UNIVERSITY OF
COPENHAGEN

Testing and Debugging

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

March 31, 2026

Last Time: Optimization

Gradient Descent
Few assumptions on f, cheap iterations, but slow convergence.

Algorithm 1: Gradient descent

Data: Step size $t > 0$
repeat
  $x \leftarrow x - t\nabla f(x)$;
until *convergence*;
return x

Newton's Method
Requires second derivatives, expensive iterations, but fast convergence.

Algorithm 2: Newton method

Data: Step size $t > 0$
repeat
  $x \leftarrow x - t(\nabla^2 f(x))^{-1}\nabla f(x)$;
until *convergence*;
return x

Today

Testing

How do we know our code is correct?

Debugging

How do we find and fix bugs in our code?

Testing

Testing

Testing is the process of executing a program with the intent of finding bugs.

We have already done testing using plots and `all.equal()`.

But this is slightly ad-hoc. We can do better!

We want automatic and reproducible testing that we can run as part of our workflow.

What is a Test?

A test is a function that checks that some property holds.

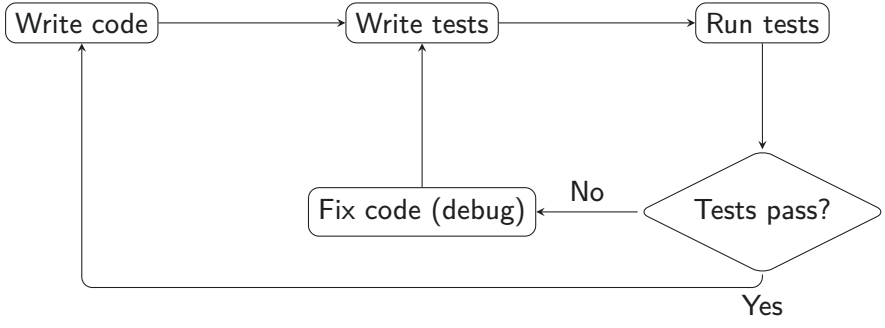
Typically, we have output from some function and want to check that it meets our expectations.

Automated Testing

So far, we have been testing manually.

But what we want is automated testing, which we can make part of our workflow.

Suggested Workflow



The testthat Package

The `testthat` package makes it easy to write and run tests in R.

It provides a simple syntax for writing tests and integrates well with RStudio.

It is also the most common testing framework for R packages.

Writing Tests with testthat

Tests in **testthat** are written using the `test_that()` function.

Inside the `test_that()` function, we use `expect_*()` functions to check that certain conditions hold.

```
library(testthat)

test_that("sum works", {
  expect_identical(sum(1, 2), 3)
  expect_identical(sum(-1, 1), 0)
})
```

```
Test passed
```

Test Failures

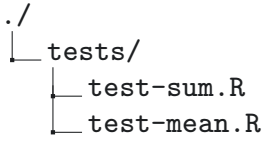
If a test fails, testthat will throw an error and provide information about the failure.

```
test_that("sum works", {
  expect_identical(sum(1, 2), 4) # This will fail
})
```

```
-- Failure: sum works ----------------------------------
sum(1, 2) not identical to 4.
1/1 mismatches
[1] 3 - 4 == -1

Error:
! Test failed
```

Organizing Tests

In your test directory, you can organize tests into multiple files.



Convention
Test files should be named `test-*.R` (or `test_*.R`), where `*` is a descriptive name for the tests in that file.

test_dir()

testthat::test_dir() runs all the tests in a specified directory.

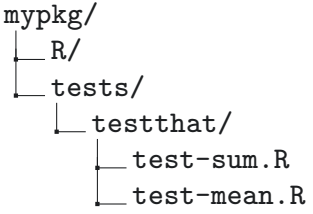
```
testthat::test_dir(here::here("tests"))
```

```
v | F W  S  OK | Context

/ |          0 | mean
v |          2 | mean

/ |          0 | sum
v |          2 | sum

== Results =====================================================
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 4 ]
```

Inside Packages

testthat was designed to be used inside R packages.

Tests are organized in files inside the `tests/testthat/` directory of an R package.



We will see more about package development later in the course.

Test Coverage

We want to test as many parts of our code as possible.

This is called test coverage.

covr
Tracks which lines of code are executed when we run our tests.

Out of scope for this course, but you can read more about it in the covr documentation.

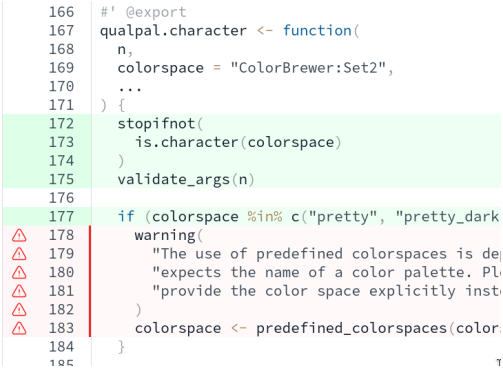


Figure 1: Test coverage report.

Reproducibility

We want our tests to be reproducible.

This means that they should give the same result every time we run them.

This is especially important for tests that involve randomness.

We can achieve this by setting a random seed using `set.seed()`.

Numerical Tolerance

When comparing numerical values, we need to account for floating-point precision.

```
test_that("numerical equality", {
  expect_identical(sqrt(2)^2, 2)
})
```

```
-- Failure: numerical equality -----------------------------
sqrt(2)^2 not identical to 2.
Objects equal but not identical

Error:
! Test failed
```

Use `expect_equal()` with a specified tolerance to handle this.

```
test_that("numerical equality with tolerance", {
  expect_equal(sqrt(2)^2, 2, tolerance = 1e-8)
})
```

```
Test passed
```

Debugging

Bugs are common, maybe inevitable.

Debugging is the process of finding and fixing bugs.

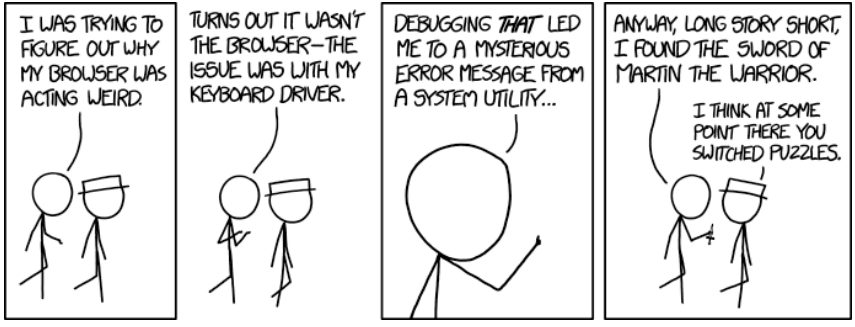


Figure 2: Down the rabbit hole.

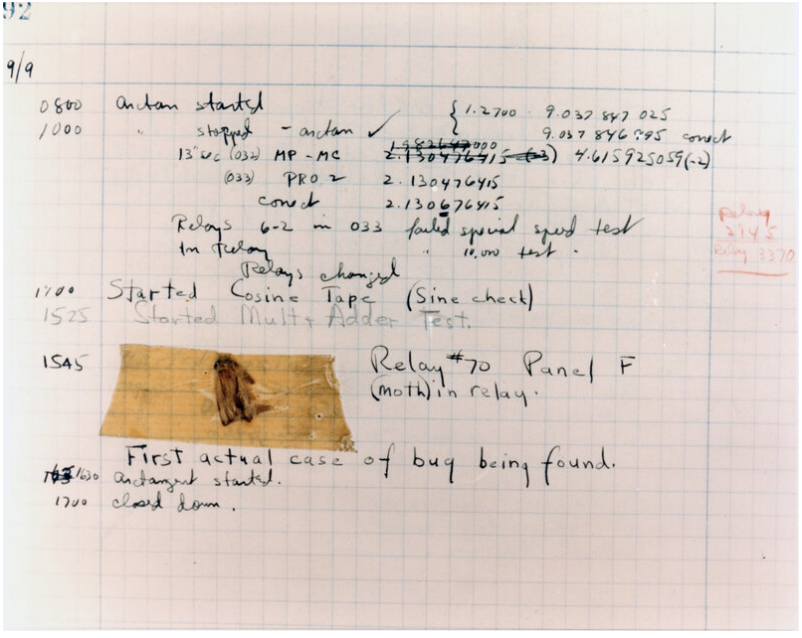


Figure 3: The "first" bug report.

Traceback

Sometimes it is enough to simply look at the call stack.

```
# debugging.R
f <- function() g()
g <- function() h()
h <- function() stop("We have a problem!")
```

```
f()
```

```
Error in h(): We have a problem!
```

```
traceback()
```

```
4: stop("We have a problem!") at debugging.R#3
3: h() at debugging.R#2
2: g() at debugging.R#1
1: f()
```

Poor Man's Debugging

Sprinkle print statements throughout your code:

```
f <- function(x, y) {
  cat("x:\n")
  print(x)

  z <- g(x, y)

  cat("z:\n")
  print(z)

  list(sum_z = sum(z))
}
```

Okay in the simplest of cases and easy to get going with, but will cost you more time in the long run.

Returning Diagnostics

For more complex functions, adding a `debug`/`diagnostics` flag, which if `TRUE` returns some extra output.

```
f <- function(x, y, debug = FALSE) {
  z <- g(x, y)

  out <- list(sum_z = sum(z), diagnostics = NULL)

  if (debug) {
    out$diagnostics <- list(z = z)
  }

  out
}
```

Interactive Debugging

For more complex problems, you typically need to debug interactively.

There are two common ways to debug code interactively in R.

Browser

`browser()` can be inserted anywhere in the code.

When execution reaches `browser()`, it stops and enters debug mode.

RStudio Breakpoints

Visual breakpoints can be set in RStudio by clicking in the margin next to a line number.

When execution reaches a breakpoint, it stops and enters debug mode.

No need to modify the code, but cannot be conditionally set.

AI Debugging

Debugging with AI (Copilot, Claude, Gemini etc.) is a quick and easy way to debug.

Make sure to provide context, especially when working with statistical topics.

AIs are typically good at finding syntax errors, but not so good at finding algorithmic errors.

Will almost always try to be helpful, so may provide incorrect answers and find bugs that are not there.

Pair Debugging

Pair debugging is a collaborative debugging technique where you work in a pair to find and fix bugs.

One person walks through the code, explaining it line-by-line, while the other listens and asks questions.

This can help you see the code from a different perspective and find bugs that you might have missed.

Rubber Duck Debugging

If you don't have anyone around to pair up with, you can always team up with a rubber duck!

The idea is that explaining the code out loud helps you see it from a different perspective.

Worked Example

We are going to fit a Poisson regression model using two different methods: gradient descent and Newton's method.

The Poisson regression model is given by

$$\mathrm{E}(\boldsymbol{y} \mid \boldsymbol{X}) = \exp(\boldsymbol{X}\boldsymbol{\beta}),$$

where $\boldsymbol{X}$ is a $n \times p$ matrix of predictors and $\boldsymbol{y}$ is our response.

The problem is to find

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left(\frac{1}{n} \sum_{i=1}^{n} \exp(\boldsymbol{x}_i^T \boldsymbol{\beta}) - y_i \boldsymbol{x}_i^T \boldsymbol{\beta} \right).$$

This is a convex optimization problem.

Data

Here, we will generate some synthetic data, for a simple Poisson regression model with one predictor.

```
n <- 1000
x <- rnorm(n)
y <- rpois(n, exp(x))
X <- model.matrix(y ~ x)

head(X, 5)
```

```
  (Intercept)          x
1           1 -0.8060639
2           1  0.2447500
3           1  1.4836670
4           1 -1.3406945
5           1  2.2039393
```

Implementation of Objective and Gradient

```
objective <- function(beta, X, y) {
  Xty <- drop(crossprod(X, y))
  (sum(exp(X %*% beta)) - beta %*% Xty) / nrow(X)
}

gradient <- function(beta, X, y) {
  Xty <- drop(crossprod(X, y))
  (colSums(drop(exp(X %*% beta)) * X) - Xty) / nrow(X)
}
```

Implementations of Optimization Algorithms

Gradient descent and Newton method implementations can be found in `debugging.R`.

```
source(here::here("R", "debugging.R"))
```

Testing

Let's run the gradient descent implementation to see if it works.

```
res_gd <- gradient_descent(
  c(0, 0),
  objective,
  gradient,
  t0 = 1,
  epsilon = 1e-8,
  X = X,
  y = y
)

res_glm <- glm(
  y ~ x,
  family = "poisson"
)
```

```
res_gd$par
```

```
(Intercept)           x
-0.05620544  1.03907663
```

```
coef(res_glm)
```

```
(Intercept)           x
-0.05620545  1.03907664
```

Buggy Newton Method

First, we define the Hessian function.

```
hessian <- function(beta, X, y) {
  (crossprod(X, drop(exp(X %*% beta)) * X)) / nrow(X)
}
```

Then, we can run the Newton method implementation: `newton_method()`.

Exercise: Debugging `newton_method()`

1. Download and open `debugging.R` in RStudio.
2. Create another script, and place some code for generating data there.
3. Source `debugging.R` and test that `gradient_descent()` works. Now try `newton_method()`. What happens?
4. Create a test for the `newton_method()` function using the `testthat` package that fails.
5. Start debugging by setting some breakpoints in `newton_method()`.
6. Run `newton_method()` again. Execution should stop at the first breakpoint. Use the RStudio interface to step through the code and inspect variables. What do you see?
7. There are two bugs in `newton_method()`. Can you find and fix them?

Next Time

Likelihood Optimization

We study a practical example of optimizing a multinomial likelihood.

We will introduce algorithms for solving constrained optimization problems.