Organizing bioinformatic projects

Jakob Willforss

January 28, 2020

Contents

Chapter 1

Instructions

1.1 Acknowledgements

Jakob Willforss wrote this material with input from Line Kofod Møller and proofreading by Björn Canbäck and Deborah Oliveira. The ideas presented in it have been developed during extensive discussions with several people including Johan Bentzer, Dag Ahren and Björn Canbäck.

1.2 Aim

These instructions are designed to be used together with any of the other exercises, and is to a large extent based on the article "A Quick Guide to Organizing Computational Biology Projects".

The main goals are:

- Store files in clear hierarchical structure.
- Document the environment (programs with versions).
- Document exact processing steps so that it can be rerun.
- Describe the reasoning about each step with comments.
- Making the analysis fully reproducible using Snakemake.

Here we describe one approach. You will likely adjust it to fit your needs better - it is meant to act as a starting point.

1.3 Organizing your files

1.3.1 A proposed setup

A clear folder structure is an important step towards getting any bioinformatic project in order. The entire analysis should be contained in a single directory. Within that, you will have a set of clearly defined subfolders. One example could be:

doc	General documentation related to the analysis (such as this document).
data	Raw starting data.
src	Scripts developed during the analysis. During analysis, execute them directly from this folder.
bin	Any external programs you retrieve and use, which is not preinstalled.
results	Files generated from your processing.

The bulk of the analysis will be contained in a script named `Snakefile` residing in the top directory (more on this in the section about Snakemake). Other documentation will be contained in a file called `README.md`.

1.3.2 The results folder

You will be generating a number of files into this folder. *W. S. Noble.* recommends that you organize the results based on which date the analysis was performed. Here, we would recommend that you create a subfolder for each main processing step. They could potentially be numbered to maintain the logical order of the analysis.

One example of subfolders to the `results` directory (if you are working on the amplicon sequencing exercise) could be:

- 0_preparedata
- 1_qualitycheck
- 2_trimming
- 3_mergereads
- 4_precluster
- 5_chimera

- 6_clustering
- 7_classify
- 8_visualize

The reasons for prefixing each folder with a number and an underscore is twofold: They will appear in the same order when listed, and they can quickly be accessed using tab completion. But as long as your structure is well thought through any approach here is fine.

1.4 Tracing your steps

When performing analysis in the wet lab, it is standard practice to note all taken steps in a lab notebook. The same should be true for computer work. You need to keep track of the exact commands and programs used. This helps both you and others reproduce the analysis, and is a great help when you, later on, want to update or do a similar analysis.

Here, two files will be used for documentation:

- A Snakemake workflow file called `Snakefile`. It will contain the bulk of the analysis.
- A README file called `README.md`. It will contain general information about your analysis.

Sections in the `README.md` file:

- **Project overview** Outline the general purpose and background information for the analysis. What is the biological question, where do the data come from, potential issues encountered in the experimental parts.
- **People** Describe who is involved in the project and the responsibility of each person. Bioinformatics projects usually involve several people with different responsibilities (experimental, interpretation, data analysis).
- **Software versions** Document versions of all software you use. **This is critical for reproducibility.**
- **Analysis** Document *all* steps you take to go from raw data to final results which aren't present in the `Snakefile`. Also give an overview of the different parts of your analysis - how it works conceptually, and any issues you have encountered.

You can explore commands to find out what works best without writing down all exploration, but make sure to document the final sequence of commands you decide to use either by including them in the Snakefile if it is part of the 'core analysis' or else in the README.

Avoid doing changes by hand. It makes things harder to reproduce and document. If you must, make sure to note what you did. For instance, if using graphical software, you could take a screenshot of the parameters to include in your README.

1.5 Task description

The idea is for you to get hands-on experience with performing your analysis in an ordered and reproducible manner. The first time you do this, it will take extra work, but it pays back big time in the long run. We want to emphasize that this is one way to do things. Later on, you will likely want to tweak, extend or replace part of this setup with something that works better for you. You will develop your style and opinions on how to do things.

- Keep track of your files in a folder structure as specified above.
- Document exact versions for environment and programs.
- Make sure every analysis step from raw data to final results is documented. Keep as much as possible of your analysis within the Snakefile so that it can easily be reproduced.
- Document the purpose of each logical step so that you or someone else could follow the reasoning behind the analysis. You don't need to describe why you perform commands such as `head -10 seq.fa > sub_seq.fa`, rather describe overall concepts.

At the end of this exercise, share your documentation with a friend. Review each other's analyses on the criteria:

- Is the analysis easy to follow and understand?
- Did you find any errors in the analysis?
- Would it be possible to redo the complete analysis based on the documentation?
- Could the analysis be easily rerun?

1.6 Crash course in Snakemake

Snakemake is a software for making analyses easily reproducible. It keeps track of what output is expected from each analysis step and will by default know what parts of the analysis it needs to rerun. Furthermore, it makes it easy to run commands in parallel, which can significantly speed up the analysis. It is based on Python and one of the most popular tools for making reproducible analyses.

1.6.1 A gentle introduction to Snakemake

Here, we will step through the basics of the Snakemake workflow with the goal of giving you enough foundation to start applying Snakemake in your own analyses. There are many things you can do in Snakemake, and we can only cover a few of them here. For further input, this FAQ is a great place to start looking:
https://snakemake.readthedocs.io/en/stable/project_info/faq.html

Setting up things

In order to run this exercise you need to create three FASTA files called "A.fasta", "B.fasta" and "C.fasta", each containing at least two entries in single-line format, and place them within a folder called `data`. They could for example contain:

```
>entry1
ATCGATCGATGCATCG
>entry2
AGCTAGCTAGCTA
```

Outside this directory, you put the snakemake file itself called `Snakefile`. This file will contain your Snakemake pipeline.

Now your folder structure should be:

```
my_analysis/
    Snakefile
    data/
        A.fasta
        B.fasta
        C.fasta
```

Now (assuming that your working directory is set to `my_analysis`) this can be executed as follows:

```
$ snakemake
```

Running a first command in Snakemake

We start by writing a minimal rule into our `Snakefile` document extracting the first two lines from a FASTA file (i.e. the first ID and the first sequence). For now, we make it specific for the file `A.fasta`.

```
rule get_first_entry:
    input:
        in_fasta="data/A.fasta"
    output:
        out_fasta="output/A.first.fasta"
    shell:
        """
        head -2 {input.in_fasta} > {output.out_fasta}
        """
```

Some things to note:

input You can specify one or several inputs (separated by a comma). These can be accessed from the script part using curly braces `{input.in_fasta}`. If having only a single input, it can also be accessed using `{input}`.

output Same as `input` - one or more can be specified, and they are used in the same way within the script part - here as `{output.out_fasta}`.

shell There are multiple ways to execute code from within Snakemake. Here, one way to execute shell scripts is used. Within the docstrings `"""` one or multiple commands can be placed. These can be bash code, or could, for instance, call to a Python script you have written yourself.

Now try executing it. If run successfully you should obtain something similar to the following output:

```
Building DAG of jobs...
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       get_first_entry
        1

rule get_first_entry:
    input: data/A.fasta
    output: output/A.first.fasta
    jobid: 0

Finished job 0.
1 of 1 steps (100) done
```

If you now subsequently try to execute the command again after successfully running it once you will obtain a different output.

```
$ snakemake
Building DAG of jobs...
Nothing to be done.
```

The reason is that snakemake can detect that the wanted output file already is generated - there is a file called `output/A.first.fasta`. Snakemake is lazy in this way - it will only rerun what is needed to get the output asked for.

Linking multiple commands - rule all

Now we will run a sequence of two commands - a mini pipeline. To do this, we will need to add two things. Snakemake files generally include a special rule at the top called `rule all` where you tell Snakemake what final output files you want to obtain. Based on the output you specify here it then figures out what other rules need to be run in order to get this, and then executes them.

Furthermore, we need a second command. This time the input of that command will be the output of our existing command. The script in its entirety will look in the following way:

```
rule all:
    input: "output/A.first.fasta.seq"

rule get_first_entry:
    input:
        in_fasta="data/A.fasta"
    output:
        out_fasta="output/A.first.fasta"
    shell:
         """
         cp {input.in_fasta} {output.out_fasta}
         """

rule get_sequence:
    input:
        in_fasta="output/A.first.fasta"
    output:
        out_fasta="output/A.first.fasta.seq"
    shell:
         """
         tail -1 {input.in_fasta} > {output.out_fasta}
         """
```

The sequence of events when executing this script will be:

1. Ask the `rule all` - which files do I need to generate?
   Answer: `output/A.first.fasta.seq`
2. Are there any commands that can prove this? Yes, `rule get_sequence`.
3. Is the input to this rule `output/A.first.fasta` available on disk? No, then are there any rules that can generate it? Yes, `rule get_first_entry`. So, we need to execute that one first.
4. Is the input file `data/A.fasta` available for this command? Yes, perfect, then we are all set.
5. Processing starts, running first `rule get_first_entry` and next `rule get_sequence`

Exercises:

1. After running the mini pipeline, try rerunning it. Next, remove the file `output/A.first.fasta` and try running it another time. What happens?

2. If you remove the output and add the flag `--dryrun`, what information gets printed to the terminal? What output is produced? This is a very useful command to verify that your Snakemake workflow is correctly defined before trying to run the often time-consuming steps.

Running Snakemake for multiple input files

Snakemake starts to shine when you run it for multiple input files. Now let's execute the same steps also for two more files called `B.fasta` and `C.fasta`. These should be placed in the same directory as `A.fasta`.

To make Snakemake run multiple input files, we specify a list with the base names specific for each input file. Beyond this, the names of the input files are expected to be identical. Remember that Snakemake is built in Python, so we can use the Python data structures that we are used to.

The `expand` statement tells Snakemake that the names in the list `sample_names` should be inserted in the output path as specified by the `{sample}` pattern.

```
sample_names = ['A', 'B', 'C']

rule all:
    input: expand("output/{sample}.first.fasta.seq",
       sample=sample_names)
```

Furthermore, the `{sample}` pattern should be used in all rules we plan to execute to all of these input files. After making this change, your script should look as follows:

```
sample_names = ['A', 'B', 'C']

rule all:
    input: expand("output/{sample}.fasta.out.tail",
       sample=sample_names)

rule get_first_entry:
    input:
        in_fasta="data/{sample}.fasta"
    output:
        out_fasta="output/{sample}.first.fasta"
    shell:
        """
        cp {input.in_fasta} {output.out_fasta}
        """

rule get_sequence:
    input:
        in_fasta="output/{sample}.first.fasta"
    output:
        out_fasta="output/{sample}.first.fasta.seq"
    shell:
        """
        tail -1 {input.in_fasta} > {output.out_fasta}
        """
```

Exercises:

1. See what happens if you remove one file in the output and run the workflow again.

2. What happens if you change the list to `sample_names = ['A', 'C']`? Or to `sample_names = ['A', 'B', 'C', 'D']`?

A more complex example

Now you should be ready to take a look at a real-word bioinformatics pipeline.

https://snakemake.readthedocs.io/en/stable/tutorial/basics.html

Some points to inspect closer (looking at the final script found at the bottom of the page):

- For two rules (`bwa_map` and `bcftools_call`) they use multiple input files, but the arguments are only named in the latter case. Can you follow the inputs that are used here?

- Note that the final rule is executed using the command `script` rather than `shell`. What is the difference? When could this be useful?
- In the rule `samtools_sort` they use the syntax `wildcards.sample`. This allows accessing the individual names (in this case "A" or "B") used when generating the input- and output- paths. This is often very useful.
- Note that in this case the `expand` command is not used within the `rule all`. The reason for this is that within the `bcftools_call` they want to use all the files as input to a *single rule*, giving only one output file.

1.6.2 Snakemake exercise

The goal of this exercise is to have a miniature version of a Snakemake workflow that you could use for a full analysis. Start by preparing three small FASTA files called `s1.fa`, `s2.fa` and `s3.fa`. These are your raw data.

- Start with writing out the `rule all` at the top. Here you specify the expected output for your workflow.
- For a single FASTA file, make a single rule which reads the files and writes the headers and the sequence of the input file to two separate files.
- Extend the workflow so that it can process all three files at once.
- Finally, make a rule merging the headers from the three FASTA files into a single file. Tip: the files can be merged with a simple `cat` command, the trick is to figure out how to get all of the files into a single rule. An example of this was shown in the "A more complex example" section.

1.7 (Optional) Further reading

1.7.1 Other documentation tools

There are also other, more advanced tools available. Some are geared towards data analysis rather than the processing we are focusing on here (Jupyter, KnitR) while others are workflow languages designed precisely to set up analysis pipelines.

Workflow engines

These are languages designed for setting up reusable pipelines. Compared to pure Bash, it keeps track of generated files allowing it to do partial analyses and can easily be generalized to different datasets.

Snakemake (`http://snakemake.readthedocs.io/en/latest`) One of the most popular workflow languages.

Nextflow (`https://www.nextflow.io`) Another workflow language. Less established than Snakemake, but with built-in support for running the analyses in so-called containers - Clearly defined environments with particular software versions which can be distributed and run on other platforms.

Lab notebook software

These are built for keeping track of performed analysis steps. Both Jupyter and KnitR allows generating a final document containing both documentation, code, output and generated figures. This final document can easily be exported to PDF or HTML and used as a reference later on. Both also allows interactive analysis where part of the code can be executed at a time.

Jupyter (`http://jupyter.org/`) is a notebook software rapidly gaining popularity. It was originally developed for Julia, Python and R (hence the name Ju-pyt-(e)-r) but has support for a number of languages including Bash. It allows interactive analysis where the code is grouped into cells and where you directly can see its output.

KnitR (`https://yihui.name/knitr/`) allows you to write your document in markdown where the code is separated from comments. The document is executed and generates a PDF where figures can be generated directly into the document.

1.7.2 Version control

A version control system lets you keep track of each version your code and documentation have gone through.

It is in particular useful for when you are developing scripts. Often you use a specific version of a script, which later is updated further. If your script is in version control, you can always return to the exact version you used at a later point, instead of making copies of the previous versions.

The currently most popular version control system is Git (https://git-scm.com/). Other less popular systems are SVN (https://tortoisesvn.net/) and Mercury (https://www.mercurial-scm.org/). Git, together with GitHub for sharing the code is the current standard.

If you want to get some hands-on practice with Git, we recommend you take a look at the following online tutorial:

https://www.codecademy.com/learn/learn-git

You can also take a look at the course materials written by me:

http://ponderomatics.com/git.html

1.7.3 Managing dependencies with containers

It is essential to track the environment of the analysis. If you aren't running the analysis with the same version of the program, there is no guarantee that the result is the same. Documenting the environment is a minimum. But there are tools available to simplify the dependency management which can be used both to guarantee that the same versions are used and simplify the setup for later analysis both for you and others.

The most popular tool providing so-called containers is Docker (https://www.docker.com/). Another alternative is Singularity (http://singularity.lbl.gov/) which is a newcomer but with the benefit that it is less of a hassle to run on a Linux server environment.

1.8 Concluding words

All of these tools are useful and powerful in many contexts. Why aren't they all used by everyone? One aspect is that each new tool has a learning curve. It is not always that the benefit from it outweighs the extra effort it requires to use it. Many of them pay off in the long run.

In our opinion, a good strategy is always to be open to new strategies and continuously try learning new approaches. Of course, you need to prioritize here. It is not always learning a new tool pays of. Discuss with other bioinformaticians and try forming a picture of what would benefit you the most to learn next.

Also, we really think the strategies outlined in the initial part of this document is crucial. A clear folder structure, enough documentation to make your analysis reproducible and preferably the ability to execute it in one step. This is the baseline. After that, put in the effort to learn a version control system. It takes some effort to get started, but it will pay you back in plenty.

Good luck with your future analyses!