Volume Title
ASP Conference Series, Vol. **Volume Number**
Author
© **Copyright Year** Astronomical Society of the Pacific

drPACS : a simple UNIX execution pipeline

Peter Teuben

Astronomy Department, University of Maryland

Abstract.
We describe a very simple yet flexible and effective pipeliner for UNIX commands. It creates a Makefile to define a set of serially dependent commands. The commands in the pipeline share a common set of parameters by which they can communicate. Commands must follow a simple convention to retrieve and store parameters. Pipeline parameters can optionally be made persistent across multiple runs of the pipeline. Tools were added to simplify running a large series of pipelines, which can then also be run in parallel.

1. Introduction

Pipelines are in common use at large observatory data centers, but are very sophisticated (e.g. Ballester (2011)) and often overkill, or simply beyond the patience or skills, of many casual users. Yet, many theoretical or observational projects could very well benefit from this concept of being able to pipeline their dataflow. Often they are an afterthought in small-scale projects. If some effort is taken to pipeline these, the threshold to experiment is lowered, opening the way to more insight. This paper introduces such a concept.

Imagine a series of programs that run a simulation and/or reduce data. If you have made it flexible, it will also have parameters to control this process. But now imagine you have to run many of these pipelines, each on a different set of data (ideally in a different directory) and potentially have to re-run them when something in your pipeline has changed. For example if one of the programs in your pipeline had a bug, or you add or insert steps into the pipeline. You will have to rerun portions of your pipeline for all your directories.

I will describe a generic UNIX solution to this dilemma, to make this process easy. We have used this pipeline in CARMA for PACS data reduction, as well as converted an existing CARMA data reduction pipeline for the STING project. We maintain a website where more information can be found where to download and install the code from[1].

[1] http://carma.astro.umd.edu/tools/drpacs

2. Overview

2.1. Concepts

For our purposes a pipeline consists of a serial set of programs controlled by a common pool of parameters that these programs can read and write. These programs can be shell scripts, python scripts, compiled programs or anything that you would normally run from the command line. Normally a pipeline is run in a project (or run) directory.

2.2. Define and Run the Pipeline

First the pipeline needs to be defined, by providing it the names of the programs that need to run in a certain sequence that we call the pipeline. The simplest way this can be done is by providing them via the command line to the `pipeline` program. Our example starts with 5 program steps, named `step1` through `step5`:

```
pipeline 5 step1 step2 step3 step4 step5 > Pipefile
```

Next, you would probably want to set the values for some of the parameters for the various programs in the pipe. The pipeline programs will have to read them through provided procedures:

```
pipepar -c project=c0184.3B_108PG2130.13 carmaRefant=2
```

and finally you can run the pipeline in one of many ways. For example:

```
pipe all
pipe step3 all
pipe clean all
```

would run the whole pipeline (or formally whatever needed to be done), would only run `step3` and then the remainder of the pipeline, and in the third example clean the pipe from the start and then run the whole pipeline again.

2.3. Advanced Usage

Now imagine for all your runs (directories) you would need to re-run the pipeline with a new parameter in `step3`. Thus `step1` and `step2` do not need to be run again. The `piperun` command can do this in a single line as follows:

```
piperun dirs.txt 'pipepar foo=1.3 ; pipe step3 all'
```

Essentially this will execute the second argument, the text between the singles quotes, in each directory listed in the text file `dirs.txt` provided as the first argument to `piperun`.

Now imagine that each pipeline computed two interesting numbers, `foo` and `bar`, and these were stored in the pipeline parameter database. To produce a simple ascii table of these two variables, which can be used for further analyis, you would issue

```
piperun dirs.txt pipepar -v foo -v bar > foo_bar.tab
```

where the file `foo_bar.tab` contains two columns with the (textual) values as they were stored in the parameter database.

Finally, another useful feature of `drpacs` is the option to save the parameters for a particular "project". This is useful in case the run directory is a scratch directory. The parameter database, which is normally local, will need to be saved, which `drpacs` will do based on it's project name. Assuming this had been done, a new project can be created from scratch as follows:

```
pipesetup a=1 b=2 project=test123
```

and

```
pipe all
```

and

```
pipesave
```

would save the parameter set based on its project name "`test123`".

2.4. Examples

We show two simple examples of scripts that will be `drpacs` compliant, i.e. will read parameters from `drpacs` first, and then the commandline, and also save them at the end. A C-shell example:

```
#! /bin/csh -f
#
# (1) define default values in case not given, problem specific
set a=1
set b=2
# (2) pipeline interface to grab old defaults, drpacs compliant
pipepar -s csh > tmp$$.par; source tmp$$.par; rm tmp$$.par
# (3) poor man's command line processor to override parameters
foreach _arg ($*)
  set $_arg
end
# (4) The Actual Code where the work can be done, all problem specific
echo A=$a B=$b
# (5) write pipeline parameters back, drpacs compliant
pipepar a=$a b=$b c=3
```

and a python example:

```
#! /bin/env python
#
import parfile, sys
a=1
b=2
if __name__ == "__main__":
  p = parfile.Parfile('drpacs.def')
  p.argv(sys.argv)
```

```
p.set('a',a)
if p.has('b'):
  b = p.get('b')
else:
  p.set('b',b)
p.set('sum',a+b)
p.save()
```

for which you will need the `parfile` module, a simple keyval pair module supplied with the drpacs code. Several more advanced ones are on the market, which can be used as effectively. For example `configobj`, the one that `pytools.teal` uses (Sontag (2011)).

2.5. Drawbacks

Simplicity and the usage of the UNIX `make` program under the hood also comes with a price.

First of all, you can only run one pipeline in a given directory. This is not a serious handicap, as the pipelines are easily redefined from a textfile, so a set of `Pipefile`'s could be stored locally. However, the status of each pipeline (e.g. which step was last correctly executed) will be lost. Only the active one will be valid, and switching from one to the other has to be done with care. Parameters between pipelines will have to be shared, there is no namespace.

Secondly, the current version has no dependancy on the keywords in the pipeline. For example, if one would change a parameter that is normally defined in `step3`, the user would need to know this parameter came from `step3` and re-execute the pipeline from this step onwards. In principle these dependancies can be coded quite easily via "make" dependancies, in a similar way how the program steps have been coded, but this has not been implemented in the current version.

For each "project" we assume a "directory", although their hierarchy is not specified.

One cannot use the same program twice in the pipeline.

Acknowledgments. The author wishes to thank Ashley Zauderer, Dalton Wu and Roger Curley for their help during the develpment of `drpacs`.

References

Ballester, P. 2011, in ADASS XX, edited by I. N. Evans, A. Accomazzi, D. J. Mink, & A. H. Rots (San Francisco: ASP), vol. TBD of ASP Conf. Ser., TBD

Sontag, C. 2011, in ADASS XX, edited by I. N. Evans, A. Accomazzi, D. J. Mink, & A. H. Rots (San Francisco: ASP), vol. TBD of ASP Conf. Ser., TBD