



Snakemake for reproducible research

Decorating and optimising a Snakemake workflow

Antonin Thiébaut & Rafael Riudavets Puig

antonin.thiebaut@chuv.ch

Rafael.RiudavetsPuig@empa.ch

What could we improve? (again)

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

What could we improve? (again)

- Avoiding hard-coded parameters → config file
- Processing list of files → expand() syntax
- Optimising resource usage → Threads directive
- (Using non-conventional outputs) → (temp(), directory()...)

What could we improve? (again)

- Avoiding hard-coded parameters → config file
- Processing list of files → `expand()` syntax
- Optimising resource usage → `Threads` directive
- (Using non-conventional `outputs`) → `(temp(), directory()...)`

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!
- Imported file with **configfile** keyword in Snakefile
 - `configfile: 'path/to/config.yaml'` (relative to working directory)
- 2 possible formats: JSON and YAML
 - Personal opinion: YAML is easier to write, understand and can be commented

```
{  
  "retries": 5,  
  "samples": [  
    "file1",  
    "file2"  
  ],  
  "resources": {  
    "threads": 8,  
    "memory": "500M"  
  }  
}
```

JSON

```
retries: 5 # Single value  
samples: # Multiple values  
- file1  
- file2  
  
resources: # Nested parameters  
  threads: 8  
  memory: 500M
```

YAML

Avoiding hard-coded parameters: config file

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

```
{  
  "retries": 5,  
  "samples": [  
    "file1",  
    "file2"  
  ],  
  "resources": {  
    "threads": 8,  
    "memory": "500M"  
  }  
}
```

JSON

```
retries: 5 # Single value  
samples: # Multiple values  
- file1  
- file2  
resources: # Nested parameters  
  threads: 8  
  memory: 500M
```

YAML

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**)

What could we improve? (again)

- Avoiding hard-coded parameters → config file
- Processing list of files → expand() syntax
- Optimising resource usage → Threads directive
- (Using non-conventional outputs) → (temp(), directory()...)

Processing list of files: the expand syntax

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

Processing list of files: the expand syntax

- `expand()`: Snakemake function to expand a wildcard expression to several 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 merge_files:
    input:
        'data/test_1.txt',
        'data/test_2.txt',
        'data/test_3.txt'
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

```
rule merge_files:
    input:
        expand('data/test_{file}.txt', file=[1, 2, 3])
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```

- The `rule merge_files` uses all three **input** files to generate a single **output** file
 - `expand()` does not apply the `rule` three times, once per **input**!

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**

```
files=['test_A','test_B']
nbs = [1, 2]

rule merge_files:
    input:
        expand('data/{file}_{nb}.tsv', file=files, nb=nbs)
    output:
        'results/total.tsv'
    shell:
        'cat {input} > {output}'
```



```
input:
    ['data/test_A_1.tsv', 'data/test_A_2.tsv',
     'data/test_B_1.tsv', 'data/test_B_2.tsv']
```


Processing list of files: the expand syntax

- The **wildcards** in `expand()` are **independent** from wildcards in the **rule**

Processing list of files: the expand syntax

- The **wildcards** in `expand()` are **independent** from wildcards in the **rule**

```
files=['test_A','test_B']
nbs = [1, 2]

rule merge_files:
    input:
        expand('data/{file}_{nb}.tsv', file=files, nb=nbs)
    output:
        'results/{file}.tsv'
    shell:
        'cat {input} > {output}'
```

➤ Here, {file} value will NOT be propagated to the **input**

What could we improve? (again)

- Avoiding hard-coded parameters → config file
- Processing list of files → expand() syntax
- Optimising resource usage → **Threads directive**
- (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 type of value: numeric (integer)
 - When executed locally, '--cores' controls the total number of threads allocated to Snakemake; **threads** is automatically decreased if it's lower than '--cores'
 - **Check whether software can actually multithread!**

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

What could we improve? (again)

- Avoiding hard-coded parameters → config file
- Processing list of files → expand() syntax
- Optimising resource usage → Threads directive
- (Using non-conventional outputs) → (temp(), directory()...)

Using non-conventional outputs

- Snakemake has built-in utilities to assign properties to 'special' outputs

Property	Syntax	Function
Temporary	<code>temp('path/to/file.txt')</code>	File is deleted as soon as it is not required by any future jobs
Protected	<code>protected('path/to/file.txt')</code>	File cannot be overwritten after the job ends (useful to prevent erasing a file by mistake, for example files requiring heavy computation)
Ancient	<code>ancient('path/to/file.txt')</code>	Ignore file timestamp and assume file is older than any outputs: file will not be re-created when re-running the workflow, except when <code>--force</code> parameters are used
Directory	<code>directory('path/to/directory')</code>	Output is a directory instead of a file (use 'touch' instead if possible)
Touch	<code>touch('path/to/file.txt')</code>	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:
 - Use a config file
 - Process list of inputs
 - Modularise a workflow
 - Aggregate outputs
 - (Optimise resource usage)
 - (Manage non-conventional outputs)

