x} = \alpha x - \beta xy$, $\dot{y} = \delta xy - \gamma y$. Plot populations over time and a phase portrait.
2. Add stochastic noise to the SIR model using `SDEProblem`. Compare 10 stochastic trajectories with the deterministic solution.
3. Minimize the Rastrigin function $f(\mathbf{x}) = 10n + \sum_i [x_i^2 - 10\cos(2\pi x_i)]$ for $n = 5$.
4. Build a 2D Laplacian as a sparse matrix. Solve the Poisson equation with direct and iterative solvers. Compare timings for $n = 50, 100, 200$.
5. Rewrite the SIR model with ModelingToolkit.jl and verify it gives the same solution.
6. Extend the parameter fitting to fit both infected and recovered curves simultaneously.

8.9 Chapter summary

- DifferentialEquations.jl solves ODEs, SDEs, and more with automatic algorithm selection.
- Define ODE functions in-place (`f!`) for performance; use `Tsit5()` for non-stiff and `Rodas5()` for stiff problems.
- Optimization.jl provides a unified interface; use `AutoForwardDiff()` for gradient-based methods.
- ModelingToolkit.jl enables symbolic problem definition with automatic simplification.
- Sparse matrices and iterative solvers are essential for large-scale linear systems.
- Parameter fitting combines ODE solving with optimization to match models to data.

Chapter 9

Machine learning with Julia

> *"Machine learning is the science of getting computers to learn from data without being explicitly programmed."* — Arthur Samuel

9.1 The MLJ.jl framework

MLJ provides a unified interface to dozens of ML models:

```
using MLJ

# The MLJ workflow:
# 1. Load data → 2. Load model → 3. Wrap in machine → 4. Fit → 5. Predict → 6.
↪  Evaluate
models()                 # list all registered models
models("RandomForest")   # search by name
```

Julia tip

MLJ separates the *model* (hyperparameters) from the *machine* (model + data). You can change hyperparameters without reloading data, and swap models without changing pipeline code.

9.2 Classification: decision trees and random forests

```
using MLJ, DataFrames
iris = load_iris()
X, y = iris[1], iris[2]   # features and target
train, test = partition(eachindex(y), 0.7, shuffle=true, rng=42)

# Decision tree
DecisionTreeClassifier = @load DecisionTreeClassifier pkg=DecisionTree
dtc = DecisionTreeClassifier(max_depth=5)
mach = machine(dtc, X, y)
fit!(mach, rows=train)
```

```
y_pred = predict_mode(mach, rows=test)
acc = sum(y_pred .== y[test]) / length(test)
println("Decision tree accuracy: $(round(acc, digits=3))")
```

```
# Random forest
RandomForestClassifier = @load RandomForestClassifier pkg=DecisionTree
rf = RandomForestClassifier(n_trees=100, max_depth=10)
mach_rf = machine(rf, X, y)
fit!(mach_rf, rows=train)
y_pred_rf = predict_mode(mach_rf, rows=test)
acc_rf = sum(y_pred_rf .== y[test]) / length(test)
println("Random forest accuracy: $(round(acc_rf, digits=3))")
```

9.3 Regression

```
using MLJ
X, y = make_regression(200, 5; noise=0.5, rng=42)
train, test = partition(eachindex(y), 0.7, shuffle=true, rng=42)

# Linear regression
LinearRegressor = @load LinearRegressor pkg=MLJLinearModels
mach_lr = machine(LinearRegressor(), X, y)
fit!(mach_lr, rows=train)
rmse_lr = rms(predict(mach_lr, rows=test), y[test])

# Ridge regression (L2 regularization)
RidgeRegressor = @load RidgeRegressor pkg=MLJLinearModels
mach_ridge = machine(RidgeRegressor(lambda=1.0), X, y)
fit!(mach_ridge, rows=train)
rmse_ridge = rms(predict(mach_ridge, rows=test), y[test])
println("Linear RMSE: $(round(rmse_lr,digits=3)), Ridge: $(round(rmse_ridge,digits=3))")
```

9.4 Cross-validation and hyperparameter tuning

```
using MLJ

# 5-fold cross-validation
evaluate(rf, X, y;
    resampling = CV(nfolds=5, shuffle=true, rng=42),
    measures   = [accuracy, log_loss, kappa])

# Stratified CV (preserves class proportions)
evaluate(rf, X, y;
    resampling = StratifiedCV(nfolds=5, shuffle=true, rng=42),
```

```
    measures   = [accuracy, balanced_accuracy])
```

```
# Grid search
r_ntrees = range(rf, :n_trees, lower=10, upper=200)
r_depth  = range(rf, :max_depth, lower=2, upper=20)

tuned_rf = TunedModel(model=rf, ranges=[r_ntrees, r_depth],
    tuning=Grid(resolution=10), resampling=CV(nfolds=5), measure=accuracy)
mach_tuned = machine(tuned_rf, X, y)
fit!(mach_tuned)

best = fitted_params(mach_tuned).best_model
println("Best: n_trees=$(best.n_trees), max_depth=$(best.max_depth)")

# Random search (faster for large search spaces)
tuned_rand = TunedModel(model=rf, ranges=[r_ntrees, r_depth],
    tuning=RandomSearch(rng=42), n=50, resampling=CV(nfolds=5),
    ↪  measure=accuracy)
```

9.5 MLJ pipelines

```
Standardizer = @load Standardizer pkg=MLJModels
PCA = @load PCA pkg=MultivariateStats
KNN = @load KNNClassifier pkg=NearestNeighborModels

pipe = Standardizer() |> PCA(maxoutdim=3) |> KNN(K=5)

evaluate(pipe, X, y; resampling=CV(nfolds=5), measures=[accuracy])

# Tune within a pipeline
r_K = range(pipe, :(knn_classifier.K), lower=1, upper=30)
tuned_pipe = TunedModel(model=pipe, ranges=[r_K],
    tuning=Grid(resolution=15), resampling=CV(nfolds=5), measure=accuracy)
```

Julia tip

Random search often finds good hyperparameters faster than grid search. With n evaluations, random search explores each dimension in n unique values, while grid search uses only $\sqrt{n}$.

9.6 Flux.jl: deep learning

```
using Flux

model = Chain(
```

```
    Dense(4 => 32, relu),      # 4 inputs → 32 hidden units
    Dense(32 => 16, relu),
    Dense(16 => 3),            # 3 outputs
    softmax
)

sum(length, Flux.params(model))   # 723 parameters
model(rand(Float32, 4))           # forward pass → 3 probabilities
```

9.7 MNIST digit classifier with Flux

```
using Flux, MLDatasets, Statistics

# Load and preprocess
train_x, train_y = MLDatasets.MNIST(split=:train)[:]
test_x, test_y   = MLDatasets.MNIST(split=:test)[:]

x_train = Float32.(reshape(train_x, 784, :))
x_test  = Float32.(reshape(test_x, 784, :))
y_train = Flux.onehotbatch(train_y, 0:9)
y_test  = Flux.onehotbatch(test_y, 0:9)

train_loader = Flux.DataLoader((x_train, y_train); batchsize=128, shuffle=true)
```

```
# Model, loss, optimizer
model = Chain(Dense(784=>256,relu), Dropout(0.3),
              Dense(256=>128,relu), Dropout(0.3), Dense(128=>10))
loss(m, x, y) = Flux.logitcrossentropy(m(x), y)
opt = Flux.setup(Adam(0.001), model)

# Training loop
for epoch in 1:10
    for (xb, yb) in train_loader
        grads = Flux.gradient(m -> loss(m, xb, yb), model)
        Flux.update!(opt, model, grads[1])
    end
    y_pred = Flux.onecold(model(x_test), 0:9)
    acc = mean(y_pred .== test_y)
    println("Epoch $epoch: test_acc=$(round(acc, digits=4))")
end
```

Performance tip

For GPU acceleration: `model = model |> gpu` and `x_train = x_train |> gpu`. Requires `CUDA.jl` and an NVIDIA GPU. The training loop stays unchanged.

9.8 Practical: MLJ classical models vs Flux neural net

```julia
using MLJ, Flux, MLDatasets, Statistics

# Prepare MNIST for MLJ (5000 samples for speed)
train_x, train_y = MLDatasets.MNIST(split=:train)[:]
x_flat = Float64.(reshape(train_x, 784, :))'
X_mlj = MLJ.table(x_flat)
y_mlj = coerce(categorical(train_y), Multiclass)
idx = shuffle(1:60000)[1:5000]
X_sub, y_sub = selectrows(X_mlj, idx), y_mlj[idx]

# Random forest
RandomForestClassifier = @load RandomForestClassifier pkg=DecisionTree
e_rf = evaluate(RandomForestClassifier(n_trees=100), X_sub, y_sub;
    resampling=CV(nfolds=5), measures=[accuracy], verbosity=0)
println("RF accuracy: $(round(e_rf.measurement[1], digits=4))")

# KNN
KNN = @load KNNClassifier pkg=NearestNeighborModels
e_knn = evaluate(KNN(K=5), X_sub, y_sub;
    resampling=CV(nfolds=5), measures=[accuracy], verbosity=0)
println("KNN accuracy: $(round(e_knn.measurement[1], digits=4))")

# Neural net test accuracy from training above
# Typical results: RF ~0.95, KNN ~0.96, NN ~0.97
```

Caution

Classical models (random forest, KNN, SVM) are often competitive with neural networks on tabular and small-scale data. Use neural networks for large datasets or image/text/sequence data. Start simple and increase complexity only when needed.

Exercise

1. Train a decision tree, random forest, and KNN on iris. Compare 5-fold CV accuracies.
2. Build a pipeline: standardize, PCA to 2D, random forest. Evaluate with cross-validation.
3. Tune KNN's K from 1 to 30 on iris. Plot accuracy vs. K.
4. Build a Flux regression network to predict house prices. Report test RMSE.
5. Modify the MNIST classifier to use `Conv` layers. Compare accuracy and training time.
6. Implement early stopping: stop when validation loss has not improved for 3 epochs.

CHAPTER 9. MACHINE LEARNING WITH JULIA

9.9 Chapter summary

- MLJ.jl provides a unified interface: load model, create machine, fit, predict, evaluate.
- Use `evaluate()` with `CV` or `StratifiedCV` for robust performance estimation.
- `TunedModel` supports grid search and random search for hyperparameter tuning.
- Pipelines chain preprocessing and modeling into a single object.
- Flux.jl is Julia's deep learning library: `Chain`, `Dense`, `Conv`, and custom training loops.
- For MNIST, a simple dense network achieves $\sim$97% accuracy in under a minute.
- Start with classical models; use neural networks when data scale or structure demands it.

Chapter 10

Capstone projects

"The best way to learn programming is to build something you care about."

10.1 Overview

Five capstone projects integrate concepts from the entire course. Each requires data handling, computation, visualization, and a written report. Choose one (or more) and work through it over 2–3 weeks.

> **Caution**
>
> These are open-ended projects, not step-by-step tutorials. You are expected to consult documentation, experiment, debug, and make design decisions.

10.2 Project 1: Lotka–Volterra predator-prey simulation

The Lotka–Volterra equations: $\dot{x} = \alpha x - \beta xy$, $\dot{y} = \delta xy - \gamma y$.

1. Implement using DifferentialEquations.jl with parameters $\alpha = 1.1$, $\beta = 0.4$, $\delta = 0.1$, $\gamma = 0.4$.
2. Create three plots: population vs. time, phase portrait (x vs. y), direction field.
3. Vary α from 0.5 to 2.0 and overlay phase portraits.
4. Add stochastic noise via `SDEProblem`. Plot 20 trajectories alongside the deterministic solution.
5. Write a function that computes the oscillation period numerically.

```
using DifferentialEquations, Plots

function lotka_volterra!(du, u, p, t)
    x, y = u;  ,  ,  ,   = p
    du[1] =  *x -  *x*y
```

```
    du[2] = *x*y - *y
end

prob = ODEProblem(lotka_volterra!, [10.0,10.0], (0.0,100.0), (1.1,0.4,0.1,0.4))
sol  = solve(prob, Tsit5())
# Your work: create plots, add noise, analyze sensitivity
```

10.3 Project 2: Image classifier with Flux.jl

FashionMNIST: 70,000 grayscale 28×28 images in 10 clothing categories.

1. Load with MLDatasets.jl. Visualize 16 random samples with labels.
2. Build a dense network (2+ hidden layers, dropout). Train 15 epochs.
3. Build a CNN (`Conv`, `MaxPool`, `Dense`). Train 15 epochs.
4. Plot training loss and test accuracy vs. epoch for both architectures.
5. Compute and display the confusion matrix on the test set.
6. Identify the 10 most-misclassified images with predicted and true labels.

```
using Flux, MLDatasets
train_x, train_y = MLDatasets.FashionMNIST(split=:train)[:]

cnn = Chain(
    Conv((3,3), 1=>16, relu, pad=1), MaxPool((2,2)),
    Conv((3,3), 16=>32, relu, pad=1), MaxPool((2,2)),
    Flux.flatten, Dense(32*7*7=>128, relu), Dropout(0.3), Dense(128=>10))
# Your work: train both models, compare, build confusion matrix
```

10.4 Project 3: African health data pipeline

Build a complete data analysis pipeline from public health data sources.

1. Download real data from at least two sources: WHO (https://www.who.int/data/gho), World Bank (https://data.worldbank.org), Our World in Data (https://ourworldindata.org).
2. Clean: handle missing values, inconsistent country names, duplicates.
3. Join datasets on country and year. Filter for African Union member states.
4. Compute summary statistics by region (West, East, Central, Southern, North Africa).
5. Create four publication-quality CairoMakie figures: (a) life expectancy by country, (b) time series for 5 countries, (c) GDP vs. health indicator, (d) faceted regional comparison.

6. Save all figures as PDF.

```
using CSV, DataFrames, Downloads, CairoMakie, Statistics

url = "https://raw.githubusercontent.com/owid/owid-datasets/" *
      "master/datasets/Life%20expectancy%20-%20OWID/" *
      "Life%20expectancy%20-%20OWID.csv"
Downloads.download(url, "life_exp.csv")
df = CSV.read("life_exp.csv", DataFrame)

african = ["Nigeria","Ethiopia","Egypt","South Africa","Kenya",
    "Tanzania","Ghana","Senegal","Cameroon","Rwanda",
    "Morocco","Tunisia","Algeria","Benin","Togo"]
df_africa = filter(:Entity => in(african), df)
# Your work: clean, join additional sources, analyze, visualize
```

Julia tip

Always use real public data before resorting to synthetic data, especially for African countries. The World Bank, WHO, and Our World in Data provide excellent open datasets with good coverage.

10.5 Project 4: Stochastic SIR epidemic model

1. Implement the deterministic SIR as an `ODEProblem`.
2. Add multiplicative noise: $dS = -\beta SI\,dt - \sigma SI\,dW$.
3. Solve with `SDEProblem` and `EM()` (Euler–Maruyama).
4. Generate 100 trajectories using `EnsembleProblem`.
5. Plot the mean with a 95% confidence band (2.5th and 97.5th percentiles).
6. Compute the distribution of peak infection time and fraction.
7. Fit deterministic parameters to the ensemble mean using Optimization.jl.
8. Investigate the effect of noise intensity σ on epidemic outcome.

```
using DifferentialEquations, Plots, Statistics

function sir_drift!(du, u, p, t)
    S,I,R = u;  ,,  = p; N = S+I+R
    du[1] = - *S*I/N; du[2] =  *S*I/N -  *I; du[3] =  *I
end
function sir_noise!(du, u, p, t)
    S,I,R = u;  ,,  = p; N = S+I+R
    du[1] = - *S*I/N; du[2] =  *S*I/N; du[3] = 0.0
end
```

```
prob = SDEProblem(sir_drift!, sir_noise!, [0.99,0.01,0.0], (0.0,200.0),
↪  (0.3,0.1,0.05))
ensemble_sol = solve(EnsembleProblem(prob), EM(); dt=0.1, trajectories=100)
# Your work: plot confidence bands, analyze peak distribution
```

10.6 Project 5: Portfolio optimization

Markowitz optimization: $\min_{\mathbf{w}} \mathbf{w}^\top \Sigma \mathbf{w}$ subject to $\mathbf{w}^\top \boldsymbol{\mu} \geq r_{\min}$, $\mathbf{w}^\top \mathbf{1} = 1$, $w_i \geq 0$.

1. Generate (or download) daily returns for 10 assets over 5 years.
2. Compute expected return $\boldsymbol{\mu}$ and covariance Σ.
3. Solve the minimum-variance portfolio with Optimization.jl.
4. Trace the efficient frontier (50 target returns).
5. Plot the frontier, individual assets, and minimum-variance portfolio.
6. Add a risk-free asset and compute the tangency portfolio (max Sharpe ratio).
7. Backtest: compare optimized, equal-weight, and market-cap-weighted portfolios.

```
using Optimization, OptimizationOptimJL, LinearAlgebra, Plots
using Random; Random.seed!(42)

n_assets = 10
  = 0.05 .+ 0.15 .* rand(n_assets)
A = randn(n_assets, n_assets)
Σ = (A * A') / 1260 .+ 0.01 * I

portfolio_var(w, p) = w' * p[1] * w   # objective
# Your work: add constraints, trace frontier, backtest
```

10.7 Report template

Your project report should follow this structure:

1. **Title and abstract** (200 words). State the problem, approach, and main result.
2. **Introduction** (1 page). Context, motivation, and objectives.
3. **Methods** (2–3 pages). Mathematical formulation, algorithms, Julia packages, implementation details.
4. **Results** (2–3 pages). Figures, tables, quantitative analysis. Every figure must have a caption.
5. **Discussion** (1 page). Interpretation, limitations, possible extensions.
6. **Code appendix**. Complete runnable code or a link to a Git repository.

10.8 Grading rubric

Criterion	Points	Description
Correctness	30	Code runs without errors, results are correct, edge cases handled.
Code quality	20	Clean Julia code. Type-stable functions, no globals in hot paths, proper dispatch.
Visualization	20	Publication-quality figures with labels, legends, captions. At least 3 plot types.
Report	15	Clear writing, logical structure, proper math notation, honest limitations.
Creativity	15	Going beyond the minimum: extra analyses, real data, interactive elements.

Performance tip

Use a Git repository from day one. Commit early and often. Include a `Project.toml` so anyone can reproduce your environment with `Pkg.activate(".")` and `Pkg.instantiate()`.

10.9 Presentation guidelines

Each project will be presented in a 15-minute talk (12 minutes presentation + 3 minutes questions):

1. **Slide 1:** Title, your name, project number.
2. **Slides 2–3:** Problem statement and motivation. Why does this problem matter?
3. **Slides 4–5:** Methods. What packages did you use? What algorithms? Show key equations.
4. **Slides 6–8:** Results. Show your best figures. Explain what each figure reveals.
5. **Slide 9:** Discussion. What worked well? What was difficult? What would you do differently?
6. **Slide 10:** Conclusion and key takeaways.

Caution

Do not put code on slides unless it illustrates a specific design decision or a clever use of Julia. Slides should show results, figures, and high-level ideas — not line-by-line code. Keep the code in your report appendix.

Exercise

1. Choose a project. Create a Git repository with `Project.toml` and an initial script.
2. Write a one-page project plan: data sources, methods, planned figures, weekly timeline.
3. Implement a minimal working version that produces at least one correct plot. Commit as "v0.1."
4. Iterate: improve the model, add analyses, refine figures. Commit each improvement.
5. Write your report and prepare your presentation. Submit the repository, PDF report, and slides.

10.10 Chapter summary

- Five capstone projects integrate all course topics: data wrangling, visualization, performance, scientific computing, and machine learning.
- Project 1 (Lotka–Volterra) focuses on ODEs, phase portraits, and stochastic simulation.
- Project 2 (FashionMNIST) compares dense and convolutional neural networks.
- Project 3 (African health data) builds a pipeline from public sources to publication figures.
- Project 4 (stochastic SIR) explores ensemble simulation and uncertainty quantification.
- Project 5 (portfolio optimization) applies constrained optimization to finance.
- Good scientific computing requires clean code, reproducible environments, honest reporting, and clear visualization.

Appendix A: Julia installation guide

Option 1: juliaup (recommended)

Install the Julia version manager:

```
# Windows (PowerShell)
winget install julia -s msstore

# macOS / Linux
curl -fsSL https://install.julialang.org | sh
```

Then install Julia 1.10:

```
juliaup add 1.10
juliaup default 1.10
```

Option 2: Direct download

Download from https://julialang.org/downloads/. Install with default settings.

VS Code setup

Install VS Code from https://code.visualstudio.com. In the Extensions panel, search for "Julia" and install the official Julia extension by julialang.

Required packages

Open the Julia REPL and run:

```
using Pkg
Pkg.add([
    "Pluto", "DataFrames", "CSV", "Plots", "CairoMakie",
    "AlgebraOfGraphics", "BenchmarkTools", "DifferentialEquations",
    "Optimization", "OptimizationOptimJL", "ModelingToolkit",
    "MLJ", "Flux", "MLDatasets", "RDatasets", "IJulia"
])
```

Appendix B: Key Julia resources

Resource	Description	URL
Official docs	Julia language documentation	docs.julialang.org
Julia Discourse	Community forum for questions	discourse.julialang.org
JuliaHub	Package registry and search	juliahub.com
Pluto.jl	Reactive notebooks for Julia	github.com/fonsp/Pluto.jl
SciML	Scientific ML ecosystem	sciml.ai
MLJ.jl	Machine learning framework	alan-turing-institute.github.io/MLJ.jl
Flux.jl	Deep learning library	fluxml.ai
Julia Academy	Free online courses	juliaacademy.com