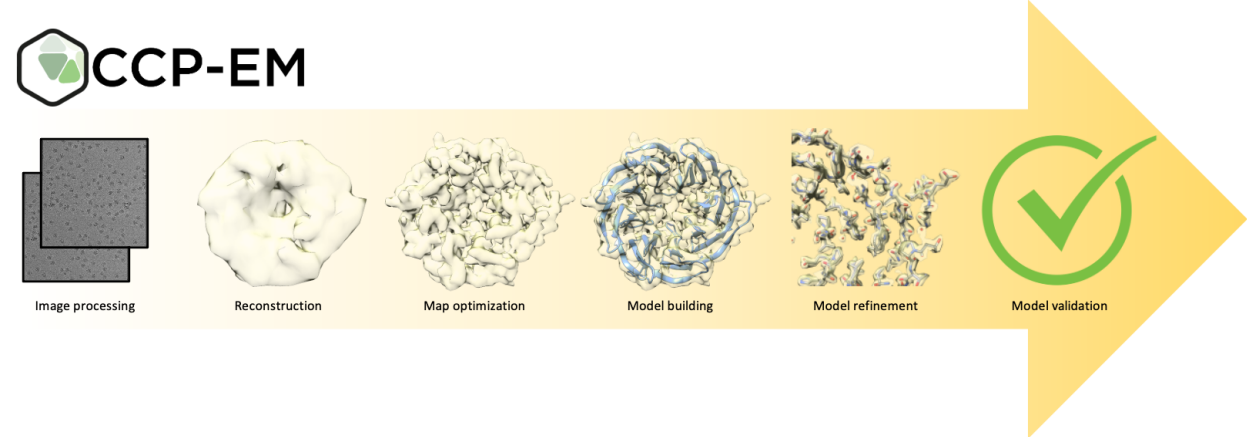
CCPEM-pipeliner

Matthew G Iadanza, Colin J Palmer, Jola Mirecka

May 23, 2022

CONTENTS



The CCPEM-pipeliner is an integrated suite of software tools for processing single particle cryoEM data, from preprocessing raw image data through building and fitting atomic models.

The pipeliner brings together a variety of 3rd party software in a single unified framework for seamless integration of the different programs, along with tools for management and analysis of the project.

CHAPTER

ONE

GENERAL INFO AND INSTALLATION

1.1 CCPEM-pipeliner

source/docs/source/header.png

1.1.1 Installation

Once the package has been downloaded navigate into the `ccpem-pipeliner` directory and install the pipeliner with the command:

```
pip install -e .
```

1.1.2 Check the installation

Once the pipeliner is installed use the command `check_setup.py` to check that the setup is complete and the pipeliner can find the Relion programs it needs to run.

1.1.3 Documentation

Documentation is available online at: ccpem-pipeliner.readthedocs.io/en/latest/

To build the documentation yourself, install the documentation build requirements as follows:

```
pip install -r requirements-docs.txt
```

Then navigate to the `ccpem-pipeliner/docs` directory and run the command:

```
make html
```

Then open the file `ccpem-pipeliner/docs/_build/html/index.html` in a web browser to access the documentation.

1.1.4 For Developers

It's a good idea to work in a virtual environment for this project. Set one up as follows:

```
python3 -m venv venv/
source venv/bin/activate
pip install -r requirements.txt
pip install -r requirements-dev.txt
pre-commit install
```

This project uses pre-commit to run Black for code formatting, flake8 for linting and some other simple checks.

You might want to install Black separately yourself too, so you can run it from your IDE.

According to the flake8 documentation, flake8 should not stop a git commit from going ahead unless `flake8.strict` is set in the git config. That doesn't actually seem to work: with the current configuration, commits fail if flake8 finds any problems. There are some flake8 warnings that have not been fixed yet, so to get around this, flake8 checks can be disabled: `SKIP=flake8 git commit ...`

1.1.5 Unit tests

Run the tests with `pytest`.

Some of the tests are quite slow or require some interaction so these tests are skipped by default. Set the environment variable `PIPELINER_TEST_SLOW` to a non-empty string to run the slower tests as well. Set the environment variable `PIPELINER_TEST_INTERACTIVE` to a non-empty string to also run the tests that require interaction.

CHAPTER

TWO

GETTING STARTED

2.1 Getting started

The CCPEM-Pipeliner provides easy access to a variety of software tools for all steps of processing cryoEM data from data preprocessing through model building and validation. The workflow is tracked and tools for visualising the full project and analysing the results are provided.

ccpem-pipeliner serves the back end for its companion sofware *doppio<https://gitlab.com/ccpem/doppio>* which provides a full graphical user interface

2.1.1 Start with a *Project*

The **Project** is contained in a single project directory. Paths used by the various steps in the pipeliner are generally relative to this project directory.

To create a project or access an existing project using the API:

```
from pipeliner.api.manage_project import PipelinerProject

my_project = PipelinerProject()
```

To start a new project from the command line

```
$ CL_pipeline --start_new_project
```

2.1.2 A *Project* is made up of *Jobs*

The project is made of up **Jobs**. Each job is one type of operation on the data, although jobs can have several steps. Jobs are defined by their **jobtype**. The format of the jobtype is:

<program>.<function>.<keywords>

with:

> *<program>* as the main piece of external software used by the job *<function>* as the task being performed and an unlimited number of *<keywords>* that serve to further differentiate the jobtype

To get information about a specific job type from the command line:

```
$ CL_pipeline --job_info <job type>
```

Jobs are written in their own job directories with the format:

<function>/job<nnn>/ with the job number automatically incremented as the project progresses.

Note: A job's directory is also its name, which is used to identify it.

The job's name EX: *AutoPick/job004/ requires* the trailing slash at the end.

2.1.3 Jobs are created from parameter files

Jobs can be created by reading from either or two types of **Parameter Files**: *run.job* or *job.star*

Both files define the jobtype, if the job is new or a continuation of an old job, and the parameters or **JobOptions**.

run.job files are more verbose and easier to manually edit:

```
job_type == relion.autopick.log
is_continue == false
Pixel size in micrographs (A) == 1.02
Pixel size in references (A) == 3.54
...
```

job.star files have a more complicated format but have the advantage that the Pipeliner has functions to dynamically edit them:

```
data_job

_rlnJobType          relion.autopick.log
_rlnJobIsContinue    0

data_joboptions_values
loop_
_rlnJobOptionVariable #1
_rlnJobOptionValue #2
angpix         1.02
angpix_ref     3.54
...
```

Note: *job.star* and *run.job* files can be used interchangeably for almost all applications in a project.

2.1.4 Getting a *run.job* or *job.star* file

A parameter file with the default values for any job can be generated with `default_runjob()` or `default_jobstar()`

API:

```
from pipeliner.api.api_utils import default_runjob, default_jobstar
default_runjob("relion.autopick.log")
default_jobstar("relion.autopick.log")
```

Command line:

```
$ CL_pipeline --default_runjob <job type>
$ CL_pipeline --default_jobstar <job type>
```

This will create the files *relion_autopick_log_job.star* and *relion_autopick_log_run.job*

2.1.5 Running a job

With the paramter file created the job can now be run with *run_job()*:

API:

```
my_project.run_job("relion_autopick_log_job.star")
```

Command line:

```
$ CL_pipeline --run_job relion_autopick_log_job.star
```

This will create and run the job *AutoPick/job001/*

2.1.6 Continuing a job

Some jobs can be continued from where they finished. When a job is run a file *continue_job.star* is written in its job directory. This file contains only the parameters that are allowed to be modified when the job is continued. Edit this file if any parameters need to be changed and then continue the job with:

API:

```
my_project.continue_job("AutoPick/job001/")
```

Command line:

```
$ CL_pipeline --continue_job AutoPick/job001/
```

Note: The job's full name was used to continue it, *not* the name of the *continue_job.star* file

2.1.7 Modifying a parameter file

The python API can modify *job.star* parameter files on-the-fly using *edit_jobstar()*. This avoids manual editing of the parameter files when stringing together multiple jobs:

```
from pipeliner.api.api_utils import edit_jobstar

movie_jobstar = my_project.write_default_jobstar("relion.import.movies")
edit_jobstar(movie_jobstar, {"fn_in_raw": "Movies/*.mrcs"})
movie_job_name = my_project.run_job(movie_jobstar)

mocorr_jobstar = my_project.write_default_jobstar("relion.motioncorr.own")
edit_jobstar(mocorr_jobstar, {"fn_in": movie_job_name + "movies.star"})
mocorr_job_name = my_project.run_job(mocorr_jobstar)
```

2.1.8 Running schedules

Scheduling allows for sets of jobs to be run multiple times via *schedule_job()* and *run_schedule()*

Note: When a job is scheduled placeholder files are created for all of its outputs so these files can be used as if they already exist.

Here is running the same jobs as above, except using the scheduling functions to run the set of import and motion correction jobs 10 times:

API:

```
movie_jobstar = my_project.write_default_jobstar("relion.import.movies")
edit_jobstar(movie_jobstar, {"fn_in_raw": "Movies/*.mrcs"})
movie_job_name = my_project.schedule_job(movie_jobstar)

mocorr_jobstar = my_project.write_default_jobstar("relion.motioncorr.own")
edit_jobstar(mocorr_jobstar, {"fn_in": movie_job_name + "movies.star"})
mocorr_job_name = my_project.schedule_job(mocorr_jobstar)

my_project.run_schedule(
        fn_sched="my_schedule",
        job_ids=[movie_job_name, mocorr_job_name],
        nr_repeat=10,
        )
```

To accomplish this from the command line the parameter files for the Import and MotionCorr jobs must already have been created

```
$ CL_pipeline --schedule_job <import job param file>
$ CL_pipeline --schedule_job <motion corr job param file>
$ CL_pipeline --run_schedule --name my_schedule --jobs job001 job002 --nr_repeat 10
```

Note: The command line tool intelligently parses job names, so for the job named *Import/job001/* it would accept *job001* or *1* as well as the full job name

2.1.9 Other job tools

A variety of other tool exist for modifying jobs in the project. See the api documentation for how to use these functions:

- *set_alias* - Give a job an more descriptive name
- *run_cleanup* - Move intermediate files from jobs into the trash to save disk space
- *delete_job* - Move a job to the trash
- *undelete_job* - Remove a job from the trash and restore it to the project
- *empty_trash* - Permanently delete files in the trash
- *draw_flowcharts* - Draw flowcharts for visualising the workflow of a project
- *get_job_metadata* - Get metadata about a specific job
- *get_network_metadata* - Get metadata about an entire project

- *create_archive* - Make archives to for storing and reproducing projects

CHAPTER

THREE

PIPELINER API

3.1 CCPEM-Pipeliner API

The pipeliner api provides access to all of the main functions of the pipeliner

3.1.1 PipelinerProject

To interact with a pipeliner project it must be created as a *PipelinerProject* object

class pipeliner.api.manage_project.**PipelinerProject**(*pipeline_name: str = 'default'*, *project_name: Optional[str] = None*, *description: Optional[str] = None*)

Bases: object

This class forms the basis for a project.

pipeline_name

The name of the pipeline. Defaults to *default* if not set. There is really no good reason to give the pipeline any other name.

Type str

pipeline

The ProjectGraph containing all the info about the project

Type *ProjectGraph*

project_name

A short descriptive name for the project, editable by the user

Type str

description

A verbose description of the project, editable by the user

Type str

cleanup_all(*harsh: bool = False*) → bool

Runs cleanup on all jobs in a project

Parameters **harsh** (*bool*) – Should harsh cleaning be performed?

Returns True

compare_job_parameters(*jobs_list: List[str]*) → dict

Compare the running parameters of multiple jobs

Parameters **jobs_list** (*list*) – The jobs to compare

Returns {parameter: [value, value, value]}

Return type dict

Raises

- **ValueError** – If any of the jobs is not found
- **ValueError** – If the jobs being compared are not of the same type

continue_job(*job_to_continue: str*, *wait_for_queued: bool = True*, *comment: Optional[str] = None*) → str

Continue a job that has already been run

To change the parameters in a continuation the user needs to edit the continue_job.star file in the job's directory

Parameters

- **job_to_continue** (*str*) – The name of the job to continue
- **wait_for_queued** (*bool*) – If the job is being sent to a queue, should the pipeliner wait for the job to finish before starting the next job. Jobs run locally always wait for the job to finish before the next is started
- **comments** (*str*) – Comments for the job's jobinfo file

Returns The name of the job that will be continued

Return type str

Raises

- **ValueError** – If the continue_job.star file is not found and there is no job.star file in the job's directory to use as a backup
- **ValueError** – If the job is of a type that needs a optimizer file to continue and this file is not found
- **ValueError** – The job has iterations but the parameters specified would result in no additional iterations being run

create_archive(*job: str*, *full: bool = False*, *tar: bool = True*) → str

Creates an archive

Archives can be full or simple. Simple archives contain the directory structure of the project, the parameter files for each job and a script to rerun the project through the terminal job. The full archive contains the full job dirs for the terminal job and all of its children

Parameters

- **job** (*str*) – The name of the terminal job in the workflow
- **full** (*bool*) – If `True` a full archive is written else a simple archive is written
- **tar** (*bool*) – Should the newly written archive be compressed?

Returns A message telling the type of archive and its name

Return type str

delete_job(*job: str*) → bool

Delete a job

Removes the job from the main project and moves it and its children it to the Trash

Parameters **job** (*str*) – The name of the job to be deleted

Returns `True` If a job was deleted, `False` if no jobs were deleted

Return type bool

draw_flowcharts(*job: Optional[str] = None*, *do_upstream: bool = False*, *do_downstream: bool = False*, *do_full: bool = False*, *save: bool = False*, *show: bool = False*) → Union[bool, Tuple[Optional[str], Optional[str], Optional[str]]]

Prepare flowcharts for visulizing a a project

Parameters

- **job** (*str*) – The job for the flowchart to start or end on.
- **do_upstream** (*bool*) – Should an upstream flowchart from the job be prepared?
- **do_downstream** (*bool*) – Should a downstream flowchart from the job be prepared?
- **do_full** (*bool*) – Should a full project flowchart be drawn?
- **save** (*bool*) – Should the flowchart be saved as a file?
- **show** (*bool*) – Should an interactive flowchart be shown?

Returns

Names of the upstream, downstream, and full flowchart files written.

The entry is `"Not saved"` if the flowchart was drawn but only used the interactive viewer and `None` no flowchart was draw for this option

Return type tuple

edit_comment(*job_name: str*, *comment: Optional[str] = None*, *overwrite: bool = False*, *new_rank: Optional[int] = None*)

Edit the comment of a job

Parameters

- **job_name** (*str*) – The name of the job to eddit the comment for
- **comment** (*str*) – The comment to add/append
- **overwrite** (*bool*) – if *True* overwrites otiginal comment, otherwise appends it to the current comment
- **new_rank** (*int*) – New rank to assign to job, use -1 to revert the rank to *None*

Raises **ValueError** – If the new rank is not *None* or an integer

empty_trash()

Deletes all the files and dirs in the Trash directory

Returns `True` if any files were deleted, `False` If no files were deleted

Return type bool

find_job_by_comment(*contains: Optional[List[str]] = None*, *not_contains: Optional[List[str]] = None*, *job_type: Optional[str] = None*, *command: bool = False*) → List[str]

Find Jobs by their comments or command

Parameters

- **contains** (*list*) – Find jobs that contain all of the strings in this list
- **not_contains** (*list*) – Find jobs that do not contain any of these strings
- **job_type** (*str*) – Only consider jobs who's type contain this string
- **command** (*bool*) – If *True* searches the job's command history rather than its comments

Returns Names of all the jobs found

Return type list

Raises **ValueError** – If nothing is specified for contains and not_contains

find_job_by_rank(*equals: Optional[int] = None*, *less_than: Optional[int] = None*, *greater_than: Optional[int] = None*, *job_type: Optional[str] = None*) → List[str]

Find jobs by their rank

Ignores jobs that are unranked

Parameters

- **equals** (*int*) – Find jobs with this exact rank
- **less_than** (*int*) – Find jobs with ranks less then this number
- **greater_than** (*int*) – Find jobs with ranks higher than this number
- **job_type** (*str*) – Only consider jobs that contain this string in their job type

Returns Names of the matching jobs

Return type list

Raises

- **ValueError** – If nothing is specified to search for
- **ValueError** – If both equals and less_than/greater than are specified

get_job_metadata(*jobname: str*, *output_name: Optional[str] = None*) → dict

Runs the gather_metadata function of a single job

Parameters

- **jobname** – The name of the job to run on
- **output_name** – File to write json to. If `None`, no file is written

Returns Metadata dict for for the job

Return type dict

get_job_runtime(*job: str*) → tuple

Returns info about how long a job took to run

Parameters **job** (*str*) – The name of the job to run on

Returns

total times and list of steps and their times (total_real, total_user, total_sys, {step: (real, user, sys)}, job status)

Return type Tuple

`get_network_metadata`(*jobname: str*, *output_name: Optional[str] = None*) → dict

Returns a full metadata trace for a job and all upstream jobs

Parameters

- **jobname** – The name of the job to run on
- **output_name** – File to write json to. If None, no file is written

Returns Metadata dict for all the jobs

Return type dict

`initialize_existing_project`()

make sure the pipeline for the project is current

This function is called by most other functions before running. There is usually no need to call it directly.

`parse_proclist`(*list_o_procs: list*, *search_trash: bool = False*) → list

Finds full process names for multiple processes

Returns full process names IE: *Import/job001/* from *job001* or *1*

Parameters

- **list_o_procs** (*list*) – A list of string process names
- **search_trash** (*bool*) – Should the trash also be search?

Returns All of the full process names

Return type list

`parse_procname`(*in_proc: str*, *search_trash: bool = False*) → Optional[str]

Find process name with the ability for parse ambigious input.

Returns full process names IE: *Import/job001/* from *job001* or *1* Can look in both active processes and the Trash Can, accepts inputs containing only job number and process type and alias IE *Import/my_alias*

Parameters

- **in_proc** (*str*) – The text that is being checked against the list of
- **search_trash** (*processes*) – Should it return the process name if the process is in the trash?

Returns the process name

Return type str

Raises

- **ValueError** – if the process was in the trash but search_trash is false
- **ValueError** – if the process name is not in the pipeliner format, jobxxx, or a number. IE: An unrelated string
- **ValueError** – if the process name is not found

`run_cleanup`(*jobs: list*, *harsh: bool = False*) → bool

Run the cleanup function for multiple jobs

Each job defines its own method for cleanup and harsh cleanup

Parameters

- **jobs** (*list*) – List of string job names to operate on
- **harsh** (*bool*) – Should harsh cleaning be performed

Returns `True` if cleanup is successful, otherwise `False`

Return type bool

run_job(*jobinput: Union[str, dict,* pipeliner.pipeliner_job.PipelinerJob*]*, *overwrite: Optional[str] = None*, *wait_for_queued: bool = True*, *comment: Optional[str] = None*) → str

Run a new job in the project

If a file is specified the job will be created from the parameters in that file If a dict is input the job will be created with defaults for all options except those specified in the dict.

If a dict is used for input it MUST contain at minimum {"_rlnJobTypeLabel": <the jobtype>}

Parameters

- **jobinput** (*str*, *dict*, PipelinerJob) – The path to a run.job or job.star file that defines the parameters for the job or a dict specifying job parameters or a PipelinerJob object
- **overwrite** (*str*) – The name of a job to overwrite, if `None` a new job will be created. A job can only be overwritten by a job of the same type
- **wait_for_queued** (*bool*) – If the job is being sent to a queue, should the pipeliner wait for the job to finish before starting the next job. Jobs run locally always wait for the job to finish before the next is started
- **comment** (*str*) – Comments to be added to the job's info file

Returns The name of the job that was run

Return type str

Raises `ValueError` – If this method is used to continue a job

run_schedule(*fn_sched: str*, *job_ids: List[str]*, *nr_repeat: int = 1*, *minutes_wait: int = 0*, *minutes_wait_before: int = 0*, *seconds_wait_after: int = 5*) → str

Runs a list of scheduled jobs

Parameters

- **fn_sched** (*str*) – A name to assign to the schedule
- **job_ids** (*list*) – A list of string job names to run
- **nr_repeat** (*int*) – Number of times to repeat the entire schedule
- **minutes_wait** (*int*) – Minimum number of minutes to wait between running each subsequent job
- **minutes_wait_before** (*int*) – Initial number of minutes to wait before starting to run the schedules.
- **seconds_wait_after** (*int*) – Time to wait after running each job

Returns The name of the schedule that is run

Return type str

Raises `ValueError` – If the schedule name is already in use

schedule_continue_job(*job_to_continue: str*, *params_dict: Optional[dict] = None*, *comments: Optional[str] = None*) → str

Schedule a job to run

Adds the job to the pipeline with scheduled status, does not run it

Parameters

- **job_to_continue** (*str*) – the name of the job to continue
- **params_dict** (*dict*) – Parameters to change in the continuation job.star file. {param name: value}
- **comments** (*str*) – comments to add to the job's jobinfo file

Returns The name of the scheduled job

Return type str

schedule_job(*job_input: str*, *comment: Optional[str] = None*) → str

Schedule a job to run

Adds the job to the pipeline with scheduled status, does not run it

Parameters

- **job_input** (*str*) – The path to a run.job or job.star file that defines the parameters for the job or a dictionary containing job parameters
- **comments** (*str*) – Comments to put in the job's jobinfo file

Returns The name of the scheduled job

Return type str

set_alias(*job: str*, *new_alias: str*) → bool

Set the alias for a job

Parameters

- **job** (*str*) – The name of the job to set the alias for
- **new_alias** (*str*) – The new alias

Returns `True` if the alias was changed, else `False`

Return type bool

stop_schedule(*schedule_name: str*) → bool

Stops a currently running schedule

Kills the process running the schedule and marks the currently running job as aborted. Works to stop schedules that were started using the RELION GUI or pipeliner.

Parameters **schedule_name** (*str*) – The name of the schedule to stop

Returns `True` If the schedule was stopped, `False` if the schedule could not be found to stop

Return type bool

undelete_job(*job: str*) → bool

Restores a job from the Trash back into the project

Also restores the job's alias if one existed

Parameters **job** (*str*) – The job to undelete

Returns `True` If a job was restored, otherwise `False`

Return type bool

update_job_status(*job: str*, *new_status: str*) → bool

Mark a job as finished, failed or aborted

If is_failed and is_aborted are both `False` the job is marked as finished.

Parameters

- **job** (*str*) – The name of the job to update
- **new_status** (*str*) – The new status for the job; Choose from *"Running"*, , *"Scheduled"*, *"Succeeded"*, *"Failed* or *"Aborted"*. Status names are not case sensitive

Returns `True` if the status was updated, otherwise `False`

Return type bool

Raises **ValueError** – If the new status is not one of the options

pipeliner.api.manage_project.**convert_pipeline**(*pipeline_file: str*) → bool

Converts a pipeline file from the RELION 2.0-3.1 format

This format has integer node, process, and status IDs. The pipeliner format uses string IDs

Parameters **pipeline_file** (*str*) – The name of the file to be converted

Returns

The result of the conversion

`True` if the pipeline was converted, `False` if the pipeline was already in pipeliner format

Return type bool

pipeliner.api.manage_project.**get_commands_and_nodes**(*job_file: str*) → tuple

Tell what commands a job file would return and nodes that would be created

Parameters **job_file** (*str*) – The path to a run.job or job.star file

Returns

Three lists

- A list of commands. Each item in the commands list is a list of commands arguments. IE: `[[com1-arg1, com1-arg2],[com2-arg1]]`
- A list of input nodes that would be created. Each item in the list is a tuple: `[(name, type), (name, type)]`
- A list of output nodes that would be created. Each item in the list is a tuple: `[(name, type), (name, type)]`

Return type tuple

pipeliner.api.manage_project.**look_for_project**(*pipeline_name: str = 'default'*) → Optional[dict]

See if a pipeliner project exists in the current directory

Parameters **pipeline_name** (*str*) – The name of the pipeline to look for. This is the same as the pipeline file name with _pipeline.star removed

Returns

(bool:was the pipeline found?, dict info about the project)

Return type tuple

3.1.2 api_utilities

Utility functions do not require an existing project

pipeliner.api.api_utils.**edit_jobstar**(*fn_template: str*, *params_to_change: dict*, *out_fn: str*) → str

Modify one or more parameters in a job.star file

Parameters

- **fn_template** (*str*) – The name of the job.star file to use as a template
- **params_to_change** (*dict*) – The parameters to change in the format `{param_name: new_value}`
- **out_fn** (*str*) – Name for the new file to be written

Returns The name of the output file written

Return type str

pipeliner.api.api_utils.**get_available_jobs**(*search_term: str = '*ALL*'*) → List[Tuple[str, bool, *pipeliner.pipeliner_job.JobInfo*]]

Returns all the available job types and info about them

Parameters **search_term** (*str*) – Only return jobs with this string in their jobtype name

Returns

A list with a tuple for each job; (jobtype, can it run?, JobInfo object) (`str`, `bool`, *pipeliner.pipeliner_job.JobInfo*)

Return type list

pipeliner.api.api_utils.**get_job_info**(*job_type: str*) → Optional[*pipeliner.pipeliner_job.JobInfo*]

Get information about a job

Parameters **job_type** (*str*) – The type of job to return info on

Returns JobInfo object with info about the job and it's references

Return type *JobInfo*

pipeliner.api.api_utils.**job_parameters_dict**(*jobtype: str*) → dict

Get dictionary of a job's parameters

Parameters **jobtype** (*str*) – The type of job to get the dict for

Returns

The parameters dict. Suitable for running a job from `run_job()`

Return type dict

pipeliner.api.api_utils.**job_success**(*job_name: str*, *search_time: float = 0*, *raise_error: bool = False*, *error_message: str = ''*) → bool

Check that a finished job has produced the expected control files

Parameters

- **job_name** (*str*) – The name of the job to be checked in the format JobType/jobxxx/
- **search_time** (*float*) – Time in minutes to spend looking for the control files before givng up
- **raise_error** (*bool*) – Should an error be raised if the job is failed/aborted rather than returning a bool

- **error_message** (*str*) – Additional text (if any) to print before the error reason

Returns

***True* if the job was successful, *False* if the job was failed, aborted,** or no control files have appeared after the set search time and *raise_error* is *False*

Return type bool

Raises

- **RuntimeError** – If the job was failed, aborted, or no control files have appeared
- **after the set search time and raise_error is True** –

pipeliner.api.api_utils.**validate_starfile**(*fn_in: str*)

Checks for inappropriate use of reserved words in starfiles

Writes a corrected version with proper quotation if possible using the *StarfileCheck* class

Parameters **fn_in** (*str*) – The name of the file to check

pipeliner.api.api_utils.**write_default_jobstar**(*job_type: str*, *out_fn: Optional[str] = None*, *relionstyle: bool = False*)

Write a job.star file for the specified type of job

The default jobstar contains all the job options with their values set as the defaults

Parameters

- **job_type** (*str*) – The type of job
- **out_fn** (*str*) – Name of the file to write the output to. If left blank defaults to *<job_type>_job.star*
- **relionstyle** (*bool*) – Should the job.star files be written in the relion format? Relion files are compatible with the pipeliner, but the pipeliner versions are not back compatible with Relion. If this option is selected a Relion job type should be used for job_type

Returns The name of the output file written

Return type str

pipeliner.api.api_utils.**write_default_runjob**(*job_type: str*, *out_fn: Optional[str] = None*) → str

Write a run.job file for the specified type of job

The default runjob contains all the job option labels with their values set as the defaults

Parameters

- **job_type** (*str*) – The type of job
- **out_fn** (*str*) – Name of the file to write the output to. If left blank defaults to *<job_type>_run.job*

Returns The name of the output file written

Return type str

CHAPTER

FOUR

COMAND LINE TOOLS

4.1 Command Line Tools

CL_pipeline allows pipeliner functions to be run from the UNIX command line

CCPEM Pipeliner command line utility

```
usage: CL_pipeline [-h] [--new_project [project name]]
                   [--available_jobs [search string]]
                   [--run_job [run.job or job.star file]]
                   [--overwrite [job name]] [--continue_job [job name]]
                   [--create_interactive_job]
                   [--print_command [run.job or job.star file]]
                   [--schedule_job [run.job, job.star file or job name]]
                   [--run_schedule] [--name [NAME]]
                   [--jobs [job name [job name ...]]] [--min_between [n]]
                   [--nr_repeats [1]] [--wait_sec_after [2]]
                   [--wait_min_before [0]] [--stop_schedule [STOP_SCHEDULE]]
                   [--delete_job [job name]] [--undelete_job [job name]]
                   [--set_alias [job name] [new alias]]
                   [--clear_alias [job name]] [--set_status [job name]
                   {finished, failed, aborted}]
                   [--cleanup [job name [job name ...]]] [--harsh]
                   [--validate_starfile [star file]]
                   [--convert_pipeline_file [_pipeline.star file]]
                   [--default_jobstar [job type]] [--relionstyle]
                   [--default_runjob [job type]] [--job_info [job type]]
                   [--empty_trash] [--job_runtime [job name]]
                   [--metadata_report [terminal job]]
                   [--draw_flowchart [job name]] [--upstream] [--downstream]
                   [--interactive] [--full_archive [terminal job name]]
                   [--simple_archive [terminal job name]]
```

4.1.1 Arguments

--new_project Initialize a new project in this directory. Project name is optional, if none is specified the project will be called 'default' (recommended)

--available_jobs Show a list of all available jobs, optionally add a search string to limit the search, Leave blank to show all available job types

4.1.2 Running jobs

--run_job Create a job using a run.job or job.star file to get the parameters

--overwrite Use with –run_job to overwrite a current job rather than making a new job. Jobs can only be overwritten with the same job type

--continue_job Continue a job that has been previously run.Edit the job's continue_job.star file to modify parameters for the continuation

--create_interactive_job Create a job.star file for any job type with an interactive dialog. Useful for situations where the GUI cannot open

Default: False

--print_command Read a job.star or run.job file and print the command it would produce

--schedule_job Add a job to list of scheduled jobs using a run.job or job.star file to get the parameters. If followed by a job.star file it will create a new job if followed by a job name it will schedule a continuation of that job

4.1.3 Executing Schedules

--run_schedule Create a schedule choosing from the currently scheduled jobs and run it

Default: False

--name (required) Enter a name for the new schedule

--jobs (required) Enter the jobs that will be run. Make sure to list the jobs in the order which they should be run

--min_between (optional) Wait at least this many minutes between jobs

Default: 0

--nr_repeats (optional) Repeat the schedule this many times

Default: 1

--wait_sec_after (optional) Wait this many seconds after finishing before starting the next job

Default: 2

--wait_min_before (optional) Wait this many minutes before starting the schedule

Default: 0

--stop_schedule (required) Enter a name for the schedule to be stopped

4.1.4 Deleting jobs

--delete_job Remove job(s) and put in the trash, deletes the job and all of its child processes

4.1.5 Undeleting jobs

--undelete_job Restore a deleted job and any of its deleted parent processes from the trash

4.1.6 Modifying jobs

--set_alias Set the alias of a job

--clear_alias Clear the alias of a job

--set_status Set the status of a job; choose from 'finished, failed, or aborted

4.1.7 Cleaning Up Job(s)

--cleanup Delete intermediate files from these job(s) to save disk space; enter ALL to clean up all jobs

--harsh Add this argument to –cleanup to delete even more files

Default: False

4.1.8 Utilities

--validate_starfile Check a star file and make sure it is written in the correct format. If errors are found will attempt to fix them

--convert_pipeline_file Convert a pipeline file from Relion 3.1 to the CCPEM pipeliner format

--default_jobstar Make a job.star file with the default values for a specific job type for use with the ccpem-pipeliner

--relionstyle OPTIONAL: Add this argument to –default_jobstar to write job.star files which are compatible with RELION 4.0. RELIONstyle job.star files are fully compatible with the pipeliner, pipeliner job.star files may havedifferences that cause bugs in RELION

Default: False

--default_runjob Make a _run.job file with the default values for a specific job type. These files can also be used to run jobs and a more human readable

--job_info Get info about a specific job type, including any reference(s)

--empty_trash Delete the files in the trash. THIS CANNOT BE UNDONE!

Default: False

4.1.9 Project Analysis

--job_runtime Measure how long the steps in a job took to complete

--metadata_report Prepares a report in .json format for the terminal job and allof its parent jobs

--draw_flowchart Draw a flowcharts for the pipeline. If used alone; draws the entire pipeline, if followed by a job name; draws upstream and downstream flowcharts for that job. Saves the output as jobxxx_upstream.png unless the –interactive argument is also included

--upstream [optional] Add this argument to –draw_flowchart to only draw the upstream flowchart for the specified job

Default: False

--downstream [optional] Add this argument to –draw_flowchart to only draw the downstream flowchart for the specified job

Default: False

--interactive [optional] Add this argument to –draw_flowchart to show interactive flowcharts

Default: False

4.1.10 Project Archiving

--full_archive Create a full archive for a project. This will contain the entire job dirs for the stated terminal job and all of its parents

--simple_archive Create a simple archive for a project. This will contain just the directory structure and parameter files for the stated terminal job andall of it's parents along with a script to automatically re-run the project through the terminal job

CHAPTER

FIVE

ADDITIONAL TOOLS

5.1 Pipeliner benchmarking tool

The benchmarking tool uses the ccpem pipeliner to run multiple jobs whilst varying specific parameters so the effects of the parameters on the results and running times can be compared.

5.1.1 Start by generating the template files

Generate a job.star template file for each job to run.

This can be done by taking a job.star file from the job directory of a previous run, or buy using the commandline tool:

```
CL_pipeline --default_jobstar <job type>
```

5.1.2 Edit the template files

Set all of the parameters for the jobs to be run by editing the template files.

To test multiple values for a parameter put all the values to try as a comma separated list (with no spaces) enclosed in square brackets

IE: To try multiple values for number of grouped frames in a motion corr find the line:

```
group_frames 1
```

and change it to:

```
group_frames [1,2,3,4,5,6,7,8]
```

Make sure to set the running parameters for all the jobs correctly. If all jobs that are able are sent to a queue, then they can be run in parallel and the benchmarking will complete much more quickly.

Warning: If multiple parameters have more than on test value the benchmark tool will make a job for every possible permutation, which could lead to A LOT of jobs being run.

5.1.3 Organise the template files

Once the template files are edited they should all be placed in a single directory called `benchmark_templates`. Rename them so the benchmark tool knows what order to run them in. EX:

`import_job.star` `motioncorr_job.star` `ctffind_job.star`

should be renamed to:

`01_import_job.star` `02_motioncorr_job.star` `03_ctffind_job.star`

5.1.4 Get your data

Place any data that will be needed by the jobs (IE not produced by a previous job) in a directory called *benchmark_data*.

These files will be symlinked into the running directory of each benchmark job.

These files will be at the top of the job directory so if a job needs one of them as input just the file name should be used as the parameter value.

IE:

`'fn_in' my_particles_data.star`

5.1.5 Run the benchmark tool

use the command `pipeliner_benchmark_tool`

The tool will set up the directory structure and templates and determine if a parallel run can be executed.

Note: Parallel execution will only be allowed if all computationally intensive jobs are being sent to a queue.

The tool will create a directory for each permutation of the test parameters starting with `bmrk000`

The results of the benchmarking tests are written in two files

`benchmark_summary.csv`: Contains run times for each job in the benchmarks along with a total run time for each benchmark.

`benchmark_full.csv`: Contains run times for each individual command in the jobs in the benchmarks.

CHAPTER

SIX

CORE MODULES

6.1 Pipeline Tools

6.1.1 Nodes and Processes

class pipeliner.data_structure.**Node**(*name: str*, *n_type: str*)

Bases: object

Nodes store info about input and output files of jobs that have been run

name

The name of the file the node represents

Type str

type

The node type

Type str

kwds

Keywords associated with the node

Type list

ext

The node file's extension

Type str

output_from_process

The Process object for process that created the file

Type *Process*

input_for_processes_list

A list of *Process* objects for processes that use the node as an input

Type list

clear() → None

Clear the input and output processes from a Node

class pipeliner.data_structure.**Process**(*name: str*, *p_type: str*, *status: str*, *alias: Optional[str] = None*)

Bases: object

A Process represents a job that has been run by the Pipeliner

name

The name of the process. It should be in the format *"<jobtype>/jobxxx/"*. The trailing slash is required

Type str

alias

An alternate name for the process to make it easier to identify

Type str

outdir

The directory the process was written into

Type str

p_type

The process' type

Type str

status

The processes' status 'running, scheduled, successful, failed, or aborted'

Type str

input_nodes

Node objects for files the process use as inputs

Type list

output_nodes

Node objects for files the process produces

Type list

clear()

6.1.2 ProjectGraph

class pipeliner.project_graph.**ProjectGraph**(*name: str = 'default'*, *do_read_only: bool = False*)

Bases: object

The main ProjectGraph object is used for manipulating the pipeline

name

The name of the pipeline. *_pipeline.star* is added to the name to generate the pipeline file name

Type str

node_list

A *Node* object for every file that is an input or output for a job in the project

Type list

process_list

A *Process* object for each job in the project

Type list

job_counter

The number of the *next* job in the project IE: If there are 10 jobs in a project job_counter is 11

Type int

do_read_only

If this current instatiation of the object should be able to make changes to the pipeline

Type bool

add_job(*job:* pipeliner.pipeliner_job.PipelinerJob, *as_status: str*, *do_overwrite: bool*) → *pipeliner.data_structure.Process*

Add a job to the pipeline

Adds the *Process* for the job, a *Node* for each of its input and output files, and writes a mini-pipeline containing just that job

Parameters

- **job** (*PipelinerJob*) – The job to add.
- **as_status** (*str*) – The status of the job in the pipeline
- **do_overwrite** (*bool*) – If the job already exists, should it be overwritten?

Returns The *Process* for the new job

Return type *Process*

add_new_input_edge(*node:* pipeliner.data_structure.Node, *process:* pipeliner.data_structure.Process)

Add a *Node* to a *Process* as in input

Parameters

- **node** (*Node*) – The node to add
- **process** (*Process*) – The Process to add the Node to

add_new_output_edge(*process:* pipeliner.data_structure.Process, *node:* pipeliner.data_structure.Node, *mini: bool = False*)

Add a *Node* to a *Process* as in output

Parameters

- **node** (*Node*) – The node to add
- **process** (*Process*) – The Process to add the Node to

add_new_process(*process:* pipeliner.data_structure.Process, *do_overwrite: bool*) → Optional[*pipeliner.data_structure.Process*]

Add a *Process* to the pipeline

Parameters

- **process** (*Process*) – The *Process* to add
- **do_overwrite** (*bool*) – If the process already exists should it be overwritten?

Returns The *Process* that was added or the existing *Process* if it already existed

Return type (*Process*)

Raises RuntimeError – If the *Process* already exists and overwrite is False

add_node(*node:* pipeliner.data_structure.Node, *touch_if_not_exists: bool = False*) → Optional[*pipeliner.data_structure.Node*]

Add a *Node* to the pipeline

A node is only added if it doesn't already exist in the pipeline, so if a node is used as an input the keywords from the process that wrote the node will overrule any added in the node's definition from the process that used it as input

Parameters

- **node** (*Node*) – The node to add
- **touch_if_not_exists** (*bool*) – If the file for the node does not exist should it be created?

Returns The Node that was added. If this node already existed, returns the existing copy

Return type pipeline.data_structure.Node

Raises RuntimeError – If the node name is empty

check_lock(*lock_expected: bool*, *wait_time: int = 2*)

Checks to see if a lock file exists for the pipeline

Parameters

- **lock_expected** (*bool*) – Is a lock file expected to exist?
- **wait_time** (*int*) – The number of minutes to continue trying whilst waiting for a lock file to appear/disappear based of if one is expected

Raises

- RuntimeError – If no lock is expected, but a lock file is still found after waiting <wait_time> minutes for it to disappear
- RuntimeError – If a lock file is expected but not found after waiting <wait_time> minutes for it to appear

check_process_completion() → bool

Check to see if a any processes have finished running, update their status

Returns True if any statuses have been updated, else False

Return type bool

clean_up_job(*process:* pipeliner.data_structure.Process, *do_harsh: bool*) → bool

Cleans up a job by deleting intermediate files

Gets a list of files to delete from the specific job's cleanup function. First checks that none of the files that are slated for deletion are on the Node list or are of a few specific types that RELION needs. Then moves all the intermediate files to the trash

There are two tiers of clean up 'harsh' and normal. Each job defines what files are cleaned up by each cleanup type

Parameters

- **process** (*Process*) – The *Process* of the job to clean up
- **do_harsh** (*bool*) – Should harsh cleaning be performed?

Returns True if cleaning was performed False if the specified job had no clean up method or it was protected from harsh cleaning

Return type bool

cleanup_all_jobs(*do_harsh: bool*) → int

Clean up all jobs in the project

Parameters **do_harsh** (*bool*) – Should harsh cleaning be performed?

Returns The number of jobs that were successfully cleaned

Return type int

clear()

Clear the node and process lists and set the job counter to 1

create_lock(*wait_time: int = 2*)

Lock the pipeline

Creates a directory called .relion_lock and the file .relion_lock/lock_<pipeline_name>_pipeline.star. The pipeline cannot be edited when this file is present.

Parameters **wait_time** (*int*) – The number of minutes to continue trying to make the lock directory/file if a problem is encountered.

Raises

- **RuntimeError** – If a permission error is encountered trying to create or read the .relion_lock directory
- **RuntimeError** – If the lock file has not appeared after <wait_time> minutes

create_process_display_objs(*proc*)

Create the ResultsDisplay objects for a process and save them

Parameters **Proc** (*Process*) – The process to operate on

Returns The DisplayObjects for that process

Return type list

delete_job(*this_job: pipeliner.data_structure.Process*)

Remove a job from the pipeline

:param this_job (*Process*:): The job to remove

delete_node(*node: pipeliner.data_structure.Node*)

Remove a node from the pipeline

Also removes any edges that contain this node

Parameters **node** (*Node*) – The node to remove

delete_temp_node_file(*node: pipeliner.data_structure.Node*) → bool

Remove files associated with a *Node*

Also removes the directory if it is empty

Parameters **node** (*Node*) – The node to remove the file for

Returns `True` if files were deleted, Otherwise `False`

Return type bool

delete_temp_node_files(*process:* pipeliner.data_structure.Process) → bool

Delete all the files for the nodes in a specific *Process*

Parameters **process** (*Process*) – The Process to create the files for

Returns `True` if files were removed, otherwise `False`

Return type bool

export_all_scheduled_jobs(*mydir: str*) → bool

Exports all scheduled jobs in the pipeline

This function is part of the RELION import/export system which is not used by the pipeliner and will probably be removed

Parameters **mydir** (*str*) – The name of the export directory to be created. It will be written as ExportJobs/<mydir>

Returns Were any jobs exported?

Return type bool

find_immediate_child_processes(*process:* pipeliner.data_structure.Process) → list

Find just the immediate child processes of a process

Parameters **process** (*Process*) – The process to find children for

Returns The *Process* object for each job connected to the input *Process*

Return type list

find_node(*name: str*) → Optional[*pipeliner.data_structure.Node*]

Retrieve the *Node* object for a file

Parameters **name** (*str*) – The name of the file to get the node for

Returns The files's *Node* object. `None` if the file is not found.

Return type *Node*

find_process(*name_or_alias: str*) → Optional[*pipeliner.data_structure.Process*]

Retrieve the *Process* object for a job in the pipeline

Parameters **name_or_alias** (*str*) – The job name or its alias

Returns The job's *Process*

Return type `pipeliner.data_structre.Process`

Raises **RuntimeError** – If multiple processes with the same name are found

get_downstream_network(*process:* pipeliner.data_structure.Process) → List[Tuple[Optional[*pipeliner.data_structure.Node*], Optional[*pipeliner.data_structure.Process*], *pipeliner.data_structure.Process*]]

Gets data for drawing a network downstream from process

Parameters **process** (*Process*) – The process to trace

Returns Contains tuple for each edge (node, parent process, chile process) [*Node*, `Process`, *Process*]. Each edge describes one file in the network

Return type list

get_node_name(*node:* pipeliner.data_structure.Node) → str

Get the relative path of a node file with its alias if it exists

This returns the relative path (which is the same as the file name) unless the job that created the node has an alias in which case it returns the file path with the alias instead of <jobtype>/jobxxx/

Parameters **node** (*Node*) – The node to get the name for

Returns The relative path of the file to node points to with the job's alias if applicable

Return type str

get_output_nodes_from_starfile(*process:* pipeliner.data_structure.Process)

Get nodes from an exported job

Reads a RELION_OUTPUT_NODES.star file created when a job is exported and adds these nodes to a *Process*

This function is RELION-specific and will probably be deprecated

Parameters **process** (*Process*) – The process to add the nodes to. It will look in the process's directory for the RELION_OUTPUT_NODES.star file

get_pipeline_edges(*delete_nodes: Optional[List[*pipeliner.data_structure.Node*]] = None*) → Tuple[List[Tuple[str, str]], List[Tuple[str, str]]]

Find the connections between jobs

Get the connections between jobs and nodes for the pipeline if any nodes have been deleted their corresponding edges need to be removed

Parameters **delete_nodes** (*list*) – List of *Node* objects to be deleted from the pipeline

Returns ([input edges], [output edges]) input edges is a list of tuples (process name, input file) output edges is a list of tuples (process name, output file)

Return type tuple

get_pipeline_filename() → str

Get the name of the pipeline file

Returns The name of the pipeline file usually 'default_pipeline.star'

Return type str

get_process_results_display(*proc*, *forceupdate=False*)

Get the ResultsDisplay objects for a process

Atttempts to be as effecient as possible, uses already existing files if they are found

Parameters **forceupdate** (*bool*) – Force an update even if it thinks one is not necessary

get_upstream_network(*process:* pipeliner.data_structure.Process) → List[Tuple[Optional[*pipeliner.data_structure.Node*], Optional[*pipeliner.data_structure.Process*], *pipeliner.data_structure.Process*]]

Gets data for drawing a network upstream from process

Parameters **process** (*Process*) – The process to trace

Returns Contains tuple for each edge (node, parent process, child process) [*Node*, Process, *Process*]. Each edge describes one file in the network

Return type list

get_whole_project_network() → List[Tuple[Optional[*pipeliner.data_structure.Node*], Optional[*pipeliner.data_structure.Process*], *pipeliner.data_structure.Process*]]

Get the edges and nodes for the entire project

Returns Edges [nod type, parent_job, child_job, extra info] set: The names of all nodes

Return type list

import_jobs(*fn_export: str*)

Import a previously exported job

This function is part of the RELION import/export system which is not used by the pipeliner and will probably be removed

Parameters **fn_export** (*str*) – The name of the file created by an export job

is_empty() → bool

Tests of the pipeline is empty

Returns `True` if there are no jobs in the pipeline

Return type bool

prepare_archive(*process:* pipeliner.data_structure.Process, *do_full: bool = False*, *tar: bool = True*) → str

Create a archive for a job

There are two levels of archive:

- A full archive copies the full job directories for the terminal job and all of its parents
- A simple archive just recreates the directory structure, saves the parameter files for each job, and writes a script to re-run the project

Parameters

- **process** (*Process*) – The *Process* object for the terminal job in the project
- **do_full** (*bool*) – Should a full archive be created?
- **tar** (*bool*) – Should the archive be compressed after creation?

Returns

A completion message

It says the archive was created successfully and gives the name of the file created or describes any errors encountered

Return type str

read(*do_lock: bool = False*, *lock_wait: int = 2*) → *pipeliner.jobstar_reader.StarfileCheck*

Read the pipeline

Parameters

- **do_lock** (*bool*) – Should the pipeline be locked upon reading
- **lock_wait** (*int*) – If the pipeline is unable to be locked continue trying for this many minutes

Returns For the pipeline file

Return type *pipeliner.jobstar_reader.StarfileCheck*

Raises AttributeError – If the pipeline file is not found

remake_node_directory()

Erase and rewrite RELION's .Nodes directory

remove_lock()

Remove the lock file for the pipeline

replace_files_for_import_export_of_sched_jobs(*fn_in_dir: str*, *fn_out_dir: str*, *find_pattern: str*, *replace_pattern: str*)

Updates the content of files in jobs that are to be exported

This function is part of the RELION import/export system which is not used by the pipeliner and will probably be removed

Parameters

- **fn_in_dir** (*str*) – The input directory
- **fn_out_dir** (*str*) – The output directory
- **find_pattern** (*str*) – The text to replace
- **replace_pattern** (*str*) – The text to replace it with

set_job_alias(*process: pipeliner.data_structure.Process*, *new_alias: Optional[str]*) → bool

Set a job's alias

Sets the alias in the pipeline and creates the ailas directory, which is a symlink to the original directory

Parameters

- **process** (*Process*) – The *Process* to make an alias for
- **new_alias** (*str*) – The new alias for the job

Returns `True` if the new alias was successfully set

Return type bool

Raises

- **ValueError** – If the *Process* is not found
- **ValueError** – If the new alias is 'None', which is not allowed
- **ValueError** – If the new alias is shorter than 2 characters
- **ValueError** – If the new alias begins with job, which would cause problems
- **ValueError** – If the new alias is not unique

set_name(*name: str*, *new_lock: bool = True*, *overwrite: bool = False*)

Change the name of the pipeline file

Unlocks the old pipeline when the new one is created. This should really not be used as there is no reason to be changing the name of the pipeline

Parameters

- **name** (*str*) – The new name
- **new_lock** (*bool*) – Should the new pipeline be locked?
- **overwrite** (*bool*) – Should the old pipeline be removed?

Raises ValueError – If an attempt is made to change the name to on that already exists

touch_temp_node_file(*node:* pipeliner.data_structure.Node, *touch_if_not_exists: bool*) → bool

Create a placeholder file for a node that will be created later

Parameters

- **node** (*Node*) – The node to create the file for
- **touch_if_not_exists** (*bool*) – Should the file be created if it does not already exist

Returns `True` if a file was created or it already existed `False` if no file was created.

Return type bool

touch_temp_node_files(*process:* pipeliner.data_structure.Process) → bool

Create placeholde files for all nodes in a *Process*

Parameters **process** (*Process*) – The Process to create the files for

Returns `True` if files were created, otherwise `False`

Return type bool

undelete_job(*del_job: str*)

Get job out of the trash and restore it to the pipeline

Also restores any alias the job may have had as long as it does not confilct wit the current aliases

Parameters **del_job** (*str*) – The name of the deleted job

update_lock_message(*lock_message: str*)

Updates the contents of the lockfile for the pipeline

This enables the user to see which process has locked the pipeline

update_status(*the_proc:* pipeliner.data_structure.Process, *new_status: str*)

Mark a job finished

The job can be marked as "Succeeded", "Failed", or "Aborted", "Running" or "Scheduled"

Parameters

- **the_proc** (*Process*) – The *Process* to update the status for
- **new_status** (*str*) – The new status for the job

Returns Was the status changed?

Return type bool

Raises

- **ValueError** – If the new_status is not in the approved list
- **ValueError** – If a job with any other status than 'Running' is marked 'Aborted'
- **ValueError** – If a job's updated status is the same as it's current status

write(*edges: Optional[Tuple[List[Tuple[str, str]], List[Tuple[str, str]]]] = None*, *lockfile_expected: bool = True*, *lock_wait: int = 2*)

Write the pipeline file from the *ProjectGraph* object

If the 'ProjectGraph' is locked it will be unlocked after writing is finished

Parameters

- **edges** (*tuple*) – The pipeline edges as

  [0] A list of tuples (process name, input file)

  [1] A list of tuples (process name, output file)

  If this is left as `None` the edges of the current pipeline are used
- **lockfile_expected** (*bool*) – Does the pipeline expect to be locked?
- **lock_wait** (*int*) – If the locking status of the pipeline is not as expected wait this many minutes for it to resolve

pipeliner.project_graph.**update_jobinfo_file**(*current_proc: pipeliner.data_structure.Process*, *action: str*, *comment: Optional[str] = None*, *command_list: Optional[list] = None*)

Update the file in the jobdir that stores info about the job

Parameters

- **current_proc** (*Process*) – Process object for the job being operated on
- **action** (*str*) – what action was performed on the job IE: Ran, Scheduled, Cleaned up
- **comment** (*str*) – Comment to append to the job's comments list
- **command_list** (*list*) – Commands that were run. Generally *None* if action was any other than Run or Schedule

6.1.3 Metadata Tools

pipeliner.metadata_tools.**format_for_metadata**(*joboption: pipeliner.job_options.JobOption*) → Optional[Union[str, float, int, bool]]

Format data from relion starfiles for JSON.

Changes 'Yes'/'No" to True/False, None for blank vals and removes any quotation marks

Parameters **joboption** (pipeliner.job_options.JobOption) – The joboption to format

Returns

A properly formatted string, float or int or bool

Return type str

Raises **ValueError** – If a boolean job option doesn't have a value compatible with a class:bool"

pipeliner.metadata_tools.**get_job_metadata**(*this_job: pipeliner.data_structure.Process*) → dict

Run a job's metadata gathering method

Parameters **this_job** (*pipeliner.data_structure.Process*) – The *Process* object for the job to gather metadata from

Returns Metadata dict for the job Returns `None` if the job has no metadata gathering function

Return type dict

pipeliner.metadata_tools.**get_metadata_chain**(*pipeline*, *this_job: pipeliner.data_structure.Process*, *output_file: Optional[str] = None*, *full: bool = False*) → dict

Get the metadata for a job and all its upstream jobs

Metadata is gathered by each individual job's gather_metadata function

Parameters

- **pipeline** (*ProjectGraph*) – The current pipeline
- **this_job** (*Process*) – The *Process* object for the terminal job
- **output_file** (*str*) – The name of a file to write the results to; if None no file will be written
- **full** (*bool*) – Should the metadata report also contain information about continuations and multiple runs of the the jobs or just the current one

Returns The metadata dictionary

Return type dict

pipeliner.metadata_tools.**get_reference_list**(*pipeline*, *terminal_job:* pipeliner.data_structure.Process) → dict

Prepares a list of every piece of software used to generate a termianl job

Parameters

- **pipeline** (*ProjectGraph*) – The current pipeline
- **terminal_job** (*Process*) – The *Process* object for the terminal job

Returns The reference list{job_name: ([programs, used], [references])}

Return type dict

pipeliner.metadata_tools.**make_default_results_schema**(*job_type: str*) → dict

Write the json schema for the results of a job if one was not provided

The metadata schema have two parts: the running parameters part is generated automatically and the results part is written by the user. If no schema was provided for the results this function is used to write a blank plackholder one

Parameters **job_type** (*str*) – The job type to write the schema for

Returns The json schema ready to be dumped to json file

Return type dict

pipeliner.metadata_tools.**make_job_parameters_schema**(*job_type: str*) → dict

Write the json schema for the running parameters of a job

The metadata schema have two parts: the running parameters part is generated automatically and the results part is written by the user

Parameters **job_type** (*str*) – The job type to write the schema for

Returns The json schema ready to be dumped to json file

Return type dict

6.1.4 Pipeline Visualisation

Tools for visualising projects. An installation of pygraphviz, which is dependent on GraphVis is necessary. See the README for info about installation of GraphViz

class pipeliner.flowchart_illustration.**ProcessFlowchart**(*pipeline: Optional[pipeliner.project_graph.ProjectGraph] = None*, *process: Optional[pipeliner.data_structure.Process] = None*, *do_upstream: bool = False*, *do_downstream: bool = False*, *do_full: bool = False*, *save: bool = False*, *show: bool = False*)

Bases: object

A class for drawing pretty flowcharts for pipeliner projects

pipeline

The pipeline to use in drawing the flowchart

Type *ProjectGraph*

process

The process to start drawing from for upstream and/or downstream flowcharts

Type *Process*

do_upstream

Should an upstream flowchart be drawn?

Type bool

do_downstream

Should a downstream flowchart be drawn?

Type bool

do_full

Should a full project flowchart be drawn?

Type bool

save

Should the flowchart be saved as a *.png* file?

Type bool

show

Should an interactive flowchart be displayed on screen?

Type bool

drew_up

The name of the upstream flowchart saved, if any

Type str

drew_down

The name of the downstream flowchart saved, if any

Type str

drew_full

The name of the full flowchart saved, if any

Type str

downstream_process_graph() → None

Make a flowchart for a job and all downstream processes

format_edges_list(*il: List[Tuple[Optional[*pipeliner.data_structure.Node*], Optional[*pipeliner.data_structure.Process*], pipeliner.data_structure.Process]]*) → List[Tuple[str, str, str, Optional[int]]]

Convert the output of network finding functions to format for graph drawing

(node, parent, child) to (Node name, parent name, child name, count) formats: (Node, Process, Process) to (str, str, str, int)

Parameters

- **il** (*list*) – list of tuples in the format
- **[node** –
- **parent** –
- **child]** –
- **[Node** –
- **Process** –
- **Process]** –

Returns

list of tuples [Node name, parent name, child name, count], [str, str, str, int]

Return type list

full_process_graph() → None

Make a flowchart for an entire project

make_process_flowchart(*edges_list: list*, *procname: Optional[str]*, *ntype: str*) → Optional[str]

Draw a flowchart and save and/or display it

Parameters

- **edges_list** (*list*) – A list of `pipeliner.project_graph.EDGE` objects
- **procname** (*str*) – The name of the process to draw a flowchart for
- **ntype** (*str*) – The type of flowchart being drawn; *"upstream"* or *"downstream"*

Returns The name of the file created if one was saved otherwise `None`

Return type str

upstream_process_graph() → None

Make a flowchart for a job and all upstream processes

6.2 Tools for Running Jobs

6.2.1 The JobRunner

class pipeliner.job_runner.**JobRunner**(*project_name: str = 'default'*)

Bases: object

The JobRunner object handles running jobs (who would have thought?)

graph

The pipeline for the project

Type *ProjectGraph*

add_job_to_pipeline(*job:* pipeliner.pipeliner_job.PipelinerJob, *status: str*, *allow_overwrite: bool*) → *pipeliner.data_structure.Process*

Add the job to the pipeline and create its temp nodes files

Parameters

- **job** (*PipelinerJob*) – The job that is being run
- **status** (*str*) – The status of the new job either *Running* or *Scheduled*
- **allow_overwrite** (*bool*) – Is the job to be run overwriting a previous job?

Returns The job that is being run

Return type *Process*

get_commandline_job(*thisjob:* pipeliner.pipeliner_job.PipelinerJob, *current_proc: Optional[*pipeliner.data_structure.Process*]*, *is_main_continue: bool*, *is_scheduled: bool*, *do_makedir: bool*, *overwrite: bool = False*, *subsequent_scheduled: bool = False*) → list

Assemble all of the commands necessary to run a job

Parameters

- **thisjob** (*PipelinerJob*) – The job that is being run
- **current_proc** (*Process*) – The existing process object for the current job, if the job to be run is a continuation or overwrite job, otherwise None
- **is_main_continue** (*bool*) – Is the job to be run a continuation?
- **is_scheduled** (*bool*) – Has the job to be run already been scheduled?
- **do_makedir** (*bool*) – Should a directory for the job be made if necessary?
- **overwrite** (*bool*) – Is the job to be run overwriting a previous job?
- **subsequent_scheduled** (*bool*) – Is the job to be run a subsequent iteration of a job that has already been run in a currently running schedule?

Returns

[[[Actual, command], [to, be, run]], [[the, Job, commands]]] If the job is being submitted to a queue *[0]* will be the qsub command and *[1]* will the be the actual job commands. For local jobs they will be identical

This is a list of lists, Each sublist is holds the arguments for a single command.

Return type list

Raises

- **ValueError** – If an attempt is made to overwrite or contiune a job that doesn't exist
- **RuntimeError** – If no commands are generated

prepare_job_to_run(*job:* pipeliner.pipeliner_job.PipelinerJob, *current_proc:* *Optional*[pipeliner.data_structure.Process], *is_main_continue:* *bool*, *is_scheduled:* *bool*, *overwrite_current:* *bool* = *False*, *subsequent_scheduled:* *bool* = *False*) → Tuple[List[list], bool]

Do the setup for running a job

This includes: - Removing the original files if overwriting - Removing any control files

Parameters

- **job** (*PipelinerJob*) – The job that is being run
- **current_proc** (*Process*) – The existing process object for the current job, if the job to be run is a continuation or overwrite job, otherwise None
- **is_main_continue** (*bool*) – Is the job to be run a continuation?
- **is_scheduled** (*bool*) – Has the job to be run already been scheduled?
- **do_makedir** (*bool*) – Should a directory for the job be made if necessary?
- **overwrite** (*bool*) – Is the job to be run overwriting a previous job?
- **subsequent_scheduled** (*bool*) – Is the job to be run a subsequent iteration of a job that has already been run in a currently running schedule?

Returns The commands (lists of lists) and if the job is overwriting (bool)

Return type tuple

run_job(*job:* pipeliner.pipeliner_job.PipelinerJob, *current_proc:* *Optional*[pipeliner.data_structure.Process], *is_main_continue:* *bool*, *is_scheduled:* *bool*, *overwrite_current:* *bool* = *False*, *subsequent_scheduled:* *bool* = *False*, *wait_for_queued:* *bool* = *True*, *comment:* *Optional*[*str*] = *None*) → *pipeliner.data_structure.Process*

Run a job, add to the pipeline with running status and execute its commmands

Parameters

- **job** (*PipelinerJob*) – The job that is being run
- **current_proc** (*Process*) – The existing process object for the current job, if the job to be run is a continuation or overwrite job, otherwise None
- **is_main_continue** (*bool*) – Is the job to be run a continuation?
- **is_scheduled** (*bool*) – Has the job to be run already been scheduled?
- **overwrite_current** (*bool*) – Is the job to be run overwriting a previous job?
- **subsequent_scheduled** (*bool*) – Is the job to be run a subsequent iteration of a job that has already been run in a currently running schedule?
- **wait_for_queued** (*bool*) – If this job is sent to the queue should the pipeliner wait for it to finish before continuing on?
- **comment** (*str*) – Comments to add to the job's jobinfo file

Returns The job that is being run

Return type *Process*

run_scheduled_jobs(*fn_sched: str*, *job_ids: Optional[List[str]] = None*, *nr_repeat: int = 1*, *minutes_wait: int = 0*, *minutes_wait_before: int = 0*, *seconds_wait_after: int = 0*)

Run the jobs in a schedule

Parameters

- **fn_sched** (*str*) – The name to be assigned to the schedule
- **job_ids** (*list*) – A list of str job names
- **nr_repeat** (*int*) – Number of times to repeat the entire schedule
- **minutes_wait** (*int*) – Minimum time to wait between jobs in minutes. If this has been passed whilst the job is running the next job will start immediately
- **minutes_wait_before** (*int*) – Wait this amount of time before initially starting to run the schedule
- **seconds_wait_after** (*int*) – Wait this many seconds before starting each job this wait always occurs, even if the minimum time between jobs has already be surpassed

Raises

- **ValueError** – If a schedule lock file exists with the selected schedule name, suggesting another schedule with the same name is already running
- **ValueError** – (through *schedule_fail()*) If the job directory for a scheduled job could not be found
- **ValueError** – (through *schedule_fail()*) If a job.star file is not found in one of the directories of a job to be run
- **ValueError** – (through *schedule_fail()*) If an input node for a job cannot be found
- **ValueError** – (through *schedule_fail()*) : If a job in the schedule fails

schedule_fail(*message: str*, *sl_name: str*, *sched_lock: str*)

Run when a schedule fails

Write to the log and then delete the schedule lock file

Parameters

- **message** (*str*) – The message to display and write to the log file
- **sl_name** (*str*) – Name of the log file
- **sched_lock** (*str*) – Name of the lock file

Returns An error message determined by why the schedule failed

Return type str

schedule_job(*job:* pipeliner.pipeliner_job.PipelinerJob, *current_proc: Optional[*pipeliner.data_structure.Process*]*, *is_main_continue: bool*, *overwrite_current: bool = False*, *subsequent_scheduled: bool = False*, *comment: Optional[str] = None*) → *pipeliner.data_structure.Process*

Schedule a job, add to the pipeline with scheduled status

Parameters

- **job** (*PipelinerJob*) – The job that is being run
- **current_proc** (*Process*) – The existing process object for the current job, if the job to be run is a continuation or overwrite job, otherwise None

- **is_main_continue** (*bool*) – Is the job to be run a continuation?
- **is_scheduled** (*bool*) – Has the job to be run already been scheduled?
- **overwrite_current** (*bool*) – Is the job to be run overwriting a previous job?
- **subsequent_scheduled** (*bool*) – Is the job to be run a subsequent iteration of a job that has already been run in a currently running schedule?

Returns The job that is being run

Return type *Process*

wait_for_queued_job_completion(*outdir: str*)

Wait for a job that has been sent to the queue to finish

Parameters **outdir** (*str*) – The job's output directory

write_to_sched_log(*message: str*, *logfile: str*)

For real time updating of the schedule log

Parameters

- **message** (*str*) – The message to display and write to the log file
- **sl_name** (*str*) – Name of the log file

6.2.2 Check_completion

6.2.3 The job factory

The job factory functions identify the available job types and return the correct type of job from the job type specified in a parameter file

pipeliner.job_factory.**active_job_from_proc**(*the_proc:* pipeliner.data_structure.Process) → *pipeliner.pipeliner_job.PipelinerJob*

Create an active job from an existing process

Used when the functions inside a job subclass need to be called on an existing job

Parameters **the_proc** (*Process*) – The process to create a job from

Returns

The job subclass object for the process

Return type *PipelinerJob*

pipeliner.job_factory.**convert_relion4_jobtypes_to_pipeliner**(*type_name*, *joboptions*, *jobs_dict*)

Convert an ambigious Relion4 style job type to the pipeliner type

Parameters

- **type_name** (*str*) – The current relion4 type for the job
- **joboptions** (*dict*) – The job's joboptions
- **jobs_kdict** (*dict*) – The job_factory jobs and classes dict, output from gather_jobtypes()
- **Returns** –

  ***PipelinerJob*: The job object for** the convert pipeler version of the job

pipeliner.job_factory.**gather_all_jobtypes**() → dict

Assemble a dict of all the available job types

Returns

The job classes dict

The dict keys are the job process names and it returns the class for that specific job type

Return type dict

Raises **ValueError** – If a process name is being used by more than one job type

pipeliner.job_factory.**job_from_dict**(*job_input: dict*) → *pipeliner.pipeliner_job.PipelinerJob*

Create a job from a dictionary

The dict must define the job type with a '_rlnJobTypeLabel' key. Any other keys will override the default options for that parameter

Parameters **job_input** (*dict*) – The dict containing the params. At minimum it must contain *{'_rlnJobTypeLabel': <jobtype>}*

Returns The job subclass

Return type *PipelinerJob*

Raises

- **ValueError** – If the '_rlnJobTypeLabel' key is missing from the dict
- **ValueError** – If '_rlnJobIsContinue' is in the dict - this function is not for creating continuations of jobs
- **ValueError** – If the specified jobtype is not found
- **ValueError** – If any of the parameters in the dict are not in the jobtype returned

pipeliner.job_factory.**new_job_of_type**(*type_name: str*, *joboptions: Optional[dict] = None*) → *pipeliner.pipeliner_job.PipelinerJob*

Creates a new object of the correct PipelinerJob sub-type

Parameters

- **type_name** (*str*) – The job process name
- **joboptions** (*dict*) – Dict of the job's joboptions, only necessary if converting from a RELION4.0 style jobname

Returns The job subclass

Return type *PipelinerJob*

Raises **RuntimeError** – If the job type is not found

pipeliner.job_factory.**read_job**(*filename: str*) → *pipeliner.pipeliner_job.PipelinerJob*

Reads run.job and job.star files and returns the correct Pipeliner job class

Parameters **filename** (*str*) – The run.job or job.star file

Returns The job subclass

Return type *PipelinerJob*

Raises

- **ValueError** – If the file name entered does not end with run.job or job.star

- RuntimeError – If the input file is in the RELION 3.1 format and conversion fails
- RuntimeError – If the job type specified in the file cannont be found

6.3 Star File Utilities

6.3.1 Star file reading utilities

Tools for reading and modifying star files

class pipeliner.jobstar_reader.**BodyFile**(*fn_in*)

Bases: object

A star file that lists the bodies in a multibody refinement

data

A gemmi cif object containing the data from

Type gemmi.cif.Cif

bodies

The block that contains info about the bodies

Type gemmi.cif.Block

count_bodies()

Count the number of bodies in the BodyFile

Returns The count

Return type int

class pipeliner.jobstar_reader.**ExportStar**(*fn_in*)

Bases: object

Class for a star file that lists exported jobs

This functionality is from RELION 3.1 and will probably be deprecated

data

A gemmi cif object containing the data from the star file

Type gemmi.cif.Cif

jobs

The block that contains info about the exported jobs

Type gemmi.cif.Block

class pipeliner.jobstar_reader.**JobStar**(*fn_in*)

Bases: object

A class for star files that define a pipeliner job parameters

data

A gemmi cif object containing the data from the job.star file

Type gemmi.cif.Cif

pipeline_info
The cif data block for the *job* section of the file

Type gemmi.cif.Block

options
The cif data block for the 'options' section of the file

Type gemmi.cif.Block

count_blocks()
Count the number of blocks in the file

Returns The number of blocks

Return type int

count_jobopt()
Count the number of items in the job options section of the file

Returns The number of items

Return type int

count_pipeline()
Count the number of items in the pipeline section of the file

Returns The number of items

Return type int

get_all_options(*against=None*)
Get a dict of all the parameters in the file

Parameters **against** (*iterable*) – Only return joboptions that are present in this list, if *None* returns all the job options

Returns The parameters dict in the format {param_name: value}

Return type dict

get_block(*block_name*)
Get a specific block from the file

Parameters **block_name** (*str*) – The name of the block to retrieve

Returns The desired block

Return type gemmi.cif.Block

get_continue_status()

get_jobtype()

class pipeliner.jobstar_reader.**OutputNodeStar**(*fn_in*)
Bases: object

A starfile that lists output nodes for exported jobs

This functionality is from RELION 3.1 and will probably be deprecated

data
A gemmi cif object containing the data from the star file

Type gemmi.cif.Cif

jobs

The block that contains info about the exported nodes

Type gemmi.cif.Block

get_output_nodes()

Get a list of output node names and types from an OutputNodeStar

This functionality is from RELION 3.1 and will probably be deprecated

Returns [[node 1 name, node 1 type], . . . , [node n name, node n type]]

Return type list

class pipeliner.jobstar_reader.**RelionStarFile**(*fn_in*)

Bases: object

A general class for starfiles that contain data, such as particles files

The input file is checked for reserved word errors when initialising a RelionStarFile object

data

A gemmi cif object containing the data from the star file

Type gemmi.cif.Cif

count_block(*blockname=None*)

Count the number of items in a block that only contains a single loop

This is the format in most relion data star files

Parameters **blockname** (*str*) – The name of the block to count

Returns The count

Return type int

get_block(*blockname=None*)

Get a block from the star file

Parameters **blockname** (*str*) – The name of the block to get. Use *None* if the file has a single unnamed block

Returns The desired block

Return type (gemmi.cif.Block)

class pipeliner.jobstar_reader.**StarfileCheck**(*fn_in=None*, *cifdoc=None*, *is_pipeline=False*)

Bases: object

A class for checking the validity of star files and converting them

Checks for reserved word errors which can be common in star files written by some versions of RELION. If the starfile was written by RELION 3.x converts it to the pipeliner format

version

Info from the star file's #version line what program wrote it and what version number it is

Type str

fn_in

The path to the star file.

Type str

cifdoc
The cif doc containing the data for the file

Type gemmi.cif.Cif

has_been_corrected
If the file has been corrected

Type bool

has_been_converted
If the file has been converted from RELION 3.1 to RELION 4.0/Pipeliner format

Type bool

is_pipeline
If the file is a pipeline star file

Type bool

check_pipeline_version()
See if a pipeline is the RELION 3.x or the new RELION 4.0/pipeliner style

Returns The pipeline version *relion3*, *relion4*, or *undefined*

Return type str

Raises **ValueError** – If the pipeline has features of both RELION 3.x and RELION 4.0/Pipeliner versions

check_reserved_words()
Make sure the starfile doesn't contain any illegally used reserved words

Only operates on a file only as trying to read in a cif doc with reserved word errors raises a parse error. Overwrites the original file if it is corrected. The old file is saved as <filename>.old

convert_relion3_1_pipeline(*display_warning=True*)
Convert a pipeline from RELION 3.x to RELION 4.0/Pipeliner format

Backs up the original file as <filename>.old

Parameters **display_warning** (*bool*) – Should a warning be displayed on screen?

pipeliner.jobstar_reader.**bool_jobop**(*jobop*)
Checks if a value should be interpreted as True

Returns *True* if the value is true, otherwise *False*

Return type bool

pipeliner.jobstar_reader.**compare_starfiles**(*starfile1*, *starfile2*)
See if two starfiles contain the same information

Direct comparison can be difficult because the starfile columns or blocks can be in different orders

Parameters

- **starfile1** (*str*) – Name of the first file
- **starfile2** (*str*) – Name of the second file

Returns (bool, bool) [0] True if the starfile structures are identical (Names of blocks and columns) [1] True if the data in the two files are identical

Return type tuple

pipeliner.jobstar_reader.**convert_coordinates_job**(*fn_in*)

Convert RELION 3.1 style coordinates files to RELION 4 style

RELION 3.x has an empty file called coords_suffix_xxx.star RELION 4.0/pipeliner has a starfile with two columns for micrograph name and coords file. If errors are encountered with the conversion it sets the node type for that file as UNSUPPORTED_deprecated_format

Parameters **fn_in** (*str*) – Path to the RELION 3.x style file

Returns The new file name

Return type str

pipeliner.jobstar_reader.**convert_oldstyle_names**(*fn_in*)

Converts model and particle nodes from RELION 3.x to RELION 4 names

The model files is RELION 3.x have been merged in to the _optimiser files in RELION 4. The particle files have moved from a system where the name and location of the file define the particles to the particles being explicitly defined in the file itself.

Parameters **fn_in** (*str*) – The path to the file to convert

Returns The name of that file's node in RELION 4

Return type str

pipeliner.jobstar_reader.**convert_pipeline_node**(*name*, *oldtype*)

Convert node names from a RELION 3.1 to the RELION 4.0 formats

Every job type has a specific method for converting its own node types

Parameters

- **name** (*str*) – The node name
- **oldtype** (*str*) – The original node type

Returns job name or error and if it was successful: (str, bool)

Return type tuple

pipeliner.jobstar_reader.**convert_proctype**(*jobname*, *jobops=None*)

Convert a process from the RELION to pipeliner style

The pipeliner has much more specific job types than RELION 3.x/4.0 so conversion is necessary

Parameters

- **jobname** (*str*) – The name of the job
- **jobops** (*dict*) – Dict of joboptions with *{dict_name or label: value}*

Returns job name or error and if it was successful: (str, bool)

Return type tuple

pipeliner.jobstar_reader.**convert_relion20_datafile**(*starfile*, *dtype*, *outname=None*, *boxsize=None*, *og_name='convertedOpticsGroup1'*, *og_num=1*)

Convert a relion 2.x format starfile to relion 3.x/4.x format

Adds on an optics groups block and renames the main data block with the relion 3/x/4.x format. Fixes te descrepency between how Relion 2.x and Relion 3.x/4.x define pixel size. Adds missing fields to the data block.

Parameters

- **starfile** (*str*) – The name of the file to process

- **outname** (*str*) – The name of the output file, 'converted_<starfile>.star' by default.
- **dtype** (*str*) – The type of file, must be in (movies, micrographs, ctf_micrographs, particles)
- **boxsize** (*int*) – The box size of particles in pix. This is required for particles
- **og_name** (*str*) – The name for the optics group that will be created for the data in the file
- **og_num** (*int*) – Optics group number for the optics group that will be created for the data in the file

pipeliner.jobstar_reader.**count_movs_mics_parts**(*node*)

Count the number of objects in a movies, mics or particles starfile

Parameters **node** (*Node*) – The *Node* object for the star file

Returns Count of object if the file contained movies, micrographs, or particles returns `None` if the file is not found or it does not contain the correct data type

Return type int

pipeliner.jobstar_reader.**get_job_type**(*job_options_dict: dict*) → Tuple[str, str]

Get the job type from a job.star file

Includes back compatibility with the old naming convention in RELION 3.1 files (_rlnJobType vs _rlnJobTypeLabel)

Parameters **job_options_dict** (*dict*) – A dict of the parameters from a job.star file in the format `{param_name: param_value}`

Returns

(job_type (str), job_type_label (str))

The job type is the pipeliner format string job type. The job_type_label is the label used in the star file to designate the job type, which is different between RELION 3.x and RELION 4.0/Pipeliner

Return type tuple

pipeliner.jobstar_reader.**get_joboption**(*dict_name*, *label*, *jobops*)

Get a job option value from muliple dictionary types

Return a job option where the jobops dict might contain the dictionary entries (from jobstar file) or the labels (from runjob file)

Parameters

- **dict_name** (*str*) – The joboption key IE: from a job.star file
- **label** (*str*) – The joboption label IE: from a run.job file
- **jobops** (*dict*) – The dict of joboptions with *{dict_name or label: value}*

Returns The job option value

Return type str

pipeliner.jobstar_reader.**modify_jobstar**(*fn_template: str*, *params_to_change: dict*, *out_fn: str*) → str

Edits the a template job.star file and saves a copy

Parameters

- **fn_template** (*str*) – Path to the template star file to be modified
- **params_to_change** (*dict*) – The Parameters to change in the template in the format {param_name: new_value}

- **out_fn** (*str*) – Name for the output file. If the same as the template files it will be overwritten

Returns The name of the output file

Return type str

pipeliner.jobstar_reader.**star_loop_as_list**(*starfile*, *block*, *columns=None*)

Returns a set of columns from a starfile loop as a list

Parameters

- **starfile** (*str*) – The file to read from
- **block** (*str*) – The name of the block to get the data from
- **columns** (*list*) – Names of the columns to get, if None then all columns are returned

Returns (list, list) [0] The names of the columns in order, [c1, c2, c3]. [1] The rows of the column(s) as a list of lists [[r1c1, r1c2, r1c3],[r1c1, r1c2, r1c3],[r1c1, r1c2, r1c3]]

Return type tuple

Raises

- **ValueError** – If the specified block is not found
- **ValueError** – If the specified block does not contain a loop
- **ValueError** – If any of the specified columns are not found

pipeliner.jobstar_reader.**star_pairs_as_dict**(*starfile*, *block*)

Returns paired values from a starfile as a dict

Parameters

- **starfile** (*str*) – The file to read from
- **block** (*str*) – The name of the block to get the data from

Returns *{parameter: value}*

Return type dict

Raises

- **ValueError** – If the specified block is not found
- **ValueError** – If the specified block is a loop and not a pair-value

6.3.2 Starfile writing utilities

Writes star files in the same style as RELION

pipeliner.star_writer.**write**(*doc*, *filename: str*, *commentline: str = 'Relion version 4.0 / CCP-EM_pipeliner vers 0.0.1'*)

Write a Gemmi CIF document to a RELION-style STAR file.

Parameters

- **doc** (gemmi.cif.Cif) – The data to write out
- **filename** (*str*) – The name of the file to write the data to
- **commentline** (*str*) – The comment line to put in the written star file

pipeliner.star_writer.**write_jobstar**(*in_dict: dict*, *out_fn: str*)

Write a job.star file from a dictionary of options

Parameters

- **in_dict** (*dict*) – Dict of job option keys and values
- **out_fn** (*str*) – Name of the file to write to

pipeliner.star_writer.**write_to_stream**(*doc*, *out_stream: IO[str]*, *commentline: str = 'Relion version 4.0 / CCP-EM_pipeliner vers 0.0.1'*)

Write a Gemmi CIF document to an output stream using RELION's output style.

Parameters

- **doc** (gemmi.cif.Cif) – The data to write out
- **out_stream** (*str*) – The name of the file to write the data to
- **commentline** (*str*) – The comment line to put in the written star file

6.4 General Utilities

These utilites are used by the pipeliner for basic tasks such as nice looking on-screen display, checking file names, and getting directory and file names

pipeliner.utils.**check_for_illegal_symbols**(*check_string: str*, *string_name: str = 'input'*, *exclude: str = ''*)

Check a text string doesn't have any of the disallowed symbols.

Illegal symbols are !*?()^/#<>&%{}$.''' and @.

Parameters

- **check_string** (*str*) – The string to be checked
- **string_name** (*str*) – The name of the string being checked; for more informative error messages
- **exclude** (*str*) – Any symbols that are normally in the illegal symbols list but should be allowed.

Raises ValueError – If illegal symbols are present in the test string

pipeliner.utils.**clean_jobname**(*jobname: str*) → str

Makes sure job names are in the correct format

Job names must have a trailing slash, cannot begin with a slash, and have no illegal characters

Parameters **jobname** (*str*) – The jon name to be checked

Returns The job name, with corrections in necessary

Return type str

pipeliner.utils.**date_time_tag**(*compact: bool = False*) → str

Get a current date and time tag

If can return a compact version or one that is easier to read

Parameters **compact** (*bool*) – Should the returned tag be in the comapct form

Returns

The datetime tag

compact format is: *YYYYMMDDHHMMSS*

verbose form is: *YYYY-MM-DD HH:MM:SS*

Return type str

pipeliner.utils.**decompose_pipeline_filename**(*fn_in: str*) → Tuple[str, Optional[int], str]

Breaks a job name into usable pieces

Returns everything before the job number, the job number as an int and everything after the job number setup for up to 20 dirs deep. The 20 directory limit is from the relion code but no really necessary anymore

Parameters **fn_in** (*str*) – The job or file name to be broken down in the format: <job-type>/jobxxx/<filename>

Returns

The decomposed file name: (str, int, str) *[0]* Everything before 'job' in the file name

[1] The job number

[2] Everything after the job number

Return type tuple

Raises **ValueError** – If the input file name is more than 20 directories deep

pipeliner.utils.**find_common_string**(*input_strings: List[str]*) → str

Find the common part of a list of strings starting from the beginning

Parameters **input_strings** (*list*) – List of strings to compare

Returns The common portion of the strings

Return type str

Raises **ValueError** – If input_list is shorter than 2

pipeliner.utils.**fix_newlines**(*file_path: str*)

Replace LF+CR new lines in files with LF, because RELION doesn't like them

Parameters **file_path** (*str*) – The file to fix

pipeliner.utils.**get_pipeliner_root**() → pathlib.Path

Get the directory of the main pipeliner module

Returns The path of the pipeliner

Return type path

pipeliner.utils.**make_pretty_header**(*title: str*)

Make nice looking headers for on-screen display

Parameters **title** (*str*) – The text to put in the header

Returns

A nice looking header

```
-=-=-=-=-=-=-=-=-=
It looks like this
-=-=-=-=-=-=-=-=-=
```

Return type str

pipeliner.utils.**print_nice_columns**(*list_in: List[str]*, *err_msg: str = 'ERROR: No items in input list'*)

Takes a list of items and makes three columns for nicer on-screen display

Parameters

- **list_in** (*str*) – The list to display in columns
- **err_msg** (*str*) – The message to display if the list is empty

pipeliner.utils.**quotate_command_list**(*commands: List[list]*) → List[list]

Adds quotation marks to commands arguments that need them

If a command is to be run in terminal some args need to be quotated. Quotation marks are not needed if the command list is run with subprocess.run but they are if the command is run as a string in a qsub script or in the terminal

Any arg that contains a space or the set of characters !*?()^#<>&%{}$@ will be quotated

Parameters **commands** (*list*) – The commands are a list of lists. Each item in the main list is a single command, which itself is a list of the individual arguments

Returns

A correctly quotated command list

The list is in the same list of lists format

Return type list

pipeliner.utils.**smart_strip_quotes**(*in_string: str*) → str

Strip the quotes from a string in a intellegant manner

Remove leading and ending ‘ and ” but don’t remove them internally

Parameters **in_string** (*str*) – The input string

Returns the string with leading and ending quotes removed

Return type str

pipeliner.utils.**touch**(*filename: str*)

Create an empty file

Parameters **filename** (*str*) – The name for the file to create

pipeliner.utils.**truncate_number**(*number: float*, *maxlength: int*) → str

Return a number with no more than x decimal places but no trailing 0s

This is used to format numbers in the exact same way that RELION does it. IE: with maxlength 3; 1.2000 = 1.2, 1.0 = 1, 1.23 = 1.23. RELION commands are happy to accept numbers with any number of decimal places or trailing 0s. This function is just to maintain continuity between RELION and pipeliner commands

Parameters

- **number** (*float*) – The number to be truncated
- **maxlength** (*int*) – The maximum number of decimal places

pipeliner.utils.**wrap_text**(*text_string: str*)

Produces <= 55 character wide wrapped text for on-screen display

Parameters **text_string** (*str*) – The text to be displayed

CHAPTER

SEVEN

PIPELINER JOBS AND PLUGINS

7.1 Pipeliner Jobs

class pipeliner.pipeliner_job.**JobInfo**(*display_name: str = 'Pipeliner job'*, *version: str = '0.0'*, *job_author: Optional[str] = None*, *short_desc: str = 'No short description for this job'*, *long_desc: str = 'No long description for this job'*, *documentation: str = 'No online documentation available'*, *programs: Optional[list] = None*, *references: Optional[list] = None*, *software_vers: str = 'No version info available'*)

Bases: object

Class for storing info about jobs.

This is used to generate documentation for the job within the pipeliner

display_name

A user-friendly name to describe the job in a GUI, this should not include the software used, because that info is pulled from the job type

Type str

version

The version number of the pipeliner job

Type str

software_vers

The version of the outside executables that will be used

Type str

job_author

Who wrote the pipeliner job

Type str

short_desc

A one line "title" for the job

Type str

long_desc

A detained description about what the job does

Type str

documentation

A URL for online documentation

Type str

programs

A list of 3rd party software used by the job. These are used by the pipeliner to determine if the job can be run so they need too be the names of all executables the job might call. If any program on this list cannot be found with *which* then the job will be marked as unable to run.

Type list

references

A list of *Ref* objects

Type list

class pipeliner.pipeliner_job.**PipelinerJob**

Bases: object

Super-class for job objects.

Each job type has its own sub-class.

WARNING: do not instantiate this class directly, use the factory functions in this module.

jobinfo

Contains information about the job such as references

Type *JobInfo*

output_name

The path of the output directory created by this job

Type str

alias

the alias for the job if one has been assigned

Type str

is_continue

If this job is a continuation of an older job or a new one

Type bool

input_nodes

A list of *Node* objects for each file used as in input

Type list

output_nodes

A list of *Node* objects for files produced by the job

Type list

joboptions

A dict of JobOption objects specifying the parameters for the job

Type dict

final_commands

A list of commands to be run by the job. Each item is a list of arguments

Type list

is_mpi

Does the job use multi-threading?

Type bool

is_tomo

Is the job a tomography job?

Type bool

vers_com

[0] The command to run to which will print the program's version info to the *STDOUT*, *[1]* The lines from the stdout to display, if *()* all lines will be displayed, make sure that the value is a *tuple* IE: *(1,)* rather than *(1)*

Type tuple(list, tuple)

OUT_DIR = ''

PROCESS_NAME = ''

clear()

Clear all attributes of the job

create_input_nodes()

Automatically add the job's input nodes to it's nodelist

create_results_display()

This function creates the objects to be displayed by the GUI

Placeholder for individual jobs to have their own

Returns

a ResultsDisplayText saying there is no specific method for this job

Return type list

default_params_dict() → dict

Get a dict with the job's parameters and default values

Returns All of the job's parameters {parameter: default value}

Return type dict

gather_metadata()

Placeholder function for metadata gathering

Each job class should define this individually

Returns A place holder "No metadata available" and the reason why

Return type dict

get_commands()

get_current_output_nodes() → list

Get the current output nodes if the job was stopped prematurely

For most jobs there will not be any but for jobs with many iterations the most recent interation can be used of teh job is aborted or failed and then later marked as successful

Parameters **new_status** (*str*) – The new status - what actions are performed will be dependent on this

Returns of *Node* objects

Return type list

get_extra_options()

Get user specified extra queue submission options

get_job_vers()

Get the current version of the software available in this system

Returns

The version info, or *No version info available* if no version command was specified

Return type str

get_runtab_options(*mpi: bool = True*, *threads: bool = True*)

Get the options found in the Run tab of the GUI, which are common to for all jobtypes

Adds entries to the joboptions dict for queueing, MPI, threading, and additional arguments. This method should be used when initialising a *PipelinerJob* subclass

Parameters

- **mpi** (*bool*) – Should MPI options be included?
- **threads** (*bool*) – Should multi-threading options be included

initialise_pipeline(*outputname: str*, *defaultname: str*, *job_counter: int*) → str

Gets the pipeline ready to add a new job

Sets the output name and clears the input and output nodes

Parameters

- **outputname** (*str*) – Where the job should write its results. If blank it is set to the next job number based on the job counter
- **defaultname** (*str*) – The name of the job type
- **job_counter** (*int*) – The number that job will get

Returns The output name

Return type str

make_additional_args()

Get the additional arguments job option

make_queue_options()

Get options related to queueing and queue submission, which are common to for all jobtypes

parameter_validation() → list

Advanced validation of job parameters

This is a placeholder function for additional validation to be done by individual job subtypes, such as comparing JobOption values IE: JobOption A must be > JobOption B

Returns A list of error messages. If no errors are found should return an empty list

Return type list

parse_additional_args()

Parse the additional arguments job option and return a list

Returns

A list ready to append to the command. Quotated strings are preserved as quoted strings all others are split into individual items

Return type list

post_run_actions() → bool

Placeholder function for actions to do after the job has finished

Each job class should define this individually. This is used for job where somthing needs to be done after the jobhas completed, such as jobs where the number/names of output nodes is not known until the job has finished

prepare_clean_up_lists(*do_harsh: bool = False*)

Placeholder function for preparation of list of files to cleanup

Each job class should define this individually

Parameters **do_harsh** (*bool*) – Should a harsh cleanup be performed

Returns Two empty lists (`[files, to, delete], [dirs, to, delete]`)

Return type tuple

prepare_final_command(*outputname: str*, *commands: list*, *do_makedir: bool*, *ignore_queue: bool = False*) → list

Assemble commands to be run for a job

The commands are in a lists of lists format. Each item in the main list is a single command and composed of a list of the arguments for that command.

An additional command to run the check completion script is added to the commands list.

Decides if a queue submission script is needed. If so it is written and the commands list is changed to the queue submission command

Parameters

- **outputname** (*str*) – The job's output directory
- **commands** (*list*) – The commands to run as a list of lists
- **do_makedir** (*bool*) – Should the output directory be created if it doesn't already exist?
- **ignore_queue** (*bool*) – Do not make a submission script, even if the job is sent to the queue, used for generating commands for display

Returns

[[[Actual, command], [to, be, run]], [[the, Job, commands]]] If the job is being submitted to a queue *[0]* will be the qsub command and *[1]* will the be the actual job commands. For local jobs they will be identical

Return type list

`prepare_onedep_data()` → list

Placeholder for function to return deposition data objects

The specific list returned should be defined by each jobtype

Returns

The deposition object(s) returned by the specific job. These need to be of the types defined in *pipeliner.onedep_deposition*

Return type list

`read`(*filename: str*)

Reads parameters from a run.job or job.star file

Parameters **filename** (*str*) – The file to read. Can be a run.job or job.star file

Raises **ValueError** – If the file is a job.star file and job option from the *PipelinerJob* is missing from the input file

`save_job_submission_script`(*output_script: str*, *outputname: str*, *commands: list*, *nmpi: int*) → str

Writes a submission script for jobs submitted to a queue

Parameters

- **output_script** (*str*) – The name for the script to be written
- **output_name** (*str*) – The job's output name
- **commands** (*list*) – The job's commands. In a list of lists format
- **nmpi** (*int*) – The number of MPI used by the job. Should be 1 if the job is not multi-threaded

Returns The name of the submission script that was written

Return type str

Raises

- **ValueError** – If no submission script template was specified in the job's joboptions
- **ValueError** – If the submission script template is not found
- **RuntimeError** – If the output script could not be written

`set_option`(*line: str*)

Sets a value in the joboptions dict from a run.job file

Parameters **line** (*str*) – A line from a run.job file

Raises

- **RuntimeError** – If the line does not contain '=='
- **RuntimeError** – If the value of the line does not match any of the joboptions keys

validate_dynamically_required_joboptions()

Validate joboptions that only become required in relation to others

For example if job option A is True, job option B is now required

Returns

pipeliner.job_options.JobOptionValidationResult: for any errors found

Return type list

validate_input_files() → list

Check that files specified as inputs actually exist

Returns

A list of pipeliner.job_options.JobOptionValidationResult objects

Return type list

validate_joboptions() → list

Make sure all of the joboptions meet their validation criteria

Returns

tuple for each joboption that had errors [(joboption, desc, error)]

Return type list

write_jobstar(*output_dir: str*, *output_fn: str = 'job.star'*, *is_continue: bool = False*)

Write a job.star file.

Parameters

- **output_fn** (*str*) – The name of the file to write. Defaults to job.star
- **is_contine** (*bool*) – Is the file for a continuation of a previously run job? If so only the parameters that can be changed on continuation are written. Overrules is_continue attribute of the job

write_runjob(*fn: Optional[str] = None*)

Writes a run.job file

Parameters **fn** (*str*) – The name of the file to write. Defaults to the file the pipeliner uses for storing GUI parameters. A directory can be entered also and it will add on the file name 'run.job'

class pipeliner.pipeliner_job.**Ref**(*authors: Optional[Union[str, List[str]]] = None*, *title: str = ''*, *journal: str = ''*, *year: str = ''*, *volume: str = ''*, *issue: str = ''*, *pages: str = ''*, *doi: str = ''*, ***kwargs*)

Bases: object

Class to hold metadata about a citation or reference, typically a journal article.

authors

The authors of the reference.

Type list

title

The reference's title.

Type str

journal
The journal.

Type str

year
The year of publication.

Type str

volume
The volume number.

Type str

issue
The issue number.

Type str

pages
The page numbers.

Type str

doi
The reference's Digital Object Identifier.

Type str

other_metadata
Other metadata as needed. Gathered from kwargs

Type dict

7.2 Job Options

7.2.1 JobOptions

JobOptions are used to store parameters for jobs. They contain all the info necessary for on-the-fly GUI generation.

class pipeliner.job_options.**BooleanJobOption**(*, *label: str*, *default_value: bool*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *deactivate_if: list = []*)

Bases: pipeliner.job_options.JobOption

Define a job option as a boolean

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*bool*) – The default value
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

get_boolean() → bool

Get a boolean value

Returns The value

Return type bool

joboption_type = 'BOOLEAN'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation of the input parameters

Returns

The validation error of one exists

Return type JobOptionValidationResult

class pipeliner.job_options.**FileNameJobOption**(*, *label: str*, *pattern: str = ''*, *directory: str = ''*, *default_value: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*, *validation_regex: Optional[str] = None*, *suggestion: str = ''*, *node_type: Optional[str] = None*)

Bases: pipeliner.job_options.JobOption

Define a job option as a file name

GUI will open a file browser for the user to find an input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*str*) – will usually be set from the string and pattern.
- **pattern** (*str*) – Info about the search string for file types. It should be in the format <description> (<example>) e.g.: *"MRC files (*.mrc)"*
- **directory** (*str*) – The directory the GUI should to open the file browser in
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **suggestion** (*str*) – A hint that will appear in the data entry field if no no default value is set, it will not be used. IE: 'Enter file path here' If you want the suggestion used as the default then just put it in default_value
- **node_type** (*str*) – The type of input node that should be created for this file, if one is necessary. If this is *None* no node will be created

check_file() → Optional[pipeliner.job_options.JobOptionValidationResult]

Check that specified input files exist

Returns An error message if the file does not exist

Return type str

joboption_type = 'FILENAME'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation of the input parameters

Returns

The validation error of one exists

Return type JobOptionValidationResult

class pipeliner.job_options.**FloatJobOption**(*, *label: str*, *default_value: Union[str, float]*, *suggested_min: Optional[float] = None*, *suggested_max: Optional[float] = None*, *step_value: float = 1.0*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*, *hard_min: Optional[float] = None*, *hard_max: Optional[float] = None*)

Bases: pipeliner.job_options.JobOption

Define a job option as a slider for inputting numbers as floats

The min value and max value are for display only, they do not limit the actual values that can be input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*float*) – The default value for the parameter
- **suggested_min** (*float*) – The suggested minimum value for the slider
- **suggested_max** (*float*) – The suggestied maximum value for the slider
- **step_value** (*float*) – The step value for the slider
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **hard_min** (*float*) – An error will be raised if the value is < this value
- **hard_max** (*float*) – An error will be raised if the value is > this value

get_number(*required: bool = False*, *errormsg: str = ''*) → float

Get the value of a JobOption as a number

Parameters

- **required** (*bool*) – If a value is required for this parameter
- **errormsg** (*string*) – The message to display if the parameter is required and no value was entered

Returns The value

Return type float

Raises

- **ValueError** – If the value cannot be converted to a float
- **ValueError** – If the value is required and missing

joboption_type = 'FLOAT'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation check on the job option

Returns

The validation error of one exists

Return type JobOptionValidationResult

class pipeliner.job_options.**InputNodeJobOption**(*, *label: str*, *node_type: str*, *directory: str = ''*, *default_value: str = ''*, *pattern: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*, *validation_regex: Optional[str] = None*, *suggestion: str = ''*, *create_node: bool = True*)

Bases: pipeliner.job_options.JobOption

Define a job option as a file name, create an input node for it

GUI will open a file browser for the user to find an input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **node_type** (*str*) – The type of node using the standard <type>.<ext>.<keywords> format
- **default_value** (*str*) – will be set by the specific class method
- **directory** (*str*) – file browser
- **pattern** (*str*) – INfo about the search string for file types. It should be in the format <description> (<example>) IE: *"MRC files (*.mrc)"*
- **directory** – The directory the GUI should to open the file browser in
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **suggestion** (*str*) – A hint that will appear in the data entry field if no no default value is set, it will not be used. IE: 'Enter file path here' If you want the suggestion used as the default then just put it in default_value
- **create_node** (*bool*) – Should the node be automatically created, assuming it's value is not *None*. This should almost always be *True*

check_file() → Optional[pipeliner.job_options.JobOptionValidationResult]

Check if the specified file exists

Returns

The validation error of one exists

Return type JobOptionValidationResult

get_input_node() → *pipeliner.data_structure.Node*

Get a Node for an input node job option

Returns A Node object

Raises **ValueError** – If the JobOption is not an input node type

joboption_type = 'INPUTNODE'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation of the input parameters

Returns

The validation error of one exists

Return type JobOptionValidationResult

class pipeliner.job_options.**IntJobOption**(*, *label: str*, *default_value: int*, *suggested_min: Optional[int] = None*, *suggested_max: Optional[int] = None*, *step_value: int = 1*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*, *hard_min: Optional[int] = None*, *hard_max: Optional[int] = None*)

Bases: pipeliner.job_options.JobOption

Define a job option as a slider for inputing numbers as integers

The slider min value and max value are for display only, they do not limit the actual values that can be input

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **default_value** (*int*) – The default value for the parameter
- **suggested_min** (*int*) – The suggested minimum value for the slider
- **suggested_max** (*int*) – The suggested maximum value for the slider
- **step_value** (*int*) – The step value for the slider
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued
- **hard_min** (*int*) – An error will be raised if the value is < this value
- **hard_max** (*int*) – An error will be raised if the value is > this value

Raises **ValueError** – If the suggested_min, siggested_max, default, hard_min , hard_max, or step values are not integers

get_number(*required: bool = False*, *errormsg: str = ''*) → int

Get the value of a JobOption as a number

Parameters

- **required** (*bool*) – If a value is required for this parameter
- **errormsg** (*string*) – The message to display if the parameter is required and no value was entered

Returns The value

Return type int

Raises

- **ValueError** – If the value cannot be converted to a float
- **ValueError** – If the value is required and missing

joboption_type = 'INT'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation check on the job option

Returns

The validation error of one exists

Return type JobOptionValidationResult

class pipeliner.job_options.**JobOption**(*, *label: str*, *default_value: Any = ''*, *help_text: str = ''*, *in_continue: bool = False*, *is_required: bool = False*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*, *only_in_continue: bool = False*)

Bases: object

A JobOption stores a parameter, its value, and info about the GUI for that param

This is a general class with several more specialised class subclasses

joboption_type

The type of job option

Type str

label

A verbose label for the parameter. This is what appears in a run.job file

Type str

value

The value of the parameter. The type will be set by the specific class method

Type None

default_value

will be set by the specific class method

Type None

help_text

Text that will be displayed in the GUI if help is clicked

Type str

in_continue

If this parameter can be modified in a job that is continued

Type bool

is_required

Is this joboption always required?

Type bool

required_if

Validation option: a list of logical statements, if all are true the joboption will be required by the GUI. Requires a list of tuples: (joboption, operator, value). Valid operators are: '=', '!=', '>', '>=', '<', and '<='

Type list

deactivate_if

a list of logical statements, if all are true the joboption will be deactivated by the GUI. Requires a list of tuples: (joboption, operator, value) Valid operators are: '=', '!=', '>', '>=', '<', and '<='

Type list

only_in_continue

The joboption should only be available if the job is being continued

Type bool

get_string(*required: bool = False*, *errormsg: str = ''*) → str

Get the value of a JobOption as a string

Parameters

- **required** (*bool*) – If a value is required for this parameter
- **errormsg** (*string*) – The message to display if the parameter is required and no value was entered

Returns The value

Return type str

Raises **ValueError** – If the value is required and missing

joboption_type = 'ERROR: no job option type set'

set_string(*set_to: str*) → None

Set the value of a JobOption to a string

Parameters **set_to** (*str*) – The string to set the value to

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation of input parameters

will be set by individual JobOption subtypes

class pipeliner.job_options.**JobOptionValidationResult**(*result_type: str*, *raised_by: List[pipeliner.job_options.JobOption]*, *message: str*)

Bases: object

A class for handling validation of joboptions

type

Either 'warning' or 'error'

Type str

raised_by

List of JobOption objects that raised the warning/error. Generally this will only be one JobOption, unless the warning/error was raised from comparing two or more JobOptions

Type list

message

Description of the error/warning

Type str

class pipeliner.job_options.**MultipleChoiceJobOption**(*, *label: str*, *choices: List[str]*, *default_value_index: int*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required=True*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*)

Bases: pipeliner.job_options.JobOption

Define a job option as a pull down menu

Radio is a misnomer, the GUI will display a pull down menu with the options

Parameters

- **label** (*str*) – A verbose label for the parameter. This is what appears in a run.job file
- **choices** (*list*) – A list str options for the menu
- **default_value_index** (*int*) – Index of the initial option for the radio; also used as the default value
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

Raises ValueError – If the index of the initial option is not in the radio menu

joboption_type = 'MULTIPLECHOICE'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation of the input parameters

Returns

The validation error of one exists

Return type JobOptionValidationResult

class pipeliner.job_options.**StringJobOption**(*, *label: str*, *default_value: str = ''*, *help_text: str = ''*, *in_continue: bool = False*, *only_in_continue: bool = False*, *is_required: bool = False*, *required_if: Optional[list] = None*, *deactivate_if: Optional[list] = None*, *validation_regex: Optional[str] = None*)

Bases: pipeliner.job_options.JobOption

Create a job option object for a string parameter.

Parameters

- **label** (*str*) – A verbose label for the parameter.
- **default_value** (*None*) –
- **help_text** (*str*) – Text that will be displayed in the GUI if help is clicked.
- **in_continue** (*bool*) – If this parameter can be modified in a job that is continued

joboption_type = 'STRING'

validate() → Optional[pipeliner.job_options.JobOptionValidationResult]

Basic validation of the input parameters

pipeliner.job_options.**files_exts**(*name: str = 'File'*, *exts: Optional[List[str]] = None*, *exact: bool = False*) → str

Produce a description of files and their extensions

In a format compatible with PyQt5 QFileDialog

Parameters

- **name** (*str*) – The type of file IE: 'Micrograph movies'
- **exts** (*list*) – The acceptable extensions or file search strings IE: '.txt' to find u002a.txt or '_half1.star' for u002a_hal1.star
- **exact** (*bool*) – Match the file name(s) exactly, do not prepend a 'u002a'

Returns

Formatted for PyQt5 QFileDialog: 'name (u002aext1 u002aext2 u002aext3)'

Return type str

7.3 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y