UNIVERSITY OF
COPENHAGEN

R Packages and Wrap-Up

Computational Statistics

Johan Larsson

Department of Mathematical Sciences, University of Copenhagen

March 31, 2026

Today

Distributing and Organizing Code
Workshop in creating an R package

Course Summary
What did we actually do?

Oral Examination
What to think of during examination

Why Write a Package?

Organizing Code

Components

In any given project, you may have:

- Code for experiments
- Source code (which we may want to reuse)
- Tests
- Rcpp code
- Data

How do we organize this?

Sourcing

One way is to just keep R scripts and source them as needed.

But this puts the burden on you to keep track of where things are and in what order to source them.

Reusability is limited.

R Package

Another way is to make an R package!

This forces you to organize your code, and gives you tools to manage dependencies, testing, data, documentation, and C++ integration.

Including Functions

Without a Package

Need to keep track of where your functions are defined or source manually.

```
source("path/to/some/file1.R")
source("path/to/some/file2.R")

my_first_function()
my_second_function()
```

With a Package

You can just do:

```
library(mypkg)

my_first_function()
my_second_function()
```

or `mypkg::my_first_function()` if you don't want to load the whole package.

Testing

Without a Package

Previously, we needed to manually run scripts to test our code: store them in some folder and call `testthat::test_dir()`

With a Package

With a package, we can use `devtools::test()` to run all tests in the package.

And tests are run automatically when checking the package.

Dependencies

Without a Package

If your code depends on other packages, you need to:

1. Make sure they are installed using `install.packages()`.
2. Load them using `library()` at the right places (or call `pkg::fun()` directly).

With a Package

Dependencies are declared in the `DESCRIPTION` file of your package.

When installing your package, dependencies are installed automatically.

C++ Integration (Rcpp)

Without a Package

You need to manually source C++ files using `Rcpp::sourceCpp()`.

Need to specify dependencies manually using special comments in the C++ code:

```
// [[Rcpp::depends(RcppEigen)]]
```

With a Package

Rcpp code lives in `src/` and is compiled automatically when installing the package.

Dependencies on other Rcpp packages (RcppArmadillo, RcppEigen) are handled automatically through `DESCRIPTION`.

Data

Without a Package

You need to keep track of where data files are located and load them manually.

```
mydata <- read.csv(
  "path/to/mydata.csv"
)
```

With a Package

Data can be included in the package in `data/` and directly available (if exported) as `mypkg::mydata`.

Publishing

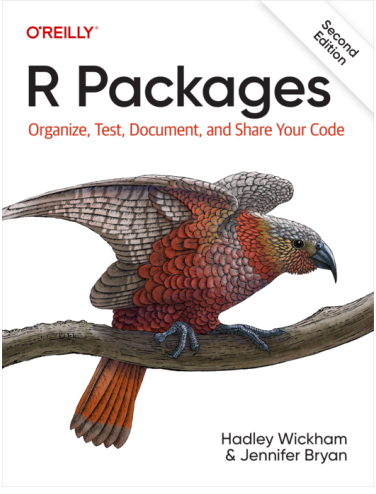
If you find out that your code is useful to others, you can publish it on CRAN or GitHub.

And if you've already made a package, this is straightforward!

Writing an R Package

R Packages

Different approaches, but we will follow **R Packages** (Wickham and Bryan 2023), which is based around the **devtools** package.



Meta-package for various helpers that aid in developing R packages (and projects).

First off, install and load **devtools**:

```
install.packages("devtools")
library(devtools)
```

usethis
This loads other packages that will be useful for setting up your package, such as the **usethis** package.

A Toy Example

Rosenbrock Package

Let's build a simple package that solves the Rosenbrock optimization problem, i.e. find

$$x^* = \arg\min_x \Big((a - x_1)^2 + b(x_2 - x_1^2)^2 \Big).$$

What We Will Learn

- Adding R functions to our package
- Testing our code
- Interfacing with Rcpp
- Adding dependencies to other packages
- Licensing our package
- Documenting the code

Create It
Use RStudio[1] or call

```
usethis::create_package("rosenbrock")
```

Gives you a **minimal** package:

```
rosenbrock/
├── R/
├── DESCRIPTION
└── NAMESPACE
```

You may also have `.Rbuildignore` and `.rosenbrock.Rproj` depending on how you created the package.

[1] `File -> New Project -> New Directory -> R Package Using Devtools`

Install It

Open up the package in your editor (R Studio[2]).

```
devtools::install()
```

Voila, you have made an R package!

If you're in RStudio, you can also click the "Build" tab and then "Install and Restart"[3].

[2] In which case it should already be opened.

[3] `Ctrl/Cmd+Shift+B`

R Code

`R/`

- All R code should live in `.R`-files in `R/`.
- These files should usually only contain **functions**.
- Many ways to organize your files: one function per file, all functions of a certain S3 class in one file etc.

Let's create a first file: `R/objective.R`.

Call `usethis::use_r("objective")` and insert

```
objective <- function(x, a = 1, b = 100) {
  (a - x[1])^2 + b * (x[2] - x[1]^2)^2
}
```

We have created a first R file, now what? Two major options:

`devtools::install()`
Installs the package, like calling `install.packages()`.

Robust but slow. Need to call `library(rosenbrock)` to load package[4].

`devtools::load_all()`[5]
Sources all of your code.

Quick but not as robust.

Good for regular development.

Try It

Try both options and see if you can call your newly defined function, `objective()`.

[4] Done automatically in R Studio
[5] `Ctrl/Cmd+Shift+L`

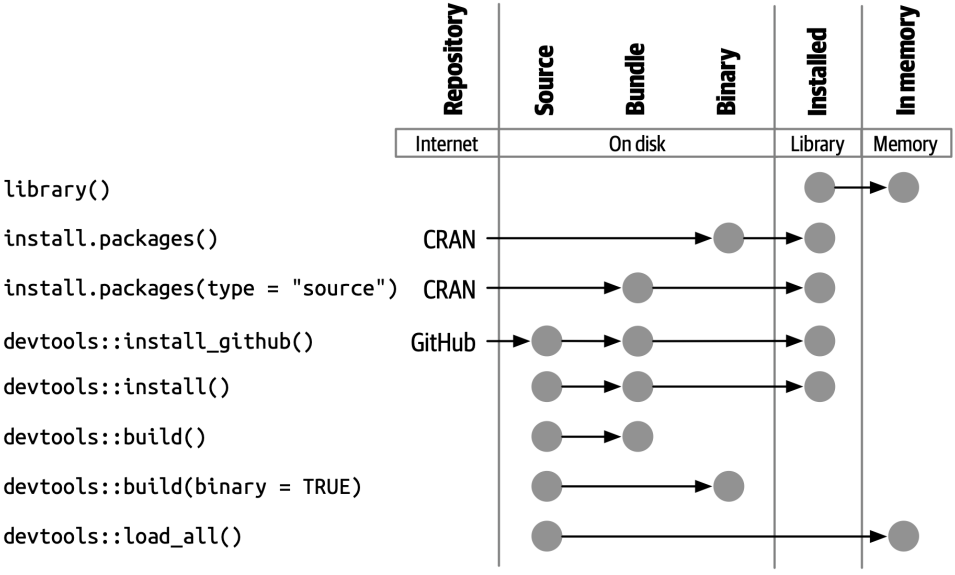


Figure 1: The various states of a package and how to move between them.

Exporting Functions

If you called `devtools::load_all()` then everything is sourced and you can just call `objective()` directly.

But if you use `devtools::install()` and `library(rosenbrock)`, then you would need to use `rosenbrock:::objective()`. The reason is that the function is not yet exported.

NAMESPACE
Decides what functions you want exported. But right now it just contains a comment:

```
# Generated by roxygen2: do not edit by hand
```

roxygen2 is a package that helps with package documentation, but it can also be used for handling the namespace.

To export a function, you need a special roxygen2 comment before the function:

```
#' @export
```

Go ahead and place this before your `objective()` definition. Then run `devtools::document()`[6] to roxygenize your package.

Now `NAMESPACE` will (should) contain this:

```
export(objective)
```

Reinstall the package and see if you can call `objective()` after loading it.

[6] Shortcut: `Ctrl/Cmd+Shift+D`

testthat

Let's start using **testthat** with our package:

```
usethis::use_testthat()
```

This creates some new files and directories:

```
rosenbrock/
└── tests/
    ├── testthat/
    │   └── <currently-empty>
    └── testthat.R
```

A First Simple Test

For the Rosenbrock function, $f^* = f(a, a^2) = f(1, 1) = 0$. Let's make sure this is the case for us too!

To create a test, we can use `usethis::use_test()`.

Call `use_test("objective")` and insert this:

```
test_that("objective at optimum", {
  # add a test using expect_equal()
})
```

Check That Everything Works

Run `devtools::test()`[7], and (hopefully) get

```
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
```

[7] Shortcut: `Ctrl/Cmd+Shift+T`

R CMD check

R contains functionality for checking that your package is built correctly and you can access this functionality through `devtools::check()`[8].

No requirement that your package needs to pass these checks (if you're using it as a project), but it's good practice to make sure it does.

ERROR	Major problem with your package
WARNING	Something that is most likely not great but not critical
NOTE	Typically small issues with your package

Now run `devtools::check()`. Is there a problem?

Yes, we're missing a license!

[8] Shortcut: `Ctrl/Cmd+Shift+E`

Metadata

The metadata for your package lives in `DESCRIPTION`. Right now it looks like this:

```
Package: rosenbrock
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R:
    person("First", "Last", , "first.last@example.com", <...>)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends
    to pick a license
Encoding: UTF-8
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.3.2
```

For now we'll leave most of these files alone, but let's fix one thing: the license

Licensing

Why Do You Need a License?

Licensing software tells other people about how they are allowed to reuse your code.

If you do not provide a license, this generally means that **nobody is allowed to copy, distribute, or modify your code.**

If you have other contributors, then "nobody" includes **you too!**

Personal Projects

For a small personal project, licensing is overkill, at least if you are not sharing it with others.

Choosing a License

So we need to pick a license: for now we'll pick the MIT license.
(Read more about picking a license at https://choosealicense.com.)

```
usethis::use_mit_license()
```

This will add new files to your package: `LICENSE`, `LICENSE.md`, and modify `DESCRIPTION`, in which you should see:

```
License: MIT + file LICENSE
```

Dependencies

In R packages, you make dependencies explicit, defined in `DESCRIPTION`.

Computing the Gradient

Let's say that we want to use **numDeriv** to numerically compute the gradient for the Rosenbrock function.

```
gradient <- function(x, a = 1, b = 100) {
  numDeriv::grad(objective, x, a = a, b = b)
}
```

Now our package depends on **numDeriv**, so we need to add it to `DESCRIPTION`:

```
usethis::use_package("numDeriv")
```

In `DESCRIPTION`, you should now see this:

```
Imports:
  numDeriv
```

Rcpp works best in a package:

- No more manual sourcing (no need to call `Rcpp::sourceCpp()`)
- You don't need to add directives for dependencies to **RcppArmadillo** and other packages.

We will rely on **roxygen2**. First, call

```
usethis::use_package_doc()
```

to set up a package doc file in `R/rosenbrock-package.R`.

Then use `usethis::use_rcpp()`, which sets up Rcpp by modifying `DESCRIPTION` and setting up the namespace via roxygen2.

It will also create a `src/` folder with a empty C++ file:

```
rosenbrock/
└── src/
    └── rosenbrock-
        package.cpp
```

Install or load your package and your code is available (but not exported).

Exporting Rcpp Code

Wrapping

Call your Rcpp function through an R wrapper:

```
my_fun <- function(x) {
  my_fun_cpp(x)
}
```

Typically easier because checking input and preparing output is easier on the R side.

Direct Export

You can add roxygen2 comments in Rcpp code too:

```
//' @export
double
my_fun_cpp()
{
  ...
}
```

Saves you having to write and maintain an R function.

Documentation

Why?

Because

- you make your code accessible to others,
- it makes you think an extra time about what your function is doing, and
- your future self will thank you.

Types

- Comments in code
- Manual (help files)
- Long-form articles (vignettes)

roxygen2

Primary purpose of the roxygen2 package. You write code in a special syntax and it converts it into manual files that R understands.

```
#' Function Title
#'
#' Here you describe what the function does, possibly
#' using several lines.
#'
#' @param x Explanation of argument x
#'
#' @return Explanation of what the function returns
#'
#' @export
my_fun <- function(x) {
  ...
}
```

Your Turn

Document `objective()` with roxygen2 syntax. No need for sensible documentation. Just make sure you have the bare minimum.

Documentation in This Course

- Not making a package for CRAN, so lower standards.
- You don't need to document to benefit from building a package.
- But it's not a bad idea to do so anyway!

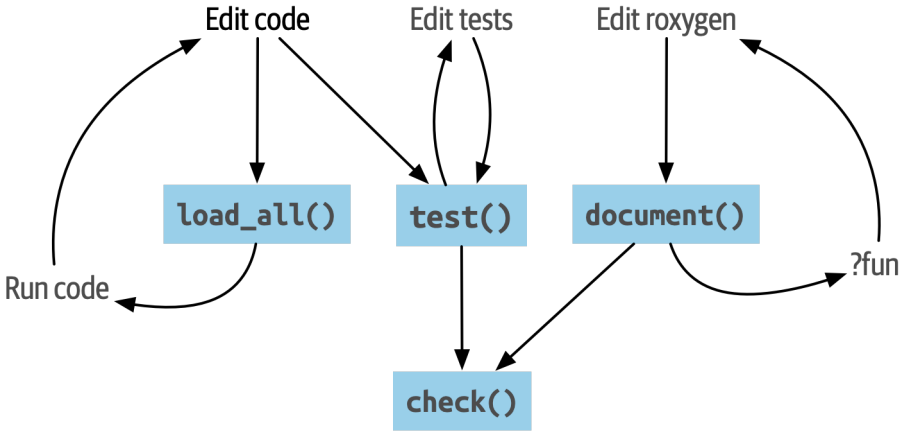


Figure 2: The whole game

Shortcuts

Use the shortcuts (and save yourself some typing/clicking):[9]

Action	Shortcut
Install package	`Ctrl+Shift+B`
Load all code	`Ctrl+Shift+L`
Document package	`Ctrl+Shift+D`
Test package	`Ctrl+Shift+T`
Check package	`Ctrl+Shift+E`

[9] Replace `Ctrl` with `Cmd` on Mac.

Projects

When you have a project, you typically need more things:

- Scripts that produce output
- Datasets stored in different formats
- Notebooks or reports

These things do not naturally fit into a package framework.

Solutions

1. Just store these things directly in the package folder.
2. Put your **package** into a *subdirectory* of your project. This cleanly separates the part of your project that contains reusable code (the package) and the part that is experiments and reports. But a little trickier to setup.

Exercise

Continue building the **rosenbrock** package!

Step 1

Write a function `optimize_banana()` that uses `optim()` to solve the Rosenbrock problem. It should take as input `a`, `b`, and a starting point `x0`, and return the optimal point and function value.

Step 2

Write an implementation of the gradient using Rcpp. Let the user choose whether to use the gradient or not in `optimize_banana()`.

Step 3

Return an S3 object of class `banana` from `optimize_banana()`, and then write an S3 plot method `plot.banana()` that plots the contours of the Rosenbrock function and the optimum found in `optimize_banana()`.

Summary

We have learned how to create an R package, add R and Rcpp code, test it, document it, and manage dependencies and licensing.

What We Didn't Cover

- Version control (git)
- How to properly format metadata (`DESCRIPTION`)
- Integrating data into our package
- Publishing to CRAN
- Enhanced reproducibility (renv)

Oral Examination

Procedure

1. After entering the room you will connect the computer and check that it works with the projector.
2. When all technical issues are settled, you will draw the assignment and find the presentation on the computer.
3. Time starts and you have 15 min for the presentation. The examiners may ask questions if something needs to be clarified.
4. After 15 min your presentation will be stopped, and the examiners will ask questions related to the assignment as well as to the general content of the course.
5. After at most 25 min the exam ends, and after assessment you will be given a grade.

Recommendations

- You should prepare a presentation that covers the assignments.
- You don't need to have covered *everything* in detail, but you should show that you understand the main concepts.
- For a good grade, you need to have studied at least part of the assignment in detail.
- We expect you to use slides. Aim for around 15-20 slides.
- Use plenty of plots. Pick out important code snippets.
- Practice your presentation.
- Keep your code available in case we want to see something.

Summary of the Course

Statistical Topics

Smoothing

- Kernel density estimation
- Smoothing splines

Monte Carlo Methods

- Inverse transform sampling
- Rejection sampling
- Importance sampling

Optimization

- Unconstrained optimization (gradient descent, Newton's method)
- Constrained optimization (interior point method)
- The EM algorithm
- Stochastic gradient descent (mini-batching, acceleration)

Computational Topics

Accuracy

- Testing your code (ad-hoc, testthat)
- Debugging techniques (interactive debugging)

Measuring Performance

- Profiling (profvis)
- Benchmarking (bench)

Improving Performance

- Using R (vectorization, avoiding copies)
- Using Rcpp (C++)
- Parallelization (foreach)

Organizing Code

- Object-oriented programming (S3)
- R Packages

Thank you, and good luck!

References

Wickham, Hadley, and Jennifer Bryan. 2023. *R Packages: Organize, Test, Document, and Share Your Code*. 2nd ed. Beijing: O'Reilly Media. https://r-pkgs.org/.