Introduction to Nextflow

Akshai Parakkal Sreenivasan

Pipeline

What is a pipeline?

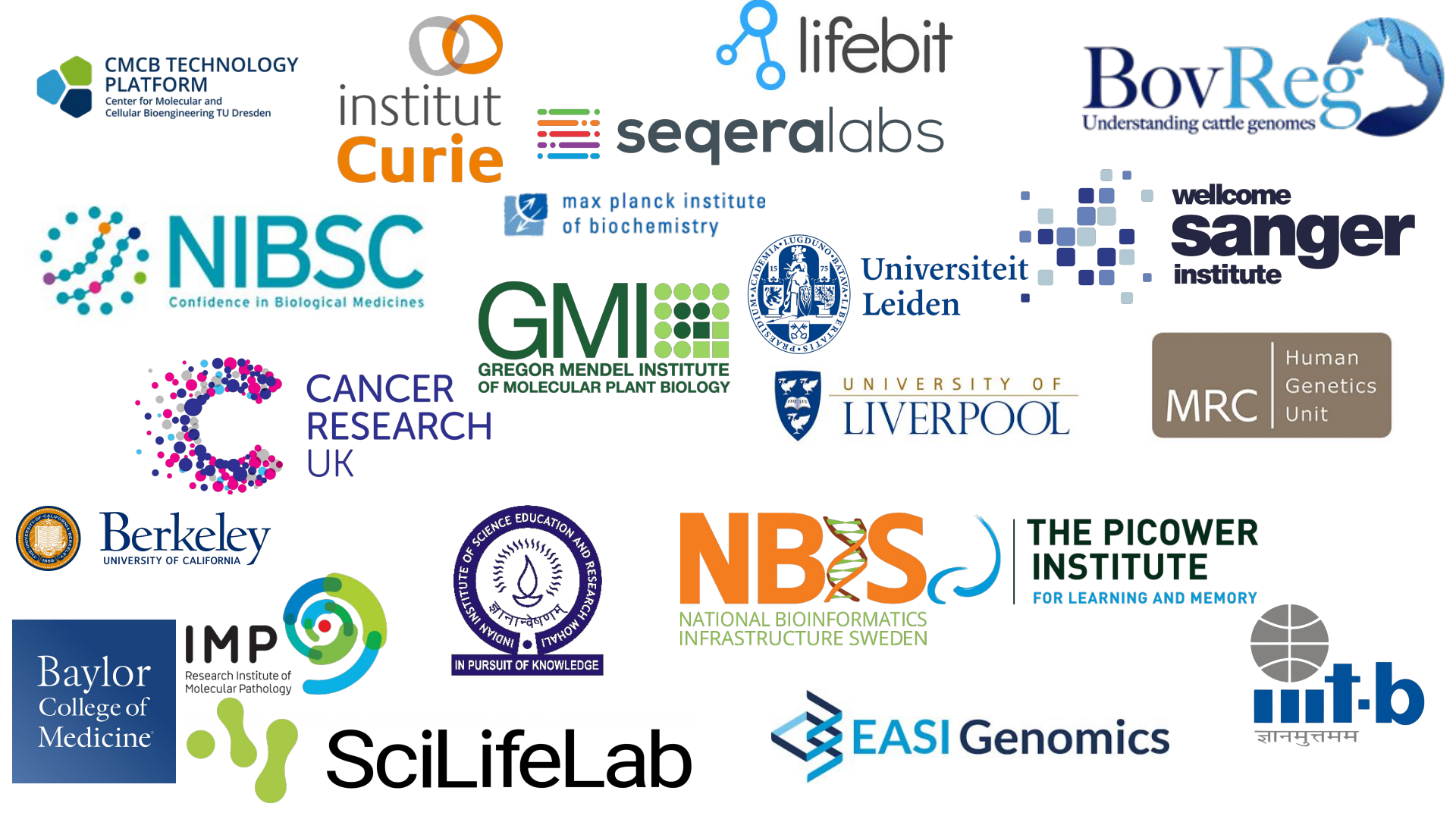
Desired benefits from a pipeline are reproducibility, avoiding manual tasks, time saving, scalable, easy to operate, error handling etc.

What is Nextflow?

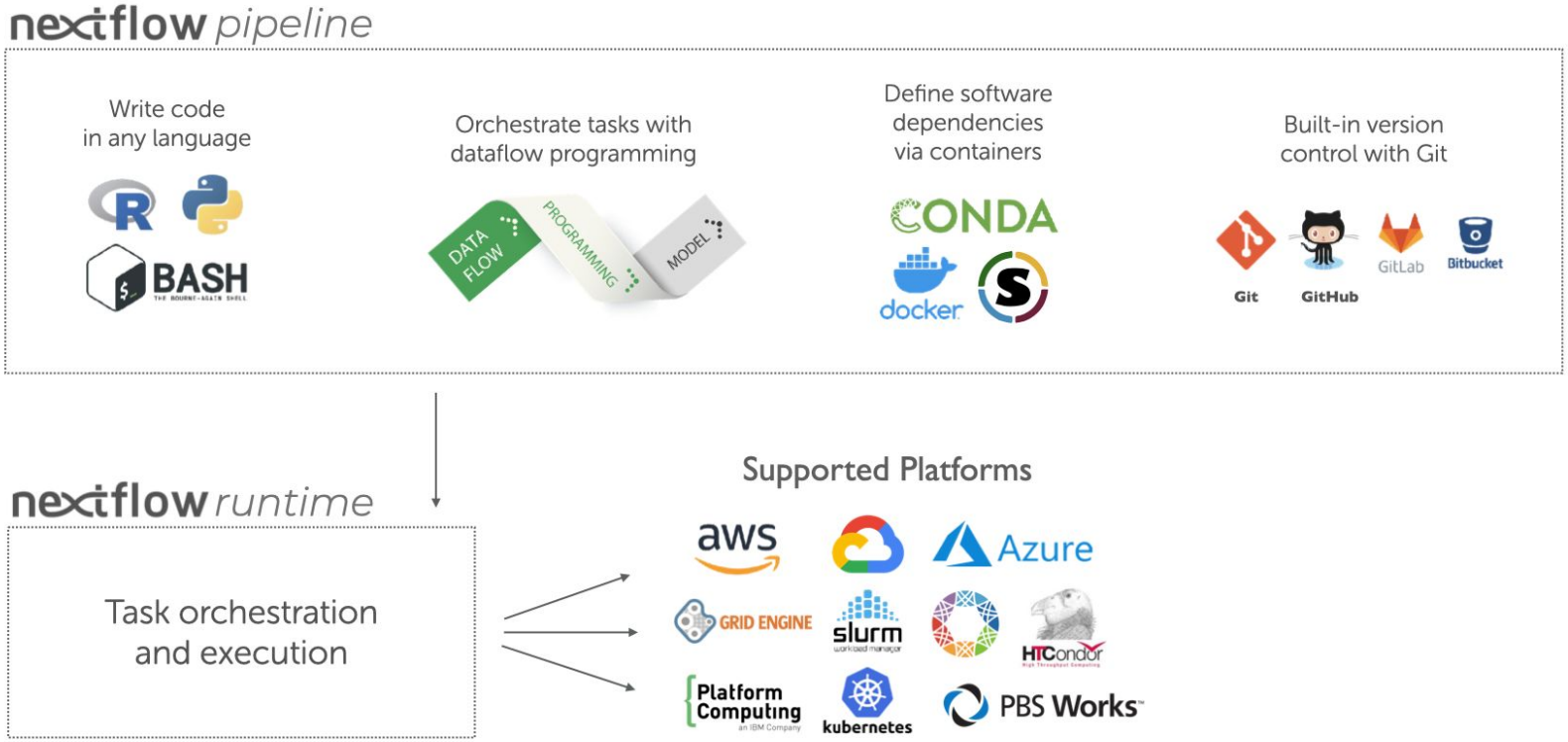
- Nextflow is a domain specific language for workflow framework
- It is an extension of Groovy programming language (super-set of Java)
- It enables scalable and reproducible workflows without worrying about filenames, error handling, environment, dependencies, resuming execution, parallelisation etc.

Nextflow

- **Fast prototyping -** Simple to put together many different tasks.
- **Reproducibility -** Support Docker, Singularity, Conda. Also GitHub for reproducibility using former configuration.
- **Portable -** Supports SLURM, AWS, Kubernetes, Microsoft Azure etc
- **Parallelism -** Tasks can scale-up or scale-out. Splits the tasks to parallel tasks
- **Continuous checkpoints -** Intermediate checkpoints enabling resuming of execution, avoiding re-execution of same step.
- **Stream oriented -** Can handle complex pipelines easily enabling resilient and reproducible pipelines


CMCB TECHNOLOGY PLATFORM
Center for Molecular and Cellular Bioengineering TU Dresden
institut Curie
lifebit
seqeralabs
BovReg
Understanding cattle genomes
NIBSC
Confidence in Biological Medicines
max planck institute of biochemistry
Universiteit Leiden
wellcome sanger institute
GMI
GREGOR MENDEL INSTITUTE OF MOLECULAR PLANT BIOLOGY
CANCER RESEARCH UK
UNIVERSITY OF LIVERPOOL
MRC Human Genetics Unit
Berkeley
UNIVERSITY OF CALIFORNIA
IMP
Research Institute of Molecular Pathology
INDIAN INSTITUTE OF SCIENCE EDUCATION AND RESEARCH MOHALI
IN PURSUIT OF KNOWLEDGE
NBIS
NATIONAL BIOINFORMATICS INFRASTRUCTURE SWEDEN
THE PICOWER INSTITUTE
FOR LEARNING AND MEMORY
Baylor College of Medicine
SciLifeLab
EASI Genomics
iit-b
ज्ञानमुत्तमम

Overview


nextflow pipeline
Write code in any language
R
BASH
Orchestrate tasks with dataflow programming
DATA FLOW
PROGRAMMING
MODEL
Define software dependencies via containers
CONDA
docker
Built-in version control with Git
Git
GitHub
GitLab
Bitbucket
nextflow runtime
Task orchestration and execution
Supported Platforms
aws
Azure
GRID ENGINE
slurm
HTCondor
Platform Computing
kubernetes
PBS Works

How does it work?

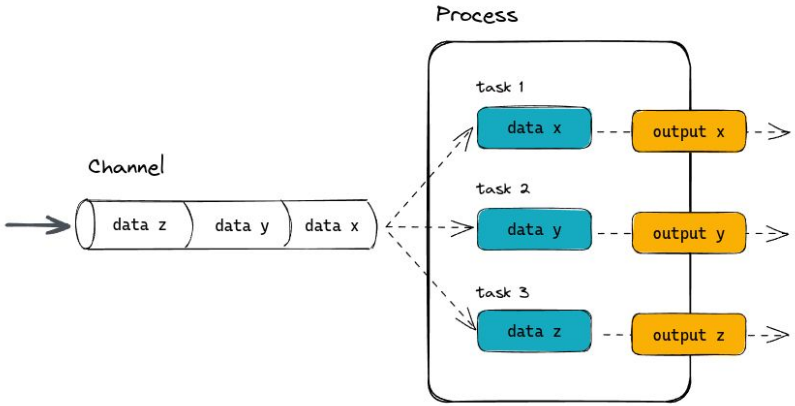
- Pipeline is made by stitching different processes together. Process can be in any scripting language executed by linux platform

- Processes are executed independently and are isolated. The communicate via async FIFO (First-in, first-out) queues called channels.

- Process contain input and output declaration connected to other process using channels.

Example

Currently Nextflow run in DSL2



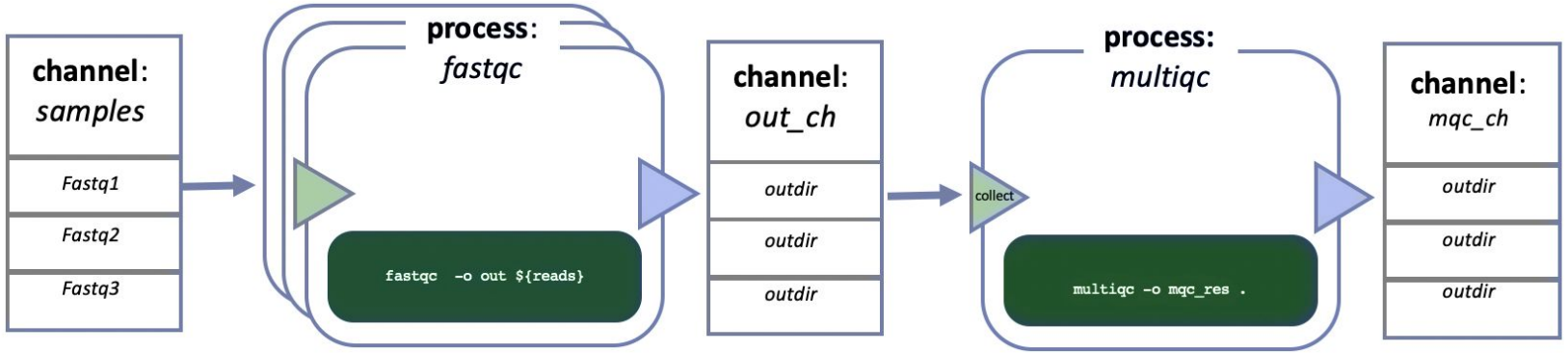

```
#!/usr/bin/env nextflow   → shebang

params.greeting = 'Hello world!' → input parameter
greeting_ch = Channel.of(params.greeting) → creating channel

process SPLITLETTERS { → Process 1

    cpus 4
    label “splitting_letters”   } Directives

    input:  → input block
    val x

    output:            → output block
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_   } Script section
    """
}

process CONVERTTOUPPER { → Process 2
    input:
    path y

    output:
    stdout

    """
    cat $y | tr '[a-z]' '[A-Z]'
    """
}
workflow {
greet_out_ch = SPLITLETTERS(greeting_ch)
read_ch = CONVERTTOUPPER(greet_out_ch)
read_ch.view()}   } Define workflow
```

Usage

- The Nextflow script is usually saved in a file called **main.nf**
- The script is run using
  - ***nextflow run main.nf***
  - ***nextflow main.nf***
- To resume an unfinished pipeline
  - ***nextflow main.nf -resume***
- Parameters are passed using
  - ***nextflow main.nf --param_name1 parameter1 --param_name2 parameter2 -resume***
    - Eg: ***nextflow main.nf --greeting “hello world”***

Basics

- **Channels**: They connect tasks/process to each other or apply data transformation
  - Two types i) Queue channel ii) Value channel
- **Processes**: To run any script
  - May contain five definition blocks
    - Directives, Input, Output, When, Script
- **Operators**: Connect and transforms channels content
  - Eg. collect, flatten
- **Workflow**
  - The data flow or logic of the workflow



Channels

- Channels connect one process to another
- Consider as data is flowed through the channels and the receiving process will do necessary operation on it
- The channels can contain different types of data based on how it is defined
- The Queue channels cannot be used more than once but the value channels can be used multiple times
- Queue channels are asynchronous, unidirectional and FIFO

How to create a channel?

Value Channel: **value()**

```
ch1 = Channel.value( 'Hello there' )
ch2 = Channel.value( [1,2,3,4,5] )
ch1.view()
ch2.view()
```

Output:

```
Hello there
[1, 2, 3, 4, 5]
```

Queue Channel: **of()**

```
ch = Channel.of( 1, 3, 5, 7 )
ch.view()
```

Output:

```
1
3
5
7
```

Queue Channel: **fromList()**

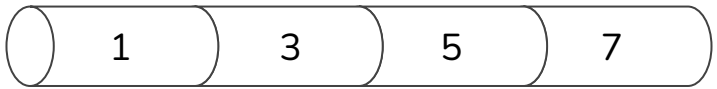
```
list = ['hello', 'world' , '!', '.']
Channel.fromList(list)
            .view()
```

Output:

```
hello
world
!
.
```

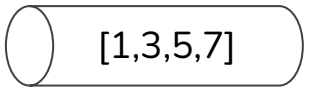
How to create a channel?

```
ch = Channel.of( 1, 3, 5, 7 )
ch.view()
```



Queue Channel: **of()**

```
ch = Channel.of( [1, 3, 5, 7])
ch.view()
```



How to create file channels?

Queue Channel: **fromPath()**

```
Channel.fromPath( './data/folder/*.csv' )
```

Output:

```
/user/project/csv_files/file_1.csv
/user/project/csv_files/file_2.csv
/user/project/csv_files/file_3.csv
```

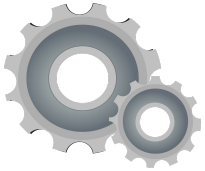
- The “*” represents the glob character. It matches the pattern in the filename.
- “**.csv” would do recursive file pattern matching
  - i.e., search for pattern in a folder inside a folder

Other channels such as Channel.fromFilePairs() and Channel.fromSRA()
We can avoid them for now

Processes

- **Directives**: Initial declarations that are optional (eg: cpus, label, memory etc)
- **Input**: Defines the input channel
- **Ouput**: Defines the output channel
- **When**: Optional statement to check, If condition is true, the process will run or else otherwise.
- **Script**: The necessary script to be executed

```
process < name > {

  [ directives ]

  input:
  < process inputs >

  output:
  < process outputs >

  when:
  < condition >

  [script|shell|exec]:
  """
  < user script to be executed >
  """
}
```

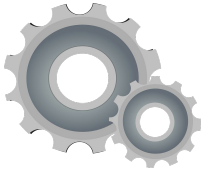


Directives

- This is an optional settings for the execution of the current process without affecting the semantic of the task itself.

- It should be on top of the process body, i.e., above input and output blocks

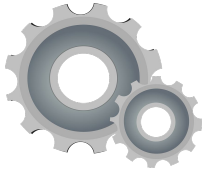
```
process < name > {

  [ directives ]

  input:
  < process inputs >

  output:
  < process outputs >

  when:
  < condition >

  [script|shell|exec]:
  """
  < user script to be executed >
  """
}
```



Directives

- **cpus** : number of logical cores to be used
- **time** : defines how long a process should run (helpful when program is taking too long than it should)
- **memory** : the amount of RAM to be used in the process
- **tag** : gives a custom label to the process (easier to identify when the process is running)
- **disk** : the space the process can use

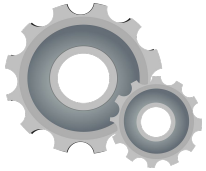
```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    cpus 4
    tag “splitting_letters”
    time ‘1d’
    memory ‘8 GB’
    disk ‘15 GB’

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}

workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - publishDir

- The publishDir outputs the results of the process to a separate folder given. The workflow outputs are usually stored in work directory.
- The publishDir directive can be specified more than one time in to publish the output files to different target directories
- In this example, these files chunk_aa, chunk_ab, chunk_ac are created inside results folder. The result folder is created in the same directory as the process is ran.
- The results will get overwritten if the process produces files with the same name.

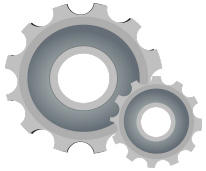
```
#!/usr/bin/env nextflow

params.greeting = ['Hello world!','world','hello']
greeting_ch = Channel.fromList(params.greeting)

process SPLITLETTERS {
    echo true
    cpus 4
    memory '8 GB'
    disk '15 GB'
    publishDir "results"

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}

workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - publishDir

- To avoid overwriting, process specific and output specific files or folders have to be created.
- In this process we create dynamic folders using “/$x”. This creates three folders inside result directory named “hello world!”, “world”, “hello”

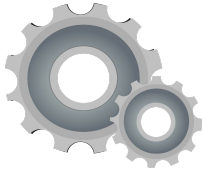
```
#!/usr/bin/env nextflow

params.greeting = ['Hello world!','world','hello']
greeting_ch = Channel.fromList(params.greeting)

process SPLITLETTERS {
    echo true
    cpus 4
    memory '8 GB'
    disk '15 GB'
    publishDir "results/$x"

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - publishDir

- publishDir creates symbolically linked files than copying the files.
- The files that are linked to are present in the work directory
- Hence deleting the work directory makes the results in the publishDir folder unusable.
- Hence to make a copy of the file,
    - mode: 'copy'

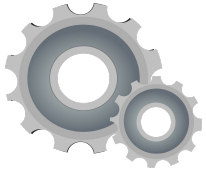
```
#!/usr/bin/env nextflow

params.greeting = ['Hello world!','world','hello']
greeting_ch = Channel.fromList(params.greeting)

process SPLITLETTERS {
    echo true
    cpus 4
    memory '8 GB'
    disk '15 GB'
    publishDir "results/$x" , mode: 'copy'

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - conda

- The **conda** directive tells the process to use dependencies using conda package manager
- Nextflow will automatically set the conda environment for the process to run
- Specific package can be downloaded and used by specifying channel of the package eg: “bioconda::bwa=0.7.15”
- Already existing environment can be also provided by specifying the path of the environment directory
- Multiple packages can be specified separating them with a blank space eg. "Openms xcms". The name of the channel from where a specific package needs to be downloaded can be specified using the usual Conda notation i.e. prefixing the package with the channel name as shown here bioconda::Openms=2.4.0

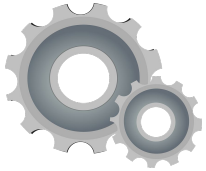
```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    conda ‘bwa=0.7.15’

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - module

- Environment Modules is a package manager that allows you to dynamically configure your execution environment and easily switch between multiple versions of the same software tool
- If it is available in your system you can use it with Nextflow in order to configure the processes execution environment in your pipeline
- In a process definition you can use the module directive to load a specific module version to be used in the process execution environment
- Multiple modules can be specified in a single module directive by separating all the module names by using a : (colon) character

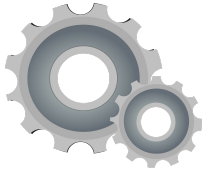
```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    module 'openms/2.4.0'

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - container

- The **container** allows to execute the process script in a docker container
- It requires the Docker daemon to be running in machine where the pipeline is executed, i.e. the local machine when using the *local* executor or the cluster nodes when the pipeline is deployed through a *grid* executor.

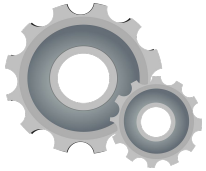
```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    cpus 4
    container ‘dockerbox:tag’

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```



Directives - stageInMode

- The stageInMode defines how the data is stored in the work directory
- Nextflow by default uses symbolic link to stage files
- Set this to “copy” if you have the same file names in input and output
- Will increase overhead of the pipeline significantly

```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    cpus 4
    stageInMode copy

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```

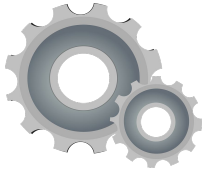


Directives - tag

- The tag directive allows to give custom name for the process.
- Makes it easier to identify them in the log file or in the trace execution report

```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    tag “$x”

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```

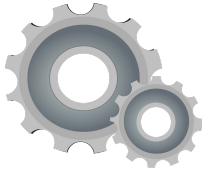
```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [astonishing_laplace] - revision: cef8def432
executor >  local (1)
[6b/5047df] process > SPLITLETTERS (Hello world!) [100%] 1 of 1 ✔
```



Directives - label

- The label directive allows the annotation of processes with an identifier of your choice
- The same label can be applied to more than a process and multiple labels can be applied to the same process using the label directive more than one time
- Labels are useful to organise pipeline processes in separate groups which can be referenced in the configuration file to select and configure subset of processes having similar computing requirements

```
#!/usr/bin/env nextflow

params.greeting = 'Hello world!'
greeting_ch = Channel.of(params.greeting)

process SPLITLETTERS {

    cpus 4
    label “basic_process”

    input:
    val x

    output:
    path 'chunk_*'

    """
    echo '$x' | split -b 6 - chunk_
    """
}
workflow {
out_ch = SPLITLETTERS(greeting_ch)
}
```

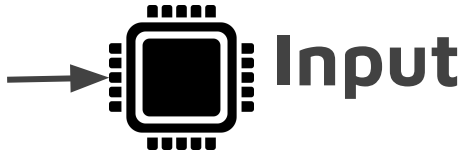


Directives - some more

maxForks → Defines maximum number of process instance that can be executed in parallel

executor → Defines where the processes are executed (eg: awsbatch, azurebatch, slurm etc)

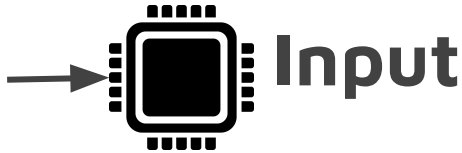
debug → setting debug to true will print the stdout on the terminal. Useful while creating a pipeline



Input

- Processes in nextflow are isolated and they are connected by sending values through channels
- Input determines the parallel execution of the process
- An input block can contain multiple input declarations
- A process is started when there is new data to be consumed from the input channel

```
process < name > {

  [ directives ]

  input:
  < process inputs >

  output:
  < process outputs >

  when:
  < condition >

  [script|shell|exec]:
  """
  < user script to be executed >
  """
}
```



Input

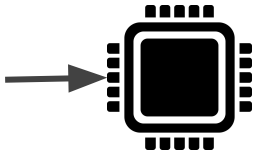
- The input **qualifier** declares the type of data to be received.
- The value can be accessed in the process script by using the specified input **name**,

What type of input the process is getting (file, value etc)?

```
input:
<input qualifier>      <input name>
```

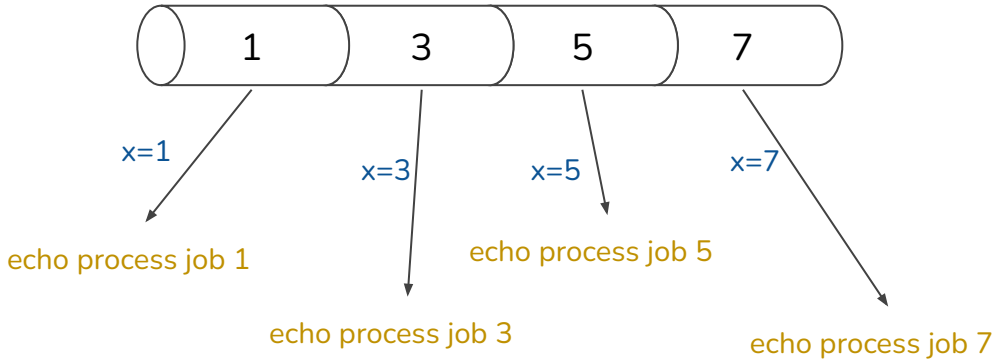
A name used to access the received value?

Qualifier	Semantic
val	Access the input value by name in the process script.
file	(DEPRECATED) Handle the input value as a file, staging it properly in the execution context.
path	Handle the input value as a path, staging the file properly in the execution context.
env	Use the input value to set an environment variable in the process script.
stdin	Forward the input value to the process stdin special file.
tuple	Handle a group of input values having any of the above qualifiers.
each	Execute the process for each element in the input collection.



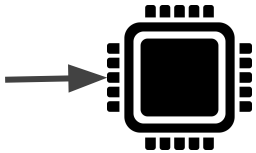
Input - val

- The val qualifier allows you to receive data of any type as input. It can be accessed in the process script by using the specified input name




```
num = Channel.from( 1, 3, 5, 7 )

process basicExample {
echo true

input:
val x

script:
"echo process job $x"
}

workflow {
basicExample(num)
}
```

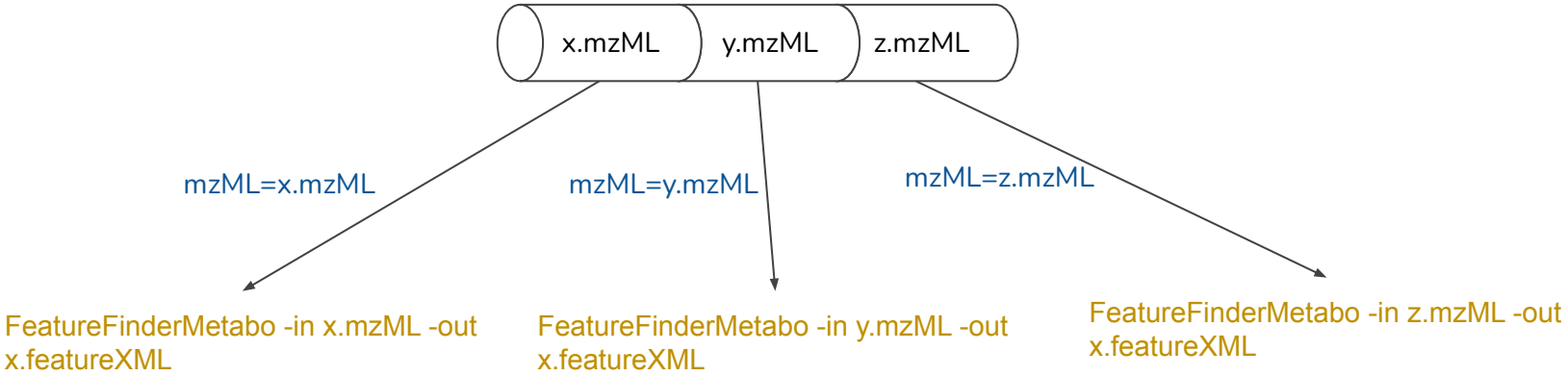
```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [tiny_swartz] - revision: 54fa7ea911
executor >  local (4)
[4e/03bd91] process > basicExample (2) [100%] 4 of 4 ✔
process job 1

process job 7

process job 5

process job 3
```



Input - file

- The file qualifier allows the handling of file values in the process execution context. This means that Nextflow will stage it in the process execution directory, and it can be access in the script by using the name specified in the input declaration

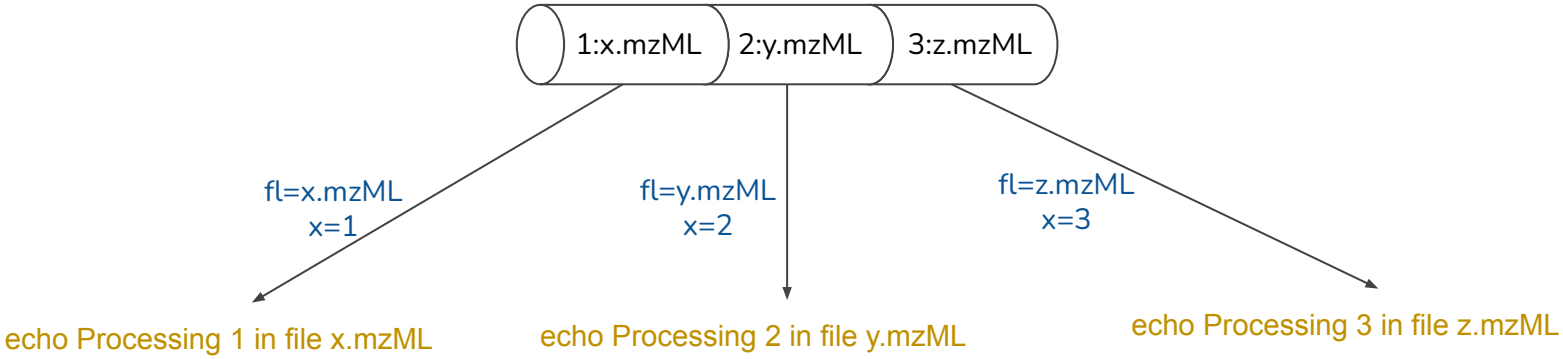
```
mzMLFiles = Channel.fromPath( '/some/path/*.mzML' )

process featureFinder {

Input:
file mzML

"FeatureFinderMetabo -in ${mzML} -out
${mzML.baseName}.featureXML"
}

workflow {
featureFinder(mzMLFiles)
}
```

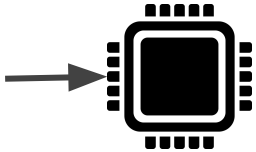


Input - tuple

- The tuple qualifier allows you to group multiple parameters in a single parameter definition. It can be useful when a process receives tuples of values that need to be handled separately

```
values = Channel.of( [1, "x.mzML"], [2, 'y.mzML'], [3,
 'z.mzML'] )

process tupleExample {

input:
tuple val(x), val(fl)

 """
echo Processing $x in file $fl
"""
 }

workflow {
tupleExample(values)
}
```

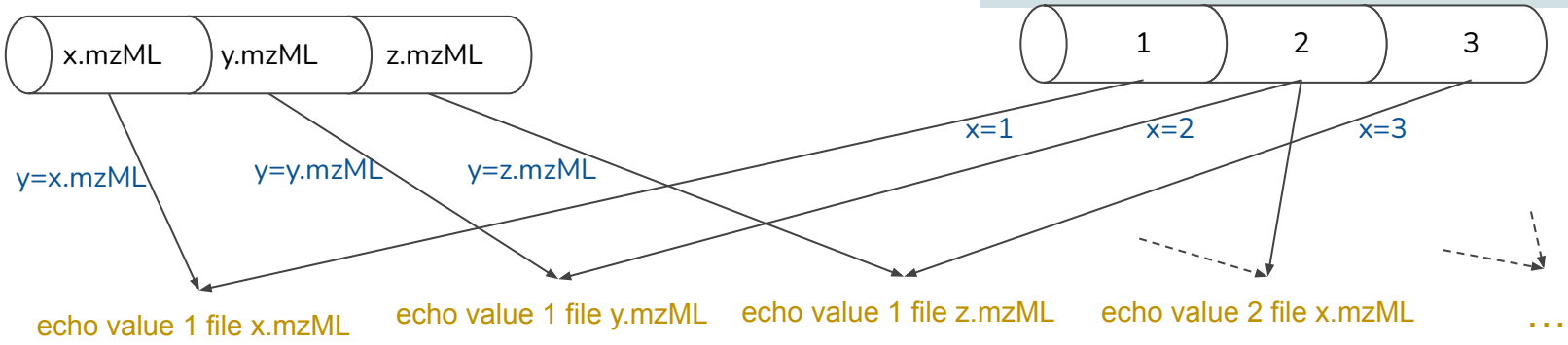


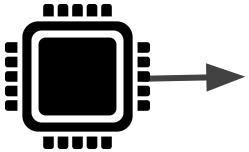


Input - each

- The each qualifier allows you to repeat the execution of a process for each item in a collection, every time a new data is received

```
mzMLFiles = Channel.fromPath('../xcms_pipeline/test-datasets/*.*' )
num = Channel.from( 1, 2, 3 )

process featureFinder {

input:
each x
file y

"""
echo value $x file $y
"""
}

workflow {
featureFinder(num,mzMLFiles)
}
```

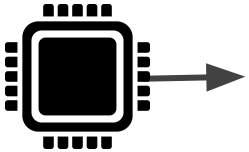




Output

- The output declaration block allows you to define the channels used by the process to send out the results produced. You can only define one output block at a time and it must contain one or more output declarations

```
process < name > {

  [ directives ]

  input:
  < process inputs >

  output:
  < process outputs >

  when:
  < condition >

  [script|shell|exec]:
  """
  < user script to be executed >
  """
}
```



Output

What type of output the process is giving (file, value etc)?

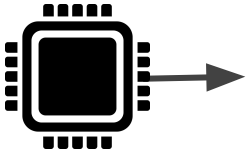
Optional attributes

```
output:
<output qualifier>   <output name> [, <option>: <option value>]
```

The actual name of value, variable or file the process is output. E.g., results of the analysis!

- Output definitions start by an output qualifier and the output name, followed by the keyword into and one or more channels over which outputs are sent. Finally some optional attributes can be specified.

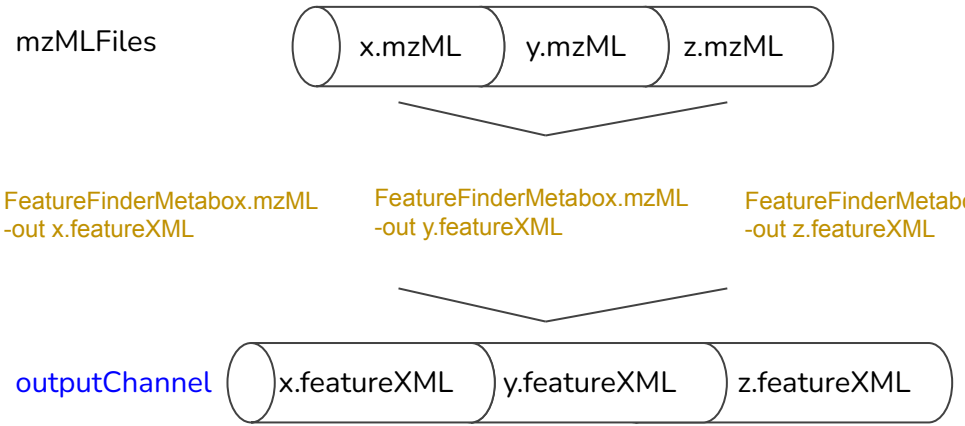
Qualifier	Semantic
val	Emit the variable with the specified name.
file	(DEPRECATED) Emit a file produced by the process with the specified name.
path	Emit a file produced by the process with the specified name.
env	Emit the variable defined in the process environment with the specified name.
stdout	Emit the stdout of the executed process.
tuple	Emit multiple values.

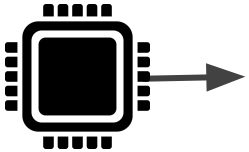


Output - file

- The file qualifier allows you to output one or a collection of files to another channel that can be consumed by another process

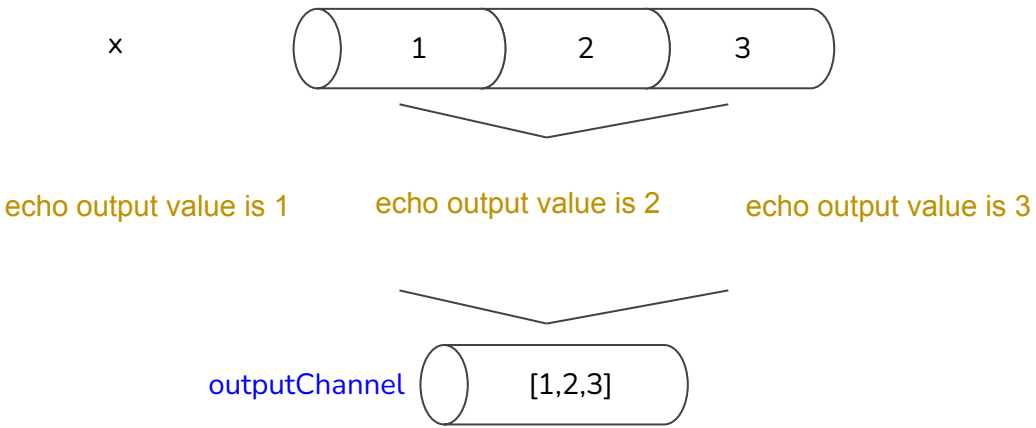
```
mzMLFiles = Channel.fromPath( '/some/path/*.mzML' )

process featureFinder {

input:
file x

output:
file "${x.baseName}.featureXML"

"""
FeatureFinderMetabo -in ${x}.mzML -out
${x.baseName}.featureXML
"""
}

workflow {
outputChannel = featureFinder(mzMLFiles)
outputChannel.view()
}
```





Output - val

```
values = Channel.of( 1,2,3 )

process basicExample {
input:
 val x

 output:
 val "$x"

"""
echo output value is $x
"""
 }
workflow {
outputChannel = basicExample(values).collect()
outputChannel.view()
}
```




```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [deadly_marconi] - revision: 86ada0ceef
executor >  local (3)
[cc/363d68] process > basicExample (1) [100%] 3 of 3 ✓
output value is 1

output value is 3

output value is 2

['1', '3', '2']
```

Script

- The **script** block defines the command to be executed in the process
- It is the last block of the process body, i.e., after input, output, and when blocks
- It can be a command, script or combination of them that would normally use terminal shell or bash
- The script block can be defined using single-quotes or double quotes and multiline strings can be defined in three single-quotes or three double-quotes

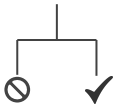
```
process < name > {

  [ directives ]

  input:
  < process inputs >

  output:
  < process outputs >

  when:
  < condition >

  [script|shell|exec]:
  """
  < user script to be executed >
  """
}
```

Script

- Double quotes are used to run bash script
- Single quotes are used to run shell script
- Exec will run the native groovy commands
- “$” sign will replace the value with nextflow variable
- To access native bash variable “\$” is used
    - Eg: echo \$PATH

When running a pipeline, the assumption is that the tools are available on the system where you are running the pipeline

```
var = "world"

process SIMPLYECHO_BASH {
    input:
    val var
    script:
    """
    echo 'hello $var'
    """
}

process SIMPLYECHO_SHELL {
    input:
    val var
    shell:
    '''
    echo  hello !{var}
    '''
}

process SIMPLYECHO_NATIVE {
    input:
    val var
    exec:
    println "hello $var"
}

workflow {
SIMPLYECHO_BASH(var)
SIMPLYECHO_SHELL(var)
SIMPLYECHO_NATIVE(var)
}
```



When

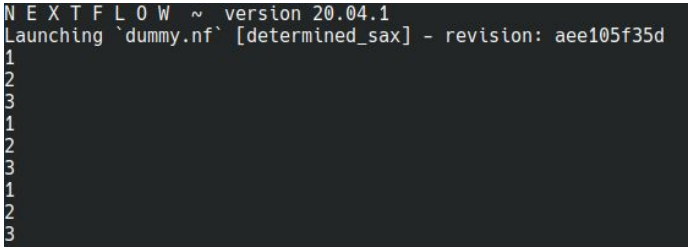
- The **when** declaration defines a condition that must be verified in order to execute the process. This can be any expression that evaluates a boolean value
- It is useful to enable/disable the process execution depending on the state of various inputs and parameters.
- In this example the process is executed only when the value of "$x" becomes “world”

```
#!/usr/bin/env nextflow

params.greeting = ['Hello world!','world','hello']
greeting_ch = Channel.fromList(params.greeting)

process SPLITLETTERS {

    input:
    val x

    output:
    path 'chunk_*'

    when:
    "$x" == "world"

    """
    echo '$x' | split -b 6 - chunk_
    """
}

workflow {
outputChannel = SPLITLETTERS(greeting_ch)
}
```

Operators - view, print, println

- These operators print the items emitted by a channel to the console standard output
- The println operator prints the items emitted by a channel to the console standard output appending a new line character to each of them
- Each operators have additional options, read more in the documentation

```
Channel.from(1,2,3).view()

Channel.from(1,2,3).print()

Channel.from(1,2,3).println()
```



Operators - set, into

- The set operator assigns the channel to a variable whose name is specified as a closure parameter
- The into operator connects a source channel to two or more target channels in such a way the values emitted by the source channel are copied to the target channels
- A second version of the into operator takes an integer n as an argument and returns a list of n channels, each of which emits a copy of the items that were emitted by the source channel.

```
Channel.from(10,20,30).set { my_channel }
my_channel = Channel.from(10,20,30)


Channel.from(10,20,30).into { my_channel1; my_channel2 }
( my_channel1, my_channel2)=Channel.from(10,20,30).into(2)
```

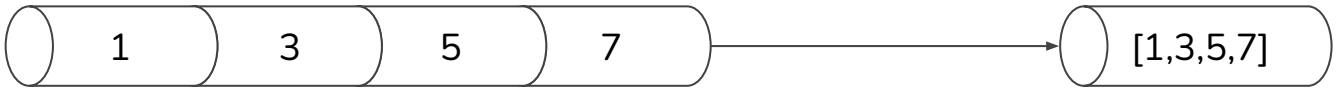
Operators - collect

- The collect operator collects all the items emitted by a channel to a List and return the resulting object as a sole emission

```
Channel.from( 1, 2, 3, 4, 5 ).collect()
.view()
```

```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [dreamy_faggin] - revision: 8ef337eb40
[1, 2, 3, 4, 5]
```

Eg: Collect all the output from the previous process



Output of process 1

Collect to get as a list

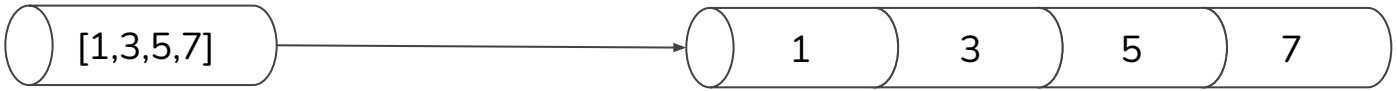
Operators - flatten

- The flatten operator transforms a channel in such a way that every item of type Collection or Array is flattened so that each single entry is emitted separately by the resulting channel

- This is opposite to collect

```
Channel.of( [1, 2, 3, 4, 5] )
.flatten()
.view()
```

```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [distraught_keller] - revision: 9d8f7b5079
1
2
3
4
5
```



Output of process 1

Flatten to four emission

Operators - map

- The map operator applies a function of your choosing to every item emitted by a channel, and returns the items so obtained as a new channel. The function applied is called the mapping function and is expressed with a closure

```
Channel.from( 1, 3, 5, 7 )
.map { it * it }.view()
```

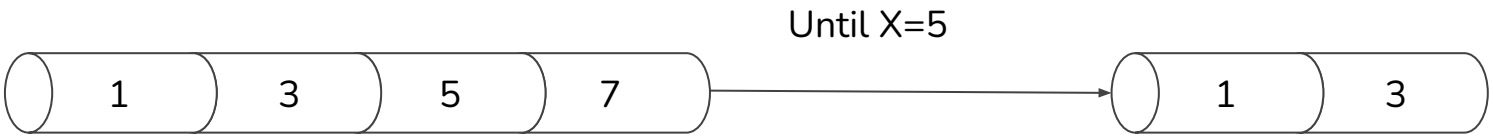
```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [insane_kare] - revision: 8a5994884a
1
9
25
49
```

Operators - until

- The until operator creates a channel that returns the items emitted by the source channel and stop when the condition specified is verified

```
Channel.from( 1, 3, 5, 7 )
.until { it==5}.view()
```

```
N E X T F L O W  ~  version 20.04.1
Launching `dummy.nf` [romantic_minsky] - revision: 392712081d
1
3
```



Workflow

- The workflow definition starts with the keyword `workflow`, followed by an optional name, and finally the workflow body delimited by curly braces.

- The order in which a task is executed is determined only by its dependencies, so a task will be executed as soon as all of its required inputs are available.

```
workflow workflow_name{
    foo() → process
   }


workflow {
    Output_channel = foo1(input_channel)
    Output_channel2 = foo2(Output_channel)
    Output_channel2.collect().view()
   }


// When there is two input and output channels
workflow {
    (output_channel1,output_channel2) =
foo(input_channel1,input_channel2)
   }

// When there is only one input and output channel, another way
to write (similar to bash pipe)
workflow {
    foo1 | foo
   }
```

Configuration file

- Nextflow supports a wide range of execution platforms, from running locally, to running on HPC clusters or cloud infrastructures
- Many of these settings are set in the configuration file
- When a pipeline script is launched. Nextflow looks for a file named nextflow.config in the current directory and in the script base directory (if it is not the same as the current directory). Finally it checks for the file $HOME/.nextflow/config

```
// This is the configuration file e.g nextflow.config

executor {
        $slurm{
                queueSize = 200
                pollInterval = '30 sec'
                clusterOptions = { "-A
                $params.project
                ${params.clusterOptions ?: "}" }
        }
$local {cpus = 8}
}
```

Nextflow - how to write a pipeline

- Think first before start coding!
- Decide on the parameters (inputs, outputs, other arguments etc)
- Choose what platforms you are supporting (Containers, cloud etc)
- Open your favorite text editor
- Make two files (main.nf and nextflow.config)
- Define parameters both for the executors and the pipeline environment in the nextflow.config
- Write the actual workflow in the main.nf
- Test and test and test!
- Publish on Github + Dockerhub (for the tools part of the workflow)

More tutorials

- Most text in this presentation have been copied from here:

  https://www.nextflow.io/docs/latest/

- Nextflow google group: https://groups.google.com/g/nextflow
- And of course google!
- More tutorials:
  - https://nf-co.re/usage/nextflow
  - https://training.seqera.io
  - https://carpentries-incubator.github.io/workflows-nextflow
- Standardization of Nextflow workflows

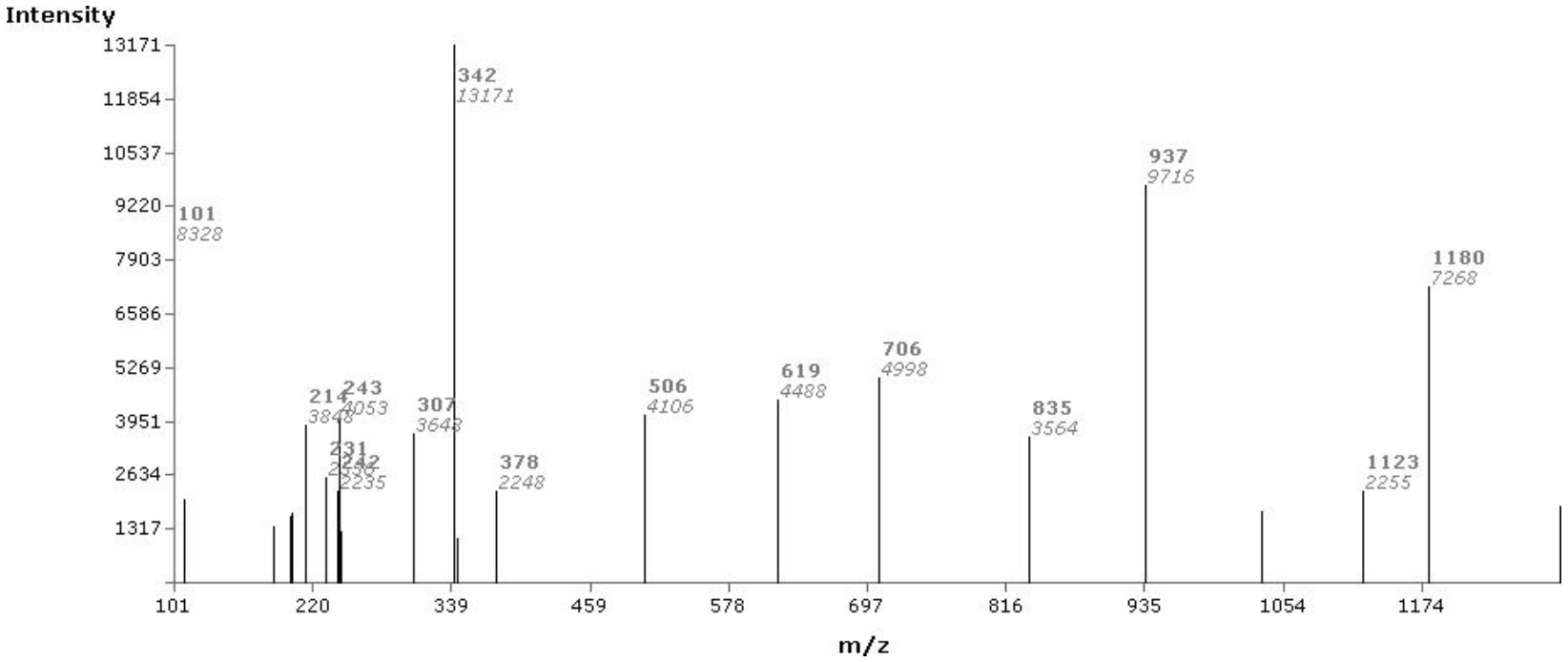
nf-core

Questions

Demo session

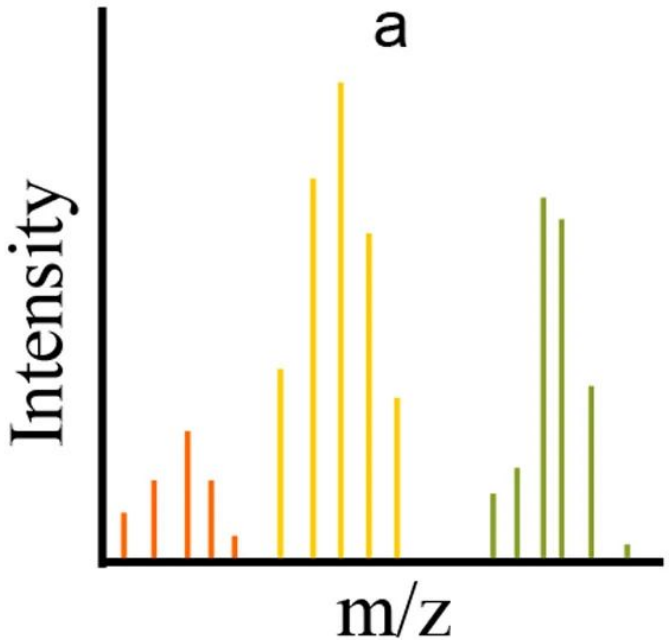
Mass spectrometry and data analysis

Data - snapshots



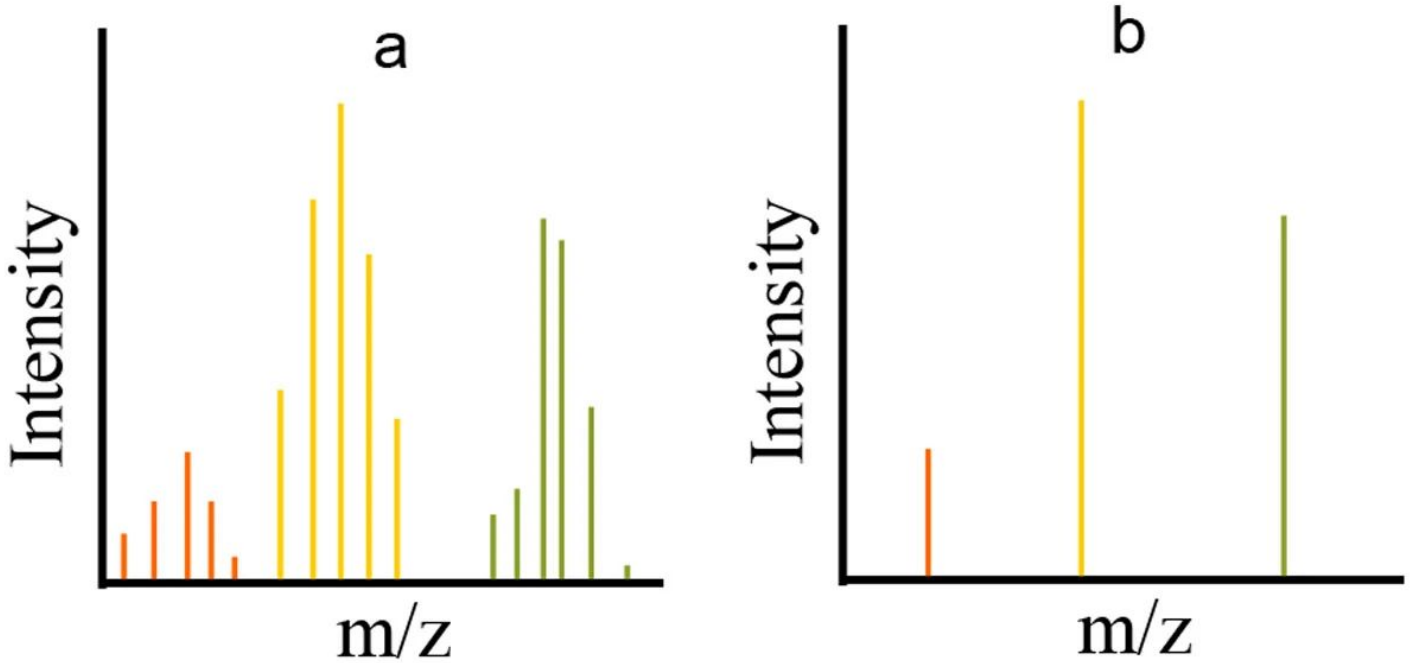
McHugh L, Arthur JW. Computational methods for protein identification from mass spectrometry data. PLoS Comput Biol. 2008 Feb;4(2):e12. doi: 10.1371/journal.pcbi.0040012. PMID: 18463710; PMCID: PMC2323404.McHugh L, Arthur JW. Computational methods for protein identification from mass spectrometry data. PLoS Comput Biol. 2008 Feb;4(2):e12. doi: 10.1371/journal.pcbi.0040012. PMID: 18463710; PMCID: PMC2323404.

Data - centroiding



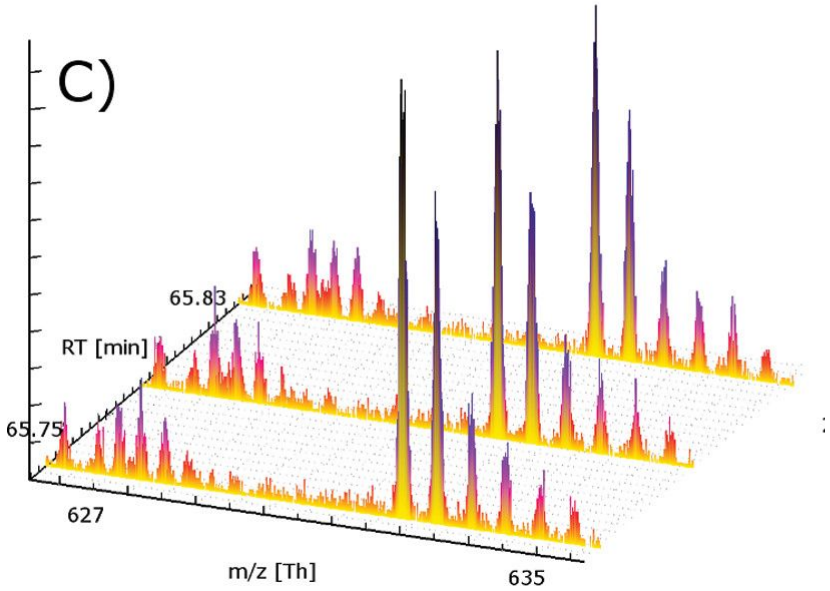
Smith, R., Mathis, A.D., Ventura, D. *et al.* Proteomics, lipidomics, metabolomics: a mass spectrometry tutorial from a computer scientist's point of view. *BMC Bioinformatics* 15 (Suppl 7), S9 (2014). https://doi.org/10.1186/1471-2105-15-S7-S9

Data - centroiding



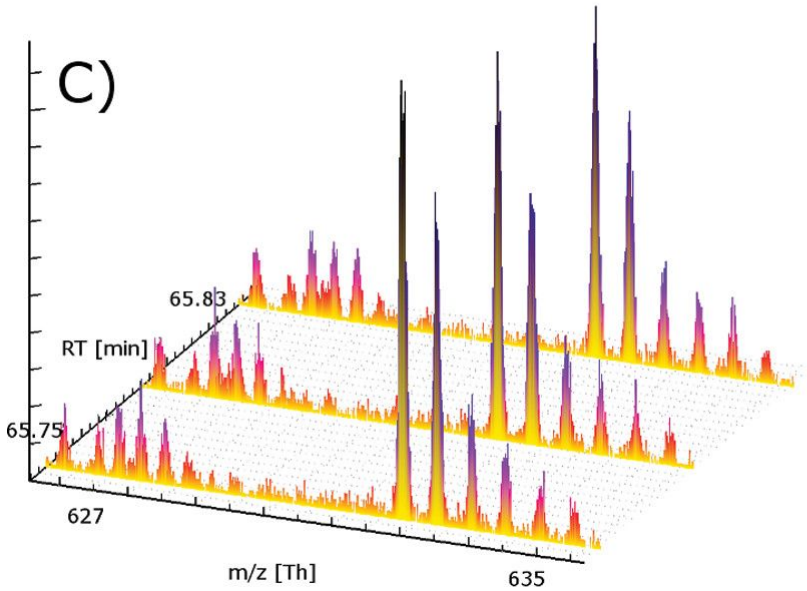
Smith, R., Mathis, A.D., Ventura, D. *et al.* Proteomics, lipidomics, metabolomics: a mass spectrometry tutorial from a computer scientist's point of view. *BMC Bioinformatics* 15 (Suppl 7), S9 (2014). https://doi.org/10.1186/1471-2105-15-S7-S9

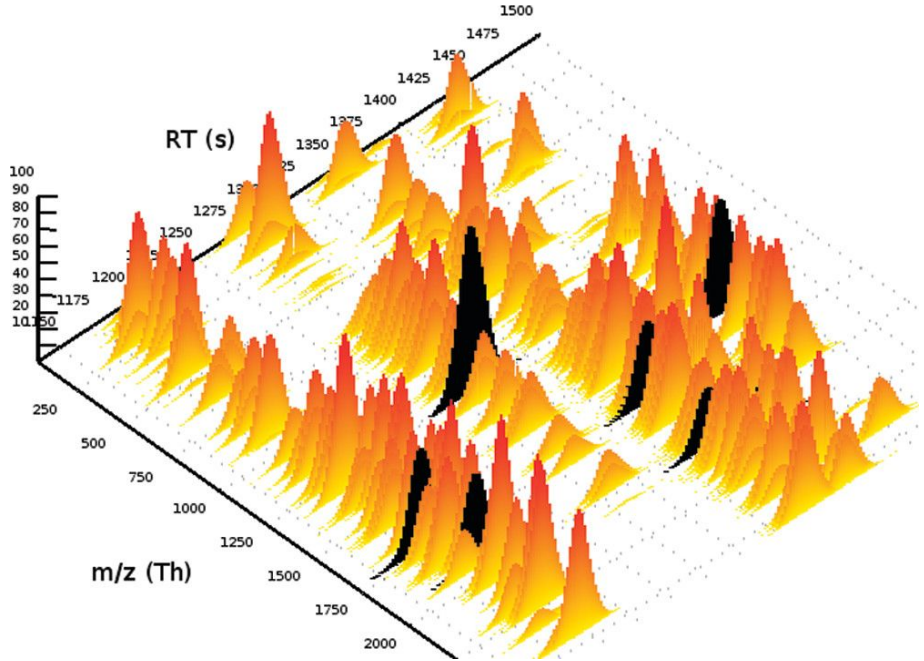
Data - over time



Bielow C, Aiche S, Andreotti S, Reinert K. MSSimulator: Simulation of mass spectrometry data. J Proteome Res. 2011 Jul 1;10(7):2922-9. doi: 10.1021/pr200155f. Epub 2011 Apr 28. PMID: 21526843.

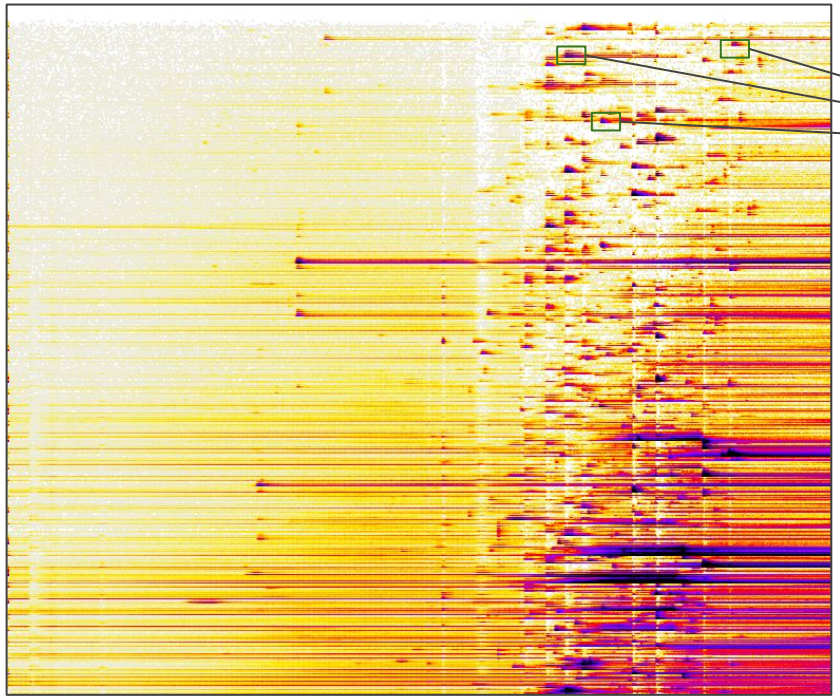
Data - over time



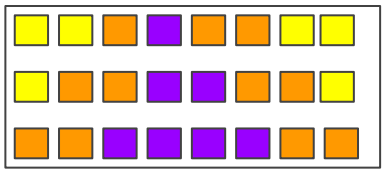


Bielow C, Aiche S, Andreotti S, Reinert K. MSSimulator: Simulation of mass spectrometry data. J Proteome Res. 2011 Jul 1;10(7):2922-9. doi: 10.1021/pr200155f. Epub 2011 Apr 28. PMID: 21526843.

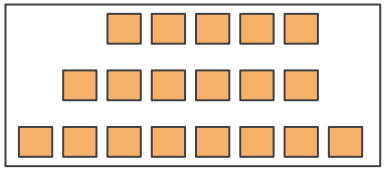
Data - 2D-View



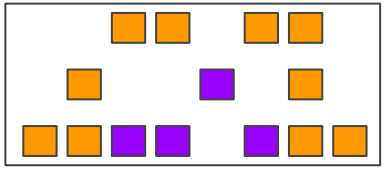
Data - features



Ideal feature

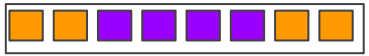


Low intensity

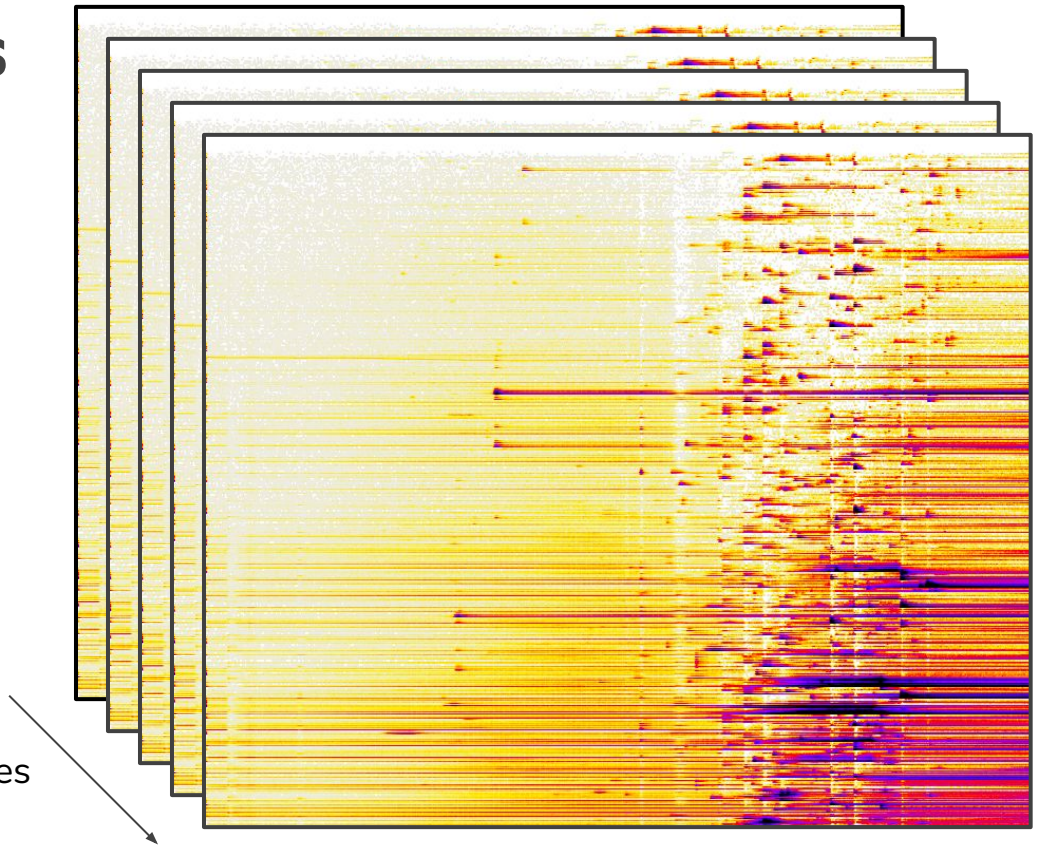


Missing values

Many other problems



Missing mass traces

Data - samples

Connecting features across samples