CLAUDE Code Style Guide for CEVE 543 Labs

This document outlines the coding and documentation standards for creating high-quality computational labs in CEVE 543.

Instructions for AI Assistants

Your Role

As an AI assistant working on CEVE 543 labs, limit your assistance to:

- **Code implementation**: Write, debug, and optimize Julia code
- **Formatting assistance**: Ensure proper Markdown, LaTeX, and document structure
- **Technical feedback**: Review code quality, performance, and best practices
- **Style compliance**: Check adherence to this style guide
- **Error diagnosis**: Help identify and fix technical issues

Stay Within Your Bounds

You should **NOT**:

- **Design educational content**: Do not create learning objectives, assessment criteria, or educational sequences
- **Make authoritative claims**: Avoid definitive statements about statistical methods or domain knowledge
- **Plan curriculum**: Do not design lab progressions or course structure
- **Evaluate pedagogy**: Do not assess the educational value of exercises or responses
- **Make disciplinary judgments**: Do not determine what concepts are important to teach

Remember Your Limitations

- **Instructors are the experts**: They understand pedagogy, students, and course context
- **You lack domain expertise**: You may not grasp nuances of hydrology, statistics, or engineering
- **Context matters**: Only instructors know their students' needs and backgrounds
- **Defer to instructor judgment**: When in doubt, ask rather than assume

Best Practices

- Act as a **technical assistant**, not an educational consultant
- Suggest improvements to code and formatting, not content or pedagogy
- Ask clarifying questions when instructions are unclear
- Acknowledge when requests exceed your appropriate scope

This style guide ensures consistent, professional, and educational computational labs that effectively teach statistical concepts while maintaining code quality and readability.

Text and Markdown Style

Sentence Structure

- **One sentence per line** in Markdown text.
- This improves version control diffs and readability.
- Keep sentences concise and clear.

Markdown Formatting Rules

- **Always include blank lines before and after headers**
- **Always include blank lines before and after lists** (numbered or unnumbered)
- This ensures proper Markdown parsing and consistent rendering
- Example:

```
This is a paragraph.

## Header Example

This follows the header.

1. First item
2. Second item

This follows the list.
```

Code References in Text

- **Always use backticks** when referring to code elements in prose:
  - Packages: `Extremes.jl`, `Turing.jl`, `Makie.jl`
  - Functions: `gevfit()`, `quantile()`, `@chain`
  - Macros: `@model`, `@filter`, `@arrange`
  - Variables: `μ_extremes`, `station_data`
  - File names: `index.qmd`, `util.jl`

Mathematical Notation

- **Use LaTeX math in Markdown**: `$\mu$`, `$\sigma$`, `$\xi$`
- **Never use Unicode math in Markdown text**
- **Unicode is acceptable in Julia code**: `μ = 4.0`, `σ = 1.2`
- Use `LaTeXStrings.jl` for plot labels: `L"\mu"`, `L"\sigma^2"`

Writing Style

- Write in **plain, simple language**
- Avoid jargon without explanation
- Use active voice when possible
- Be direct and concise
- Explain the "why" behind each step, not just the "what"

Julia Code Style

Package Management

- **Never modify `Project.toml` directly**
- Add packages using Julia package manager: `] add PackageName`
- Document required packages in setup sections
- Use `using PackageName` not `import PackageName` unless specific functions needed

Plotting with Makie

- **Always use Makie.jl** for all plotting

- Prefer `CairoMakie` backend: `CairoMakie.activate!(type="svg")`
- Use descriptive variable names: `fig`, `ax`, `ga` (for GeoAxis)
- Set figure sizes explicitly: `Figure(size=(800, 600))`
- Use LaTeX strings for mathematical labels: `ylabel=L"Return Level [\mathrm{inches}]"`

Code Annotations

- **Use code annotations sparingly** - only when truly helpful
- Format: `# <1>` in code, then numbered explanations after code block
- Always include blank line between code block and explanations
- Keep explanations concise and specific
- Example:

```
extremes_fit = gevfit(y)
μ = location(extremes_fit)[1]
```

Line 1

Fit GEV distribution using maximum likelihood estimation

Line 2

Extract location parameter from fitted model

Variable Naming

- Use descriptive names: `station_data` not `data`
- Mathematical parameters can use Unicode: μ, σ, ξ
- Functions use snake_case: `plot_time_series`, `calc_distance`
- Constants use descriptive names: `return_periods`, `sample_sizes`

Data Processing

- Use `@chain` macro for data pipelines when appropriate
- Prefer explicit operations over implicit ones
- Handle missing data explicitly: `skipmissing()`, `dropmissing()`
- Convert units clearly: `ustrip.(u"inch", rainfall)`

Document Structure

Quarto YAML Headers

- Include comprehensive metadata:
  - `title`, `subtitle`, `author`, `date`
  - `topics`, `objectives`, `ps_connection`
  - Both HTML and PDF output formats
  - Code annotation settings: `code-annotations: hover` for HTML

Section Organization

- Use clear hierarchical headings
- Include learning objectives upfront
- Provide setup instructions before code
- Structure analysis with numbered responses
- End with synthesis and reflection

Student Instructions

- Use callout boxes for critical instructions
- Make requirements **bold** and explicit
- Provide specific success criteria
- Include progress checkpoints throughout
- Use `{.callout-important}` for essential instructions

Response Structure

- Create numbered response placeholders at document start
- Use `{.callout-note}` for response prompts throughout
- Connect code outputs to specific responses
- Provide word count or bullet point guidance
- Include evaluation criteria

Technical Best Practices

Error Handling

- Include `try-catch` blocks for potentially failing operations
- Provide informative error messages
- Skip failed iterations with `continue` when appropriate
- Test edge cases and document limitations

Reproducibility

- Set random seeds: `Random.seed!(543)`
- Cache expensive computations when possible
- Document software versions in setup
- Use relative paths and check file existence

Performance

- Avoid global variables in loops
- Pre-allocate arrays when size is known
- Use vectorized operations: `y .~ Distribution(...)`
- Profile expensive code sections

Data Visualization

- Always include axis labels with units
- Use consistent color schemes across plots
- Add legends and titles
- Set appropriate axis scales (log, linear)
- Include error bars or uncertainty when relevant

Code Comments and Documentation

When to Comment

- Explain complex algorithms or mathematical concepts
- Document assumptions and limitations
- Clarify non-obvious variable transformations
- Provide context for magic numbers

Comment Style

- Use clear, complete sentences
- Explain the purpose, not just the action
- Update comments when code changes
- Remove outdated or obvious comments

File Organization

Project Structure

```
Lab-X/
├── index.qmd          # Main lab document
├── util.jl            # Utility functions
├── Project.toml       # Julia environment (don't edit manually)
├── Manifest.toml      # Package versions (auto-generated)
└── CLAUDE.md         # Style guide (this file)
```

Utility Functions

- Place reusable functions in `util.jl`
- Document function parameters and return values
- Include type hints when helpful
- Test functions with simple examples

Quality Assurance

Before Submitting

- Render document to both HTML and PDF
- Check all code blocks execute without errors
- Verify all code annotations have corresponding explanations
- Test with fresh Julia environment
- Proofread all text for clarity and correctness

Common Issues to Avoid

- Missing backticks around code references
- Multiple sentences per line in Markdown
- Undefined variables in code blocks
- Inconsistent mathematical notation
- Overly complex code annotations
- Missing or incorrect file paths

Example Implementation

Good Text Style

```
We'll use the `gevfit()` function from `Extremes.jl` to estimate parameters.
The location parameter $\mu$ represents the mode of the distribution.
This approach uses maximum likelihood estimation (MLE) for parameter inference.
```

Good Code Style

```
# Fit GEV distribution using MLE
extremes_fit = gevfit(y)
μ_extremes = location(extremes_fit)[1]
σ_extremes = scale(extremes_fit)[1]

# Create return level plot
fig = Figure(size=(800, 600))
ax = Axis(fig[1, 1],
    xlabel="Return Period [years]",
    ylabel=L"Return Level [\mathrm{inches}]",
    xscale=log10)
```

Line 2

Fit GEV using maximum likelihood estimation

Line 3

Extract location parameter (mode of distribution)

Line 4

Extract scale parameter (measure of variability)

Bibliography