A Reproducible Research Compendium

cf. list of contributors at
https://github.com/rr-mrc-bsu/reproducible-research/graphs/contributors

2019-03-04

Contents

Chapter 1

A 'Living Book' - aims and scope

This book is a little different from your ususal statistics foliant - it is written entirely using **Markdown** and rendered to html, pdf, and epub publishing formats using the R package **bookdown**. Its entire Markdown source code is publicly available on GitHub.com at https://github.com/rr-mrc-bsu/reproducible-research. A pre-build version is hosted as static html website using **GitHub pages** at https://rr-mrc-bsu.github.io/reproducible-research/. This structure allows to easily discuss changes using GitHub issues https://github.com/rr-mrc-bsu/reproducible-research/issues, organize further development using milestones and projects, contribute corrections or even entire chapters by creating pull requests, and to manage editions by GitHub releases. It also means that everybody - and yes, that does include you - can become a contributor by creating pull requests in the GitHub repository. Since the contents are thus evolving over time as long as there are active contributors to the project, the book is 'living'.

The overall purpose of the *Reproducible Research Compendium* is threefold:

1. Provide a platform for discussing aims and objectives as well as best practices for reproducible research with a clear focus on applications in biostatistics.
2. Build-up a lasting compendium for knowledge sharing around various issues and methods for coping with them that may broadly be subsumed under the term 'reproducible research'.
3. The book project itself acts as a learning-by-doing example for its contributors with the goal of anybody participating becoming knowlegable about organizing collaborative open-source [mostly coding] projects.

The complete documentation for **bookdown** can be found at https://bookdown.org/yihui/bookdown/. Note that R is a prerequisite but only for building the book - the contents itself are completely language-agnostic.

Chapter 2

How to contribute

Since this book is a collaborative effort, the most important thing is enabling people to contribute! This chapter is a hands-on tutorial and does not go into the details of the required steps. Each of the techniques will be explained in more detail in the subsequent chapters.

2.1 Before you start

In the following we will assume that you are working on a Linux or MacOS machine. The instructions are similar for Windows users, but we do not discuss the details here. Using an open source operating system such as Linux is preferable since all collaborators are able to use the exact same operating system that you are running.

A great way to get started with Linux is Ubuntu and the easiest way to install it on an existing Windows or MacOS machine might be via a virtual machine. Detailed instructions on how to do so can be found here.

2.2 Cloning: Getting the book's source code

The book is compiled from a collection of R markdown files which is a special text file format that allows to combine code and text within the same file as well as basic markup[1]. The motivation behind markdown is to separate the content entirely from the layout and stick to an absolute minimum number of markups (e.g. headings, enumeration, hyperlinks) to be able to compile the document to as many different output formats as possible. The actual compilation will be done in 'pandoc' and will be explained later [ref].

A very important pillar of reproducibility is version control, i.e., some mechanism to keep track of changing files over time and enabling roll-backs to previous versions. For more details on why version control is so important in reproducible research and how to implement it, cf. [ref: version control chapter]. For now, we will just focus on how to install and use a specific program, **git**, to obtain the source code for the book and make changes to it. Assuming that you have a linux (we will assume an Ubuntu installation) or MacOS system running, you will need to install the version control system git [link to chapter]. The easiest way to do this is via the command line package manager 'apt' on linux ('homebrew' is the equivalent for MacOS).

On **linux**, open a terminal window and execute the command

[1] Markups are simple meta information on text such as 'this is a headline' etc. If you are familiar with LaTeX you are already a markup professional since LaTeX supports a huge number of different markup expressions. HTML is also a markup language.

```
sudo apt -y install git
```

On **MacOS**, the following command will prompt git installation if it is not already installed

```
git --version
```

This will download and install all required packages from the official repository.

You may now 'clone' the online repository of the book from its GitHub.com website: https://github.com/rr-mrc-bsu/reproducible-research. Here 'cloning' does exactly what it says: it downloads an exact copy of the entire source code including its complete history of previous changes to your local computer to work on.

Assuming that you have a terminal window opened and the working directory is your home directory '~' you clone the repository by invoking

```
git clone https://github.com/rr-mrc-bsu/reproducible-research.git
```

This will create a new folder 'reproducible-research' in your current working directory and download all necessary files. You should then change the working directory to the new folder via

```
cd reproducible-research
```

Note that the source code only needs to be cloned **once**. After you have a local copy of the source code, you should ensure that this is kept up to date by 'pulling' from the remote master branch. This ensures that you are editing the most up-to-date version of the project.

The following command copies the remote master branch to your local device (this need not be done if you have just cloned the source code)

```
git pull origin master
```

2.3 Creating a new 'branch'

Branches are different variants of the source code that may exist in parallel and one major job of git is making it possible to bring these branches together.

By default your git repository will now be on its 'master' branch.

You may verify that via

```
git status
```

This is a useful command to check that you are never working on the master branch. For this project, we have envoked a branch protection rule meaning that you are not able to work on the local master and then push this directly to the remote master. Instead, you must first create a branch that you edit and then push back to the remote, before opening a pull request to merge with the remote master.

The master branch is special in that it is usually considered the current 'best' variant of a project. For most smaller projects, a single master branch might be sufficient but things do get a bit hairy when many people could potentially change this common master branch at the same time. Also, for this book project, each time the master branch in the online repository changes, the entire book is recompiled and published at https://rr-mrc-bsu.github.io/reproducible-research/. Therefore the books contributing guidelines require that no changes are made directly to the master branch. Instead, all work is done on separate feature branches, e.g., on 'my-cool-new-chapter' if you want to add a new chapter.

To create this branch you run the following git commands

```
git branch my-cool-new-chapter
git checkout my-cool-new-chapter
```

This creates the new branch and checks it out (activates it). All changes that you now make to files in the directory only affect the version of the book associated with your local branch 'my-cool-new-chapter' (after you commit them, that is).

Of course, you can always check that you are on your new branch using

```
git status
```

2.4 Creating a new chapter

You may now add a new chapter simply by placing a new numbered .Rmd file in the top level of the book projects directory, e.g. the new file could be called '99-my-cool-new-chapter.Rmd'. Do feel free to browse the other chapters of the book already present to learn more about the R markdown syntax used to write the book. Note that although R markdown relies on R to compile the files, the contents of the file may not contain any R code at all. For more details on R markdown see [link chapter].

Once the new chapter file is created and you added some content you should check whether the altered book still compiles without errors. This is critically important for any piece of source code since even small changes might break the entire thing. Since you will not be able to incorporate you changes in the online version of the book without passing some automated checks, you may just as well check that everything is working locally before attempting to add your changes online.

To compile the book you will need R and some packages. The easiest way to install everything is again via the icommand line.

Linux users should envoke

```
sudo apt-get install r-base r-base-dev
Rscript -e 'install.packages("bookdown")'
```

Whilst MacOS users should use the following, which installs XCode CLT and homebrew prior to the installation of R (if XCode and/or homebrew are already installed you will see a message warning and you can skip these steps)

```
xcode-select --install
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew install r
Rscript -e 'install.packages("bookdown")'
```

You may then attempt to build the new book with your new chapter by invoking the build script

```
bash _build.sh
```

which essentially runs the R command

```
bookdown::render_book("index.Rmd")
```

to build the book and cleans up afterward. You should now see a new folder '_book/'and be able to open'_book/index.html' in your browser. This just opens the newly build version of the book locally. Make sure that everything is as you want it to be.

2.5 Committing your changes

Next you will want to 'commit' your changes to your local 'my-cool-new-chapter' branch. Commiting means that you store your changes in the git repository thus creating a snapshot in time that you may always return to irrespective of any further changes to the repository.

The repository should be configured in such a way as to ignore the _book/ folder that you created since this is just output. You can therefore simply add all changes and commit them with a short description of what you did

```
git add -A
git commit -m "added cool new chapter"
```

2.6 Publishing your changes

Changes to the master branch in the online repository are organized as pull requests. This is a feature on GitHub.com that allows you to publicly propose merging a branch back to master. Usually this is straightforward to do since the master branch will change very slowly (cf. [ref merge conflicts]). The pull request will then have to be reviewed by at least one other collaborator to the repository before you are able to actually merge the changes into the master branch - only at that point are they actually integrated in the published book.

To do so, you first have to be a contributor to the project [TODO: explain how to do that without being a contributor / how to become one]. Next, you will need to push your local branch to the online repository via

```
git push -u origin my-cool-new-chapter
```

Switch to a browser and open https://github.com/rr-mrc-bsu/reproducible-research. In the top/middle you then need to switch from the '<> Code' tab to the pull requests tab. Create a new pull request by clicking on the button. This opens a panel where you can define your pull request. A pull request always proposes to merge one branch onto another. In our case we want to merge 'my-cool-new-chapter' onto 'master'. That meas that we leave the 'base:' branch master as it stands by default. However, in the rop-down menu for 'compare:' you can now select you new branch. Note that the arrow between the two already indicates that the 'compare:' branch is supposed to be merged onto the 'base:' branch. In the panel below you will then see a git diff, i.e., a listing of all the differences between the two branches (gree: additions, red: deletions). Since you only added new stuff in this example and the master (probably) did not change between you downloading the latest copy of the repository and creating your changes, there are no merge conflicts. Confirm by clicking on 'create pull request'.

This will first create the pull request and the immediately trigger a build script on the continuous integration system Travis [reference to continuous integration]. The continuous integration system will spin up a virtual machine in the cloud, install all required software, download the repository and check whether the build script will still run without errors after merging you pull request. This process will take a few minutes and once it is completed the status of the build will be shown in your pull request. Even if the build script ran perfetly fine on your local machine the Travis build might fail if you introduced new dependencies without altering the Travis build configuration. For normal changes (only editing .Rmd files), the build should work without any errors. The advantage of having a CI system is that anybody reviewing your changes will immediately know that your pull request did not introduce any breaking changes and that the book can still be compiled on the default minimal build system defined in the travis.yml file.

Once another contributor has reviwed your changes and approved them, your are then free to merge your pull request. Only this last action will actually change the master branch of the repository. This change will again trigger a build on Travis, this time for the newly merged master branch. Since the pull request was

already checked, there should not be any further problems. Additionally, any build of the master branch will also execute the _deploy.sh script which will take the compiled book and push the output to the gh-pages branch of the repository. This branch is special in that it only contains the generated output and not the corresponding source code. GitHub.com offers the possibility of hosting static html pages free of charge via GitHub pages and the project is configured such that the contents of the gh-pages branch (feel free to inspect it) are used to populate the GitHub pages homepage of the project. This means that the version of the book displayed at https://rr-mrc-bsu.github.io/reproducible-research/ should automatically corresponds to the version obtained from the last commit to the repositories master branch.

Chapter 3

Version control

This chapter is dedicated to version contol

Chapter 4

R package development

4.1 Continuous integration

4.2 Unit testing

4.3 Documentation

Chapter 5

Build automation

Large projects can be a pain to manage. Small changes may break your software, or may deem your previously obtained analysis results useless. Build automation refers to a collection of tools that atempt to automate steps in your workflow, thereby simplifying your the whole process. Many processes may be automated, but here we will mainly discuss build automation using Makefiles.

5.1 Makefiles

Most research projects consist of several different connected components. For example, the end product might be a manuscipt, which depends on intermediate components such as a data analysis script and an R package. In this case, the manuscript depends on the data analysis script, which in turn depends on the R package. Such a hierarchy implies that every time a file changes, all files downstream in the hierarchy should be updated as well. In the previous example, we might adjust a function in the R package, which might change the outcome of the data analysis, and as a result, we'd have to re-run the data analysis script. The outcome of the data analysis might change the manuscript, so we'd have to re-compile that as well. In large projects, this quickly becomes tedious and difficult to maintain by hand. Luckily there is software available to streamline this process.

Consider the more complicated example in Figure 5.1 with the following corresponding project directory:

```
./
  rpackage
     DESCRIPTION
     functions.R
  code
     analysis.R
     simulation.R
  docs
     manuscript.Rnw
     presentation.Rnw
     refs.bib
  README.md
```

[to do: make diagram with DiagrammeR]

Re-running, -building, and -compiling all the files after we made a change to the anyone of the intermediate files would be a tedious task. Ideally, we would like to have a command that re-runs/compiles/builds the different files everytime an upstream change is made. This is exactly what the GNU software Make does. Make works through a Makefile, a file that describes how a target file, depends on its dependencies, and

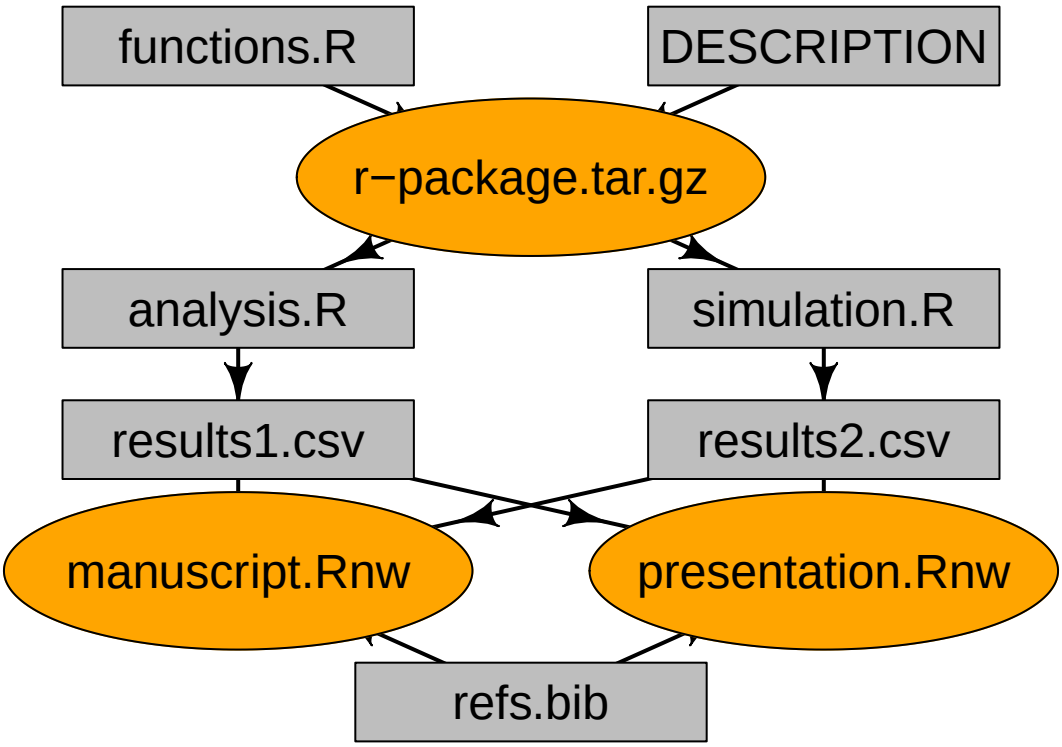

Figure 5.1: Example project workflow

how these in turn on their dependencies, and so on. If a Makefile is run, a file is compiled if any of its dependencies has changed since the last time the file was compiled. In other words, the Makefile starts at the top of the hierarchy and updates a file if its creation time is older than the creation time of its dependencies. In our example in Figure 5.1, if we make a change to the functions.R file, we trigger the recompilation of the recompilation of the r-package.tar.gz file, which in turn triggers a rerunning of the analysis.R and simulation.R scripts, and so on, until all files are up to date again.

[add example Makefile]

Chapter 6

Dependency management

Reproducible research, in the narrow sense in which it is defined in this book, ultimately means that an entire analysis - however complicated - can be repeated at the push of a button (or the command line equivalent: typing 'make') to yield the exact same figures, tables, files, or reports when aplied to the exact same data. In mathematical terms, one could go as far as requriing that an analysis must act as a function on data: there may very well be two data sets that produce the same output but the same input data must always produce the same analysis results. The core tools for achieving this are certainly literate programming, which allows to closer integrate documentation and code from a documentation-first perspective, and any form of build automation/workflow management system (GNU make, snakemake, CWL, etc.).

It is certainly worthwhile to take a step back here and reflect on the complexity of the approach that was put forward so far. None of the steps suggested is excessively complex or requires a particularly deep understanding of 'the command line' but in combination a sizeable stack of software dependencies has piled up:

1. On the base layer, there is the operating system itself.
2. The analysis is conducted by interacting with the operating system, ideally, via some form of terminal and shell.
3. Next, a workflow management system or build system like GNU make, snakemake, or CWL-runner should be used to 'tie everything together'
4. Version control softwae (usually git) is required to ensure integrity of the project
5. Usually some form of output is to be produced and the most reliable way to preserve digital documents at the moment seems to be .pdf. Note that changes in html/browser support may lead to different depiction of the same html file over the years! To create this output one will typically rely on some form of literate programming which has further dependencies. In the case of Rmarkdown and pdf output that would be R + pandoc + LaTeX.
6. Finally, the analysis itself will require even more complex software (R packages, python modules). In a scientific context there might also be bleeding edge software with no stabel release at all, which needs to be build from source.

All in all, this software stack is incredibly complex even for the simplest of analyses! Over time, each of these component layers might change/be updated at different pace or the developmet might simply cease. This situation can be pictured as a completed analysis being a delicate skyscraper: the moment it is finished it starts to slowly crumble away. [picture lening tower of pisa] Even if all all scripts and report files were preserved exactly in the state they were in when the analysis was conducted, the slowly evolving software ecosystem around them will still change over time and it might very well be that the excat same code would produce different results, or much more likely, simply stop working at some point.

It is thus not enough to simply maintain a versioned repository of all analysis scripts. Rather, complete reproducibility also requires a perfect snapshot of the software environment at the time of execution to be preserved for future execution - a time capsule of code if you will. A good first step in this direction

is certainly to report as many version numbers of software used as possible. This approach, however, is cumbersome, error prone and ultimately futile since users are typically not even aware of the phletora of low level software they implicitly rely on. A better approach would be to define explicitly the computing environment required for the intended computation and to preserve an exact 'image' of this environemnt which can then be replicated at a later point in time to re-run the analysis in it's original environment.

6.1 Example problem

Consider the toy example in https://github.com/rr-mrc-bsu/containerization-example.

Let's sort throught the individual files and folder one by one. Any well-organized repository should contain `README.md`, `LICENSE`, and `.gitignore` files. Their respective roles are described in the git chapter [write chapter, REFERENCE].

The `.travis.yml` configuration file contains the configuration of the continuous integration system (here: Travis-CI) linked to the repository. Continuous integration services allow automatic builds/checks of the code in a repository and greatly facilitate quality checking of new pull requests. To avoid a lengthly configuration file, the folder `.travis` contains further scripts which are referenced from the actual Travis configuration file. For details on continuous integration, see chapter 7.

The `Makefile` and the `Snakemake`-file are explained in more detail in chapter **??**chpt-workflow-automation). These files can be used to organize complex workflows using either GNU make (in simpler cases) or snakemake (more sophisticated). We will discuss how these tow common workflow automation systems can be used together with containers at the end of this chapter in section 6.4.

Finally, the Rmarkdown file `r-and-python.Rmd` and the `docker` folder are the elements of the example repository most important to the contents of this chapter. Assume that `r-and-python.Rmd` contains an important analysis which we want to render fully reproducible. As the file name suggests, the analysis relies on both R as well as python as well as a potentially large number of packages/modules for these languages. In fact, the file contains both R as well as python code. To learn more about the language agnostic nature of Rmarkdown and especially python/R interoperability, see [write chapter, REFERENCE]. The `docker`-folder contains a build script `docker/build` and a `dockerfile`. We will return to this folder in section 6.2.1. Before learning more about containers, it is a good idea to have a rough understanding of so called 'virtual machines'.

6.2 Virtual machines and containers

A virtual machines are (software) emulations of entire computer systems. As such they allow running various guest operating systems from withing a single host system, e.g., Linux within Windows or vice versa. To achieve this, a virtual machine acts as intermediate layer between the host system (and its hardware) and the guest operating system. I.e., all low-level operatons of the guest system are mapped through the virtual machine and the host operating system to the host hardware. A common software for running virtual machines is VirtualBox which can, e.g., be used to run an Ubuntu linux from within a Windows system. When using virtual machines for reproducible reseach, care should be taken to make the process of creating the virtual machine as transparent as possible. Command line tools like Vagrant are much better suited for theses use cases. We strongly discourage setting up a full-fledged Ubuntu system within VirtualBox and installing required software manually from within the virtual machine since this process is itself not reproducible. A virtual machine created in this way may render a particular piece of code reproducible but can itself not easily be recreated to the exact same specifications.

Instead, the reproducible research community is mainly embracing containers to create portable computing environments. For our purposes, containers can bee seen as a more leightweight alternative to virtual ma-

chines. Snakemake [write chapter, REFERENCE] for instance supports running workflows where individual steps are executed in their respective individual containers.

> In a nutshell, we may see container images as lightweight portable pieces of software which can be used to spawn container instances thus creating a **portable computing environment** by simply distributing the respective container image file.

If all dependencies of a particular analysis or an individual step in a larger workflow are contained in a container image, these dependencies will be available in any instance spawned from the image. By far the most common container software is Docker.

6.2.1 Docker

Docker constitutes the de-facto standard in terms of container software and hugely contributed to popularizing the concept of containerization in recent years. As most of the tools discussed in this book, Docker was nnever designed with reproducibility in mind but rather to enable the fast spinning-up of leightweight application containers to handle web-services etc. ('micro virtualization').

Since it is the de-facto standard, Docker is very well documented and t he docker community edition is an open source project and available free of charge. The company behind Docker als runs an online repository, Docker Hub, for docker images which can be seen as the GitHub/GitLab equivalent for docker images. Docker Hub can be used free of charge and thus enables the storage and sharing of custom container images via the world wide web.

A major drawback of Docker is, that it was never intended as a user-space application but mainly as a tool for server administrators. As such it requires root access to operate. While this is fine for buidling containers from so called `dockerfiles` (cf. `docker/dockerfile` in the exampel repository), the execution of an analysis should not require root access of the user. This is especially important when computations are conducted on shared resource like HPC systems where users do not have root access.

6.2.2 Singularity

Only relatively recently, the singularity container software was introduced to address this issue. Singularity was created exactly with HPC sytems and reproducibility in science in mind (see this video]. It does not require root access to run (but to build container images!) and thus enables HPC users to locally build container images which they can then use to run analyses on a high-performance cluster. A guiding pricipal in the development of singularity was to maintain compatibility with docker containers, i.e., singularity can be used to run standard docker containers. Since singularity is still rather a niche product, community help for docker is much easier to get online, and dockerhub is arguably the safest (and cheapest) place to store and distribute container images we still propose to use Docker for building the actual container images.

6.3 The dockerfile

A `dockerfile` contains the 'recipe' for building a docker container image. The complete reference of the syntax of dockerfiles can be found here. For our purposes, only a few commands will suffice to set up a docker container that contains all dependencies needed to render the Rmarkdown file `r-and-python.Rmd` of the example project. In fact, the entire contents of `docker/dockerfile` boil down to:

```
FROM rocker/verse:latest

MAINTAINER blameme@ihavenoclue.co.uk

# update apt
```

```
RUN sudo apt-get update

# install required R packages
RUN R -e "install.packages('reticulate')"

# install python
RUN sudo apt-get install -y python3-pip python3-dev python3-tk
RUN sudo pip3 install -U pip
RUN sudo pip3 install -U pandas matplotlib
```

Only three statements are used.

1. The `FROM` statement indicates the basis of the container. This allows to build on existing containers which may then be modified or extended to save time and storage space since images are composed of individual layers. By re-using previous layers with `FROM` the respective layer must only be saved once per container repository. Here, the container is derived from the latest version of `rocker/verse`, a relatively large container consisting of a stable debian linux system with R, Rstudio, the tidyverse packges, and all software required to render Rmarkdownr reports (LaTeX) pre-installed.
2. The `MAINTAINER` field simplycontains an email address to complain to.
3. The `RUN` statement can be used to execute commmands inside the container during the build. Here we use it to update the distibution package manager before installing the R package `reticulate` which enables interoperability between R and python before installing python, pandas and matplotlib.

For more information on using R and python together, see [TODO; REFERENCE].

We will now build the container image locally. To that end, clone the example repository to your local filesystem (cf. [TODO; REFERENCE GIT]) and `cd` to the example repository.

```
git clone https://github.com/rr-mrc-bsu/containerization-example
cd containerization-example/docker
```

Next, install docker and execute the build script in `docker/`

```
sudo docker build --no-cache -t mycontainer .
```

Note that you require root access to build the container! This command will trigger the build process (and take a while). Afterwards, a success message is displayed together with a unique sha256 hash value for the container image. This hash value can later be used to uniquely identify a particular versions of a container (similar to git commit hashes, cf. ???). Should you have a Docker Hub account you could then push the image by

```
docker push yourname/mycontainer
```

Otherwise, the image is only available locally. Note that having access to the image (or at least its exact hash) is extremely important to guarantee reproducibility. Simply rebuilding the image from the same dockerfile at a later timepoint will almost always result in a slightly different image when the build is configured to use the most recent package sources and versions of the software required. To avoid this, one may take steps to specify the software versions explicitly in the dockerfile but this never guarantees reproducible builds due to the many uncontrollable external dependencies. Still, keeping the dockerfile is important, since it may be seen as a guide as to how a similar computing environment can be set up manually at a later timepoint. Even if there are reasons not to use singularity to re-run an analysis, as good dockerfile might be the best way to specify software dependencies.

[TODO link to continuous integration for dockerfiles]

Switch back to the top level of the containerization-example folder now.

```
cd ..
```

A major advantage of singularity over docker over virtual machines is the ease with which singularity enables

execution of commands **inside of a docker container instance** but **within the host file system**, i.e., in contrast to docker, one does not need to manually mount a particular directory when starting a container but simply may invoke

```
singularity exec docker://kkmann/rr-containerization-example touch test
```

to execute the command `touch test` in an instance of the publicly built container image but **within the host filesystem**. This means that after executing the line in a shell, a new file 'test' should have been created in the current working directory. The touch command, however, was not executed in the host system but within the container! This example is obviously useless since `touch` is just as well available in the host shell but it immediately demonstrates the ease of using singularity with the host file system!

6.4 Containers and workflow management

6.4.1 GNU make

A much more useful application in our context is executing GNU make in a container! This means that we can render the Rmarkdown file in our container by

```
singularity exec docker://kkmann/rr-containerization-example make
```

Since all dependencies are pre-installed in the container image, this command only depends on the verson of singularity and the exact container image. To query the exact sha256 hash value, use `docker images`

```
docker images --digest | grep containerization
```

which will list the locally available corresponding container images and there exact hash values. To re-use a particular version, one may then invoke, e.g.,

```
singularity exec docker://kkmann/rr-containerization-example@sha256:5225e53f934d749bf3017f140e5169b0d2ea
```

6.4.2 Snakemake

Snakemake is a much more powerful tool whe it comes to complex workflows since it supports cluster execution and more flexible rule definitions (cf chapter [TODO, REFERENCE]). It is also designed with reproducibility in mind and thus allows to run either specific rules or an entire workflow using singularity. The contents of the `Snakefile` in the containerization-example folder are

```
singularity:
    "docker://kkmann/rr-containerization-example@sha256:5225e53f934d749bf3017f140e5169b0d2eadc0512799b8

rule build_report:
    input:
        "r-and-python.Rmd"
    output:
        "r-and-python.html"
    shell:
        """
        Rscript -e "rmarkdown::render(\\"{input}\\")"
        """
```

The first line specifies the singularity container to use. Here, the publicly available image from dockerhub is used with a particular hash value. One could just as well point to a local image as well though. Such a global singularity parameter will run every rule of the workflow in the same default container. It is also

possible to specify custom containers on a per-rule basis by specifying the `singularity` parameter of the rule.

```
rule build_report:
    input:
        "r-and-python.Rmd"
    output:
        "r-and-python.html"
    singularity:
        "docker://kkmann/rr-containerization-example@sha256:5225e53f934d749bf3017f140e5169b0d2eadc05127
    shell:
        """
        Rscript -e "rmarkdown::render(\\"{input}\\")"
        """
```

This will override potential gloabl singularity container settings (cf. [REFERENE SINGULARITY CHAPTER]).

6.5 Containerization vs. package managers

The approach to dependency management presented in this chapter might be considered a bit overpowered for some use cases. The main reason why we chose to demonstrate what we consider the 'gold-standard' of dependency management over more language/environment specific approaches is to is to showcase how simple it actually is. As long as you are capabale of running a few simple commands in a linux command line you are good to go. The container-based approach to dependency managemnt is also the most generic in that it is capable of managing arbitrary dependencies - as long as your computing environment can be set up on a linux operating system you are good to go! It does not matter which (or even how many different) programming languages you use, how much messy custom research software you need. As long as you are able to install it to a plain linux system there is no environment that cannot be mapped to a container (or several!). Still, for completeness' sake, a few different, potentially more accessible methods for partial dependency management will be discussed as 'honorable mentions' in the following.

6.5.1 R only - wrap everything up in a package

In case your entire workflow is R-based, it might be worthwhile to write an R packge for your analysis. This essentially relies on R's package management system to resolve dependencies (in this case: R packages only). An R analysis package tends to not contain much code in the `R/` folder, but rather encapsulates any analyses in vignettes. [TODO: elaborate on this idea, example repository]

6.5.2 R only - packrat

[TODO]

6.6 Caveats

Depending on your perspective, there are a few restrictions to a container-based dependency management approach.

1. **Licensing:** Reproducibility and the open-source philosophy are closely connected in that true reproducibility requires the software dependencies to be openly available. If the required dependencies cannot be installed in a container that may be distributed openly, results will not be reproducible by

everybody. For instance, while the SAS system for statistical analyses can be used inside containers, licensing and license consts may be a major obstacle in doing so.

2. **Containers are linux based:** The effective restriciton to open-source software also restricts the choice of container operating systems. In practice, containers are almost exclusively linux based. In case of, e.g., Windows dependencies, this means that the less flexible approach via virtual machines might be required to encapsulate the analysis environment.

While these restrictions might be seen as caveats, in fact, they can also be seen as an encouragement to conducting research in a more open (as in open-source-software-based) way.

Chapter 7

Continuous Integration

Did you ever wonder what the green/yellow/red 'badges' in some Readme.md files on, e.g., Github.com actually mean? How are they created, what are they for and why should you care?

This section will hopefully shed light on the meaning of some of these badges (those refering to a 'build status') and you will learn how to use these techniques for you own repositories. The key term here is 'continuous integration' (CI) which refers to a concept in software development where all working copies (in git one would refer to branches) of a project are frequently integrated into the mainline (in git terms: the master branch). The rationale being that frequent/continuous integration prevents diverging development branches. Since the philosophy of git is to create feature branches for small, contained changes of master which are to be merged back as soon as possible CI and git are a natural fit.

In practice, however, frequent changes to master are dangerous. After all, the master branch should maintain the last at least working if not stable version of the project that other feature branches can be started from. It is thus crucial to prevent erroneous changes to be merged into the master branch too often. This means that CI requires some kind of automated quality checks that preemptively check for new bugs/problems before a pull request from a feature branch on master is executed. It is this particular aspect of CI that is most interesting to collaborative work on scientific coding projects - being able to automatically run checks/tests on pull requests proposing changes to the master branch of the project.

[Random thought: we should have an example repository for demonstraiting the different states of PRs etc. instead of just including pictures. Readers could then inspect the 'frozen' repository directly and see the PRs etc.!]

To enable this kind of feature on Github, a cloud-based farm or build servers is required where users can run build scripts in virtual machines and retrieve reports on the build status (0 worked, 1 failed). It is these build-statuses that the green/yellow/red badges report visually (yellow/gray being a pending build)! There are multiple companies offering these services (within reasonable bounds) for free for public repositories and as of 2018 the free academic account for GitHub also enables free builds using TravisCI for private repositories. It must be stressed though that, since everything is running in the cloud, the same constraints as for storing data on GitHub servers apply to downloading or processing data in buildscripts for CI services. [point to Jenkins as on-premis solution]

The obvious setting to use automated builds in is package development. This is by far the most common application and the current tools are definitely geared towards that use case. We will later discuss how to extend the scope to non-package situations. For instance, the repository containing the source code for this book also uses TravisCI for build automation even though it is not an R-package itself.

Chapter 8

Python package development

8.1 Continuous integration

I agree, Kevin, we should some content.

8.2 Unit testing

8.3 Documentation

Chapter 9

Metadata

- what is metadata?
- indexing and search engines?
- long term storage/accessibility

9.1 FAIR principle

[todo]

9.2 Digintal Object Identifiers - DOI

[todo]

9.3 Zenodo.org

[todo]