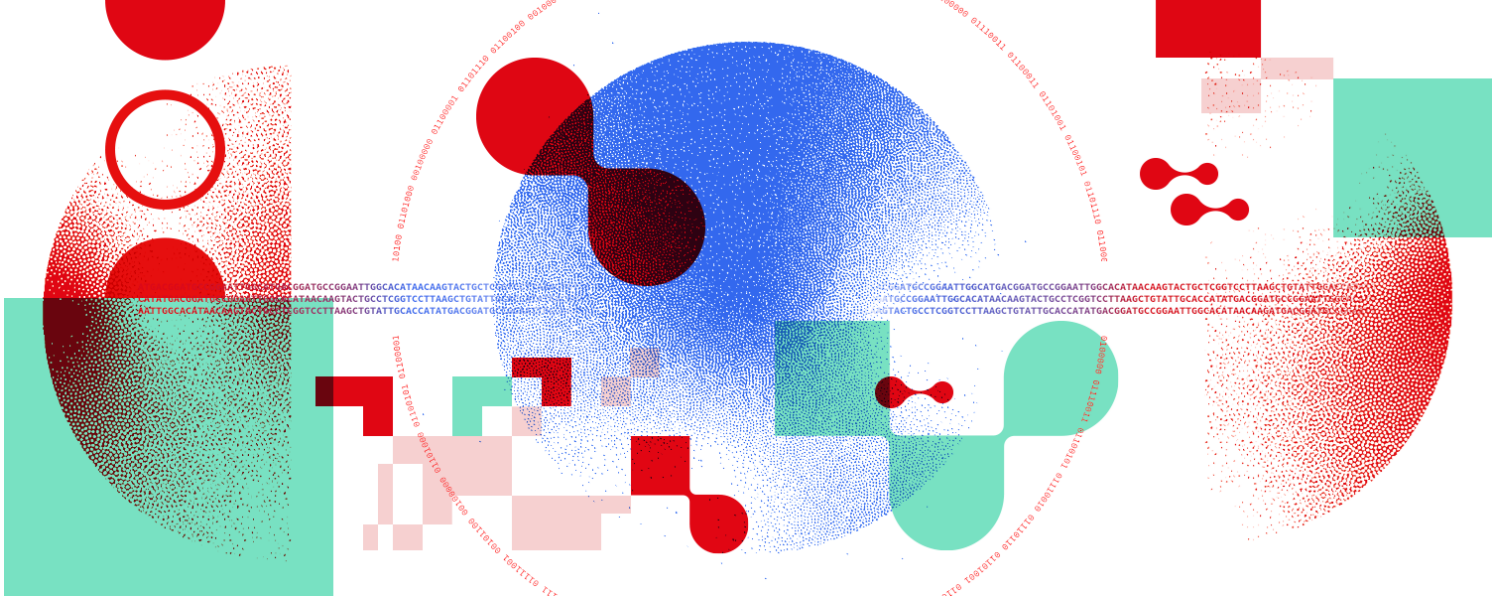

Snakemake for reproducible research

Decorating and optimising a Snakemake workflow

Centre hospitalier universitaire vaudois

Antonin Thiébaut
antonin.thiebaut@chuv.ch

SIB Swiss Institute of Bioinformatics

What could we improve? (again)

- Optimising resource usage
- Avoiding hard-coded parameters
- Processing list of files
- (Using non-conventional outputs)

What could we improve? (again)

- Optimising resource usage → Directives resources and threads
- Avoiding hard-coded parameters → config file
- Processing list of files → expand() syntax
- (Using non-conventional outputs) → (temp(), directory()…)

Optimising resource usage: threads

- ‘threads’ is a directive; its value is the number of threads to allocate to each job spawned by a rule
  - New kind of directive value: numeric (integer)
  - **Check whether software can actually multithread!**

Optimising resource usage: threads

- ‘threads’ is a directive; its value is the number of threads to allocate to each job spawned by a rule
  - New kind of directive value: numeric (integer)
  - **Check whether software can actually multithread!**

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    threads: 4
    shell:
        'command --threads {threads} {input} > {output}'
```

  - In local mode, total number of threads allocated to Snakemake is constrained by the execution parameter ‘--cores’

Optimising resource usage: memory and runtime

- ‘resources’ is a directive; its values aim to set the resources available for a job
  - New kind of directive value: pair of <key>=<value>

Optimising resource usage: memory and runtime

- ‘resources’ is a directive; its values aim to set the resources available for a job
  - New kind of directive value: pair of <key>=<value>
- mem_<unit>
  - Amount of memory needed by the job
  - <unit>: mb, gb, tb…
- runtime_<unit>
  - Amount of wall clock time a job needs to run
  - <unit>: s, m, h, d…

```
rule example:
    input:
        'data/first_step.tsv'
    output:
        'results/first_step.txt'
    resources:
        mem_gb=1,
        runtime_s=3600
    shell:
        'command {input} > {output}'
```

Avoiding hard-coded parameters: config file

- Snakemake can use configuration files to render workflows more flexible
  - Change config instead of code!

Avoiding hard-coded parameters: config file

- Snakemake can use configuration files to render workflows more flexible
  - Change config instead of code!

- Import file with configfile keyword
  - configfile: ‘path/to/config.yaml’ (relative to working directory)

- Accessed via global variable config
  - Imported as a Python dictionary (use keys to access values): config[‘sample’]

Avoiding hard-coded parameters: config file

- Snakemake can use configuration files to render workflows more flexible
  - Change config instead of code!
- Import file with configfile keyword
  - configfile: ‘path/to/config.yaml’ (relative to working directory)
- Accessed via global variable config
  - Imported as a Python dictionary (use keys to access values): config[‘sample’]
- 2 possible formats: JSON and YAML
  - Personal take: YAML is easier to write, understand and can be commented

```yaml
lines_number: 5  # Single value
samples:  # Multiple values
   - sample1
   - sample2
resources:  # Nested parameters
   threads: 4
   memory: 4G
```

YAML

```json
{
  "lines_number": 5,
  "samples": [
    "sample1",
    "sample2"
  ],
  "resources": {
    "threads": 4,
    "memory": "4G"
  }
}
```

JSON

Config file?

- Question 5

What should appear in a config file?

- Ideally, everything that should not be hard-coded:
  - File locations
  - Sample names and associated information
  - Rule computing resources
  - Etc…

What should appear in a config file?

- Ideally, everything that should not be hard-coded:
  - File locations
  - Sample names and associated information
  - Rule computing resources
  - Etc…

- But it is preferable to use paths to other smaller config files
  - Same as Snakefile and snakefiles
  - Example:
    - Table containing the sample names and information: config/samples_info.tsv
      - Tab-separated format is easy to write, read and parse
    - In the config file: samples: 'config/samples_info.tsv'
    - Add a function in a Snakefile to parse the table

What should **NOT** appear in a config file?

- Credentials: access tokens, passwords…

➔ Use environment variables (envvars)

Processing list of files: the expand syntax

- expand(): Snakemake function to automatically expand a wildcard expression to several wildcard values
  - Useful to define multiple inputs or outputs with a common pattern

Processing list of files: the expand syntax

- expand(): Snakemake function to automatically expand a wildcard expression to several wildcard values
  - Useful to define multiple inputs or outputs with a common pattern
  - Syntax: expand('{wildcard_name}', wildcard_name=<values>)
    - <values>: iterable (*i.e.* list, tuple, set) containing the wildcard values

```
rule example:
    input:
        'data/A.tsv',
        'data/B.tsv',
        'data/C.tsv'
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

```
rule example:
    input:
        expand('data/{sample}.tsv', sample=['A','B','C'])
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

➢ The rule example uses all three input files to generate a single output file. expand() does not apply the rule separately to the three inputs!

Processing list of files: the expand syntax

- When there are several wildcards, expand() creates **all possible combinations**

Processing list of files: the expand syntax

- When there are several wildcards, expand() creates **all possible combinations**

```
samples=[‘A,‘B’]
replicates = [1, 2]

rule example:
    input:
        expand('data/{sample}_{replicate}.tsv', sample=samples, replicate=replicates)
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

➢ input = 'data/A_1.tsv data/A_2.tsv data/B_1.tsv data/B_2.tsv'

Processing list of files: the expand syntax

- The wildcards in expand() are **independent** from any other wildcard in the rule

Processing list of files: the expand syntax

- The wildcards in expand() are **independent** from any other wildcard in the rule

```
samples=[‘A,‘B’]
replicates = [1, 2]

rule example:
    input:
        expand('data/{sample}_{replicate}.tsv', sample=samples, replicate=replicates)
    output:
        'results/{sample}.tsv'
    shell:
        'cat {input} > {output}'
```

➢ In this case, the value of the {sample} wildcard will NOT be propagated to the input

Using non-conventional outputs

- Snakemake has built-in utilities to assign properties to ‘special’

Property	Syntax	Function
Temporary	temp('path/to/file.txt')	File is deleted as soon as it is not required by any future jobs
Protected	protected('path/to/file.txt')	File cannot be overwritten after the job ends (useful to prevent erasing a file by mistake, for example files requiring heavy computation)
Ancient	ancient('path/to/file.txt')	Ignore file timestamp and assume file is older than any outputs: file will not be re-created when re-running the workflow, except when --force options are used
Directory	directory('path/to/directory')	Output is a directory instead of a file (use ‘touch’ instead if possible)
Touch	touch('path/to/file.txt')	Create an empty flag file ‘file.txt’ regardless of the shell command (if the command finished without errors)

Exercises

- Through the day:
  - Develop a simple RNAseq analysis workflow, from reads (fastq files) to Differentially Expressed Genes (DEG)

- For this session:
  - Optimise resource usage
  - Use a config file
  - Process list of inputs
  - Modularise a workflow
  - Aggregate outputs
  - (Manage non-conventional outputs)