UNIVERSITY OF
COPENHAGEN

Introduction

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

August 26, 2025

What is Computational Statistics?

It is a broad field, where meaning depends on context.

One definition is that it is **the use of computational methods to solve statistical problems**, for instance

- simulation,
- optimization,
- numerical integration,
- data analysis, and
- visualization.

A Running Example

Let's try to get a bit of flavor of what we will be doing in the course.

Throughout the course we will use a data set of amino acid angles, Φ and Ψ, from protein structures.

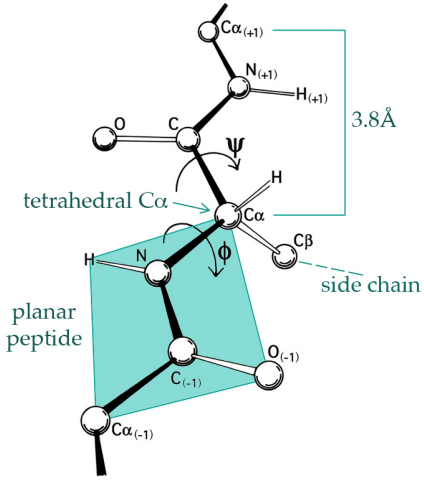
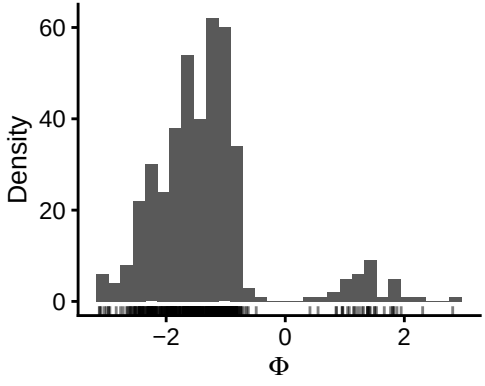


Figure 1: Amino Acid Angles

Histograms

A simple way to analyze the distributions of the angles Φ and Ψ is the **histogram**.

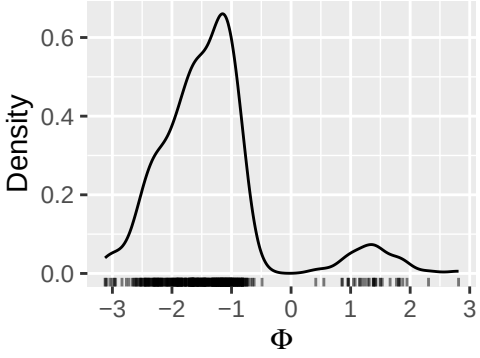
```
ggplot(phipsi, aes(x = phi)) +
  geom_histogram() +
  geom_rug(alpha = 0.5) +
  labs(
    x = expression(Phi),
    y = "Density"
  )
```



Density Estimation

Histograms are not very smooth. If we allow ourselves to make stronger assumptions, we can get a smoother estimate of the distribution, using **kernel density estimation** (KDE).

```
ggplot(phipsi, aes(x = phi)) +
  geom_density() +
  geom_rug(alpha = 0.5) +
  labs(
    x = expression(Phi),
    y = "Density"
  )
```



But how is this KDE actually computed? Doing this efficiently is a **computational statistics** problem.

Statistical Topics of the Course

The course can be broken down into a number of **statistical** and **computational** topics.

There are three statistical topics in the course:

Statistical Topics of the Course

The course can be broken down into a number of **statistical** and **computational** topics.

There are three statistical topics in the course:

Smoothing
We will learn how to compute efficient kernel density estimates and scatterplot smoothers.

Simulation
We will learn to efficiently simulate from probability distributions using inversion, rejection, and importance sampling.

Optimization
We will learn to solve optimization problems that arise in statistics, for instance in maximum likelihood estimation (MLE), using the EM algorithm and gradient-based optimization.

Computational Topics of the Course

Implementation

We will learn how to implement statistical methods in R, using object-oriented programming and functional programming.

Correctness

We will learn how to ensure that our code is correct, using testing and debugging.

Efficiency

We will learn how measure performance and find bottlenecks in our code using profiling and benchmarking, and how to optimize it.

Teaching Staff

Instructor
Johan Larsson, postdoctoral researcher

Figure 2: Johan

Teaching Assistant
Jinyang Liu, PhD student in machine learning

Figure 3: Jin

Contact
Use Absalon for course-related questions and email (see Absalon) for personal matters.

Assignments

Four assignments make up the bulk of the course work.

For each assignment, there are two alternatives (A and B). You will pick one.

Each assignment is tied to a particular **topic**:

1. Smoothing
2. Univariate simulation
3. The EM algorithm
4. Stochastic optimization

Presentations

There will be four presentation sessions (week 3, 4, 6, 7)[1]

You will divide into groups of 2-3 students and present your solution to one of the assignments during one of the sessions.

You will register for groups and assignments in Absalon.

Presentation is compulsory but not graded. We expect solutions to be work in progress.

[1] Not counting the potato harvesting week when you are off.

Oral Examination

The main examination is an oral exam based on your assignments.

You will prepare four presentations, one for each assignment you picked.

At the exam, you will present one of these at random.

Schedule

Lectures

- Tuesdays and Thursdays, 10:15–12:00 (Johan)

Exercise Sessions

- Thursdays, 08:15–10:00 (Jinyang)

Presentations

- Thursdays, 13:15–15:00 (Johan)
- Only weeks 3, 4, 6, and 7

Examination

- November 6-8 (8.15-17.30, tentative)
- Rooms to be announced

Course Literature

Computational Statistics with R

Main textbook for the course, written by Niels Richard Hansen.

- Available online at https://cswr.nrhstat.org/
- Not yet complete, but we only use parts that are.
- Companion package: install with `pak::pak("github::nielsrhansen/CSwR/CSwR_package")`.

Advanced R

Auxiliary textbook, written by Hadley Wickham.

- Available online at https://adv-r.hadley.nz/
- Covers more advanced R programming topics.
- We will use selected chapters.

Online Resources

Absalon
Main source for information and communication about the course. Accessed at absalon.ku.dk.

CompStat Web Page
Course content will be uploaded to github.io/math-ku/compstat.

This is where you will a detailed schedule of the course, slides, and the assignments.

Figure 4: Absalon

Generative AI

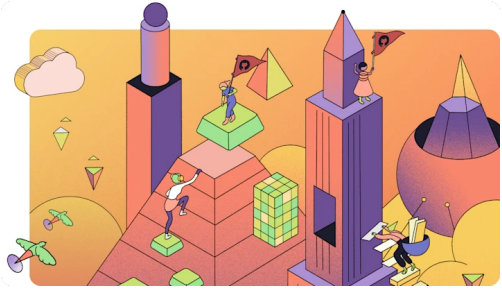
Generative AI (e.g. ChatGPT, Copilot, Bard, etc.) are powerful tools.

You are allowed to use them in this course, but with some caveats:

- You must understand the results.
- You must acknowledge their use in your assignments and how you used them.

Copilot

Access to GitHub Copilot is available for free to students, via GitHub Education.



Programming in R

What is R?

R is a programming language and application (command-line interface) for statistical computing and graphics.

It is widely used among statisticians and data miners for developing statistical software and data analysis.

Why R?
It is free, open source, and cross-platform.

Why not Python/Julia?
R has a large number of packages for statistical computing and graphics.

My personal opinion is that:

- R is better suited for visualization and exploratory data analysis, while
- Python is better suited for general-purpose programming and machine learning, and
- Julia is a best for numerical computing.

Prerequisite R Knowledge

We expect knowledge of

- data structures (vectors, lists, data frames),
- control structures (loops, if-then-else),
- function calling,
- interactive and script usage (`source`) of R.

All of this is covered in chapters 1-5 of Advanced R.

We do not expect that you are an expert in R.

Getting Help with R

Google It

Especially good for error messages.

Generative AI

- Also great for error messages and debugging
- *Caution*: You need to understand the results, especially when you ask it to create something for you.

Absalon Discussion Forum

Use the fact that there are twenty other people in the course with exactly the same problem.

Functions

Functions in R

Everything that happens in R is the result of a function call. Even +, [and <- are functions.

An R function takes a number of *arguments*, and when a function call is evaluated it computes a *return value*.

Functions can return any R object, including functions!

Implementations of R functions are collected into source files, which can be organized into packages.

Why Functions?

Technically, you could write all your code in a single script. So why use functions?

Functions help you structure your code, make it reusable, and make it easier to test and debug.

A well-designed function has a single purpose, which makes it easier to understand and reason about.

Rule of Thumb
If you find yourself writing the same piece of code more than, say, twice, then it is probably a good idea to turn it into a function.

Function Syntax

Here is a simple function that takes two arguments and returns their sum.

```
f <- function(x, y) {
  x + y
}
```

A function has three components: **arguments**, **body**, and **environment**.

Arguments

In this case, `x` and `y` are the arguments.

Body

The body of the function is everything inside the curly braces `{}`, that is, `x + y`.

Environment

The environment is where the function was created. It is used to look up variables that are not defined inside the function.

```
environment(f)
```

```
<environment: R_GlobalEnv>
```

Arguments

Arguments are specified in the parentheses after the function name.

Arguments can have default values

```
g <- function(x, y = 2) {
  x + y
}

g(3) # y takes the default value 2
```

```
[1] 5
```

Named arguments can be passed in any order.

```
g(y = 3, x = 2)
```

```
[1] 5
```

Copy-on-Write

R uses **copy-on-write** semantics for function arguments, which means that arguments are only copied if they are modified inside the function.

This makes function calls efficient, and also means that you can modify arguments without shooting yourself in the foot.

```
h <- function(x) {
  x[1] <- 100
  x
}

a <- 1:5

h(a) # a is not modified
```

```
[1] 100   2   3   4   5
```

```
a # a is still 1:5
```

```
[1] 1 2 3 4 5
```

<<- Operator

You can use the <<- operator to modify variables in the parent environment, but please don't.

Environment and Scoping

When a function is called, a new environment is created for the function.

This environment is used to look up variables that are not defined inside the function.

The new environment has as its parent the environment where the function was created.

This is called **lexical scoping**.

```
x <- 10
f <- function(y) {
  x + y
}
f(5)
```

```
[1] 15
```

Assuming that `x` is not defined inside `f()`, R looks for `x` in the environment where `f()` was created, which is the global environment in this case.
As a rule of thumb, avoid using variables from the parent environment inside functions.

Return Values

The return value of a function is the value of the last expression in the body.

Unless you explicitly use the `return()` function, which immediately exits the function and returns the specified value.

```r
f <- function(x, y) {
  return(x + y)
  x * y # This line is never reached
}
```

Whether or not to use `return()` is a matter of style. Personally, I prefer to not use it unless returning early.

Functional Programming

Functions are **first-class citizens**: they can be passed as arguments to other functions, returned as values from functions, and assigned to variables. R is a **functional programming** language.

This allows for a high degree of abstraction and code reuse, for instance through the use of the `apply` family of functions.
Let's write our own apply function.

```
our_apply <- function(x, fun) {
  val <- numeric(length(x))
  for (i in seq_along(x)) {
    val[i] <- fun(x[[i]])
  }
  val
}
```

Testing Our Apply Function

```
sapply(1:10, exp)
```

```
 [1]     2.718282     7.389056    20.085537    54.598150   148.413159
 [6]   403.428793  1096.633158  2980.957987  8103.083928 22026.465795
```

```
our_apply(1:10, exp)
```

```
 [1]     2.718282     7.389056    20.085537    54.598150   148.413159
 [6]   403.428793  1096.633158  2980.957987  8103.083928 22026.465795
```

Assumptions of `our_apply()`

`x` is a "list-like" structure, `fun` takes a single argument, and `fun` returns a numeric.

What if `fun` Needs Additional Arguments?

Then we get an error:

```
our_apply(1:10, rpois)
```

```
Error in fun(x[[i]]): argument "lambda" is missing, with no default
```

Anonymous Functions

We can use an anonymous function to pass additional arguments to `fun()`.

```
our_apply(
  1:10,
  function(lambda) rpois(1, lambda = 0.9)
)
```

```
 [1] 0 0 1 2 0 2 3 1 1 0
```

Ellipsis (...)

An arbitrary number of arguments can be forwarded via the ... (ellipsis) argument, which allows us to pass additional arguments to `fun()`.

```
our_apply <- function(x, fun, ...) {                    ①
  val <- numeric(length(x))
  for (i in seq_along(x)) {
    val[i] <- fun(x[[i]], ...)                          ②
  }
  val
}
```

① ... in the argument list of `our_apply()` collects additional arguments.
② ... in the call to `fun()` passes these additional arguments to `fun()`.

```
our_apply(1:10, rpois, n = 1)
```

```
 [1]  0  1  4  3  7  6  8 16  8 12
```

Assertions

It is a good idea to assert that the arguments to a function are valid, and stop early if they are not.

The simplest way to do this is to use the `stopifnot()` function.

```
my_mean <- function(x) {
  stopifnot(is.numeric(x))
  sum(x) / length(x)
}

my_mean(c("asdf", "qwer"))
```

```
Error in my_mean(c("asdf", "qwer")): is.numeric(x) is not TRUE
```

If `x` is not numeric, we would otherwise get an error deep inside `sum()`.

Custom Error Messages

For more control over the error message, use `if` and `stop()`.

```
my_mean <- function(x) {
  if (!is.numeric(x)) {
    stop("x must be numeric")
  }
  sum(x) / length(x)
}
```

Document Your Functions

Document your functions, including their arguments, return values, and any side effects.

Documentation can be either ad-hoc comments or using a documentation system like roxygen2.

Aim to document **why** something is done, not just **what** is done. The latter is often obvious from the code itself.

Naming Functions

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> *–Phil Karlton*

Favor Descriptive Names

Begin to see if you can use a *verb*. Better long and descriptive than short and cryptic.

Honor Common Conventions

Avoid `.` in names; it is used for **methods** (upcoming).

Use a Consistent Style

- `lowercase`
- `snake_case` (tidyverse)
- `camelCase`
- `UpperCamelCase`

Namespace Clashes

Avoid names of existing functions.

Keep Functions Short and Focused

Functions should do one thing and do it well. If a function is too long or complex, consider breaking it up into smaller functions.

Testing

Write tests for your functions to ensure that they work as expected. (More on this later in the course.)

Debugging

Use `browser()`, `traceback()`, and `debug()` to debug your functions. (More on this later in the course.)

Summary

- Functions are the building blocks of R code.
- Functions help you structure your code, make it reusable, and make it easier to test and debug.
- Functions have arguments, a body, and an environment.
- Functions can be passed as arguments to other functions, returned as values from functions, and assigned to variables.
- Use `...` to pass additional arguments to functions.
- Document your functions and use descriptive names.
- Write tests for your functions and use debugging tools.

Exercises

Exercises

Exercise 1: Write a Function with a Default Argument

Write a function `greet` that takes a name as an argument and prints "Hello, !". If no name is given, it should print "Hello, world!".

Exercise 2: Count Missing Values

Write a function `count_na` that takes a vector and returns the number of missing (`NA`) values in it.

Exercise 3: Check if a Number is Even

Write a function `is_even()` that returns `TRUE` if its argument is even, `FALSE` otherwise.