732A94 Advanced R Programming
Computer lab 3

Krzysztof Bartoszek, Bayu Brahmantio.
(designed by Leif Jonsson and Måns Magnusson)
(converted to LaTeX by Arash Haratian)

8 September 2025 (U2)

Lab session: **12 September, 08:15** (SU10, SU11, SU12, SU13, SU14)
Seminar date: **19 September, 10:15** (U2)
Lab deadline: **22 September, 23:59**

Instructions

- Ideally this lab should be conducted by students **two by two**.
- The lab consists of writing a package that is version controlled on `github.com` or `gitlab.liu.se`.
- All students in the group should **contribute equally much** to the package. All group members have to contribute to, understand and be able to explain all aspects of the work. In case some member(s) of a group do not contribute equally this has to be reported and in this situation a formal group work contract will be signed, stipulating the consequences for further unequal contributions.
- Other significant collaborations/discussions should be acknowledged in the solution.
- To copy other's code is **NOT** allowed. Your solutions will be checked through URKUND.
- Copying solutions of others and from any online or offline resources is **NOT** allowed.
- Commit continously your addition and changes.
- Collaborations should be done using GitHub (ie you should commit using your own github account) or using GitLab.
- In the lab some functions can be marked with an *. Students **MUST do AT LEAST ONE** exercise marked with an * for each of the Labs 3 - 6 and Bonus. If only one exercise is marked with an *, then it **MUST** be done.
- The deadline for the lab is on the lab's title page.
- The lab should be turned in using an url to the repository containing the package on github/gitlab.liu.se using **LISAM**. This should also include name, liu-id and, if applicable, github user names of the students behind the project. In case of problems or if you do not have access to LISAM the url may be emailed to `baybr79@liu.se` or `krzysztof.bartoszek@liu.se`.
- **NO resubmissions will be allowed for the Bonus lab.**
  NO late submissions will be allowed for the Bonus lab.
- Inside your package you may not depend on any global variable (unless it is a standardR one, like pi). Using them will result in an immediate failure of your code.If at any stage your code changes any options, these changes have to be reverted before your code finishes.
- All notes raised by Travis/GitHub Actions/GitLab CI have to be taken care of or explicitly defended in your submission.
- The seminars are there to discuss your solutions and obtain support with problems. Every group has to present at least once during the seminars in order to pass the lab part.

Contents

Chapter 1

To create a package in R

In this lab we will create our first R package in R. To be able to get everything to work you need to have the following software installed:

- R, with packages:
  - `devtools`
  - `usethis`
  - `testthat`
- R-Studio
- Git

This lab will be a walkthrough on how to create a package. **NOTE: This is not the only way to do this but one way that works for most.**

1.1 Write the R code

In this first R package we will implement two famous algorithms, the Euclidian algorithm to find the greates common divisor of two integers and Dijkstra's shortest path algorithm in a graph. For both these algorithms you will have pseudocode for the algoritm, so the job is to implement these algorithms in R. Store each function in their own R file with the name of the function.

1.1.1 `euclidean()`

The first algorithm to implement is the Euclidean algorithm to find the greatest common divisor of two numbers. The description of the algorithm with pseudocode can be found **here** `https://en.wikipedia.org/wiki/Euclidean_algorithm`. Assert that the arguments are numeric scalars or integers.

Below is an example of the `euclidean()` function.

```
euclidean(123612, 13892347912)

[1] 4

euclidean(100, 1000)

[1] 100
```

1.1.2 * `dijkstra()`

The next algorithm to implement is one of the most famous algorithms in computer science, Dijkstra's algorithm. The algorithm takes a graph and an initial node and calculates the shortest path from the initial node to every other node in the graph. A description with pseudocode can be found at the

Wikipedia page **here** (https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm). If you are not very familiar with graphs, vertices and edges, see **this** (https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) Wikipedia page for a fast introduction.

The function should be named `dijkstra()` and have the argument `graph` and `init_node`. The graph should be a `data.frame` with three variables (`v1`, `v2` and `w`) that contains the edges of the graph (from `v1` to `v2`) with the weight of the edge (`w`). The dijkstra function should return the shortest path to every other node from the starting node as a vector.

Assert that the graph argument have the above structure and that `init_node` is a numeric scalar that exist in the graph.

Below is code to create the first graph at the "ikipedia page (this is not the most memory efficient way to express the edges but it makes it easier to implement the function) and the results of the function `dijkstra()`.

```
wiki_graph <-
data.frame(v1=c(1,1,1,2,2,2,3,3,3,3,4,4,4,5,5,6,6,6),
v2=c(2,3,6,1,3,4,1,2,4,6,2,3,5,4,6,1,3,5),
w=c(7,9,14,7,10,15,9,10,11,2,15,11,6,6,9,14,2,9))

dijkstra(wiki_graph, 1)

[1]  0  7  9 20 20 11

dijkstra(wiki_graph, 3)

[1]  9 10  0 11 11  2
```

1.2 Create the R package

To create a package can be done in many different ways. This is one suggestion on how to do it.

1.2.1 Create the package

1. Create a new R project

From R-Studio, click on the **Project** button (top-right) and select **New project**. Select the first option (**New Directory**), and then the **R package** option. Fill in the package name and choose a directory for the package. Check the **Create a git repository** option, as this will be handy in the next step. This will create a backbone of your R package containing the necessary files. However, by default it creates files that we do not need (`hello.R` in the `./R` directory and `hello.Rd` in `./man`). Feel free to remove those files. The `NAMESPACE` should contain the functions of the package that we want to export. For now, we delete this file since it will be generated again later.

2. Fill out the `DESCRIPTION` file

The `DESCRIPTION` file contains basic information of your package. Edit the Title, Version, Authors, and Description fields. It is advisable to use an open-source license for your package. There are many open-source licenses to choose from, and each has its own terms. The `usethis` package provides an easy way to write a license for your package. For example, one can run the following line to apply the MIT license to the package:

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

3. Move your R files

Move the R file(s) that contains the functions `euclidean()` (and `dijkstra()` $*$) into the `./R` directory.

4. Document the package using `roxygen2`

The next step is to document the functions and the data using `roxygen2`. `roxygen2` makes it easy to include documentation in direct connection to the functions, making it much easier to both document and read the documentation when you inspect the code. See the. Using some functions from the `devtools` package, the function documentation workflow can be summarized by the following steps:

1. Add roxygen comments on top of your functions, that is, write useful information about your functions using roxygen's format. The documentation should include:
   (a) Arguments,
   (b) Description of the algorithm,
   (c) What the function returns,
   (d) A reference to the Wikipedia page of each algorithm.

   Don't forget to use the `@export` tag so the functions are visible to the user of your package.
2. Document the functions by running `devtools::document()`. This will write documentations in the `./man` directory, provided that you have written roxygen documentations on the corresponding functions. At the same time, it will rewrite the `NAMESPACE` file so that it contains exported functions. In general, it is not recommended to edit the `.Rd` and `NAMESPACE` files by hand.
3. Check if the documentations are properly written by running the command `?function_name`, where `function_name` refers to the name of the documented function (`euclidean` or `dijkstra` for this lab).

See chapter 16, "Function documentation", in [3] for more details.

5. Include the dataset `wiki_graph` in the package

The next step is to include the dataset `wiki_graph` created above as dataset in the package. See chapter 7, "Data" in [3] for more details on how to do this. Document the dataset using the following information using `roxygen2`:

1. The variables in the `data.frame`
2. A reference to the Wikipedia page

6. Check the package

Now that you have written and documented your functions and dataset, it is time to check if your package is working properly and have no errors or missing requirements. There are several ways to do this and they lead to the same functionality:

- Run the command `devtools::check()`
- Under the "Build" tab in R-studio, click "Check".
- From the "Build" drop-down menu in R-studio, click "Check Package".

You want to ensure that your package receive 0 errors and 0 warnings, and ideally 0 notes. See Appendix A - R CMD Check in [3] for more details on how checks are performed on your R package. Additionally, you can locally install the package by running the `devtools::install()` command, or by clicking the "Clean and install" option under the Build tab in R-studio.

1.2.2 Make a repository for your package

The next step is to publish your package to a repository. Here, GitHub is used as an option for publishing your package. However, it is possible to use other alternatives, such as GitLab, and the process would be similar.

To begin, we need to make sure that Git is available (if not, see [1], ch. 6), and create a GitHub account. Next, make sure that your computer is connected to your GitHub account (Connect Git, GitHub, RStudio, [1]).

Assuming that the package is already made by following the previous section, you need to connect your existing local directory to GitHub ([1], ch. 17). It is also possible to create a repository first and create your package later, by choosing the option "Version Control", and then "Git", and providing the link to the GitHub repository when creating a new project in R-Studio ([1], ch. 12).

If you prefer to use GitLab, you can go to gitlab.liu.se and use your LiU-id as your account.

1.2.3 Enable continuous integration

We want to check the package continuously to ensure that it passes the check with the latest changes made. Other alternatives are Travis CI and GitLab CI, which also will be discussed. In deciding which of the three CI tools to use, you may want to note the following:

- We recommend using GitHub Actions or Travis when hosting your repository on GitHub, and GitLab CI when hosting your repository on gitlab.liu.se.
- Travis CI provides a limited number of free credits, irrespective of whether your GitHub repository is Public or not. Every time you use Travis CI service, you will use up some Travis credits, and if and once you have no credits left, you will not be able to continue using their service, unless you pay for more credits (or you have a (legal) alternative account with free Travis credits). Furthermore, there are student reports that the free credits provided by Travis were insufficient for completing all the labs in the course and also that Credit Card information was sought by Travis even to use a free plan
- GitHub Actions provides unlimited free credits for Public repositories but limits free credit for non-Public repositories. Credit Card information is not sought by GitHub for free services

1.2.3.2. Working with GitHub Actions

GitHub Actions will check the R package automatically every time you push new code to the GitHub repository and inform you if the package is working (Green) or Failing (Red).

1. Make sure that there is a `README.md` file for your repository. Otherwise, run `usethis::use_readme_md()`. This will create a README file that can act as a quick information page for your package on GitHub, and also something to put the GitHub Actions badge on.
2. Execute the following R command `usethis::use_github_action("check-standard")`.
3. Commit the files created/modified and push changes to GitHub. Now GitHub Actions should try to build your packages. If your package fails, correct the bugs until the package passes. Pass/Fail status can be seen in the GitHub Actions build badge on repository's root folder in the README section.

1.2.3.1. Working with Travis CI

Travis will build the R package automatically every time you push new code to the GitHub repository and inform you if the package is working (Green) or Failing (Red).

1. Create an account on Travis CI and connect it to your GitHub account: `https://travis-ci.com/`
2. Mark your repository you want to be built on Travis (the package in this lab).
3. Add a `.travis.yml` file with the following content (this will build an/your R package and save installed R packages used):

   ```
   language: r
   cache: packages
   ```

   More information on how you can handle Travis builds can be found here:
   `https://docs.travis-ci.com/user/languages/r/`
4. Commit the `.travis.yml` file and push it to GitHub. Now Travis should try to build your packages. If your package fails, correct the bugs until the package passes.

5. Add a Travis build badge (markdown) to the repository README file (at the top) so it is easy to see if your package is passing or failing. More information can be found here:

   `https://docs.travis-ci.com/user/status-images/`

1.2.3.3. Working with GitLab CI

Like Travis CI and GitHub Actions, GitLab CI will automatically build the R package every time you push new code to the gitlab.liu.se repository and inform you if the package is working (Green) or Failing (Red). The CI tool in GitLab is called pipeline.

1. When viewing your repository on gitlab.liu.se go to "Build", and then "Pipeline Editor" on the sidebar on the left.
2. Click on "Configure Pipeline"
3. Copy the following code into the .gitlab-ci.yml file:

```
image: rocker/tidyverse

stages:
  - build
  - test
  - deploy

building:
  stage: build
  script:
    - R -e 'remotes::install_deps(dependencies = TRUE)'
    - R -e 'devtools::check()'

# To have the coverage percentage appear as a gitlab badge follow these
# instructions:
# https://docs.gitlab.com/ee/user/project/pipelines/settings.html#test-coverage-parsing
# The coverage parsing string is
# Coverage: \d+\.\d+

testing:
    stage: test
    allow_failure: true
    when: on_success
    only:
        - master
    coverage: '/coverage: \d+.\d+% of statements/'
    script:
        - Rscript -e 'install.packages("DT")'
        - Rscript -e 'install.packages("covr")'
        - Rscript -e 'covr::gitlab(quiet = FALSE)'
    artifacts:
        paths:
            - public

# To produce a code coverage report as a GitLab page see
# https://about.gitlab.com/2016/11/03/publish-code-coverage-report-with-gitlab-pages/

pages:
    stage: deploy
    dependencies:
        - testing
    script:
```

```
        - ls
    artifacts:
        paths:
            - public
        expire_in: 30 days
    only:
        - master
```

4. Commit .gitlab-ci.yml file and push to gitlab.liu.se. Now GitLab CI should try to build your packages. If your package fails, correct the bugs until the package passes.

5. Add a "Pipeline Status" badge to the README file. It is enough to copy the correct markdown command for the badge to the top of the file. More information on how to find the markdown command can be found here:
`https://docs.gitlab.com/ee/user/project/badges.html#view-the-url-of-pipeline-badges`

1.2.4 Include the test suite in your package

The last step is to include unit tests for your package (later you will write unit tests yourself). See chapter 13, "Testing basics", in [3], or [2]. Unit tests should be designed in a way that it is possible to introduce a bug in the code and you will find out that we have introduced that bug.

1. Set up the testthat framework for your package with `use_testthat()` in the `usethis` package.

2. Add the test suite for `euclidean()` (and `dijkstra()`), that can be found at:
`https://github.com/STIMALiU/AdvRCourse/tree/master/Testsuites`

3. Run "Test" under the Build tab in R-Studio to check that your functions passes all tests, or run the command `devtools::test()`. Commit and push your tests to github.

4. If your functions do not pass the tests, find the bug and try again. You're not done until all tests are passed.

1.2.5 Finalize your package

Now everything should be working in your package. As a final step we should check that everything works with your package. Do the following steps:

1. Perform checks again (step 6 of section 1.2.1) on your package and make sure that there are no warnings and errors.

2. Push your final package to GitHub (or GitLab) and test that it is possible to install your package using the `devtools::install_github` command.

3. Create a release of your package (ex. v. 1.0) on GitHub (or GitLab).

You can consider (not obligatory) to use p4merge to resolve conflicts. See the file 732A94_AdvancedRHT2019_Seminar_Lab3_Mourao_Valencia.pdf, courtesy of Agustín Valencia and Marcos Mourão.

1.3 Seminar and examination

During the seminar you will bring your own computer and demonstrate your package and what you found difficult in the project.

We will present as many packages as possible during the seminar and you should

1. Show that the package can be built using R Studio and that all unit tests is passing.

2. Present the unit tests you've written.

3. We will try to introduce a bug in the code and check that this bug is found by the unit tests (and by git).

1.3.1 Examination

Turn in a the address to your github or gitlab repo with the package using LISAM. To pass the lab you need to:

1. Have the R package up on GitHub (or GitLab) with a GitHub Actions (or Travis CI, GitLab CI) pass/fail badge.
2. The test suites for the implemented function(s) should be included in the package.
3. The package should build without warnings (pass) on GitHub Actions (or Travis CI / GitLab CI).
4. All issues raised by GitHub Actions (or Travis CI / GitLab CI) should be taken care of or justified why they are not a problem or cannot be corrected. Be careful with namespace issues, these you HAVE to take care of.

Bibliography

[1] Jennifer Bryan. Happy git and github for the user. URL: `https://happygitwithr.com/`.

[2] Hadley Wickham. testthat: Get started with testing. *The R Journal*, 3(1):5–10, 2011.

[3] Hadley Wickham and Jennifer Bryan. *R packages*. " O'Reilly Media, Inc.", 2023. URL: `https://r-pkgs.org/`.