Towards reproducible computational biology: An introductory tutorial

Release 2019.01

Sebastian Schmeier (https://sschmeier.com)

Jan 20, 2019

CONTENTS

CHAPTER

ONE

AN INTRODUCTORY TUTORIAL

This is an introductory tutorial for working in a reproducible manner in bioinformatics/genomics and related fields of study. You will learn how to analyse some next-generation sequencing (NGS) data. The idea here is not to facilitate the best possible analysis of this data or to use the best tools available, but rather to learn some tools that are available to us for making our analysis as reproducible as possible. The data you will be using is real research data, albeit down-sampled to make the analyses finish in a reasonable time.

Currently, Dr. Sebastian Schmeier[1] is teaching this material at Massey University in Auckland, New Zealand.

More information about other bioinformatics material, tutorials, and our research can be found on the webpages of the Schmeier Group[2] (https://sschmeier.com).

Note: A online version of this tutorial can be accessed at https://reproducibility.sschmeier.com.

1.1 Introduction

This is an introductory tutorial for learning to work reproducible in bioinformatics/genomics research. This tutorial makes extensive use of the command-line interface.

We will be analysing RNA-seq/transcriptomics samples from yeast. However, in the context of this tutorial the biological background or relevance is of no real importance to us. The aim here is not to understand the data, or bioinformatics tools for that matter, but rather how to structure the analysis steps in a way that you or someone else can redo the analysis and derive at the same results. Yes, the bioinformatics tools we are using here are real and the analyses performed here can be applied to other datasets too, however, all tools can be substituted for alternatives.

Note: The focus in this tutorial is not on the bioinformatics tools, but rather on the tools that facilitate reproducible analyses.

1.1.1 Prerequisites

- This tutorial generally assumes you work in a Unix/Linux type of environment (a minimal tutorial can be assessed here[3]), however, MacOS is fine too.

[1] https://sschmeier.com
[2] https://sschmeier.com
[3] https://linux.sschmeier.com

- You should be relatively comfortable using the command-line interface (you can update your knowledge using the excellent introductory material of the Software Carpentry[4]: here[5]).
- You should have a basic knowledge of version control. We are going to use Git[6] here. A good tutorial, again, is available from the Software Carpentry[7]: here[8]
- Any bioinformatics knowledge is a bonus but not strictly required (a Genomics tutorial can be assessed here[9]).

1.1.2 Learning outcomes

During this tutorial you will learn to:

- Use conda[10] and in particular Bioconda[11] *[GRUENING2018]* (page 35) for tool installation and tracking of the version numbers of the used tools.
- Use Snakemake[12] *[KOESTER2012]* (page 35) to create workflows / pipelines to analyse your data in a way that is accessible and reproducible.
- Understand how containerization of software can facilitate reproducibility in an operating system independent manner.
- Use Git[13] as a means to version control our analysis workflow and make it publicly available.

1.1.3 The data we will be using

In this tutorial we will analyse public data stemming from a transcriptomics experiment (RNA-sequencing) using next-generation sequencing (NGS). The associated publication is entitled *"Dynamics of the Saccharomyces cerevisiae Transcriptome during Bread Dough Fermentation"* and can be found here[14] *[ASLANKOOHI2013]* (page 35). The associated data has been deposited at the Short Read Archive[15] and can be found here (accession: PRJNA212389)[16]. The final aim in this tutorial is to quantify the expression of genes in each sample.

Note: The data has been downloaded already and is being made available within a Git[17] repository accompanying this tutorial. To facilitate timely analyses during this tutorial, **the original data has been down-sampled**.

An overview of a typical RNA-seq experiment can be seen in Fig. 1.1. RNA gets extracted from samples of interest and reverse transcribed and sequenced as a proxy for gene expression of the sample (either a set of cells or single cell).

1.1.4 The analysis workflow

We will be using a traditional set-up for analysing RNA-seq data, where sequenced reads will be cleaned, mapped to a reference genome, and finally reads per transcript/gene-model counted. The workflow is summarised in Fig. 1.2.

1. **Quality control** - We will be filtering reads based on read quality.

[4] https://software-carpentry.org
[5] http://swcarpentry.github.io/shell-novice/
[6] https://git-scm.com/
[7] https://software-carpentry.org
[8] http://swcarpentry.github.io/git-novice
[9] https://genomics.sschmeier.com
[10] http://conda.pydata.org/miniconda.html
[11] https://bioconda.github.io/
[12] http://snakemake.readthedocs.io/en/latest/
[13] https://git-scm.com/
[14] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3837736/
[15] https://www.ncbi.nlm.nih.gov/sra
[16] https://www.ncbi.nlm.nih.gov/bioproject/PRJNA212389
[17] https://git-scm.com/

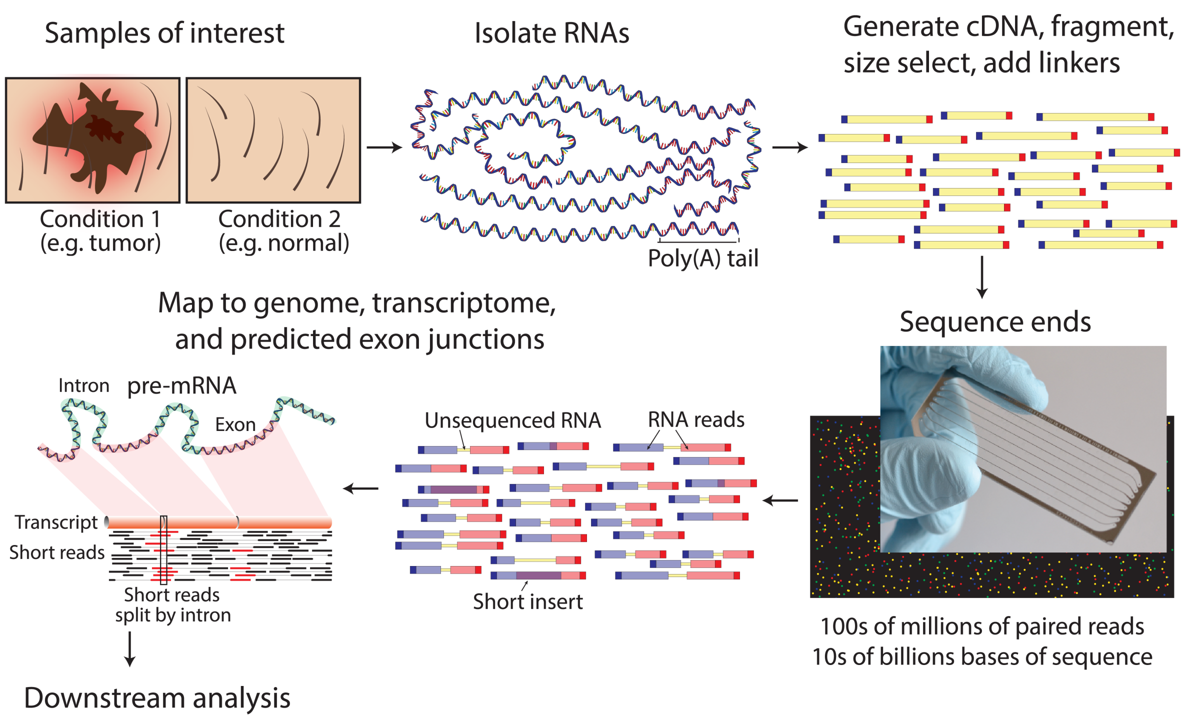

Samples of interest
Isolate RNAs
Generate cDNA, fragment, size select, add linkers
Condition 1 (e.g. tumor)
Condition 2 (e.g. normal)
Poly(A) tail
Sequence ends
Map to genome, transcriptome, and predicted exon junctions
Intron
pre-mRNA
Exon
Unsequenced RNA
RNA reads
Transcript
Short reads
Short reads split by intron
Short insert
100s of millions of paired reads
10s of billions bases of sequence
Downstream analysis

Fig. 1.1: RNA-seq overview (from https://doi.org/10.1371/journal.pcbi.1004393) *[GRIFFITH2015]* (page 35).

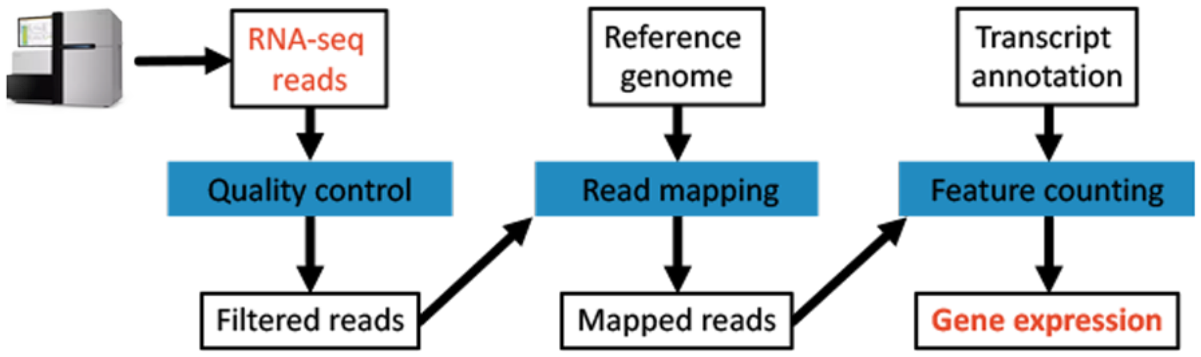


Fig. 1.2: The tutorial will analyse data using this workflow.

2. **Read mapping** - We Will be using a tool for mapping reads to the human genome.
3. **Feature counting** - We will count reads per gene model to derive gene expression values.

1.2 Working reproducible

1.2.1 What is the problem?

The current norm, when publishing your research manuscript does, for the most part, not require you to submit a detailed analysis protocol to the journal. This is true for experimental as well as computational work. This inhibits anyone (including the original authors) to reproduce the exact analysis steps that were performed using only the information contained within the journal article. Hence, deriving at the same results as the original study is based on luck, and for the most parts currently impossible.

While experimentalists, generally speaking, take care of recording their analyses in lab notebooks, these are by no means publicly available and often buried somewhere in the lab facility and often lost. The situation is worse in computational research, where lab notebooks are not the norm, and analyses are often done "as you go". This situation seems strange, as on the computational side one would expect that keeping score of the steps involved in an analysis should be easier.

1.2.2 How to tackle this problem?

In this tutorial we are dealing with the computational side of things. We will focus on using a set of tools that helps us developing analyses workflows that are reproducible, or at least as close as possible to being reproducible. The tools here are by no means the only ones and many other are available to help you in the task of keeping score of what you have done.

Our approach for getting a reproducible analysis in place will require:

1. **Keeping track of the used tools and their versions.** (addressed in *Tool and package management* (page 4))
2. **Keeping track of the commands used to analyse the data, including tool parameters.** (addressed in *Creating analysis workflows* (page 8))
3. **Publishing & versioning the workflow information, as to keep track of when workflows change and what changes occurred.** (addressed in *Creating analysis workflows* (page 8))

1.2.3 Background reading on reproducibility

- NIH plans to enhance reproducibility. *[COLLINS2014]* (page 35).
- A Framework for Improving the Quality of Research in the Biological Sciences. *[CASADEVALL2016]* (page 35)
- All hail reproducibility in microbiome research. *[RAVEL2014]* (page 35)
- Quantifying reproducibility in computational biology: The case of the tuberculosis drugome. *[GARIJO2013]* (page 35)
- A quick guide to organizing computational biology projects. *[NOBLE2009]* (page 35)
- Investigating reproducibility and tracking provenance. *[KANWAL2017]* (page 35)

1.3 Tool and package management

Here, we will tackle the first item on the list towards more reproducibility:

Keeping track of the used tools and their versions.

This can be achieved by using an appropriate package manager. As an example we are going to use conda[62] with the Bioconda[63] software channel.

[62] http://conda.pydata.org/miniconda.html
[63] https://bioconda.github.io/

1.3.1 Installing the Conda package manager

We will use the package/tool managing system conda[64] to install some programs that we will use during the course. It is not installed by default, thus we need to install it first to be able to use it.

```
# download latest conda installer
$ curl -O https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh

# run the installer
$ bash Miniconda3-latest-Linux-x86_64.sh

# delete the installer after successful run
$ rm Miniconda3-latest-Linux-x86_64.sh
```

Note: Should the conda installer download fail. Please find links to alternative locations on the *Downloads* (page 30) page.

Update `.bashrc` and `.zshrc` config-files

Before we are able to use conda we need to tell our shell where it can find the program. We add the right path to the conda installation to our shell config files:

```
$ echo 'export PATH="~/miniconda3/bin:$PATH"' >> ~/.bashrc
$ echo 'export PATH="~/miniconda3/bin:$PATH"' >> ~/.zshrc
```

So what is actually happening here? We are appending a line to a file (either `.bashrc` or `.zshrc`). If we are starting a new command-line shell, either file gets executed first (depending on which shell you are using, either bash or zsh shells). What this line does, is to put permanently the directory `~/miniconda3/bin` first on your `PATH` variable. The `PATH` variable contains directories in which our computer looks for installed programs, one directory after the other until the program you requested is found (or not, then it will complain). Through the addition of the above line we make sure that the program `conda` can be found any time we open a new shell.

Close shell/terminal, **re-open** new shell/terminal. Now, we should be able to use the conda command:

```
$ conda update conda
```

Installing conda channels to make tools available

Different tools are packaged in what conda calls channels. We need to add some channels to make the bioinformatics and genomics tools available for installation. In particular we need the Bioconda[65] channel, that pre-packages many bioinformatics tools.

```
# Install some conda channels
# A channel is where conda looks for packages
$ conda config --add channels defaults
$ conda config --add channels conda-forge
$ conda config --add channels bioconda
```

1.3.2 Using conda to search and install tools

Let us first look for a tool, e.g. the aligner BWA[66]:

```
# Look for available tools/packages
$ conda search bwa
```

(continues on next page)

[64] http://conda.pydata.org/miniconda.html
[65] https://bioconda.github.io/
[66] http://bio-bwa.sourceforge.net/

(continued from previous page)

```
Loading channels: done
# Name                  Version           Build  Channel
bwa                       0.5.9               0  bioconda
bwa                       0.5.9               1  bioconda
bwa                       0.6.2               0  bioconda
bwa                       0.6.2               1  bioconda
bwa                      0.7.3a               0  bioconda
...
```

We can see that the tool is available and several versions can be installed. To install software (here BWA) using conda, one uses the command `conda install`:

```
# install a tool into the environment
$ conda install bwa
# to install a particular version of a tool do
$ conda install bwa=0.6.2
```

Note: Without a version number conda[67] tries to install the latest version for you.

While conda was in the first place not developed for bioinformatics/genomics type of tools/packages, clever people took the system and packaged bioinformatics tools into the conda system. To not confuse things with the original conda system, people are using "channels" to distribute software that is related. We already made three software "channels" available to our conda installation: conda-forge, defaults, bioconda. Specifically, the Bioconda channel is of importance to us as it makes ~3000 bioinformatics packages available to us *[GRUENING2018]* (page 35).

1.3.3 Create isolated environments

While having one software manager for all your bioinformatics tools is great already, conda has one particular strength that we are going to exploit often during the course of this tutorial. Conda can create isolated environments for sets of user-defined tools. The tools and their version numbers within environments, once created, can be easily saved in a file. Using these files we can easily re-create an environment from scratch with the same tool-set with the same version numbers. *Awesome!*

```
# Create a base environment
$ conda create -n tutorial python=3
# Activate the environment
$ conda activate tutorial
```

So what is happening when you type `conda activate tutorial` in a shell. The `PATH` variable (mentioned above) gets temporarily manipulated and set to:

```
$ conda activate tutorial
# Lets look at the content of the PATH variable
(tutorial) $ echo $PATH
~/miniconda3/envs/tutorial/bin:~/miniconda3/bin:/usr/local/bin: ...
```

Now it will look first in your environment's `bin`-directory (here `~/miniconda3/envs/tutorial/bin`) and only afterwards in the general conda[68] `bin`-directory (`~/miniconda3/bin`). So basically everything you install generally with `conda install` (without being in an environment) is also available to you but gets overshadowed if a similar program is in `~/miniconda3/envs/tutorial/bin` and you have activated the `tutorial` environment.

Note: To tell if you are in the correct conda environment, look at the command-prompt. Do you see the name of the environment in round brackets at the very beginning of the prompt, e.g. (tutorial)? If not,

[67] http://conda.pydata.org/miniconda.html
[68] http://conda.pydata.org/miniconda.html

activate the `tutorial` environment with `conda activate tutorial` before installing the tools.

To leave an environment just type:

```
(tutorial) $ conda deactivate
# Lets look at the content of the PATH variable
$ echo $PATH
~/miniconda3/bin:/usr/local/bin: ...
```

The command `conda list` will show you the packages that are installed within the environment:

```
$ conda activate tutorial
# list all installed
(tutorial) $ conda list
```

Looks like the tools `bwa` we wanted is installed.

Ok, now we want to get a snapshot of the current environment so that we could recreate it either here or on another machine running the same operating system.

```
# Lets export the environment into a yaml-file
(tutorial) $ conda env export > tutorial.yaml
```

Lets have a look into the `tutorial.yaml` file.

```
(tutorial) $ cat tutorial.yaml
```

To deactivate the environment again type:

```
# Deactivate environment
(tutorial) $ conda deactivate
```

Now we delete the environment, specifying the name again with `-n tutorial`

```
# Delete original "tutorial" environment
$ conda env remove -n tutorial
```

Now, we can use the created `yaml`-file to recreate the former `tutorial` environment, we submit the file with `--file` to `conda env create`.

```
# Lets recreate an environment using the tutorial.yaml file
$ conda env create -n tutorial --file tutorial.yml

# Activate environment
$ conda activate tutorial
```

Done! So we learned that we can create conda environments for a certain tool or toolset/packages and store the installed tools and their installed version numbers in a `yaml`-file that can be used to recreate the environment. This enables us in a very easy way to keep track of the tools and versions used in our analysis.

Note: It is good practice to include a `yaml` file of your environment in your analysis directory and submit it together with the rest of your code.

1.3.4 General Conda commands

```
# to search for packages
conda search PACKAGE

```

(continues on next page)

(continued from previous page)

```
# Install
conda install PACKAGE

# To update all packages
conda update --all --yes

# List all packages installed
conda list [-n ENV]

# conda list environments
conda env list

# create new environment with packages
conda create -n ENV PACKAGE [PACKAGE] ...

# activate environment
conda activate ENV

# deavtivate environment
conda deactivate

# export env
conda env export > env.yaml

# recreate env from file
conda env create -n ENV -f env.yaml
```

1.4 Creating analysis workflows

1.4.1 What is a workflow management system?

A workflow management system provides an infrastructure for the setting up, performing and monitoring defined sequences of commands, hence addressing our second requirement:

Keeping track of the commands used to analyse the data, including tool parameters.

There are many such systems and some have been specifically designed for genomics and bioinformatics tasks. An recent overview can be accessed here *[LEIPZIG2017]* (page 35).

1.4.2 What is Snakemake?

Snakemake[86] is a workflow engine, developed for creating scalable bioinformatics and genomics workflows *[KOESTER2012]* (page 35). It borrows ideas from an old system for compiling and link a program Make[87] and extends the ideas to help with bioinformatics pipelines. We will be using Snakemake to run our complete analysis for us, from start to end. Snakemake will make sure that our jobs are run in the correct order and will recognise if jobs have already been run and thus do not have to be run again. It will also recognise if input files changed and thus jobs have to be re-run. Correctly configured, Snakemake will take care of error logging, benchmarking, and simultaneous execution of our jobs (it is also able to distribute jobs to computer clusters). We will see that combined with conda[88] it makes for a powerful system for developing and running reproducible analyses workflows.

1.4.3 Setup

Install Snakemake

We are going to create a general conda environment and install Snakemake into it.

[86] http://snakemake.readthedocs.io/en/latest/
[87] https://www.gnu.org/software/make/manual/make.html
[88] http://conda.pydata.org/miniconda.html

```
$ conda create -n tutorial python=3 snakemake
# activate the environment
$ source activate tutorial
```

Download data

I prepared a Git[89] repository that contains the some data we will work with. You can download the repository using:

```
$ git clone https://gitlab.com/schmeierlab/reproduce-tutorial.git

# Change into the created directory
$ cd reproduce-tutorial

# Let delete the associated git remote
$ git remote rm origin
```

Let's investigate the directory:

```
$ tree
.
├── README.md
├── Snakefile
├── analyses
│   └── results
│       └── README.md
├── data
│   ├── Saccharomyces_cerevisiae.R64-1-1.92.gtf.gz
│   └── Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa
├── envs
│   ├── map.yaml
│   ├── sickle.yaml
│   ├── snakemake-kubernetes.yaml
│   ├── snakemake.yaml
│   └── subread.yaml
├── examples
│   ├── Snakefile_v2
│   ├── ...
├── fastq
│   ├── SRR941826.fastq.gz
│   ├── SRR941827.fastq.gz
│   ├── SRR941830.fastq.gz
│   └── SRR941831.fastq.gz
└── help
    ├── bwa.help
    ├── featureCounts.help
    ├── sickle.se.help
    └── snakemake.help
```

This directory contains all the files we need to do this tutorial. There are **four** fastq-files in the `fastq` directory that we want to clean and map to the reference genome. Finally, we will count the reads per gene and per sample. The complete workflow is depicted in Fig. 1.3.

Note: The repository contains the downloaded the genome, the annotation, and the samples already. This can be done as well via Snakemake[90] but goes beyond the topic of this tutorial. Should you be interested to see how this was done later on, you can have a look here[91] and here[92].

[89] https://git-scm.com/
[90] http://snakemake.readthedocs.io/en/latest/
[91] https://gitlab.com/snippets/1723667
[92] https://gitlab.com/snippets/1723667

1.4.4 The analysis without a workflow management system

We can of course do this analysis without any workflow management system and write down the commands one by one. Given that we only have four samples, this is not particular difficult. However, this process does not scale well if we decide to do it for 400 samples. So we are later going to use a workflow management system that creates for us the commands for each sample without us doing it explicitly. However, here we are going to look at each step and write down the command for one of the samples (`SRR941826`) to understand what is involved.

Activate our conda[93] environment for the tutorial session:

```
conda activate tutorial
```

Data QC

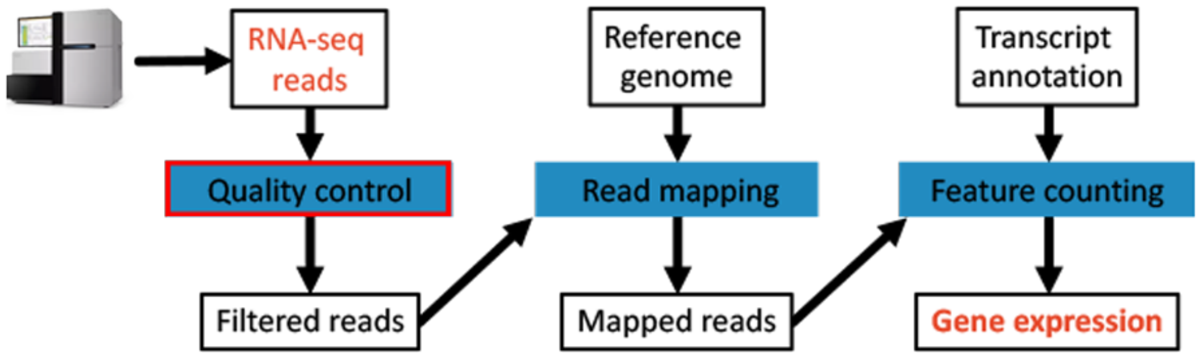


Fig. 1.3: The workflow: Data QC step.

The purpose of this step is to remove bases from the ends of the reads that are of bad quality. There are many tools that can do this for us. Here, we are going to use the program Sickle[94] to perform this task *[JOSHI2011]* (page 35). The program is easy to use.

After installing Sickle[95], running sickle by itself will print the help:

```
# install sickle
$ conda install sickle-trim
$ sickle
```

Running sickle with either the single-end (*se*) or paired-end (*pe*) reads:

```
sickle se
sickle pe
```

Here the command for our single-end fastq-file:

```
$ sickle se -g -t sanger -f fastq/SRR941826.fastq.gz -o analyses/results/SRR941826.trimmed.fastq.gz
```

- `-g`: will facilitate gzip output
- `-t`: specifies the quality metric used in the fastq-file
- `-f`: input filename
- `-o`: output filename

Easy enough! Lets map reads.

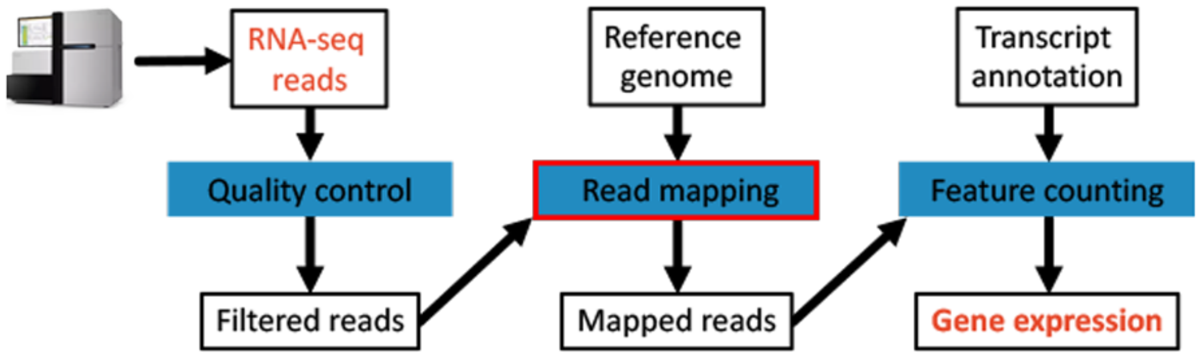


Fig. 1.4: The workflow: The mapping step.

Read mapping

We use BWA[96] and SAMtools[97] to get mapped reads in bam-format. In order to map the reads to the genome we first need to index the genome:

```
$ conda install bwa samtools
$ bwa index data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa
```

Now we can take a sample and map it:

```
$ bwa mem -t 8 data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa analyses/results/SRR941826.
↪trimmed.fastq.gz
  | samtools view -Sbh > analyses/results/SRR941827.bam
```

- BWA[98]:`-t 8`: specify how many threads can be used at the same time.
- SAMtools[99] view: `-Sbh`: Include the header in the output and creates bam format output.

Feature counting

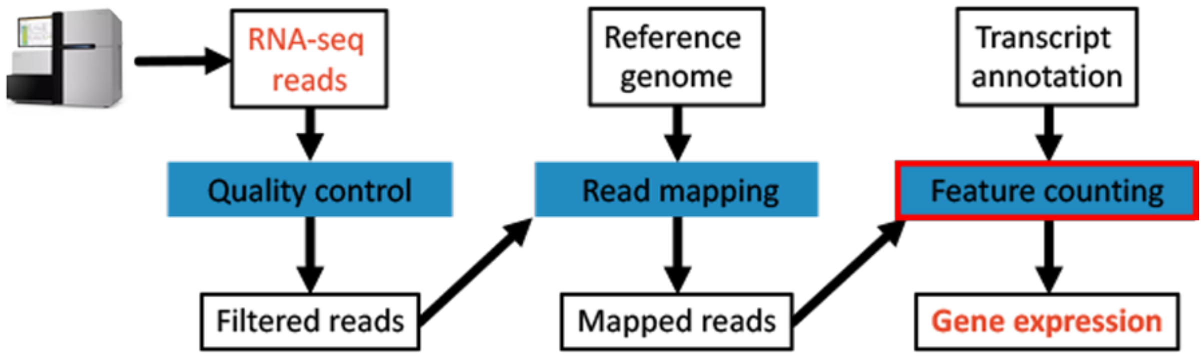


Fig. 1.5: The workflow: Expression quantification step.

We are using the `featureCounts` tool of the `subread` package to count reads per feature.

```
$ conda install subread
$ featureCounts -T 4 -t exon -g gene_id
```

(continues on next page)

93 http://conda.pydata.org/miniconda.html
94 https://github.com/najoshi/sickle
95 https://github.com/najoshi/sickle
96 http://bio-bwa.sourceforge.net/
97 http://samtools.sourceforge.net/
98 http://bio-bwa.sourceforge.net/
99 http://samtools.sourceforge.net/

(continued from previous page)

```
  -a data/Saccharomyces_cerevisiae.R64-1-1.92.gtf.gz
  -o counts.txt analyses/results/SRR941826.bam
```

- `-T`: Number of threads to use at the same time
- `-t`: Specify feature type in GTF annotation. Features used for read counting will be extracted from annotation using the provided value.
- `-g`: Specify attribute type in GTF annotation
- `-a`: The annotation file with the features
- `-o`: Output file

Saving tool version information

Great, we have done our analysis of all four samples. Now, we can export our conda[100] environment and save the information in a file:

```
$ conda activate tutorial
$ conda env export > tutorial.yaml
```

This file contains the tools and their versions that we used in this analysis. We could give this file to someone else and they could, given they work on the same operating system, recreate the same conda[101] environment and redo the analysis.

Summary

So, generally we could use the programs on the command-line like shown above with one sample after the other. However, we do not want to do this, as in this manner we do not save the information about the commands we used. We could of course put all commands in a bash-script and in this manner remember all commands that have been run.

However, we would still not be able to keep this approach general to run it again and again on different (numbers) of samples. **We can do better!** This is were Snakemake[102] comes into play.

1.4.5 Using a workflow management system

Let's look at the complete workflow again for one samples:

```
# 1 Trimming
$ sickle se -g -t sanger -f fastq/SRR941826.fastq.gz -o analyses/results/SRR941826.trimmed.fastq.gz

# 2 Genome indexing
$ bwa index data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa

# 3 Read mapping
$ bwa mem -t 8 data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa analyses/results/SRR941826.
↪trimmed.fastq.gz
     | samtools view -Sbh > analyses/results/SRR941827.bam

# 4 Counting reads per features
$ featureCounts -T 4 -t exon -g gene_id
    -a data/Saccharomyces_cerevisiae.R64-1-1.92.gtf.gz
    -o counts.txt analyses/results/SRR941826.bam
```

[100] http://conda.pydata.org/miniconda.html
[101] http://conda.pydata.org/miniconda.html
[102] http://snakemake.readthedocs.io/en/latest/

We will engineer now this workflow in the workflow management system Snakemake[103]. However, it will be general with no filenames hard-coded, so that we can run the same workflow on any arbitrary number of fastq-files of the same type, here single-end reads.

Snakemake

Snakemake[104] uses rules that define how to get from an *input* to an *output*. These rules are defined in a `Snakefile` that is read upon Snakemake execution. A basic structure of a rule looks like this:

Listing 1.1: Example rule in a Snakefile

```
rule do-something:
    input:
        "{sample}.fastq"
    output:
        "{sample}.out"
    shell:
        "SOMECOMMAND {input} {output}"
```

In this example, we have an input file and an output file, as well as a way to get from the input to the output via a shell command. Rules can either use shell commands, plain Python code or external scripts to create output files from input files. Curly brackets define *wildcards* that get substituted.

Once, you have written lots of rules, Snakemake determines the **rule dependencies by matching file names**.

Let us write a rule for trimming to see how this works in practice.

Creating rules

Let us build a general rule for the first step, the read trimming via `sickle`. The original command was:

```
$ sickle se -g -t sanger -f fastq/SRR941826.fastq.gz -o analyses/results/SRR941826.trimmed.fastq.gz
```

We will use this command in our first rule and substitute the input and output part, as well as some of the parameters with **wildcards**.

In the working directory their is an empty `Snakefile`. We will add to this file during the tutorial. Open this file in a text editor.

Listing 1.2: File: Snakefile_v1

```
rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
```

The command in the shell keyword section will be used to trim the data. However, before it is executed, Snakemake will replace the *wildcards* in the command with the proper values defined in the other sections of the rule. Of note, we introduce here as well a keyword `params` (highlighted lines), with which one can add more flexibility to the values that get substituted in the shell command.

Fine, but what is happening with the strange `{sample}` *wildcard*? The *wildcard* will be replaced by Snakemake to try and match our requested final targets. However, we have **not** defined any targets yet.

[103] http://snakemake.readthedocs.io/en/latest/
[104] http://snakemake.readthedocs.io/en/latest/

Snakemake needs to know for what to run this rule. We need to define result or target files we want to create.

Note: 1. Snakemake works by matching file-names, i.e. finding rules that can generate the requested files from other files. **2.** Snakemake automatically creates directories if they do not already exist (e.g. `analyses/results/`).

Lets define some targets. We are creating a *pseudorule* (`all`) that only defines inputs, which are our expected final targets. In this way, Snakemake finds the rule `all` and tries to figure out which rules can be run to create the desired output target files. It will find that our rule `trimse` can accomplish this when run four times with four different input files by substituting the `{sample}` *wildcard*.

Listing 1.3: File: Snakefile_v2

```
rule all:
    input:
        ["analyses/results/SRR941826.trimmed.fastq.gz",
         "analyses/results/SRR941827.trimmed.fastq.gz",
         "analyses/results/SRR941830.trimmed.fastq.gz",
         "analyses/results/SRR941831.trimmed.fastq.gz"]

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
```

We defined four target files. Let us look what happens here for one of the target files when you run Snakemake.

- Snakemake finds that you request to create the file `analyses/results/SRR941826.trimmed.fastq.gz`.
- Snakemake will scan all rules, to identify which rule can create this file (Snakemake will try to substitute any *wildcards* like `{sample}` and try to match file names).
- In our case, it will find that substituting `{sample}` in the output section of rule `trimse` (`analyses/results/{sample}.trimmed.fastq.gz`) with `SRR941826` will match the requested target file name `analyses/results/SRR941826.trimmed.fastq.gz`
- Snakemake will check if the input file of rule `trimse` (`fastq/{sample}.fastq.gz`) with a substituted `{sample}` part with `SRR941826` (`fastq/SRR941826.fastq.gz`) exists.
- If this file can be found the rule will be scheduled for execution. If the input cannot be found, Snakemake will complain that the requred input for creating the requested file is missing.

Ok, this `Snakefile` can already be run with the `snakemake` command. We can do a dry-run (without actually running anything) to see the commands that Snakemake would execute. We use the `snakemake` flag `-n` for dry-run and `-p` to see the commands that Snakemake would execute:

```
$ snakemake -np
Building DAG of jobs...
Job counts:
    count   jobs
    1       all
    4       trimse
    5
```

(continues on next page)

(continued from previous page)

```
rule trimse:
    input: fastq/SRR941830.fastq.gz
    output: analyses/results/SRR941830.trimmed.fastq.gz
    jobid: 3
    wildcards: sample=SRR941830

sickle se -g -t sanger -f fastq/SRR941830.fastq.gz -o analyses/results/SRR941830.trimmed.fastq.gz
...
```

Nice, the Snakemake correctly substituted the files in input and output to create the correct commands for trimming.

Note: We used explicit file names for the expected target files in rule `all` based on the input sample file names we knew existed in the `fastq` directory . However, we want to be general to be able to run any files located in the `fastq` directory without explicitly listing them. Snakemake should identify the input files and create expected target file names automatically based on the input file names.

Lets rewrite it a bit to make the workflow more general, so that any file located in the `fastq` directory that has the right file name structure is found by Snakemake:

Listing 1.4: File: Snakefile_v3

```
SAMPLES, = glob_wildcards("fastq/{sample}.fastq.gz")

rule all:
    input:
        expand("analyses/results/{sample}.trimmed.fastq.gz", sample=SAMPLES)

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
```

- We use an inbuilt function `glob_wilcards` to scan the `fastq` directory for files of a certain structure and extract the sample identifier out of the file names.
- We use another inbuilt function `expand` in rule `all` to create a new "target" file name for each sample identifier collected in the step before.

Running the same `snakemake -np` command again, will yield the same result as in the explicit case. However, now it would not matter if we would add another 100 files to the `fastq` directory. Snakemake would find them, without us doing anything else.

Error logging and benchmarking

There are a few things we can still add to our rule to facilitate error logging and benchmarking (the process of testing how long a task takes):

Listing 1.5: File: Snakefile_v4

```
SAMPLES, = glob_wildcards("fastq/{sample}.fastq.gz")

rule all:
    input:
```

(continues on next page)

(continued from previous page)

```
        expand("analyses/results/{sample}.trimmed.fastq.gz", sample=SAMPLES)

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
    benchmark:
        "analyses/benchmarks/{sample}.trimse"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
        " 2> {log}"
```

- We are adding another keyword (`log`), specify the file where we want the logging to end up in (of note: it also contains the same *wildcard* as the input and output files, thus gets substituted as well), and add the error redirection to the *wildcard* `log` in the shell command.
- We add another keyword (`benchmark`) and specify the file where we want the data to end up in. We do not do anything else. Snakemake will take care of benchmarking for us.

Note: You should realise that the shell command can be written over several lines, like in the example above.

If you rerun the `snakemake` command logging and benchmarking will be visible in the execution plan of Snakemake. Because we specified the same *wildcard* in the benchmark file name, it gets substituted with the sample identifier too.

```
$ snakemake -np
Building DAG of jobs...
Job counts:
        count   jobs
        1       all
        4       trimse
        5

rule trimse:
    input: fastq/SRR941830.fastq.gz
    output: analyses/results/SRR941830.trimmed.fastq.gz
    log: analyses/logs/SRR941830.trimse
    jobid: 3
    benchmark: analyses/benchmarks/SRR941830.trimse
    wildcards: sample=SRR941830

sickle se -g -t sanger -f fastq/SRR941830.fastq.gz -o analyses/results/SRR941830.trimmed.fastq.gz 2>
↪ analyses/logs/SRR941830.trimse
...
```

Integrating package management

Now that we are after reproducibility, we need to integrate package management into the workflow. This is easily done with Snakemake[105] using conda[106]. We can add another keyword argument to our rule that specifies the conda environment that will be activated before running the particular rule.

You can find a minimal conda environment file for `sickle` in the `envs` directory: `sickle.yaml`.

[105] http://snakemake.readthedocs.io/en/latest/
[106] http://conda.pydata.org/miniconda.html

Listing 1.6: File: envs/sickle.yaml

```
channels:
    - bioconda
    - conda-forge
    - defaults
dependencies:
    - sickle-trim ==1.33
```

The file specifies that the rule should be run with `sickle` version 1.33. Before running anything, Snakemake will create the environment and store it in the current working directory in a subdirectory of `.snakemake`. It will only be recreated if the yaml file changes. Otherwise, if Snakemake is rerun it will use the already created environment.

Let us integrate conda and the environment in our `sickle` rule:

Listing 1.7: File: Snakefile_v5

```
SAMPLES, = glob_wildcards("fastq/{sample}.fastq.gz")

rule all:
    input:
        expand("analyses/results/{sample}.trimmed.fastq.gz", sample=SAMPLES)

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
    benchmark:
        "analyses/benchmarks/{sample}.trimse"
    conda:
        "envs/sickle.yaml"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
        " 2> {log}"
```

In order to tell the `snakemake` command that we want to make use of conda environments in our rules, we need to specify the `--use-conda` flag when running Snakemake.

Note: Integrating conda's environment capability in Snakemake is a powerful way to keep track of what tools and versions of tools have been used.

Running Snakemake

Now that we have our first rules established, let us run Snakemake and look at the outputs.

In order to actually run the Snakemake workflow we need to adjust the `snakemake` command a bit:

```
$ snakemake -p --use-conda
```

We got rid of the `-n` flag that signalled a dry-run. We also added the `--use-conda` flag as we want Snakemake to use conda environments when running our rules.

Let us look at the results in the `analyses` folder:

```
$ tree analyses
analyses/
├── benchmarks
│   ├── SRR941826.trimse
│   ├── SRR941827.trimse
│   ├── SRR941830.trimse
│   └── SRR941831.trimse
├── logs
│   ├── SRR941826.trimse
│   ├── SRR941827.trimse
│   ├── SRR941830.trimse
│   └── SRR941831.trimse
└── results
    ├── SRR941826.trimmed.fastq.gz
    ├── SRR941827.trimmed.fastq.gz
    ├── SRR941830.trimmed.fastq.gz
    └── SRR941831.trimmed.fastq.gz

3 directories, 12 files
$ cat analyses/benchmarks/SRR941826.trimse
s       h:m:s   max_rss max_vms max_uss max_pss io_in   io_out  mean_load
0.7001  0:00:00 4.79    19.03   0.86    0.95    0.00    0.60    0.00
$ cat analyses/logs/SRR941826.trimse
$
```

Our result files, as well as benchmarks and log files, have been created.

If we would rerun snakemake, it would tell us that there are no jobs needed to run as all requirements (targets) are satisfied.

```
$ snakemake -p --use-conda
Building DAG of jobs...
Nothing to be done.
Complete log: .../reproduce-tutorial/.snakemake/log/2018-06-12T142827.292869.snakemake.log
```

Let us delete a particular result file to test if Snakemake will realize that one sample is missing and still needs to be run:

```
$ rm analyses/results/SRR941826.trimmed.fastq.gz
$ $ snakemake -n --use-conda
Building DAG of jobs...
Job counts:
        count   jobs
        1       all
        1       trimse
        2

rule trimse:
    input: fastq/SRR941826.fastq.gz
    output: analyses/results/SRR941826.trimmed.fastq.gz
    log: analyses/logs/SRR941826.trimse
    jobid: 3
    benchmark: analyses/benchmarks/SRR941826.trimse
    wildcards: sample=SRR941826


localrule all:
    input: analyses/results/SRR941827.trimmed.fastq.gz, analyses/results/SRR941826.trimmed.fastq.gz,
↪ analyses/results/SRR941831.trimmed.fastq.gz, analyses/results/SRR941830.trimmed.fastq.gz
    jobid: 0

Job counts:
```

(continues on next page)

(continued from previous page)

```
        count   jobs
        1       all
        1       trimse
        2
```

Indeed, Snakemake finds that we still need to run the rule `trimse` once to fulfil all requirements in rule `all`.

Note: Benchmark and logging files will be overwritten in subsequent runs for the same sample.

Visualising the workflow graph

Internally, Snakemake is creating a directed acyclic graph (DAG) of the rules and their dependencies. We can generate a graphical visualisation of the graph and thus workflow (Fig. 1.6) using Snakemake and Graphviz[107]:

```
$ snakemake --dag | dot -Tpng > dag.png
```

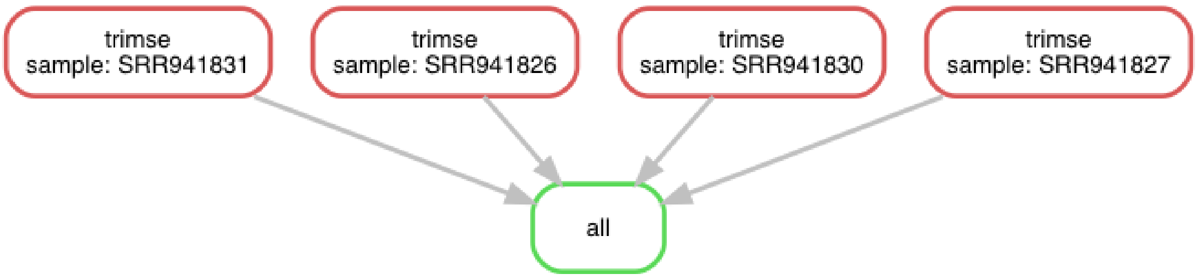


Fig. 1.6: The workflow v5 as a directed acyclic graph.

Note: This visualisation becomes bigger and bigger with more and more samples.

Building the remaining rules

We are adding two more rules. One rule for indexing of the genome using BWA and the another that will take the index, as well as a sample `fastq` file, and map the reads to the genome.

We are adding to our `Snakefile`.

Listing 1.8: File: Snakefile_v6

```
SAMPLES, = glob_wildcards("fastq/{sample}.fastq.gz")

rule all:
    input:
        expand("analyses/results/{sample}.bam", sample=SAMPLES)

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
```

(continues on next page)

107 http://graphviz.org/

(continued from previous page)

```
14      benchmark:
15          "analyses/benchmarks/{sample}.trimse"
16      conda:
17          "envs/sickle.yaml"
18      params:
19          qualtype="sanger"
20      shell:
21          "sickle se -g -t {params.qualtype} -f {input} -o {output}"
22          " 2> {log}"
23
24  rule makeidx:
25      input:
26          fasta = "data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa"
27      output:
28          touch("data/makeidx.done")
29      log:
30          "analyses/logs/makeidx.log"
31      benchmark:
32          "analyses/benchmarks/makeidx.txt"
33      conda:
34          "envs/map.yaml"
35      shell:
36          "bwa index {input.fasta} 2> {log}"
37
38  rule map:
39      input:
40          reads = "analyses/results/{sample}.trimmed.fastq.gz",
41          idxdone = "data/makeidx.done"
42      output:
43          "analyses/results/{sample}.bam"
44      log:
45          "analyses/logs/{sample}.map"
46      benchmark:
47          "analyses/benchmarks/{sample}.map"
48      threads: 8
49      conda:
50          "envs/map.yaml"
51      params:
52          idx = "data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa"
53      shell:
54          "bwa mem -t {threads} {params.idx} {input.reads} | "
55          "samtools view -Sbh > {output} 2> {log}"
```

There are a few points that need a closer look:

1. On *line 5* we are changing the final output to `bam` files, so that the mapping is run for all samples.
2. The shell command to index the genome with BWA does not create an output. Thus, we create a pseudo output. On *line 28* We are using the function `touch` to create an empty file after the rule is successfully run. We require this file as input for the map rule on *line 41* so that the indexing rule is run before any mapping happens.
3. *Line 41* also shows that we can have more than one input to a rule.
4. We see a new keyword in a rule on *line 48* `threads`. This can be used to specify the number of threads allowed for the rule when running `snakemake` with the `--jobs NUM` flag.
5. *Line 54 and 55* show that we can chain shell commands easily in a Snakemake rule.
6. We also added a separate conda environment for indexing and mapping.

Finally, we add the rule to count the reads per features.

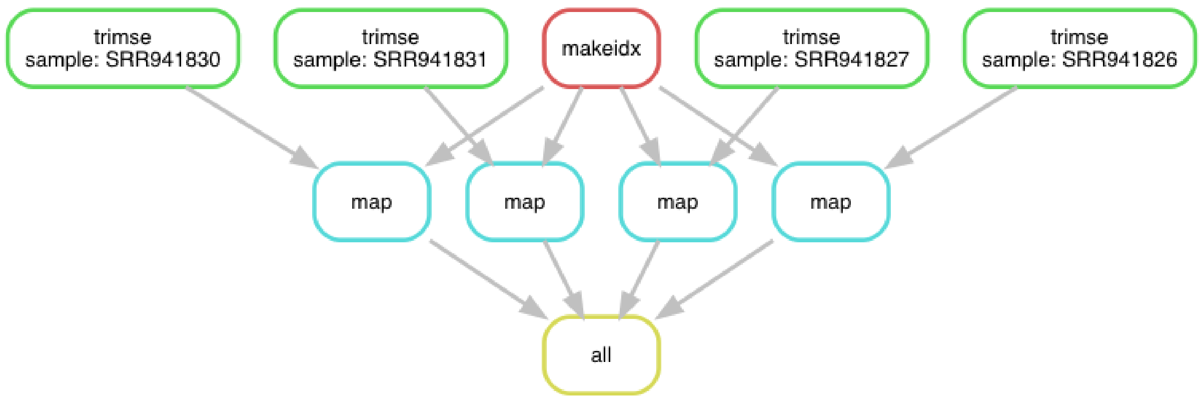


Fig. 1.7: The workflow v6 as a directed acyclic graph.

Listing 1.9: File: Snakefile_v7

```
SAMPLES, = glob_wildcards("fastq/{sample}.fastq.gz")

rule all:
    input:
        "analyses/results/counts.txt"

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
    benchmark:
        "analyses/benchmarks/{sample}.trimse"
    conda:
        "envs/sickle.yaml"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
        " 2> {log}"

rule makeidx:
    input:
        fasta = "data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa"
    output:
        touch("data/makeidx.done")
    log:
        "analyses/logs/makeidx.log"
    benchmark:
        "analyses/benchmarks/makeidx.txt"
    conda:
        "envs/map.yaml"
    shell:
        "bwa index {input.fasta} 2> {log}"

rule map:
    input:
        reads = "analyses/results/{sample}.trimmed.fastq.gz",
        idxdone = "data/makeidx.done"
    output:
```

(continues on next page)

(continued from previous page)

```
43            "analyses/results/{sample}.bam"
44        log:
45            "analyses/logs/{sample}.map"
46        benchmark:
47            "analyses/benchmarks/{sample}.map"
48        threads: 8
49        conda:
50            "envs/map.yaml"
51        params:
52            idx = "data/Saccharomyces_cerevisiae.R64-1-1.dna_sm.toplevel.fa"
53        shell:
54            "bwa mem -t {threads} {params.idx} {input.reads} | "
55            "samtools view -Sbh > {output} 2> {log}"
56
57    rule featurecount:
58        input:
59            gtf = "data/Saccharomyces_cerevisiae.R64-1-1.92.gtf.gz",
60            bams = expand("analyses/results/{sample}.bam", sample=SAMPLES)
61        output:
62            "analyses/results/counts.txt"
63        log:
64            "analyses/logs/featurecount.log"
65        benchmark:
66            "analyses/benchmarks/featurecount.txt"
67        conda:
68            "envs/subread.yaml"
69        threads: 4
70        shell:
71            "featureCounts -T {threads} -t exon -g gene_id -a {input.gtf} -o {output} {input.bams}"
72            " 2> {log}"
```

1. We need to change the input of rule `all` again to specify our final target (*line 5*).
2. On *line 60* we are specifying all `bam` files as input, as `featureCounts` can be run on all samples at the same time to produce one table of counts for all samples.

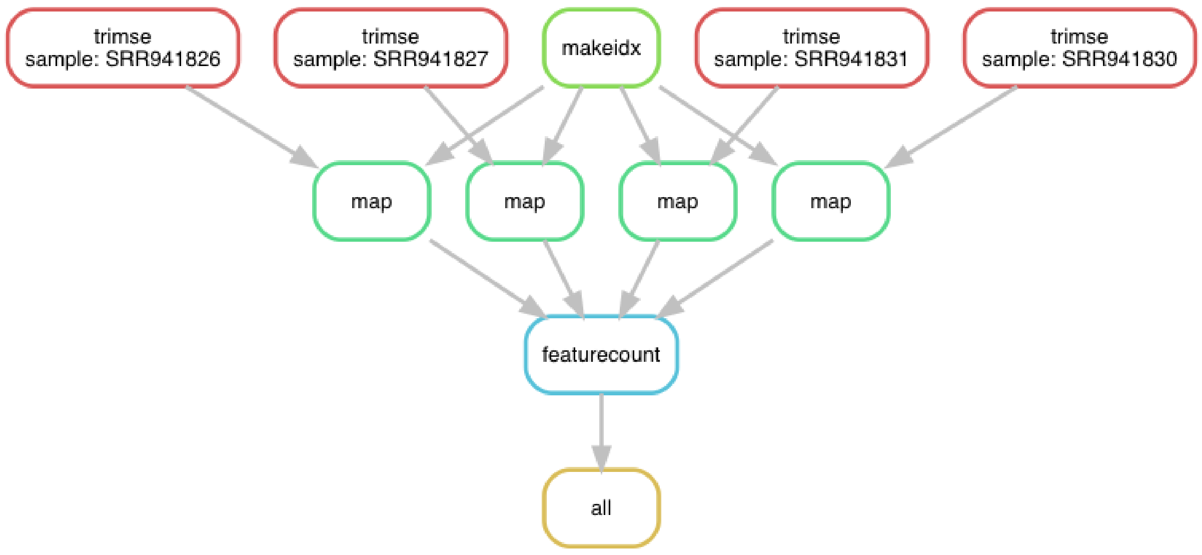


Fig. 1.8: The workflow v7 as a directed acyclic graph.

This looks very similar to our original workflow cartoon from the beginning.

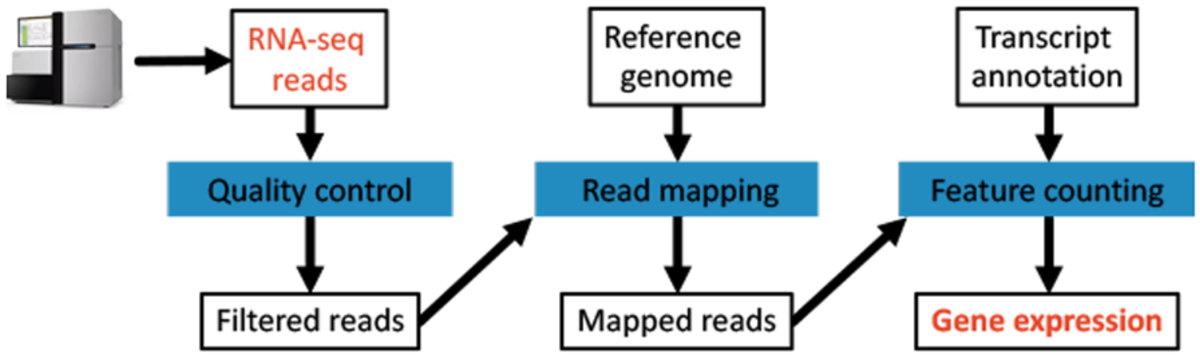


Fig. 1.9: The original workflow.

1.4.6 Making your work available

We are left with one requirement that we wanted to address in this tutorial:

Publishing & versioning the workflow information, as to keep track of when workflows change and what changes occurred.

After we created our workflow and our tool specifications in form of `yaml` files, we can make sure that others are able to easily get to our workflow specifications. The easiest way to facilitate this, is to put your directory under Git[108] version control and push your repository to a remote provider like GitLab[109] or GitHub[110]. The URL to this repository can be added to manuscripts or sent via emails to collaborators so that others can easily download the repository and thus redo the analysis with the same settings, hence we are fulfilling our last requirement above.

```
$ git init
$ git add Snakefile
$ git add envs/*
# commit your changes
$ git commit -m "Init"
# Create a remote on GitLab, GitHub, Bitbucket, etc.
# and associate remote to repository
$ git remote add origin git@gitlab.com/...
$ git push -u origin master
```

Note: It might be tempting to add the input data as well to the repository. However, in most cases it will be very big and you are better of using a remote storage location for the input data, e.g. Dropbox, Google Cloud Storage, FTP, etc. Snakemake can deal with all of those remote locations and can download data on demand. An example can be seen in the example file "Snakemake_v8" (see also below), where I request the input data from a Google Cloud Storage bucket[111]. *Of course there might be fees associated with storing data in the cloud*. For proper reproducibility it should be a persistent location. For genomic data, free storage solutions like NCBI Short Read Archive[112] or Gene Expression Ominbus[113] can be used.

Listing 1.10: File: Snakefile_v8

```
from snakemake.remote.GS import RemoteProvider as GSRemoteProvider
GS = GSRemoteProvider()

# Read samples from a Google Cloud Storage bucket.
```

(continues on next page)

[108] https://git-scm.com/
[109] https://gitlab.com/
[110] https://github.com/
[111] https://cloud.google.com/storage/
[112] https://www.ncbi.nlm.nih.gov/sra
[113] https://www.ncbi.nlm.nih.gov/geo/

(continued from previous page)

```
# Samples will be downloaded, processed locally and uploaded to google storage
# Call snakemake with the --default-remote-provider GS --default-remote-prefix BUCKET-PREFIX
SAMPLES, = GS.glob_wildcards("schmeier-reproduce-bucket/fastq/{sample}.fastq.gz")

rule all:
    input:
        expand("analyses/results/{sample}.trimmed.fastq.gz", sample=SAMPLES)
...
```

1.5 Containerization

1.5.1 What is containerization?

Until recently when people were talking about virtualization, one would think of VirtualBox[132] and full GUI accessible virtual machines (VM), e.g. running a Ubuntu[133] Linux VM on a Windows host machine. However, containerization takes this concept to another level. Generally, containerization in enterprise environments is used for micro-service virtualization, that is, to ship a specific application as a self-contained unit, including operating system.

Containers are useful to ensure application and environment compatibility on different computing resources. If you distribute applications or software through a container, whoever uses this container does not have to deal with installing missing dependencies and the like. The container contains all of it, while still being manageable in size.

The most famous container engine is Docker[134]. It is designed primarily for network micro-service virtualization and facilitates creating, maintaining and distributing container images that are relatively easy to install, well documented, and standardized. Docker containers are also kind of reproducible.

So why not use Docker? Using Docker on your local resources is great but unfortunately Docker containers are not designed for the use with traditional high performance computing.

Singularity[135] to the rescue. Singularity is a containers engine developed at the Berkeley Lab and designed for the needs of scientific workloads *[KURTZER2017a]* (page 36). Some points of note with regards to Singularity:

- Containers are stored in a single file.
- No system, architectural or workflow changes are necessary to integrate on HPC systems.
- Limits user's privileges (inside user = outside user).

[132] https://www.virtualbox.org
[133] https://www.ubuntu.com
[134] https://www.docker.com/
[135] http://singularity.lbl.gov/

- Unlike Docker, there is no need for a root owned container daemon.
- However, Singularity is able to run Docker images.
- Simple integration with resource managers, InfiniBand, GPUs, MPI, file systems, and supports multiple architectures.
- Security: When a container is launched by a user all programs inside the container run as that user. If you want to be root inside the container, you must first be root outside of the container

1.5.2 What does it accomplish for us?

Once you have installed Singularity, you can use container images with pre-configured operating systems and software installed. In our case of course software for analysing genomic data. You can build "relatively" easy your own container (see an example below "*Building your own Singularity container locally* (page 25)") but you can also use preconfigured containers at will, e.g. from Biocontainers[136]. In this way we can achieve another layer of reproducibility as we can package the tools we use in a container, that can be run like it is on different host systems.

1.5.3 Using a Singularity container

First, we need to have Singularity[137] installed on our system where we want to run the container (see installation instructions here[138]). Second, we need a container. Lets download a ready-made container containing BWA[139] from the Biocontainers registry[140] to see how this works:

```
$ singularity pull docker://biocontainers/bwa
```

Next, we can use BWA from within the container on files located on our host system, e.g.

```
$ singularity exec bwa.simg bwa mem -t 8 genomeindex sample.fastq > sample.sam
```

In this way BWA is compartmentalised and if someone else would use the same container, the version of BWA, as well as the underlying operating system that executes BWA, would be the same as in our analysis.

Note: Containers can of course change and be overwritten, however, in practise one could freeze a container that was used for an analysis so that it can be re-used like in the original study.

We will integrate a container into our Snakemake[141] workflow below, first however, we are going to see how to build our own container.

1.5.4 Building your own Singularity container locally

I am going to build a Singularity[142] container that contains the same tools and versions that we used in the last section to analyse our data.

Line 1 and 2 define the base container, here a docker container containing a Miniconda3 installation. In the `%environment` part, starting on *line 7*, we define some environment variables within the container, so that the conda command is exposed and ready to use within the container. Finally, in the `%post` section, we add conda[143] channels to make Bioconda[144] tools available to the container and finally install the same software and versions we used before. We clean out any remaining downloaded packages on *line 25*

136 http://biocontainers.pro/
137 http://singularity.lbl.gov/
138 http://singularity.lbl.gov/docs-installation
139 http://bio-bwa.sourceforge.net/
140 https://biocontainers.pro/registry/#/
141 http://snakemake.readthedocs.io/en/latest/
142 http://singularity.lbl.gov/
143 http://conda.pydata.org/miniconda.html
144 https://bioconda.github.io/

Listing 1.11: File: `Singularity`

```
Bootstrap: docker
From: continuumio/miniconda3:4.4.10

%labels
AUTHOR email@email.com

%environment
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# This sets global environment variables for anything run within the container
export PATH="/opt/conda/bin:/usr/local/bin:/usr/bin:/bin:"
unset CONDA_DEFAULT_ENV
export ANACONDA_HOME=/opt/conda

%post
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# This is going to be executed after the base container has been downloaded
export PATH=/opt/conda/bin:$PATH
conda config --add channels defaults
conda config --add channels conda-forge
conda config --add channels bioconda
conda install --yes bwa=0.7.15
conda install --yes sickle-trim=1.33
conda install --yes subread=1.6.1
conda install --yes samtools=1.8
conda clean --index-cache --tarballs --packages --yes
```

If we have Singularity installed locally on our machine, we can build the container named `biotools.simg` with the following command:

```
$ sudo singularity build biotools.simg Singularity
```

As can be seen in the command above, to build a new Singularity container, we need root privileges (hence sudo) on the machine we want to build the container on. Sometimes, however, this is not possible, e.g. on shared resources, like on cluster environments we do not normally have root privileges. This brings us to the next section "*Building a container on Singularity Hub* (page 26)".

1.5.5 Building a container on Singularity Hub

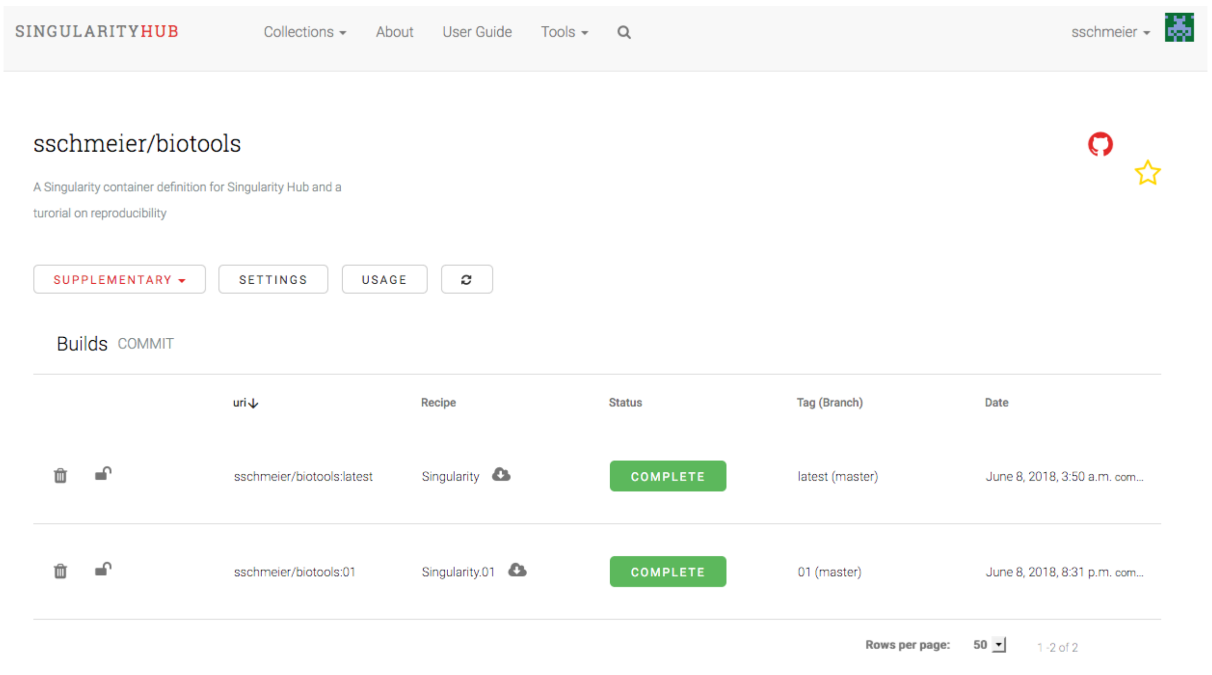
There is another method that we can use to integrate our container into our workflow. Instead of building the container locally, we can automatically build the container in the "cloud" on Singularity Hub[145]. In order to do this, we need to create a Git[146] repository (here sschmeier/biotools at https://github.com/sschmeier/biotools) and add the `Singularity` file to the repository and push the repository to GitHub[147].

```
$ git init
$ git add Singularity
$ git commit -m "Init."
$ git remote add origin git@github.com:sschmeier/biotools.git
$ git push -u origin master
```

Next, we need to create an account (for free) on Singularity Hub and connect our GitHub repository. Now, with every push of a changed `Singularity` file in our Git repository to GitHub, Singularity Hub will detect that a change occurred and (re)build the container for us (see Fig. 1.10).

Once, the container has been build and stored on Singularity Hub, it is readily available for us and others to be downloaded and used on a system that has Singularity installed, i.e. a Linux-based system.

145 https://www.singularity-hub.org
146 https://git-scm.com/
147 https://github.com/



sschmeier/biotools

A Singularity container definition for Singularity Hub and a turorial on reproducibility

Builds COMMIT

uri	Recipe	Status	Tag (Branch)	Date
sschmeier/biotools:latest	Singularity	COMPLETE	latest (master)	June 8, 2018, 3:50 a.m. com...
sschmeier/biotools:01	Singularity.01	COMPLETE	01 (master)	June 8, 2018, 8:31 p.m. com...

Rows per page: 50 1-2 of 2

Fig. 1.10: A screen shot form the Singularity Hub website.

```
$ singularity pull --name "biotools.simg" shub://sschmeier/biotools:latest
$ singularity exec biotools.simg samtools --version
samtools 1.8
Using htslib 1.8
Copyright (C) 2018 Genome Research Ltd.
```

Note: We can have different versions of our `Singularity` file, e.g. `Singularity.01`, or `Singularity.mickeymouse`, etc. Singularity Hub will build one container for each `Singularity` file. These can be later specifically "pulled" from the Hub. The original `Singularity` file will be automatically tagged as `latest`, hence `shub://sschmeier/biotools:latest` in the above shell command, in which "latest" could be ignored.

1.5.6 Using a container in our workflow

We need Singularity[148] installed on the system we are going to use the container, however, we do not need a privileged user to merely run Singularity. We will change a rule in our `Snakefile` file, so that instead of using the local conda[149] environment, Snakemake[150] is going to pull our Singularity container from Singularity Hub[151] and uses it to run the trimming step. The container is stored locally in your current working directory within the `.snakemake` directory, so that it does not have to be pulled again as long as it did not change. In the example below only the yellow part changed from the conda version to the Singularity container version.

Listing 1.12: One rule of a Snakefile

```
rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
```

(continues on next page)

[148] http://singularity.lbl.gov/
[149] http://conda.pydata.org/miniconda.html
[150] http://snakemake.readthedocs.io/en/latest/
[151] https://www.singularity-hub.org

(continued from previous page)

```
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
    benchmark:
        "analyses/benchmarks/{sample}.trimse"
    #conda:
    #    "envs/sickle.yaml"
    singularity:
        "shub://sschmeier/biotools:latest"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
        " 2> {log}"
```

Execute the workflow from the command-line with:

```
$ snakemake -p --use-singularity
```

Note: By using both flags `--use-singularity` **and** `--use-conda` we can even mix the execution, some rules that define a container will be run with the container, while others that still are defining a conda environment will be run with the local conda environment. However, in practice this is not done most of the times (see below).

Attention: Singularity[152] by default only has access to the folders under your home directory. To be able to use files that are located somewhere else in you system, you need to supply singularity explicitly with that location. This can be facilitated in Snakemake[153] by adding the following argument: `--singularity-args "--bind /folder/to/allow/access"`.

1.5.7 Using one container for the whole workflow

Instead of defining one container per rule, we could also define one container for the whole `Snakefile` and in this way have all rules run with the specified container:

Listing 1.13: Shown is part of a `Snakefile`

```
singularity: "shub://sschmeier/biotools:latest"

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
    benchmark:
        "analyses/benchmarks/{sample}.trimse"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
        " 2> {log}"

```

(continues on next page)

[152] http://singularity.lbl.gov/
[153] http://snakemake.readthedocs.io/en/latest/

(continued from previous page)

```
rule map:
    ...
```

1.5.8 Ready made containers

In any case, it needs to be pointed out that we do not necessarily need to build and use our own containers. Biocontainers[154] collects containers for lots of different purposes. However, to achieve best possible reproducibility your own containers on Singularity Hub[155] would be preferable as one has control over freezing container states and thus making them truly reproducible. In practise one could place the URL to the container (on Singularity Hub) in the Methods part of a manuscript. In this way it would be clear what versions of tools have been used for the analyses and everyone can pull and re-use the container.

1.5.9 Combining containers with conda-based package management

The former section outlined how to use one container for the whole workflow. This can be combined with conda[156]-based package management to make use of both containerization and rule-based package management. This is really powerful. We can define a global conda[157] docker container and still use per rule yaml files for package definitions. In this way, Snakemake[158] will first enter the container and create conda[159] environments based on the yaml-file definitions. Upon execution of a rule, Snakemake[160] will first enter the container and activate the environment subsequently before the rule is executed. This process gives us a bit more flexibility in terms of package management, with the added benefit of not having to create several Singularity[161] images. An example definition is shown below:

Listing 1.14: Shown is part of a Snakefile

```
singularity: "docker://continuumio/miniconda3:4.5.11"

rule trimse:
    input:
        "fastq/{sample}.fastq.gz"
    output:
        "analyses/results/{sample}.trimmed.fastq.gz"
    log:
        "analyses/logs/{sample}.trimse"
    benchmark:
        "analyses/benchmarks/{sample}.trimse"
    conda:
        "envs/sickle.yaml"
    params:
        qualtype="sanger"
    shell:
        "sickle se -g -t {params.qualtype} -f {input} -o {output}"
        " 2> {log}"

rule map:
    ...
```

1.5.10 Background reading on containers

- Singularity Containers for Science. Kurtzer GM. *[KURTZER2017a]* (page 36)
- Singularity: Scientific containers for mobility of compute. *[KURTZER2017b]* (page 36)

[154] http://biocontainers.pro/
[155] https://www.singularity-hub.org
[156] http://conda.pydata.org/miniconda.html
[157] http://conda.pydata.org/miniconda.html
[158] http://snakemake.readthedocs.io/en/latest/
[159] http://conda.pydata.org/miniconda.html
[160] http://snakemake.readthedocs.io/en/latest/
[161] http://singularity.lbl.gov/

1.6 Downloads

1.6.1 Tools

- Miniconda installer [EXTERNAL[181] | INTERNAL[182] | DROPBOX[183]]

1.6.2 Data

- The Data-set can be downloaded from GitLab using: `git clone https://gitlab.com/schmeierlab/reproduce-tutorial.git`.
- Alternatively, you can visit https://gitlab.com/schmeierlab/reproduce-tutorial and download a zipped version of the repository.

[181] https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
[182] http://compbio.massey.ac.nz/data/203341/Miniconda3-latest-Linux-x86_64.sh
[183] https://www.dropbox.com/s/tz2wocdzjr4grdy/Miniconda3-latest-Linux-x86_64.sh?dl=0

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[GRUENING2018] Grüning B, et al. Bioconda: sustainable and comprehensive software distribution for the life sciences. Nature Methods, 2018, 15:475–476.[18]

[KOESTER2012] Köster J and Rahmann S. Snakemake - A scalable bioinformatics workflow engine. Bioinformatics 2012, 10.1093/bioinformatics/bts480.[19]

[ASLANKOOHI2013] Aslankoohi E, et al. Dynamics of the Saccharomyces cerevisiae Transcriptome during Bread Dough Fermentation. Appl Environ Microbiol. 2013 Dec; 79(23): 7325–7333.[20]

[GRIFFITH2015] Griffith M, Walker JR, Spies NC, Ainscough BJ, Griffith OL. Informatics for RNA Sequencing: A Web Resource for Analysis on the Cloud. PLoS Comput Biol. 2015 Aug 6;11(8):e1004393. doi: 10.1371/journal.pcbi.1004393[21].

[COLLINS2014] Collins FS, Tabak LA. NIH plans to enhance reproducibility. Nature. 2014 Jan;505:612-613. doi: 10.1038/505612a[39]

[CASADEVALL2016] Casadevall A, Ellis LM, Davies EW, McFall-Ngai M, Fang FC. A Framework for Improving the Quality of Research in the Biological Sciences. MBio. 2016 Aug 30;7(4). pii: e01256-16. doi: 10.1128/mBio.01256-16[40].

[RAVEL2014] Ravel J, Wommack KE. All hail reproducibility in microbiome research. Microbiome. 2014 Mar 7;2(1):8. doi: 10.1186/2049-2618-2-8[41].

[GARIJO2013] Garijo D, Kinnings S, Xie L, Xie L, Zhang Y, Bourne PE, Gil Y. Quantifying reproducibility in computational biology: The case of the tuberculosis drugome. PLOS ONE. 2013 Nov;505:612-613. doi: 10.1371/journal.pone.0080278[42].

[NOBLE2009] Noble WS. A quick guide to organizing computational biology projects. PLoS Comput Biol. 2009 Jul;5(7):e1000424. doi: 10.1371/journal.pcbi.1000424[43].

[KANWAL2017] Kanwal S, Zaib F, Lonie A, Sinnott RO. Investigating reproducibility and tracking provenance – A genomic workflow case study. BMC Bioinformatics, 2017, 18:337, doi: 10.1186/s12859-017-1747-0.[44]

[LEIPZIG2017] Leipzig J. A review of bioinformatic pipeline frameworks. Briefings in Bioinformatics, Volume 18, Issue 3, 1 May 2017, Pages 530–536,[114]

[JOSHI2011] Joshi NA, Fass JN. Sickle: A sliding-window, adaptive, quality-based trimming tool for FastQ files (Version 1.33) [Software]. (2011) Available at https://github.com/najoshi/sickle.

[18] http://dx.doi.org/10.1038/s41592-018-0046-7
[19] https://doi.org/10.1093/bioinformatics/bts480
[20] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3837736/
[21] http://doi.org/10.1371/journal.pcbi.1004393
[39] http://doi.org/10.1038/505612a
[40] http://doi.org/10.1128/mBio.01256-16
[41] http://doi.org/10.1186/2049-2618-2-8
[42] http://doi.org/10.1371/journal.pone.0080278
[43] http://doi.org/10.1371/journal.pcbi.1000424
[44] https://doi.org/10.1186/s12859-017-1747-0
[114] https://doi.org/10.1093/bib/bbw020

[KURTZER2017a] Kurtzer GM. Singularity Containers for Science. Presentation [PDF][162]

[KURTZER2017b] Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PLoS ONE 12(5): e0177459. https://doi.org/10.1371/journal.pone.0177459[163]

[162] http://www.hpcadvisorycouncil.com/events/2017/stanford-workshop/pdf/GMKurtzer_Singularity_Keynote_Tuesday_02072017.pdf

[163] https://doi.org/10.1371/journal.pone.0177459