jrnr Documentation

Release 0.2.4

ClimateImpactLab

Apr 21, 2020

Contents

Contents:

CHAPTER 1

jrnr

Job Runner for Climate Impact Lab Jobs

- Free software: MIT license
- Documentation: https://jrnr.readthedocs.io.

jrnr is a tool to help you create, manage, and monitor your highly parallelizable jobs.

1.1 Features

- Job set-up utility for running parallel jobs
- Currently only supports running slurm jobs

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install jrnr, run this command in your terminal:

```
$ pip install jrnr
```

This is the preferred method to install jrnr, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for jrnr can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ClimateImpactLab/jrnr
```

Or download the tarball:

```
$ curl  -OL https://github.com/ClimateImpactLab/jrnr/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

`jrnr` is a python library currently configured to work on systems using slurm workload managers. If your computing workflows can be parallelized, *jrnr* can help.

`jrnr` is an application that relies on click, the python command line tool.

At the top of your python module add this to the `import` section:

```
from jrnr.jrnr import slurm_runner
```

3.1 Interactive mode

Frequently, you'll want to do some basic debugging and iteration to make sure your batch jobs will run as expected. To assist this process, `jrnr` has an interactive mode that allows you to run a single job in an *ipython* session.

```
In [1]: import tas

In [2]: tas.make_tas.run_interactive(42)

2018-01-10 17:01:55,001 Beginning job
kwargs: { 'model': 'NorESM1-M', 'scenario': 'rcp45', 'year': '2054'}
2018-01-10 17:02:43,733 beginning
2018-01-10 17:02:43,733 producing_tas
Out[3]:
<xarray.Dataset>
Dimensions:  (lat: 720, lon: 1440, time: 365)
Coordinates:
  * lon      (lon) float32 -179.875 -179.625 -179.375 -179.125 -178.875 ...
  * time     (time) datetime64[ns] 2054-01-01T12:00:00 2054-01-02T12:00:00 ...
  * lat      (lat) float32 -89.875 -89.625 -89.375 -89.125 -88.875 -88.625 ...
Data variables:
    tas      (time, lat, lon) float32 272.935 272.937 272.931 272.911 ...
Attributes:
```

(continues on next page)

(continued from previous page)

```
    version:          1.0
    repo:             https://gitlab.com/ClimateImpactLab/tas/
    frequency:        annual
    oneline:          Average Daily Temperature, tavg
    file:             tas.py
    year:             2054
    write_variable:   tas
    description:      Average Daily Temperature, tavg\n\n Average Daily Temper...
    execute:          python tas.py run
    project:          gcp
    team:             climate
    dependencies:     ['/global/scratch/groups/co_laika/gcp/climate/nasa_bcsd/...
    model:            NorESM1-M
```

As you can see, if you setting up logging, the logging information will print to wherever you direct stdout. In this case, ininteractive mode, it prints to the ipython terminal. In batch mode, jrnr logs can be found in the directory you specified as `run-{job_name}-{job_id}-{task-id}.log`.

3.2 Running your job in batch mode

The `slurm_runner` decorator function in `jrnr` acts as a wrapper around your main function. Make sure that above your main function you have added `@slurm_runner()`. With this enabled, you can use the command line to launch your jobs on the slurm workload manager.

Make sure you are in the directory where your python module is located. Let's say we are running the job specified in *Example jrnr script*. Let's look at what the `help` function does.

```
$ python tas.py --help

Usage: tas.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  cleanup
  do_job
  prep
  run
  status
  wait
```

We can see that this will give us the list of options. Let's look at *run*.

3.3 `run`

Let's first have a look at the options with the run command.

```
$ python run --help

Usage: tas.py run [OPTIONS]
```

(continues on next page)

(continued from previous page)

```
Options:
  -l, --limit INTEGER          Number of iterations to run
  -n, --jobs_per_node INTEGER  Number of jobs to run per node
  -x, --maxnodes INTEGER       Number of nodes to request for this job
  -j, --jobname TEXT           name of the job
  -p, --partition TEXT         resource on which to run
  -d, --dependency INTEGER
  -L, --logdir TEXT            Directory to write log files
  -u, --uniqueid TEXT          Unique job pool id
  --help                       Show this message and exit.
```

The most important options are `u`, `j` and `L`. To specify a job you need `u` and `j` since these parameters uniquely identify a job and allow you to track the progress of your jobs. An example command is below

```
$ python tas.py run -u 001 -j tas
```

This creates a job with a unique id of *001* and a job name of *tas*.

By specifying some of the options listed above, you can adjust the behavior of your slurm jobs. For example, you can put your log files in a specific directory by specifying a value for argument `L`. Additionally, if you want to use a specific partition on your cluster you can speify the *p* option. Similarly, if your job is particularly compute intensive, with `n` you can adjust the number of jobs per node.

```
$ python tas.py run -u 001 -j tas -L /logs/tas/ -p savio2_bigmem -n 10
```

Its important to note that, by default, log files will be written to the directory where you are executing the file. Depending on how large your job is you may want to put these log files elsewhere.

If you want to fully take advantage of BRC's computing capacity you can run

```
$ python tas.py run -u 001 -j tas -L /logs/tas/ -p savio_bigmem -n 10
  run job: 98
  on-finish job: 99
$ python tas.py run -u 001 -j tas -L /logs/tas/ -p savio2_bigmem -n 10
  run job: 100
  on-finish job: 101
$ python tas.py run -u 001 -j tas -L /logs/tas/ -p savio2 -n 5
  run job: 104
  on-finish job: 105
$ python tas.py run -u 001 -j tas -L /logs/tas/ -p savio -n 5
  run job: 106
  on-finish job: 107
```

How many jobs should you run on each node?

To determine this, you'll need to divide the amount of memory per node by the amount of memory required by your job. To determine the amount of memory per node, you can look at the Savio user guide. For example, if I have a job that requires 6GB of RAM and I am running on the `savio2_bigmem` partition. Then we'll add 2GB of buffer to our 6GB RAM requirement and take the result of `128/8` to get 16 jobs.

3.4 status

You launched your job 10 minutes ago and you want to check on the status of your jobs. We can check with the `status` option. Let's look again at our `tas.py` file.

```
$ python tas.py status -u 001 -j tas

jobs:           4473
done:           3000
in progress:    1470
errored:           3
```

Notice that we use the unique id `001` and the jobname `tas` that we used when we created the job. You must use these values or we cannot compute the progress of our job.

3.5 Technical note

3.5.1 How does this `jrnr` track the status of my jobs?

In your directory where you are running your job, `jrnr` creates a *locks* directory. In this `locks` directory, for each job in your set of batch jobs a file is created with the following structure `{job_name}-{unique_id}-{job_index}`. When a node is working on a job, it adds the `.lck` file extension to the file. When the job is completed, it converts the *.lck* extension to a `.done` extension. If, for some reason, the job encounters an error, the extension will shift to `.err`. When you call the `status` command `jrnr` is just displaying the count of files with each file extension in the locks directory.

3.5.2 How does `jrnr` construct a job specification?

Each `jrnr` job can be specified with arguments from key, value dictionaries. Since these arguments are taken from a set of known possible inputs we can take each key and its associated set of possible values and compute the cartesian product of every key, value combination. In the background of `jrnr`, we take lists of dictionaries and use the python method `itertools.product` to specify the superset of possible batch jobs. A demonstration is below:

```
In [1]: def generate_jobs(job_spec):
          for specs in itertools.product(*job_spec):
            yield _unpack_job(specs)


In [2]: def _unpack_job(specs):
            job = {}
            for spec in specs:
                job.update(spec)
            return job


In [3]: MODELS = list(map(lambda x: dict(model=x), [
        'ACCESS1-0',
        'bcc-csm1-1',
        'BNU-ESM',
        'CanESM2',
        ]))

In [4]: PERIODS = (
        [dict(scenario='historical', year=y) for y in range(1981, 2006)] +
        [dict(scenario='rcp45',  year=y) for y in range(2006, 2100)])

In [5]: job_spec = [PERIODS, MODELS]
```

(continues on next page)

(continued from previous page)

```
In [6]: jobs = list(generate_jobs(job_spec))

In [7]: jobs[:100:10]
Out[7]:
[{'model': 'ACCESS1-0', 'scenario': 'historical', 'year': 1981},
{'model': 'BNU-ESM', 'scenario': 'historical', 'year': 1983},
{'model': 'ACCESS1-0', 'scenario': 'historical', 'year': 1986},
{'model': 'BNU-ESM', 'scenario': 'historical', 'year': 1988},
{'model': 'ACCESS1-0', 'scenario': 'historical', 'year': 1991},
{'model': 'BNU-ESM', 'scenario': 'historical', 'year': 1993},
{'model': 'ACCESS1-0', 'scenario': 'historical', 'year': 1996},
{'model': 'BNU-ESM', 'scenario': 'historical', 'year': 1998},
{'model': 'ACCESS1-0', 'scenario': 'historical', 'year': 2001},
{'model': 'BNU-ESM', 'scenario': 'historical', 'year': 2003}]
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/ClimateImpactLab/jrnr/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

jrnr could always use more documentation, whether as part of the official jrnr docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ClimateImpactLab/jrnr/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *jrnr* for local development.

1. Fork the *jrnr* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/jrnr.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv jrnr
$ cd jrnr/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 jrnr tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/ClimateImpactLab/jrnr/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ pytest tests.test_jrnr
```

CHAPTER 5

Credits

This repository is a project of the Climate Impact Lab

5.1 Development Lead

- Justin Simcock <jsimcock@rhg.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.2.4 (2020-04-21)

- Compatibility patch allowing commands with underscores to be normalized to dashes in click app returned by jrnr.jrnr.slurm_runner. Thanks for the digging and issue raising @simondgreenhill!

6.2 0.2.3 (2018-01-16)

- Documentation & testing improvements

6.3 0.2.2 (2017-08-28)

- Update to documentation
- `jrnr` attempts to remove `.lck` files if there is a keyboard interrupt or system exit

6.4 0.2.1 (2017-07-31)

- Fix bug in `slurm_runner.do_job` which caused job duplication when race conditions on lock object creation occur (GH #3)
- Infer filepath from passed function in `slurm_runner`. Removes need to supply filepath argument in `slurm_runner` function calls (GH #5)
- Adds `return_index` parameter to `slurm_runner` (GH #7)

6.5 0.2.0 (2017-07-31)

- Fix interactive bug – call interactive=True on `slurm_runner.run_interactive()` (GH #1)
- Add slurm_runner as module-level import

6.6 0.1.2 (2017-07-28)

- Add interactive capability

6.7 0.1.1 (2017-07-28)

- Fix deployment bugs

6.8 0.1.0 (2017-07-28)

- First release on PyPI.

CHAPTER 7

Example *jrnr* script

```python
'''

This module demonstrates the specification of a jrnr script

This jrnr script will compute daily average temperature
as a the average of daily max and min temperature.

To run, jrnr requires a parameterized job spec. This is constructed
as a list on line 105 and handed to jrnr's `slurm_runner` decorator on line 123.

Each job, interactive and batch, in jrnr receives a
dictionary which fully parameterizes the input arguments.

To parameterize your job spec jrnr takes the
cartesian product of the items in lists of dictionaries.
`JOB_SPEC` on line 112 is simply a list of those lists.
'''

import os
import logging
import time
import xarray as xr
import pandas as pd
import numpy as np
import climate_toolbox.climate_toolbox as ctb
from jrnr.jrnr import slurm_runner

#set up logging format and configuration
FORMAT = '%(asctime)-15s %(message)s'
logging.basicConfig(format=FORMAT)
logger = logging.getLogger('uploader')
logger.setLevel('DEBUG')

```

(continues on next page)

(continued from previous page)

```
description = '\n\n'.join(
        map(lambda s: ' '.join(s.split('\n')),
            __doc__.strip().split('\n\n')))

oneline = description.split('\n')[0]

__author__ = 'Justin Simcock'
__contact__ = 'jsimcock@rhg.com'
__version__ = '1.0'


READ_PATH = (
    '/global/scratch/groups/co_laika/gcp/climate/nasa_bcsd/reformatted/' +
    '{scenario}/{model}/{variable}/' +
    '{variable}_day_BCSD_{scenario}_r1i1p1_{model}_{year}/1.0.nc4')

WRITE_PATH = ('/global/scratch/groups/co_laika/gcp/climate/nasa_bcsd/reformatted/' +
                '{scenario}/{model}/{variable}/' +
                '{variable}_day_BCSD_{scenario}_r1i1p1_{model}_{year}/' +
                '{version}.nc4')


ADDITIONAL_METADATA = dict(
    oneline=oneline,
    description=description,
    author=__author__,
    contact=__contact__,
    version=__version__,
    repo='https://gitlab.com/ClimateImpactLab/make_tas/',
    file=str(__file__),
    execute='python {} run'.format(str(__file__)),
    project='gcp',
    team='climate',
    frequency='annual',
    write_variable='tas',
    dependencies='')


def make_tas_ds(tasmax, tasmin):

    tas = xr.Dataset()
    tas['tas'] = (tasmax.tasmax + tasmin.tasmin) / 2.
    return tas

PERIODS = (
    [dict(scenario='historical', year=y) for y in range(1981, 2006)] +
    [dict(scenario='rcp45',  year=y) for y in range(2006, 2100)] +
    [dict(scenario='rcp85', year=y) for y in range(2006, 2100)])


MODELS = list(map(lambda x: dict(model=x), [
    'ACCESS1-0',
    'bcc-csm1-1',
    'BNU-ESM',
    'CanESM2',
    'CCSM4',
    'CESM1-BGC',
```

(continues on next page)

(continued from previous page)

```
    'CNRM-CM5',
    'CSIRO-Mk3-6-0',
    'GFDL-CM3',
    'GFDL-ESM2G',
    'GFDL-ESM2M',
    'IPSL-CM5A-LR',
    'IPSL-CM5A-MR',
    'MIROC-ESM-CHEM',
    'MIROC-ESM',
    'MIROC5',
    'MPI-ESM-LR',
    'MPI-ESM-MR',
    'MRI-CGCM3',
    'inmcm4',
    'NorESM1-M'
    ]))


JOB_SPEC = [MODELS, PERIODS]

def validate_tas(tas):
    '''
    Make sure NaNs are not present
    '''
    msg_null = 'DataArray contains NaNs'
    msg_shape = 'DataSet dims {} do not match expected'
    tas_nan = tas.tas.sel(lat=slice(-85,85)).isnull().sum().values
    assert tas_nan == 0 , msg_null
    assert tas.dims['lat'] == 720,  msg_shape.format(tas.dims['lat'])
    assert tas.dims['lon'] == 1440, msg_shape.format(tas.dims['lon'])
    assert tas.dims['time'] in [364, 365, 366], msg_shape.format(tas.dims['time'])
    assert tas.lon.min().values == -179.875
    assert tas.lon.max().values == 179.875
    return


@slurm_runner(job_spec=JOB_SPEC)
def make_tas(metadata,
            scenario,
            year,
            model,
            interactive=False
            ):


    metadata.update(ADDITIONAL_METADATA)

    tasmin_read = READ_PATH.format(variable='tasmin', **metadata)
    tasmax_read = READ_PATH.format(variable='tasmax', **metadata)

    metadata['dependencies'] = str([tasmin_read, tasmax_read])

    tas_write = WRITE_PATH.format(variable='tas', **metadata)


    if os.path.isfile(tas_write) and not interactive:
        tas = xr.open_dataset(tas_write, autoclose=True, chunks={'time': 100}).load()
```

(continues on next page)

(continued from previous page)

```
        tas = ctb._standardize_longitude_dimension(tas)



    else:
        tasmax = xr.open_dataset(tasmax_read, autoclose=True, chunks={'time': 100}).
↪load()
        tasmin = xr.open_dataset(tasmin_read, autoclose=True, chunks={'time': 100}).
↪load()

        logger.debug('beginning')

        logger.debug('producing_tas')
        tas = make_tas_ds(tasmax, tasmin)
        tas = ctb._standardize_longitude_dimension(tas)

    tas.attrs.update(metadata)

    if interactive:
        return tas

    logger.debug('checking_tas_path')
    if not os.path.isdir(os.path.dirname(tas_write)):
        os.makedirs(os.path.dirname(tas_write))

    tas.to_netcdf(tas_write + '~', encoding={var : {'dtype': 'float32'} for var in␣
↪tas.data_vars.keys()})
    logger.debug('write_tas_path')

    validate_tas(tas)
    logger.debug('validate_tas')
    os.rename(tas_write + '~', tas_write)


    logger.debug('job_complete')


if __name__ == '__main__':
    make_tas()
```

(continues on next page)

(continued from previous page)

```
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search