
hpc-tools

Release 0.1.1dev1

Chris Finan

Feb 03, 2022

SETUP

1	Contents	3
1.1	Getting started with hpc-tools	3
1.2	General Workflow	5
1.3	Example job array	7
1.4	Command-line endpoints	15
1.5	Contributing to the project	22
1.6	Package management	23
2	Indices and tables	29

hpc-tools a Python package with some additional bash scripts/configuration files designed to simplify submitting array jobs on a cluster. Whilst it is written with Grid Engine in mind it can probably be re-purposed for other systems (such as LSF) - although this has not been tested in any way. Also, this has only been tested on the University College London (UCL) systems, Myriad and the CS cluster.

CONTENTS

1.1 Getting started with hpc-tools

version: 0.2.1dev0

[hpc-tools](#) is a Python package with some additional bash scripts/configuration files designed to simplify submitting array jobs on a cluster. Whilst it is written with Grid Engine in mind it can probably be re-purposed for other systems (such as LSF) - although this has not been tested in any way. Also, this has only been tested on the University College London (UCL) systems, Myriad and the CS cluster.

There is [online](#) documentation for hpc-tools and offline PDF documentation can be downloaded [here](#).

1.1.1 Installation instructions

At present, hpc-tools is undergoing development and no packages exist yet on PyPi. Therefore it is recommended that it is installed in either of the two ways listed below. First, clone this repository and then cd to the root of the repository.

```
git clone git@gitlab.com:cfinan/cluster.git
cd cluster
```

Installation not using any conda dependencies

If you are not using conda in any way then install the dependencies via `pip` and install hpc-tools as an editable install also via `pip`:

Install dependencies:

```
python -m pip install --upgrade -r requirements.txt
```

For an editable (developer) install run the command below from the root of the bio-misc repository (or drop `-e` for static install):

```
python -m pip install -e .
```

Installation using conda

A conda build is also available for Python v3.7, v3.8 and v3.9 on linux-64.

```
conda install -c cfin hpc-tools
```

If you are using conda and require anything different then please install with pip and install the dependencies with the environments specified in `resources/conda_env`. There is also a build recipe in `./resources/build/conda` that can be used to create new packages.

This will also install `pyjasub` and `make-jaf` as console script endpoints that will be executable from the command line.

Post Python install

After installation of `hpc-tools`, there are some manual setup steps.

Setup a mapping file

The main submission script, `pyjasub` works by building job submission commands from user arguments. Therefore there is a mapping step to map between the user arguments and the cluster arguments. This happens internally but how the mapping is performed is controlled by a mapping config file. Examples, can be seen in the `./resources/mappings` directory of the repository and is also shown below:

```
[mappings]
array=-t {0}-{1}:{2}

# Multiple arguments are separated by commas, such as two memory arguments
mem=-l hmem={0}, -l vmem={0}
time=-l h_rt={0}
nodes=-l hostname={0}
shell=-S {0}
out=-o {0}
err=-e {0}
batch=-tc {0}
tmp_size=-l tmpfs={0}
scratch_size=-l tscratch={0}
```

This follows a `.ini` format and the Python formatting characters `{0}`, `{1}` etc... will be substituted with arguments. The mappings should be under the section header `[mappings]`.

As a single the mappings file is expected per system, the location of the mappings file should be set in your `~/ .bashrc`:

```
export QSUB_PYTHON_MAPPING="/path/to/mappings.conf"
```

As the mappings file is a potential security risk, it should only be readable by you. This is checked by `pyjasub` and if any other users have any of `rwX` on the mapping file then `pyjasub` will error out with a `PermissionError`.

The abstraction of the cluster arguments has some plusses and minuses, it allows similar commands and job files can be used in different setups. However, this is at the expense of granular control. Having said that `pyjasub` can handle most everyday cluster uses and can probably be extended easily. Currently, `pyjasub` has only really been tested using `GridEngine`, however, it should be able to be adapted for `LSF`.

Add the bin directory to your PATH

There are a number of bash scripts used by pyjasub. Whilst, they are not obligatory and the user has full control of the task script, they can be used to perform some setup and cleanup operations. If you make sure that the directory `.resources/bin` is added to your PATH, for example, in your `~/ .bashrc` file:

```
export "${PATH}:/path/to/cluster/resources/bin"
```

1.2 General Workflow

Here the general idea behind hpc-tools is outlined along with some of the things that make your life on the cluster a bit easier. Once again, your experience on various HPCs may vary and this is geared towards GridEngine on UCL systems.

1.2.1 Array Jobs

An array job is a single cluster job that is split into multiple tasks. for example each task might do the same thing do different input files and produce a corresponding output file. So, you might want to perform the same actions on 10 different input files producing 10 different output files. So you would submit an array job that is 10 tasks. Obviously, you also have the option of submitting 10 separate jobs in a for loop. However, each of these will have to be scheduled and will have it's own job ID. So if you notice a problem with the jobs and they need to be killed, then you will have to kill 10 separate jobs. With an array job, you get a single job ID, so it is simple to kill with `qdel` (on GridEngine). Additionally, submitting 10 separate jobs puts unnecessary strain on the scheduler. So, if you have multiple tasks, it is a good idea to use array jobs.

So, how to you supply all the parameters required for each task in a job array? The easiest approach is to use a flat file, which has each task on a separate line and each parameter for a task in a separate column. The line in the file that corresponds to the current task can then be interpreted by the script running the task (called the task script) and the values in that line can be used as parameters to run the task. In this readme and in the code, this is referred to as a job array file and I would typically give it a `.job` extension although there is no requirement for this. However, there is no requirement for a job array file to be a flat file. It could be something more exotic such as an XML file, as long as the task script knows how to parse it to interpret what is needed to run the current task then that is all that is required.

So, how does the current task know what line it is supposed to read from the job array file? This is the job of a specially created environment variable called the `SGE_TASK_ID` (on GridEngine). This is available as `$SGE_TASK_ID` in a bash script. This will contain an integer, corresponding to the task that is currently be run. So, if it is task number 5 in the job array then `$SGE_TASK_ID` will equal 5. The task script will then know, to extract the line number 5 from the job array file and use it's contents as parameters to run the task.

The most tricky part of managing an array job is working out what tasks have failed and what tasks have been successful. Tasks can fail for a number of reasons, it could be they have exceeded their memory or time requirements, input/output files are missing or simply a bug in the code. Figuring out what has been successful and what has failed maybe easy if the array job is only 10 tasks but can be tricky/time consuming if there are 100,000s of tasks. There are several good practices that you can use to make your life a lot easier, this package implements some of them directly and assumes that you are using others:

1. Always make your task script a bash script (or whatever your favourite shells is): Whilst not an absolute requirement You can make sure that all your environment is setup (for example conda). The bash task script can then call whatever downstream scripts you need (for example python)
2. Use `err` and `log` files and name them appropriately. `err` files capture output to `STDERR` and outfiles capture output to `STDOUT`. For example make sure the variable `$TASK_ID` is in the file name specification. Also, remember

that these files will be appended to not overwritten. So if you re-run a job several times and point to the same files, then the output of all of your runs will be in the same files, this can get quite confusing.

3. Make sure that any normal output (progress) is output to STDOUT and any error output is redirected to STDERR. In Python, this will happen implicitly for `print` and `raises`, in Python a print statement can also be redirected to STDERR, using the `file=sys.stderr` argument. but in bash you will have to redirect `echo` to `1>&2` for STDERR.
4. Make sure that you can test your task script outside of the job array submission but in exactly the same way as it will be run when inside the job array. So, for example, you can submit an interactive job via `qssh` and profile the time and memory requirements, i.e. run via `/usr/bin/time -v`. It will also, allow you to efficiently debug your task without submitting the job. It is very difficult to debug a task by continuously submitting jobs and watching them fail.
5. A common cause of job failures are inaccessible input files and output file locations. These could be caused by a mistake in the file location or a permission error. It is good practice to make sure that all file/directory locations are present and accessible `__before__` the job has been submitted. This way, you do not waste time submitting loads of jobs and the deleting them when they all start to fail.
6. Many scripts will create temporary output or intermediate files. These way be created in a working output directory or in some sort of job runtime `tmp` space. What happens to these files when your jobs finishes or if your job errors out? There could be temporary files littered all over the file system and filling up precious `tmp` space on a node. It is good practice to make sure that your task script and any other processes it uses clean up after themselves both when they execute cleanly and when they fail. `bash traps` are very useful for this purpose, trapping `EXIT` and `ERR` signals and executing clean up functions in response. Similarly in Python, `try`, `except` and `finally` blocks or `context managers` can perform these tasks.
7. If you are submitting thousands of jobs, then sometimes you will get failures for unspecified reasons, it could be other users overloading nodes of filling up `/tmp`, sometimes, a job may run out of time or memory, even when you think it will be ok. Either way, the question is, how can you identify what tasks have failed and ones that have run to completion? If you have `err` and `out` files, these can be parsed to look for telltale signs of failure or completion, for example a `**** JOB COMPLETED ****` message as the last thing to `echo` out as the script exits, perhaps on a clean exit trap. However, parsing all these files can be error prone and cumbersome. A far better approach, is to look for the presence of expected output files. These paths will often be present in your job array file, so they could be checked against that to determine and flag tasks where the expected output is absent. However, what if your job runs out of memory in the middle of writing an output file? Then you will have an expected output file present but it will be half written. If you have set up an `ERROR` trap, you could have it fire in response to `ERR` exit to remove output files. However, it is unclear if these will get fired in an out of memory situation. A good practice to get into is to write output files to a temporary location and when written one of the final commands in the task script is to move the output to it's final location. That way, any during any mid run failures the output file will not be in it's final location. An additional safety check is to take the MD5 hash of the file before and after moving. If they do not match then the output file can be deleted before the script exists. This gets around the possibility of something going wrong when moving the output file. Either way, the end goal is to make sure that the output files are not present in their final location until you are 100% sure that they are the final output that you expect. Then you can safely say the job has completed by looking at the expect output locations in your job array file.

With the points above in mind, the general workflow for submitting an array job would be:

1. Create a job array file. There is a script/module `make-jaf` that can aid in doing this. See the documentation below for full details.
2. Write a task script that will perform each task of the job array. Whist, this is probably very specific for the task, there are some general skeleton task files in the `resources` directory that can be used as a base for this and adhere to many of the good practices listed above.
3. Write a job config (ini) file that details the parameters of the job.
4. Submit the job using `pyjasub` (python job array submission)

5. After the submission has finished - re-run pyjasub to determine if all the tasks were processed successfully and re-submit any failed tasks.

1.3 Example job array

Here we will run through a simple toy example of submitting a job with pyjasub. This assumes that you have installed the package and have your mapping file in place and the path to it set in your `~/ .bashrc`, if not please see [here](#). Also, to run the scripts in the example you will need to clone the repository and have access to `./cluster/resources/example` where `.` is the root of the repository.

1.3.1 Creating example files

For this there is a small script that will create a directory called `~/hpc_tools_examples` in the root of your home directory. If it already exists, it will error out. This directory, will have everything in it to run the examples. Please make sure you are in the `./cluster/resources/example` directory when running the script:

```
./create_example_files.sh
```

After running the test input files are available in `~/hpc_tools_examples/test_input` and the scripts etc are available in `~/hpc_tools_examples/example`

1.3.2 Creating a job array file

Now we will use the file and directory structure just created to build a job array file that has the following columns:

1. `rowidx` - A counter starting at 1 for the file non-header row. This always has to be present in this position in job array files used by pyjasub
2. `infile` - The location of the input file (what we just created with `./create_example_files.sh`)
3. `outfile` - The location of the output file that will be created by our task script

We will use the helper script `make-jaf` to create the basic file structure above. Now we create our job array file using the command below:

```
make-jaf -v \
  -i ~/hpc_tools_examples/test_input \
  -g '*/*.txt' \
  -o ~/hpc_tools_examples/test_output \
  --regexp 'i_am_a_big_long_file_name_with_(?P<B>\d+)_and_all_this_as_well_(?P<A>\
↵w+)\.txt' \
  --levels 1 \
  --strip_ext \
  --suffix ".out" \
  --outfile "./example_job_array.job"
```

The command above can also be run via the script `create_job_array.sh`

```
./create_job_array.sh
```

Running should produce an output similar to below:

```
=== make_general_job_array (hpc_tools v0.1.0a1) ===
[info] indir value: /home/rmjdcfi/hpc_tools_examples/test_input
[info] infiles length: 0
[info] inglob value: /*.txt
[info] levels value: 1
[info] outdir value: /home/rmjdcfi/hpc_tools_examples/test_output
[info] outfile value: ./example_job_array.job
[info] outfile_errors value: error
[info] prefix value: None
[info] regexp value: i_am_a_big_long_file_name_with_(?P<B>\d+)_and_all_this_as_well_(?P
↪<A>\w+)\.txt
[info] strip_ext value: True
[info] suffix value: .out
[info] verbose value: True
[info] 10 input found
[info] *** END ***
```

The job array file should look similar to this, although the home directory paths will differ:

```
$ cat example_job_array.job
rowidx  infile  outfile
1      /home/rmjdcfi/hpc_tools_examples/test_input/india/i_am_a_big_long_file_name_with_
↪9_and_all_this_as_well_nine.txt /home/rmjdcfi/hpc_tools_examples/test_output/india/
↪nine_9.out
2      /home/rmjdcfi/hpc_tools_examples/test_input/foxtrot/i_am_a_big_long_file_name_
↪with_6_and_all_this_as_well_six.txt /home/rmjdcfi/hpc_tools_examples/test_
↪output/foxtrot/six_6.out
3      /home/rmjdcfi/hpc_tools_examples/test_input/bravo/i_am_a_big_long_file_name_with_
↪2_and_all_this_as_well_two.txt /home/rmjdcfi/hpc_tools_examples/test_output/bravo/two_
↪2.out
4      /home/rmjdcfi/hpc_tools_examples/test_input/alfa/i_am_a_big_long_file_name_with_
↪1_and_all_this_as_well_one.txt /home/rmjdcfi/hpc_tools_examples/test_output/alfa/one_
↪1.out
5      /home/rmjdcfi/hpc_tools_examples/test_input/golf/i_am_a_big_long_file_name_with_
↪7_and_all_this_as_well_seven.txt /home/rmjdcfi/hpc_tools_examples/test_output/golf/
↪seven_7.out
6      /home/rmjdcfi/hpc_tools_examples/test_input/hotel/i_am_a_big_long_file_name_with_
↪8_and_all_this_as_well_eight.txt /home/rmjdcfi/hpc_tools_examples/test_output/
↪hotel/eight_8.out
7      /home/rmjdcfi/hpc_tools_examples/test_input/delta/i_am_a_big_long_file_name_with_
↪4_and_all_this_as_well_four.txt /home/rmjdcfi/hpc_tools_examples/test_output/delta/
↪four_4.out
8      /home/rmjdcfi/hpc_tools_examples/test_input/echo/i_am_a_big_long_file_name_with_
↪5_and_all_this_as_well_five.txt /home/rmjdcfi/hpc_tools_examples/test_output/echo/
↪five_5.out
9      /home/rmjdcfi/hpc_tools_examples/test_input/charlie/i_am_a_big_long_file_name_
↪with_3_and_all_this_as_well_three.txt /home/rmjdcfi/hpc_tools_examples/test_
↪output/charlie/three_3.out
10     /home/rmjdcfi/hpc_tools_examples/test_input/juliect/i_am_a_big_long_file_name_
↪with_10_and_all_this_as_well_ten.txt /home/rmjdcfi/hpc_tools_examples/test_
↪output/juliect/ten_10.out
```

The command above illustrates several useful features of `make_general_job_array_file`, in fact, `make_general_job_array_file` can be imported and extended to make more specific job arrays with other

parameters in the file.

So the first thing to point out is that we supplied an input directory `-i` and an input glob `-g`. These are placed together and any files matching that glob will be used. We could have also used the unix globbing and placed `/home/rmjdcfi/hpc_tools_examples/test_input/*/*.txt` after the `--outfile` argument. Also, if supplying a glob via `-g` then make sure it is single quoted so unix path expansion does not take place.

We also specified an output directory argument. If we had not, then the input directory would have been used as the output directory. The output directory is also created if it does not exist.

You will also notice that there are a few differences between the structure of the input file name and that of the output file. Firstly, they have different extensions, the input files have `.txt` and the output files have `.out`, this is achieved by a combination of `--strip_ext` and `--suffix` arguments. Also, our long unwieldy file name has been reduced considerably. This is a result of the `--regex` argument. Here we supplied a regular expression to match against the file name and capture the parts we want to keep using python named capture groups, labelled alphabetically in the order we want them in the output file name. This allows us to extract the relevant parts of the file name. If this option is used, each file name must match the regex and the output files produced must be unique in the whole job array file (see the `--outfile_errors` argument if not). Also, another important point about the `--regex` argument is that it must also be single quoted when supplied on the command line.

Finally you will also notice that we retained the “phonetic” sub-directory structure of the input files, with `alfa`, `bravo`, `charlie`, `delta` etc... . This was accomplished using the `--levels` argument. So if `--levels 0` was supplied (the default) then your output files would not have the phonetic sub directory attached, so instead of this `/home/rmjdcfi/hpc_tools_examples/test_output/juliect/ten_10.out`, we would have had this: `/home/rmjdcfi/hpc_tools_examples/test_output/ten_10.out`. Also, if we had supplied `--levels 2`, then we would have had this: `/home/rmjdcfi/hpc_tools_examples/test_output/test_input/juliect/ten_10.out`. So `--levels N` will take `N` many directory levels above the input file and add it to below the output directory. All the sub-directories defined with `--levels` are created automatically so do not have to be created by the task script.

1.3.3 Testing the task script

You will want/need to be able to test your task script from outside of your job array and that is what we will do here. The example task script is very simple. It accepts 3 required positional arguments and an optional argument

1. The job array file
2. Temp location
3. Step size
4. Task ID (optional) - if not supplied then it will default to the `$SGE_TASK_ID` environment variable

When running as a job array, the first 3 arguments are supplied by `pyjasub` and the fourth one defaults to the `$SGE_TASK_ID` supplied by `GridEngine`. However, when testing on the command line, we supply all 4 with the fourth one being the job row in the job array file we want to test.

Now, ensure that the task script is executable with `chmod +x example_task.sh` and assuming you are in the `example` directory. Running the first task from the job array should look like this:

```
~/hpc_tools_examples/example/example_task.sh ~/hpc_tools_examples/example/example_job_
array.job ~/Scratch/ 1 1
=== ./example_task.sh ===
[info] started at Wed  5 Feb 11:39:11 GMT 2020
[info] hostname=login12.myriad.ucl.ac.uk
[info] job array file=example_job_array.job
[info] temp location=/home/rmjdcfi/Scratch/
[info] step=1
```

(continues on next page)

(continued from previous page)

```
[info] job idx line (SGE_TASK_ID)=1
[info] pull line (accounting for header)=2
[info] quit line=3
[info] # infiles=1
[info] outfile=/home/rmjdcfi/hpc_tools_examples/test_output/india/nine_9.out
[info] temp file=/home/rmjdcfi/Scratch/.RUN_rmjdcfi_1_mipaXs6bp8
[info] ended at Wed 5 Feb 11:39:11 GMT 2020
*** END ***
```

And will produce an output file like this:

```
$ cat /home/rmjdcfi/hpc_tools_examples/test_output/india/nine_9.out
output file:
This is the file: /home/rmjdcfi/hpc_tools_examples/test_input/india/i_am_a_big_long_file_
↪name_with_9_and_all_this_aswell_nine.txt
```

Now we attempt to run the second task in the job array. Remember that we have set this to fail before the output is written

```
~/hpc_tools_examples/example/example_task.sh ~/hpc_tools_examples/example/example_job_
↪array.job ~/Scratch/ 1 2
=== ./example_task.sh ===
[info] started at Wed 5 Feb 11:41:45 GMT 2020
[info] hostname=login12.myriad.ucl.ac.uk
[info] job array file=example_job_array.job
[info] temp location=/home/rmjdcfi/Scratch/
[info] step=1
[error] simulated failure job index: 2
```

Whilst not indicated in the output, our second task should have output a file called `hpc_tools_examples/test_output/foxtrot/six_6.out`, it will not exist.

So we know that our task script is working ok, we can now prepare to run our job array

1.3.4 Job array arguments

When supplying arguments to pyjasub, they can either be supplied via the command line or a config file (in ini format). A config file is easier as it is less error prone. However, both can be used with command like arguments overriding those in the config file. An example config file is given to run the job. It will output the logs of the job into the root of the `~/hpc_tools_examples` directory. It is also shown below:

```
# A job array config file for extracting the top hits from the interval data
[qsub]
# The general parameters for the job. The script has defaults for many of these
step=1
batch=50
mem=1M
shell=/bin/bash
time=00:01:00
nodes=None
tmp_size=10M
tmp_dir=~/Scratch
```

(continues on next page)

(continued from previous page)

```

[logs]
# The root for all the job/run logs. If it does not exist it will be created
log_dir=~/.hpc_tools_examples/example_job_logs

[job_array]
# The location of the job array file, I have given a full path
# so there are no issues, relative paths are ok, but remember they will
# be relative to where the script is run not where the config file is
# also ~/ can be used
ja_file=~/.hpc_tools_examples/example/example_job_array.job

# The location of the task script that runs the job. Task scripts should
# accept 4 arguments.
# 1. the job array file
# 2. the location of the tmp folder i.e. scratch
# 3. the step size for the job array
# 4. the job ID i.e. SGE_TASK_ID
task_script=~/.hpc_tools_examples/example/example_task.sh

# A comma separated list of column names in the jobarray files
# where the input files are located and the output files are located
# only list them if you want the presence of the input files/output
# files checked by the submission script. if the column name contains
# a comma, protect with quotes ""
infile=infile
outfile=outfile

[script]
# Arguments to the script, at the moment it is just verbose that can be
# true or false
verbose=true

```

1.3.5 Running the jobs

Now with everything in place the jobs can be submitted with the command:

```
pyjasub ~/.hpc_tools_examples/example/example_job.cnf
```

Which should produce a report like this if successfully submitted. Note that pyjasub detects that one of the tasks has already been run (it was run during testing) so it will only submit none tasks. We expect all the evenly numbered tasks to fail as we have engineered it that way, so we expect at least 4 failures on this run, then 2 failures on the second run, 1 failure on the 3rd run and 0 failures on the fourth run. So when we run the fifth time we should get a message saying that the job is complete. Of course, the numbers above assume that no failures occur for other reasons that we can't control.

```

=== pyjasub (hpc_tools v0.1.0a1) ===
[info] step: 1
[info] batch: 50
[info] mem: 1M
[info] shell: /bin/bash

```

(continues on next page)

(continued from previous page)

```

[info] time: 00:01:00
[info] nodes: None
[info] tmp_dir: ~/Scratch
[info] scratch_size: None
[info] tmp_size: 10M
[info] log_dir: ~/hpc_tools_examples/example_job_logs
[info] ja_file: ~/hpc_tools_examples/example/example_job_array.job
[info] task_script: ~/hpc_tools_examples/example/example_task.sh
[info] infiles: ['infile']
[info] outfiles: ['outfile']
[info] full job log directory '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs'
[info] creating log directory '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs'
[info] run started at: '2020-02-05 12:29:31.538651'
[info] current run number: '1'
[info] run log path: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/run.log'
[info] run directory: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_
↳1580905771'
[info] run out files: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_
↳1580905771/out/\$TASK_ID_1_1580905771.out'
[info] run err files: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_
↳1580905771/err/\$TASK_ID_1_1580905771.err'
[info] full job array path: '/lustre/home/rmjdcfi/hpc_tools_examples/example/example_job_
↳array.job'
[info] job array md5: 'db812ee5a433b04843353f4ef813796b'
[info] full task script path: '/lustre/home/rmjdcfi/hpc_tools_examples/example/example_
↳task.sh'
[info] task script md5: '1278fb188fb52d4c6615325db565fa8a'
[info] copying task script: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_
↳1580905771/1278fb188fb52d4c6615325db565fa8a.task'
[info] run job array path: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_
↳1580905771/run_job_array.job'
[info] check results below:
[info] input file present: '10'
[info] input file absent: '0'
[info] output file present: '1'
[info] output file absent: '9'
[info] total tasks: '10'
[info] tasks to be submitted: '9'
[run] You are about to run '9' tasks over '9' rows in a job array with a step of '1',
↳using the following command:
[run] qsub -t 1-9:1 -e /lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_
↳1580905771/err/\$TASK_ID_1_1580905771.err -o /lustre/home/rmjdcfi/hpc_tools_examples/
↳example_job_logs/1_1580905771/out/\$TASK_ID_1_1580905771.out -l mem=1M -S /bin/bash -l
↳h_rt=00:01:00 -l tmpfs=10M /lustre/home/rmjdcfi/hpc_tools_examples/example/example_
↳task.sh /lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/1_1580905771/run_job_
↳array.job /lustre/scratch/scratch/rmjdcfi 1
[run] if all this looks correct, press [ENTER] to submit or [CTRL-C] to quit >

[run] submitting job...
[run] submitted array job with ID 2768482
[info] run log details below...
+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+
↳-----+-----+

```

(continues on next page)

(continued from previous page)

```

| run_time          | run_dir          | step | batch | mem | time       | total_
↳ tasks | no_of_tasks_submitted | infile_exist | infile_missing | outfile_exist |
↳ outfile_missing | job_id |
+-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+
| 2020-02-05 12:29:31.538651 | 1_1580905771 | 1    | 50    | 1M  | 00:01:00 | 10
↳ | 9          | 10          | 0    |      | 1    |          | 9
↳ | 2768482 |
+-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+
[info] *** job submission finished ***
*** END ***

```

After the 5th run you should see something like this, and you will get asked if you want to archive all the log files. Pressing ENTER will write a tar file on the run directory contents and delete the run directory:

```

$ pyjasub example_job.cnf
=== pyjasub (hpc_tools v0.1.0a1) ===
[info] step: 1
[info] batch: 50
[info] mem: 1G
[info] shell: /bin/bash
[info] time: 00:10:00
[info] nodes: None
[info] tmp_dir: ~/Scratch
[info] scratch_size: None
[info] tmp_size: 1G
[info] log_dir: ~/hpc_tools_examples/example_job_logs
[info] ja_file: ~/hpc_tools_examples/example/example_job_array.job
[info] task_script: ~/hpc_tools_examples/example/example_task.sh
[info] infiles: ['infile']
[info] outfiles: ['outfile']
[info] full job log directory '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs'
[info] run started at: '2020-02-05 14:15:24.519655'
[info] current run number: '5'
[info] run log path: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/run.log'
[info] run directory: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/5_
↳ 1580912124'
[info] run out files: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/5_
↳ 1580912124/out/\$TASK_ID_5_1580912124.out'
[info] run err files: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/5_
↳ 1580912124/err/\$TASK_ID_5_1580912124.err'
[info] full job array path: '/lustre/home/rmjdcfi/hpc_tools_examples/example/example_job_
↳ array.job'
[info] job array md5: 'db812ee5a433b04843353f4ef813796b'
[info] full task script path: '/lustre/home/rmjdcfi/hpc_tools_examples/example/example_
↳ task.sh'
[info] task script md5: '1278fb188fb52d4c6615325db565fa8a'
[info] previous task script md5: '1278fb188fb52d4c6615325db565fa8a'
[info] copying task script: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/5_
↳ 1580912124/1278fb188fb52d4c6615325db565fa8a.task'

```

(continues on next page)

(continued from previous page)

```

[info] run job array path: '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs/5_
↳1580912124/run_job_array.job'
[info] check results below:
[info] input file present: '10'
[info] input file absent: '0'
[info] output file present: '10'
[info] output file absent: '0'
[info] total tasks: '10'
[info] tasks to be submitted: '0'
[complete] no tasks to submit - job is complete
[info] run log details below...
+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+
| run_time          | run_dir          | step | batch | mem | time       | total_
↳tasks | no_of_tasks_submitted | infile_exist | infile_missing | outfile_exist |
↳outfile_missing | job_id          |
+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+
| 2020-02-05 13:17:21.707045 | 1_1580908641 | 1    | 50    | 1G  | 00:10:00 | 10
↳ | 9                      | 10          | 0    |      | 1   |          | 9
↳ | 2768778                |            |
| 2020-02-05 13:43:39.033742 | 2_1580910219 | 1    | 50    | 1G  | 00:10:00 | 10
↳ | 4                      | 10          | 0    |      | 6   |          | 4
↳ | 2768812                |            |
| 2020-02-05 14:01:20.850012 | 3_1580911280 | 1    | 50    | 1G  | 00:10:00 | 10
↳ | 2                      | 10          | 0    |      | 8   |          | 2
↳ | 2768958                |            |
| 2020-02-05 14:09:40.718982 | 4_1580911780 | 1    | 50    | 1G  | 00:10:00 | 10
↳ | 1                      | 10          | 0    |      | 9   |          | 1
↳ | 2768969                |            |
| 2020-02-05 14:15:24.519655 | 5_1580912124 | 1    | 50    | 1G  | 00:10:00 | 10
↳ | 0                      | 10          | 0    |      | 10  |          | 0
↳ | NOT_SUBMITTED |
+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+-----+-----+-----+-----+-----+
↳+-----+-----+
[info] job is complete
[info] *** job submission finished ***
[archive] do you want to archive the run files, press [ENTER] to archive or [CTRL-C] to
↳quit >

[archive] archiving to '/lustre/home/rmjdcfi/hpc_tools_examples/example_job_logs.tar.gz'
*** END ***

```

TODO: Add some notes about the run directories

1.4 Command-line endpoints

Below is a list of all the command line endpoints installed with hpc-tools. Many of these are mentioned in their respective context throughout the documentation but they are listed here all in a single place.

1.4.1 pyjasub

pyjasub is the main job array submission script and should be available after installation of hpc-tools.

Setup

If you have already followed the post Python installation tasks in getting started, then you can skip this section. If not, then please complete the steps below. pyjasub works by building job submission commands from user arguments. Therefore there is a mapping step to map between the user arguments and the cluster arguments. This happens internally but how the mapping is performed is controlled by a mapping config file. Examples, can be seen in the `./resources/mappings` directory and is also shown below:

```
[mappings]
array=-t {0}-{1}:{2}

# Multiple arguments are separated by commas, such as two memory arguments
mem=-l hmem={0}, -l vmem={0}
time=-l h_rt={0}
nodes=-l hostname={0}
shell=-S {0}
out=-o {0}
err=-e {0}
batch=-tc {0}
tmp_size=-l tmpfs={0}
scratch_size=-l tscratch={0}
```

This follows a `.ini` format and the python formatting characters `{0}`, `{1}` etc... will be substituted with arguments. The mappings should be under the section header `[mappings]`.

As a single the mappings file is expected per system, the location of the mappings file should be set in your `~/ .bashrc`:

```
export QSUB_PYTHON_MAPPING="/path/to/mappings.conf"
```

As the mappings file is a potential security risk, it should only be readable by you. This is checked by pyjasub and if any other users have any of `rwX` on the mapping file then pyjasub will error out with a `PermissionError`.

The abstraction of the cluster arguments has some plusses and minuses, it allows similar commands and job files can be used in different setups. However, this is at the expense of granular control. Having said that pyjasub can handle most everyday cluster uses and can probably be extended easily. Currently, pyjasub has only really been tested using GridEngine, however, it should be able to be adapted for LSF.

There are a number of bash scripts used by pyjasub. Whilst, they are not obligatory and the user has full control of the task script, they can be used to perform some setup and cleanup operations. If you make sure that the directory `./resources/bin` is added to your `PATH`, for example, in your `~/ .bashrc` file:

```
export "${PATH}:/path/to/cluster/resources/bin"
```

Usage

The options from the command line are as follows, these take priority over options defined in the config file.

Submit a qsub job array according to the options and configuration that has been supplied

```
usage: pyjasub [-h] [--project PROJECT] [--paid PAID] [--step STEP] [--batch BATCH] [--  
↪mem MEM] [--shell SHELL] [--time TIME] [--nodes NODES] [--scratch_dir SCRATCH_DIR] [--  
↪tmp_dir TMP_DIR] [--scratch_size SCRATCH_SIZE] [--tmp_size TMP_SIZE]  
           [--log_dir LOG_DIR] [--ja_file JA_FILE] [--task_script TASK_SCRIPT] [--  
↪infiles INFILES [INFILES ...]] [--outfiles OUTFILES [OUTFILES ...]] [-v] [-w]  
           [config]
```

Positional Arguments

config	The job config file, options in the config file are overwritten by options on the command line
---------------	--

Named Arguments

--project	Use specific project nodes and potentially a different policy
--paid	Force the tasks to use any paid resources specified by project
--step	the step size for the array job
--batch	the number of concurrent jobs that can be ran simultaneously
--mem	The memory for the job
--shell	The shell to use for the job
--time	The max time for the job
--nodes	restrict the job to running on certain nodes
--scratch_dir	The scratch location
--tmp_dir	The location or tmp
--scratch_size	The scratch space that is required
--tmp_size	The tmp space that is required
--log_dir	The space to log all the job parameters and err/out files
--ja_file	The location of the job array file
--task_script	The location of the job array task script
--infiles	space separated list of column names in the job array file that contain input files to check they are present
--outfiles	space separated list of column names in the job array file that contain output files to check they are present
-v, --verbose	Print out progress Default: False

-w, --whole-steps If step size is > 1 then this ensures that the step size is a multiple of total tasks and will throw an error if not. This treats all the rows steps in the job array file as a logical unit and ensures that all the output files for the steps in the task exists if not they are tasked for re-running again. If this is not set then the rows are not treated as logical blocks

Default: False

Job submission workflow

pyjasub is designed to make it easier to manage array job submissions and reruns of failed tasks. It can be run on the command line or be imported and the main submission class can be inherited from to tailor a job submission.

The general workflow for using pyjasub is as follows:

1. Create a job array file using a script such as `make_general_job_array_file`, or deriving from it. A job array file is a standard flat file that should have a header and be tab delimited. Its role is to supply task specific arguments to each task in the job array. The first column should be a column called `rowidx` and should contain an incremental counter with the row after the header being 1.
2. Create a job config file that will supply the parameters for the job. Whilst this is optional and the parameters can be supplied on the command line it is recommended to use a config file to avoid errors, arguments given on the command line can be used to override config file parameters. An example of a config file is shown below:

```
# A job array config file for extracting the top hits from the interval data
[qsub]
# The general parameters for the job. The script has defaults for many of these
step=1
batch=50
mem=2G
shell=/bin/bash
time=00:20:00
nodes=None
tmp_size=10G
tmp_dir=~/.Scratch

[logs]
# The root for all the job/run logs. If it does not exist it will be created
log_dir=~/.Scratch/somalogic_interval_top_hits_jobs

[job_array]
# The location of the job array file
ja_file=~/.somalogic_interval.jobs

# The location of the task script that runs the job. Task scripts should
# accept 4 arguments.
# 1. the job array file
# 2. the location of the tmp folder i.e. scratch
# 3. the step size for the job array
# 4. the job ID i.e. SGE_TASK_ID
task_script=~/.code/gwas_import_pipeline/scripts/extract_pvalue_job_array_task.sh

# A comma separated list of column names in the jobarray files
# where the input files are located and the output files are located
# only list them if you want the presence of the input files/output
```

(continues on next page)

(continued from previous page)

```
# files checked by the submission script. if the column name contains
# a comma, protect with quotes ""
infile=infile
outfile=outfile

[script]
# Arguments to the script, at the moment it is just verbose that can be
# true or false
verbose=true
```

3. Keep on running pyjasub until all the tasks have been completed. If you have some stubborn tasks the keep failing the check your error and out files. These will be located in the log directory under the different run directories.
4. Once completed, you will be asked if you want to archive your run files. Archiving creates a compressed tar archive and will delete your run directories. An example from a job array that took 4 runs to complete is shown below:

```
$ pyjasub interval.ja.conf
=== pyjasub (hpc_tools v0.1.0a1) ===
[info] step: 1
[info] batch: 50
[info] mem: 2G
[info] shell: /bin/bash
[info] time: 00:20:00
[info] nodes: None
[info] tmp_dir: ~/Scratch
[info] scratch_size: None
[info] tmp_size: 10G
[info] log_dir: ~/Scratch/somalogic_interval_top_hits_jobs
[info] ja_file: ~/somalogic_interval.jobs
[info] task_script: ~/code/gwas_import_pipeline/scripts/extract_pvalue_job_array_task.sh
[info] infile: ['infile']
[info] outfile: ['outfile']
[info] full job log directory '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_
↳ hits_jobs'
[info] run started at: '2020-02-04 18:07:12.963490'
[info] current run number: '5'
[info] run log path: '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_jobs/
↳ run.log'
[info] run directory: '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_jobs/
↳ 5_1580839632'
[info] run out files: '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_jobs/
↳ 5_1580839632/out/\$TASK_ID_5_1580839632.out'
[info] run err files: '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_jobs/
↳ 5_1580839632/err/\$TASK_ID_5_1580839632.err'
[info] full job array path: '/lustre/home/rmjdcfi/somalogic_interval.jobs'
[info] job array md5: '6b637ce3197d9fcd1ca9bfa875b88a10'
[info] full task script path: '/lustre/home/rmjdcfi/code/gwas_import_pipeline/scripts/
↳ extract_pvalue_job_array_task.sh'
[info] task script md5: '9094df2fa17e6ff390493094b2c32c9c'
[info] previous task script md5: '9094df2fa17e6ff390493094b2c32c9c'
[info] copying task script: '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_
↳ jobs/5_1580839632/9094df2fa17e6ff390493094b2c32c9c.task'
```

(continues on next page)

(continued from previous page)

```

[info] run job array path: '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_
↳ jobs/5_1580839632/run_job_array.job'
[info] check results below:
[info] input file present: '72226'
[info] input file absent: '0'
[info] output file present: '3283'
[info] output file absent: '0'
[info] total tasks: '3283'
[info] tasks to be submitted: '0'
[complete] no tasks to submit - job is complete
[info] run log details below...
+-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+
| run_time          | run_dir          | step | batch | mem | time       | total_
↳ tasks | no_of_tasks_submitted | infile_exist | infile_missing | outfile_exist |
↳ outfile_missing | job_id          |
+-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+
| 2020-01-22 17:03:34.802277 | 1_1579712614 | 1    | 50    | 2G  | 00:20:00 | 3283    |
↳ | 3283                | 72226        | 0    |      | 0    |          | 3283    |
↳ | 2582791            |              |      |      |      |          |          |
| 2020-02-03 17:38:19.356899 | 2_1580751499 | 1    | 50    | 2G  | 00:20:00 | 3283    |
↳ | 246                  | 72226        | 0    |      | 3037 |          | 246     |
↳ | 2751956            |              |      |      |      |          |          |
| 2020-02-04 05:56:59.818196 | 3_1580795819 | 1    | 50    | 2G  | 00:20:00 | 3283    |
↳ | 246                  | 72226        | 0    |      | 3037 |          | 246     |
↳ | 2755344            |              |      |      |      |          |          |
| 2020-02-04 06:34:21.751488 | 4_1580798061 | 1    | 50    | 2G  | 00:20:00 | 3283    |
↳ | 245                  | 72226        | 0    |      | 3038 |          | 245     |
↳ | 2755345            |              |      |      |      |          |          |
| 2020-02-04 18:07:12.963490 | 5_1580839632 | 1    | 50    | 2G  | 00:20:00 | 3283    |
↳ | 0                    | 72226        | 0    |      | 3283 |          | 0       |
↳ | NOT_SUBMITTED      |              |      |      |      |          |          |
+-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+-----+-----+-----+-----+-----+
↳ +-----+-----+
[info] job is complete
[info] *** job submission finished ***
[archive] do you want to archive the run files, press [ENTER] to archive or [CTRL-C] to
↳ quit >

[archive] archiving to '/lustre/scratch/scratch/rmjdcfi/somalogic_interval_top_hits_jobs.
↳ tar.gz'
*** END ***

```

Checks on the job array file

Each run of `pyjasub` will start with a check on the job array file. The following are checked, if they do not pass then we get a fatal error

1. Has the job array file changed? The job array file is crucial to the integrity of all the jobs, therefore it must stay constant for the lifetime of all runs of the job.
2. Is the first column of the job array `rowidx`, if not it is an error.
3. For each row in the job array file, is it the same length (no of columns) as the header
4. For each row in the job array file, is the first column of the row an expected incremental counter i.e. first column of the first row after the header should have the value 1.

Defining a completed job

So here it can get tricky. We have to take into account the step size when evaluating the output files. So, we have to have all output files present for all steps in the task.

1.4.2 `make-jaf`

Create a general job array file that has 3 columns. The file also has a header and is tab separated. This can be executed as a script or imported as a module to aid building of more specific job array files.

Usage

The options from the command line are as follows, these take priority over options defined in the config file.

Generate a skeleton for a job array file with input file, output file columns and an index column. All input files are checked to make sure they can be accessed and it is a fatal error if they can not, similarly, each output location is checked to see if it is writable. If the output file already exists then it is a fatal error

```
usage: make-jaf [-h] [-f OUTFILE] [-i INDIR] [-g INGLOB] [-o OUTDIR] [-l LEVELS] [-e
↪ {error,ignore,flatten}] [-p PREFIX] [-s SUFFIX] [-r REGEXP] [-x] [-v] [infiles_
↪ [infiles ...]]
```

Positional Arguments

infiles

The input files for the job array, if none are supplied then `-indir` should be set and all the files in `indir` are used as input

Named Arguments

-f, --outfile	the output file, if not provided, output is to STDOUT
-i, --indir	the input directory, this should be set if there are no input files, it can also be used in conjunction with input files
-g, --inglob	An input directory glob that is used in conjunction with the <code>--indir</code> , the default is no pattern
-o, --outdir	An output directory, if this is not set then it is assumed that the output directory is the same directory as the directory that each input file resides in. If the outdir does not exist then it will be created
-l, --levels	active when <code>--outdir</code> is used, this uses number of levels directory structure down from infile and is appended to the outdir. e.g. <code>outdir=/home/bob/my_data</code> , <code>input file=/data/to/process/file.txt</code> . If <code>--levels = 0</code> then the outfile will be <code>/home/bob/my_data/file.txt</code> , if <code>--levels 1</code> is used, then <code>/home/bob/my_data/process/file.txt</code> and if <code>levels 2</code> is used, then <code>/home/bob/my_data/to/process/file.txt</code> Default: 0
-e, --outfile_errors	Possible choices: error, ignore, flatten action to take if there are errors with non-unique output files. 'flatten', will make the output file unique and group all the input files into a single column delimited with a pipe ' ' Default: "error"
-p, --prefix	A prefix to add to each output file, the default is ''
-s, --suffix	A suffix to add to each output file, the default is '_output'
-r, --regexp	A regular expression used to generate the output file name from the full input file path. The regexp will be applied using <code>re.search</code> . All groups will be captured, although the groups required in the output file name should be labels '1', '2', '3' using the named capture groups <code>?P<I></code> etc... . Remember to also use non-capturing groups <code>(?:)</code> where necessary.
-x, --strip_ext	shall the extension be stripped from the outfile name Default: False
-v, --verbose	shall I print progress to STDERR? Default: False

Output file

When executed as a script it will generate a three column TAB delimited job array file with the columns:

1. `rowidx` An integer counter of the job number. Note that `pyjasub` always expects the first column to be a `rowidx` column.
2. `infile` An input file column, this is not mandatory but most job tasks will probably require at least a single input file. Note, that it is valid for this column to contain a pipe '|' separated set of input files in this column also. These are generated if the `--flatten` option is active and the output file names are not unique.
3. `outfile` An output file column. Note it is also valid to have a pipe '|' separated set of output files in this column. Although, `make-jaf` does not do this.

The `make-jaf` script firsts gathers all the available input files, depending on the user arguments. The user can either supply input files or directories. If input files are checked to make sure that they are actually files and not directories, any input files that do end up being directories are ignored. If the `infile` positional argument, `--indir` and `--inglob` are all used then the full complement of files gathered by all of them will be used. All input files are checked to make sure they are readable by opening and closing the file. If this fails then it is a fatal error.

Output file names are generated from the input file names. The `--outdir`, `--levels`, `--regexp`, `--strip_ext`, `--prefix` and `--suffix` options can be used to modify the output file name. If `--outdir` is supplied then the basename of the input file is added to the output file name. If `--levels` is also supplied, then the `--levels` number of directories up from the input files are created below the `--outdir`. The default is 0, i.e. no levels down from the input directory.

The `--prefix` and `--suffix` generates an output file name with strings added to the start or the end of the input file basename. `--suffix` is added before the file extension. `--strip_ext` will generate an output file name from the input file name with the extension removed. So a new output file extension can be generated with a combination of `--strip_ext` and `--suffix`.

The `--regexp` argument allows the user to supply a regexp on the command line that will be applied to the full path of the input file and generate an output file name. Even if `--regexp` is included, all the other options still apply. This regexp uses named capture groups to extract the parts that should be used in the output file name. the capture groups should be named in uppercase alphabet characters and the order in the alphabet is the order they appear in the output file name. i.e. `r'(?P<A>first_bit)'`.

By default, if all the output file names are not unique, then an error is generated (`--outfile_errors 'error'`). This can be set to `'ignore'` if this is what is intended, i.e. it could be that the job array has a different step size. Also, this can be set to `'flatten'` and in this case the input files are flattened into a pipe `|` separated list to give a unique number of output files.

Basic API usage

This can also be imported as a module. See the reference documentation for the functions within.

```
from cluster import make_general_job_array_file
```

1.5 Contributing to the project

I welcome contributions from all users to the project:

1. Documentation fixes/updates
2. Example notebooks to illustrate features/methods/Example bash scripts for `./resources/bin`
3. Bug fixes
4. New scripts

However, we ask that contributors please follow these guidelines when contributing to the package. This allows us to retain some level of homogeneity throughout a project and will make it easy to maintain and improve in the long run.

A list of current contributors can be found in the [root of the repository](#).

1. Please ensure that all code follows the Python [PEP-8](#) standard. the easiest way to do this is to use an IDE with a good [linter](#).
2. Please ensure that your code is well commented, so that programmers of varying abilities have a good chance in following it.
3. Please make sure that any code, or bug-fixes you contribute are accompanied by a [pytest](#).

4. Please run a full `pytest` prior to merging your code and ensure all tests pass (or that your new code has not caused any new test failures).
5. Please make sure that all modules, classes, functions (even private ones) have docstrings. This enables us to easily produce high quality documentation and improves the user experience. `hpc_tools` uses Numpy docstrings for the most part. Please follow [this](#) excellent guide if unsure.
6. Any HOWTOs or extended user documentation should be written in `reStructuredText` (preferably) or `Markdown`.
7. We incorporate Jupyter notebooks into the documentation, so please ensure they are fairly quick to run and do not have reams of output associated with them. Also, please make sure that all output is collapsed (removed) prior to committing the notebook to git.
8. Please make sure that any test data you submit is small and compressed and please do not commit any individual level data that is not already in the public domain.

If you want to contribute and are unsure on any of these points please [contact us](#). We will acknowledge any contributions in future publications, either as acknowledgements or authorship positions in the case of significant contributions.

Many thanks in advance!

1.6 Package management

Here is documentation on the maintenance of the package. This is not really geared towards end users but rather a log of how version numbering is handled and how documentation is built and maintained. Throughout this section `./` indicates the root of the repository.

1.6.1 Versioning

`hpc-tools` uses 3 level versioning:

1. Major version number
2. Minor version number
3. Patch version number

For all non-stable versions (which is all of them at present), these are augmented with a:

- release - either dev, a (alpha), b (beta), rc (release candidate), prod (production)
- build number, this build number is associated with a release

So, a development version number will look like `0.2.0dev0`

While `hpc-tools` is unfinished, the versioning will be slightly ad-hoc but will stabilise once the minimal code base is in place.

`hpc-tools` uses `bump2version` to increment the version numbers throughout the package. The version numbers are located in several different places (a `.` denotes the root of the repository):

- `./bumpversion.cfg`
- `./README.md`
- `./genomic_config/__init__.py`
- `./setup.py`
- `./VERSION`
- `./docs/source/conf.py`

- ./resources/build/conda/py*/meta.yaml

The bumpversion config file is shown below but is also a hidden file in the root of the repository:

```
[bumpversion]
current_version = 0.2.0dev0
commit = True
tag = True
parse = (?P<major>\d+)\.(?P<minor>\d+)\.(?P<patch>\d+)(\-?(?P<release>[a-z]+)(?P<build>\d+))?
serialize =
    {major}.{minor}.{patch}{release}{build}
    {major}.{minor}.{patch}

[bumpversion:part:release]
optional_value = prod
first_value = dev
values =
    dev
    a
    b
    rc
    prod

[bumpversion:part:build]

[bumpversion:file:VERSION]

[bumpversion:file:./README.md]
search = __version__: `{current_version}`
replace = __version__: `{new_version}`

[bumpversion:file:./_version.py]
search = __version__ = '{current_version}'
replace = __version__ = '{new_version}'

[bumpversion:file:./genomic_config/__init__.py]
search = __version__ = '{current_version}'
replace = __version__ = '{new_version}'

[bumpversion:file:./setup.py]
search = VERSION = '{current_version}'
replace = VERSION = '{new_version}'

[bumpversion:file:./docs/source/conf.py]
search = release = '{current_version}'
replace = release = '{new_version}'
```

The procedure for incrementing the version number is as follows. This assumes that the patch number is being bumped and you are located in the root of the repository (where your `.bumpconfig.cfg` file is located):

1. run `bump2version` in “dry-run” (`-n`) mode with `--verbose` to make sure everything is ok:

```
$ bump2version --verbose -n patch
current_version=0.2.0dev0
```

(continues on next page)

(continued from previous page)

```

commit=True
tag=True
parse=(?P<major>\d+)\.(?P<minor>\d+)\.(?P<patch>\d+)(\-(?P<release>[a-z]+)(?P<build>\
↪d+))?
serialize=
{major}.{minor}.{patch}{release}{build}
{major}.{minor}.{patch}
new_version=0.2.1dev0

```

1. If it all looks good (which it does above) then run for real:

```

$ bump2version --verbose patch
current_version=0.2.0dev0
commit=True
tag=True
parse=(?P<major>\d+)\.(?P<minor>\d+)\.(?P<patch>\d+)(\-(?P<release>[a-z]+)(?P<build>\
↪d+))?
serialize=
{major}.{minor}.{patch}{release}{build}
{major}.{minor}.{patch}
new_version=0.2.1dev0

```

1. Push to git. bump2version will produce git tags (imagine these are bookmarks in your repository). Not make sure these are updated in your repository you have to push with the --tags flag.

```
git push --tags origin master
```

1.6.2 Building the documentation

Sphinx is used to document the package and the documentation is maintained as a set of HTML files and a combined PDF document. The HTML pages use the readthedocs [theme](#). This can be installed with:

```
pip install sphinx-rtd-theme
```

There are several other packages that are required for building and handling markdown ([myst-parser](#))

```
pip install myst-parser
```

Handling jupyter-notebook inclusion into the documentation ([nbsphinx](#))

```
pip install nbsphinx
```

Associated with nbsphinx is [nbssphinx-link](#) which allows notebooks to be included in the documentation when they are not physically located inside the Sphinx build directory.

The documentation configuration and static HTML files are located in ./docs.

In order to build the PDF documentation, you will need pdftex, this can be installed with, see [here](#):

```

sudo apt-get install texlive-latex-recommended
sudo apt-get install texlive-fonts-recommended
sudo apt-get install texlive-latex-extra
sudo apt-get install latexmk

```

Initialisation

This only needs to be done once to create the directory structure for Sphinx to work in. Change into the `./docs` directory and run `sphinx-quickstart` then answer the questions. Here, separate build and source directories were created.

```
$ sphinx-quickstart
Welcome to the Sphinx 3.4.0 quickstart utility.

Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: .

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]: y

The project name will occur in several places in the built documentation.
> Project name: hpc-tools
> Author name(s): Chris Finan
> Project release []: 0.1.0a1

If the documents are to be written in a language other than English,
you can select a language here by its language code. Sphinx will then
translate text that it generates into that language.

For a list of supported codes, see
https://www.sphinx-doc.org/en/master/usage/configuration.html#confval-language.
> Project language [en]:

Creating file ./cluster/docs/source/conf.py.
Creating file ./cluster/docs/source/index.rst.
Creating file ./cluster/docs/Makefile.
Creating file ./cluster/docs/make.bat.

Finished: An initial directory structure has been created.

You should now populate your master file cluster/docs/source/index.rst and create other
→ documentation
source files. Use the Makefile to build the docs, like so:
    make builder
where "builder" is one of the supported builders, e.g. html, latex or linkcheck.
```

Configuration

Ensure that the Sphinx configuration file (`./docs/conf.py`) is updated to load all the required extensions.

```
extensions = [  
    "sphinx_rtd_theme",  
    "myst_parser",  
    "nbsphinx",  
    "nbsphinx_link",  
    "sphinx.ext.autodoc",  
    "sphinx.ext.napoleon",  
    'sphinx.ext.todo',  
    'sphinx.ext.viewcode'  
]
```

And the theme:

```
html_theme = 'sphinx_rtd_theme'
```

Building HTML

The HTML documentation can be built with:

```
./docs/make html
```

Building PDF

The PDF documentation can be built with:

```
./docs/make latexpdf
```

The resulting pdf file is located at `./docs/build/latex/hpc-tools.pdf`. This is copied to `./resources/pdf/hpc-tools.pdf`.

Helper script

The build commands and copying of the PDF file are wrapped into a small helper script. This can be run by:

```
./resources/build/build_docs.sh
```


INDICES AND TABLES

- `genindex`
- `modindex`
- `search`