Appendix A

drPACS Pipeline

ABSTRACT

We describe[1] a simple yet flexible and effective pipeliner for Unix commands. It uses a Makefile (behind the scenes) to define a serial set of commands for your choice of the pipeline. The pipeline commands share a common set of parameters by which they communicate. Pipeline parameters can optionally be made persistent accross multiple runs of the pipeline. Commands must follow a simple convention to retrieve and store parameters.

A.1 Introduction

To process a large number of datasets in a very similar way is a common theme, particularly for the type of PACS data we are discussing here. We thus developed a simple infrastructure to assemble and run a pipeline comprising of a set of commands that have to be run in a certain order, and depend on each other. We call this package '**drpacs**'[2].

Technically **drpacs** consists a set of shell and python scripts and the infrastructure to setup a serial pipeline, and some aspects of running this pipeline in parallel as well.

Although **drpacs** was written for the first stages of analyzing a large set of PACS data, you can use it and construct your very own pipeline. An example is given later in this Appendix.

A summary of typical PACS pipeline commands is:

```
pipeline $DRPACS/cat/pipeline.001 > Pipefile           # generate control Pipefile
pipepar -c project=c0184.3B_108PG2130.13 carmaRefant=2  # set some parameters
pipe all                                               # run the pipe
```

A.2 Pipeline

Lets assume we have 4 programs (or scripts) called `step1`, `step2`, `step3` and `step4` that need to be run in succession in each project directory. Each of the `stepX` commands accepts its own unique set of `keyword=value` command line arguments, but the pipeline convention is that the values for each of these keywords are remembered for any subsequent run if you do not specify them explicitly. So if you would manually first run a command as "`step2 foo=bar`", the next time you would run just "`step2`",

[1] Draft version Sept 23

[2] Dalton and Roger were its first users for a PACS project, and helped developing it

it will have remembered the value `bar` for `foo`. Users of the MIRIAD shell program will recall this type of global parameter behavior. This can be very convenient, but also bite you when you least expect it.

With the "`pipeline`" command you establish which commands, and in which order, need to be executed for your pipe. This will produce a Unix-style Makefile, which by convention we call a `Pipefile`:

```
% pipeline 4 step1 step2 step3 step4 > Pipefile
```

Of course these 4 commands[3] must exist in your Unix `$PATH` and most follow the pipeline parameter convention (see below).

Unless for some bizarre reason you do not use pipeline parameters, a dummy parameter file needs to be created before you can run the pipeline:

```
% pipepar -c
```

Apart from manually running each command, the "`pipe`" command will now run these 4 steps in succession:

```
% pipe
Checking ./Pipefile
Doing step1 as step1
Doing step2 as step2
Doing step3 as step3
Doing step4 as step4
```

This command uses the Unix "`make`" command, and uses the `Pipefile` as the controllng `Makefile` with all dependancies properly defined. If for some reason `step2` fails, the pipe will be aborted at that stage. If you try and re-run the pipe after it had been successfully finished, you will probably see something like the following:

```
% pipe
Checking ./Pipefile
make: Nothing to be done for 'all'.
```

Technically, it uses Unix dot files for its dependancies. These dot files should not be touched or removed, unless you know what you are doing. They make sure commands are not run again if not needed because nothing was changed.

Here are some simple examples of running and re-running portions of the pipeline

```
pipe all               does all steps in the pipeline (if needed)
pipe step2             run just step2 (if even ran ok before)
pipe all               now will do step3 and step4
pipe step3 all         does step3 and step4
pipe clean             wipes the pipeline control files (the dot files)
```

As was hinted to before, a utility is needed to manage the global pipe parameter file. This is the file that contains the parameters that are passed between the programs and scripts in the pipeline. This file is a very simple ascii file with a set of "`par=val`" pairs, one per line. No spaces should be used before the parameter, or surrounding the '=' sign. You can actually use any text editor to modify this file. No quotes are needed as everything is interpreted as text. CHECK THIS FOR VALUES WITH SPACES

Here is the usage line for the `pipepar` command

[3] Although we gave the command simple stepX names, you can name these any way you like, e.g. `pipeline 2 foo bar` is legal

```
pipepar [-h] [-c] [-f parfile_name] [-s shell] [-d var] [-v par] [-e par] [par=val ...]

 -h     help
 -c     create empty drpacs.def (parfile_name)
 -f         use another parfile_name from the default drpacs.def (not recommended)
 -s      output for given shell (csh and bash are currently known)
 -v     show value of a parameter (multiple allowed, but do not combine with -e flag)
 -d         delete a parameter
 -e         print 1 or 0 if a parameter exists (only only occurence allowed now)
 par=val    assign (new) value to a parameter  (multiple allowed)
```

A.3 Some Examples

1. rerun the complete pipeline (from step1 onwards) with a new parameter

   ```
   pipepar foo=1.3
   pipe step1 all
   ```

2. rerun the pipeline with a new parameter, but only run the pipeline from step3 onwards. This also implies you better make sure that `foo` is not needed in `step1` and `step2`.

   ```
   pipepar foo=2.3
   pipe step3 all
   ```

3. Rerun piece of the pipeline in each of a set of directories. In `csh` notation this could be achieved as follows:

   ```
   foreach dir (`cat dirs.txt`)
     cd $dir
     pipepar foo=3.3
     pipe step3 all
     cd ..
   end
   ```

   and because this is somewhat cumbersome to program, a special command is available to help running (pipeline) commands in a set of directories:

   ```
   piperun dirs.txt 'pipepar foo=3.3 ;  pipe step3 all'
   ```

   does the same thing

4. From a set of directories, create a scatterplot of two pipeline derived parameters. Again, with the aid of the `piperun` command this can be done as follows

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

5. Run a (pipe) command on a set of project directories in parallel. Generally you will need to understand if your pipeline contains I/O and if running them in parallel is an efficient usage of your resources. And of course the number of processes you can run them under. Here is an example of running a set of pipelines on 4 processors, from step2 and onwards:

   ```
   piperun -n 4 dirs.txt pipe step2 all
   ```

6. Re-create a new pipeline using an extra step, and rerun the whole pipeline

```
pipeline $DRPACS/cat/pipeline.002  > Pipefile
pipe clean all
```

You generally will need to use an extra `clean` step, in order to clean up the old control files, since the steps are unlikely to be compatible with the old pipeline (the exception being if the new pipeline appends steps to the old one).

7. Run the PACS pipeline in parallel on a 4-way processor on a set of project directories, and store the output in pipe.log (in each project directory):

```
piperun -n 4 -c -o pipe.log dirs.txt 'pipepar -c project=%s showPlots=False; pipe clean all'
```

Note the use of the special "`%s`" construct to replace it with the active directory name from the `dirs.txt` file as it loops over the project directories. Here is the full usage line for the `piperun` command:

```
Usage: piperun [-n #procs] [-o logfile] [-c] dirs.txt cmd [args]
-h         this help
-n #procs  parallel processing using #procs processors
-o logfile output log
-c         create directories
-v         verbose (debug)
dirs.txt   text file with directory names
cmd        unix command to run
args       arguments to unix command (inlcuding ; cmd2 args2...)
```

8. Re-Run the PACS pipeline for failed projects

```
% piperun -v -o pipe1.log dirs.txt 'pipepar showPlots=False; pipe all'
```

CAVEAT: There appears to be a Unix issue aborting this command with the usual $\hat{\text{C}}$, it will instead work on the next project directory. A better way to halt the series, is to issue $\hat{\text{Z}}$, which suspends the task, and then issue "`kill %%`" the currently last suspended task.

9. Re-Run a project with some additional flagging :

```
% echo 'ant(2)'  > CARMA.uvflag
% echo 'ant(21)' > SZA.uvflag
% pipe reduction all
```

Flagging is acted upon in the `reduction` step, but only if files `CARMA.uvflag` and/or `SZA.uvflag` are present, they are created with an editor, or in this case with a simple `echo` command, and the pipeline is run again.

If you want to store these uvflag files in a more persisent way, use the advanced feature of storing them as pipefiles (see "Advanced Usage" below) together with `drpacs.def` do the following

```
% pipepar pipefile=CARMA.uvflag,SZA.uvflag
% pipesave
```

A.4 Pipeline Parameter Convention

Any script or program that is part of the pipeline must follow the same convention of retrieving named parameters from the commandline, and override any previously stored values in the (default: `drpacs.def`) pipeline parameter file. At the (successfull) end of the script new and changed parameters will be written back to this parameter file for the next program in the pipe to pick up these variables.

In short, these are the conventions:

1. arguments to a pipeline tasks should be a unique series of "*par=val*" (since parameters can be shared between pipeline tasks)
2. values are stored as a string, assigning a type (integer, real, boolean etc.) are left open to the task
3. each task should define a default value for each parameter
4. the task should then read the global database (usually from the `drpacs.def` file)
5. task can now compute
6. task should write out all parameter back to the global database

A.4.1 csh

Here is a very simple example in the `csh` scripting language how this can be accomplished:

```
#! /bin/csh -f
#

#  (1) define default values in case not given
set a=1
set b=2

#  (2) pipeline interface to grab old defaults
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
echo A=$a   B=$b

#  (5) write pipeline parameters back
pipepar a=$a b=$b c=3
```

The actual code is in (4), though (1) and (3) are fairly common techniques to make your shell script more dynamic and have it accept parameters in a simple way. (2) and (5) are the new pipeline based commands to ensure previous defaults are read before the commandline sets them (2), and ensure their values are stored back into your local pipeline database (5).

A.4.2 python

Here is a very simple example in the `python` scripting language how this can be accomplished:

```
#! /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()
```

You can find full examples of this convention for bash, csh and python in the `$DRPACS/templates/pipeline` directory. There are no examples in C or Fortran yet, but are relatively easy to implement and add to your library.

A.5 Advanced Usage

Two wrapper scripts exist that help you managing your pipeline parameters in a persistent way. Imagine your pipeline directory tree runs completely from scratch, so you would like to save and restore parameters between different versions of the pipeline. We simply store the parameter files (`drpacs.def`) in an agreed upon location, currently the **project** directory within the `$DRPACS_DEF` (if present, or else the default `$DRPACS/def`) directory), and retrieve them in a subsequent run. The commands `pipesetup` and `pipesave` are used for this

```
  pipesetup a=1 b=2 project=test
  pipe all
  pipesave
```

Another common technique to store large amounts of simulations is in a hierarchy where each level identifies a new value in a parameter. **drpacs** supports this method as well, by storing the project directories in that same hierarchy. In the example below you see three levels representing values for the 3 parameters `a, b` and `c`:

```
  mkdir par/0.4/0.1/6.0
  pushd par/0.4/0.1/6.0
  pipepar -c project=par/0.4/0.1/6.0 a=0.4 b=0.1 c=6.0
  pipe all
  pipesave
  popd

...

  mkdir test
  cd test
  pipesetup project=par/0.4/0.1/6.0
  pipe all
```

In addition to the special identifying `project` parameter, the `pipefile` parameter is also treated somewhat special by the `pipesetup` and `pipesave` wrappers: they can contain a comma separated list of filenames that will be retrieved and saved at the start and beginning of a pipeline, if you add these wrappers to your pipeline. This would then enable your pipeline to use these files in a more persistent way.

Example:

```
  pipepar pipefile=NOTES,uvflags
  echo flagging ant 2 and 13, both had issues   > NOTES
  echo 'ant(2),time(10:00,11:00)'   > uvflags
  echo 'ant(13),time(11:00,12:00)' >> uvflags
  pipesave
```

A.6 Building your own Pipeline

1. Define your pipeline commands, and for each command make sure they follow the pipeline parameter convention. Place the commands somewhere in your `$PATH`.
2. Create the Pipefile, for example

   ```
   pipeline 5 getdata calibrate map deconvolve summary > Pipefile
   ```

   You can stick this Pipefile either in each project directory you want to run the pipeline on, or make it a default by overwriting the example in `$DRPACS/cat/Pipefile`
3. Run the pipe, set parameters etc.etc.

   ```
   pipepar -c foo=bar fum=bar
   pipe all
   ```

4. If you want save/restore the pipeline parameters, the `project` parameter is the key for this:

   ```
   pipepar project=mytest1
   pipesave
   ```

   the next time you setup a pipeline in another directory, the command

   ```
   pipesetup project=mytest1
   ```

   will restore your previous defaults.

A.7 drPACS Command Summary

Although the current `$DRPACS/bin` directory also contains many PACS specific commands, the following commands are very general pipeline related, and all start with the 4 letters **pipe**:

```
pipeline            create a Pipefile for running the 'pipe' command
pipepar             set and retrieve pipeline parameters
pipe                run the pipeline
piperun             run (optionally in parallel) pipeline in set of directories

pipesetup           grab previously run pipeline parameters for a project
pipesave            save pipeline parameters for a project
```

Appendix B

Installation

Here we discuss the installation of **drpacs** and comment on some related packages that may be needed.

B.1 drpacs

We use the same CVS repository as MIRIAD and NEMO, and the installation is very similar. The module is called `drpacs` and once installed it will need MIRIAD, python + scipy/matplotlib and NEMO (at least for the actual PACS pipeline, for your own pipeline you only need whatever your pipeline programs need):[1]:

```
% cvs -d :pserver:anonymous@cvs.astro.umd.edu:/home/cvsroot co drpacs
% cd drpacs
% ./configure
% source drpacs_start.csh
```

From this point on you can put new scripts and python modules in the appropriate places (e.g. `$DRPACS/bin` and `$DRPACS/lib`, or your own style somewhere in `$PATH`[2]

B.2 MIRIAD

For the actual PACS pipeline, you will need to have MIRIAD installed. See the appropriate MIRIAD documentation.

B.3 python

You will have to have python installed with at least scipy and matplotlib. You can test this the following way, and see if you got any error messages about none existent modules:

```
% ipython -pylab
In [1]: import matplotlib
In [2]: import numpy
```

[1] In its most basic form the **drpacs** package does not need MIRIAD, NEMO or anything but the most basic pieces of python

[2] because commands are executed in another directory, the scripts in the current directory will not work.

However, if you got an error message such as

```
ERROR: matplotlib could NOT be imported!  Starting normal IPython.
or
ipython: Command not found.
```

you are in bad shape. For U of Maryland computers the command(s)

```
source /astromake/astromake_start
astroload python
```

will give you a version of (i)python with all the proper modules included, and even more. The script to ease the installation into your own workspace is called `python.install` , and a copy should be in `$NEMO/src/scripts` or `$MIR/install`.

B.4 NEMO

If you do not have NEMO installed, there is a simple way within MIRIAD to install NEMO:

```
% cd $MIR/borrow
% cvs -Q checkout nemo
% mirboss
% mir.install nemo
% source $MIR/borrow/nemo/nemo_start.csh
```

and to test a simple program that plots some garbage into a pgplot window, try this

```
% nemoinp 1:10 | tabhist -
```

Appendix C

PACS

This Appendix describes some PACS specific aspects of the drPACS pipeline.

C.1 PACS parameters

The global pipeline parameters are stored in ASCII format in a small text file, by default this is called `drpacs.def`. For our PACS pipeline the following parameters listed in column 2 are defined by the programs listed in column 1:

```
pipesetup       project         <optional step for re-running pipes with old drpacs.def files>
pipesetup       pipefile        optional comma separated list of to-be-saved filenames
pipesetup       step            most pipeline scripts store their last successful step name here

getdata         datadir         where tar.gz or miriad/mir files live [/grus1/pacsdata]
getdata         project         e.g. c0184.1B_105PG1351.10
getdata         cvis            e.g. c0184.1B_105PG1351.10.miriad - usually derived from project
getdata         svis            e.g. zrc0184.1B_105PG1351.10.mir
getdata         link            normally 1, if symlinks used, use 0 if you want a local copy

timeselect      whitelist       source1[,source1,...] - case does not matter
timeselect      blacklist       source1[,source1,...]
timeselect      cfreq           somewhere in the 1mm or 3mm band, in GHz
timeselect      sfreq           always fixed at 30.0 now, could be slightly off
timeselect      szaPatchSource  list of 'SOURCE,HH:MM:SS,DD:MM:SS' to patch bad SZA ra,dec's

antselect       goodCarAnts     list of good ants (if absent, all are good?)
antselect       goodSzaAnts

reduction       source          a prefix (foobar)
reduction       carmaRefant     [7]
reduction       szaRefant       [22]
reduction       fastInterval    [0.04]
reduction       slowInterval    [4]
reduction       showPlots       [True] or False
reduction       cleanUp         [True] or False
reduction       timeOffset      0  (1 would trigger options=time for uvedit to fix CARMA filler bug)
reduction       pipefile        files CARMA.uvflag and SZA.uvflag can be used to flag data

baselinecalc    freqRatio       force it to -1, or inherit from cfreq/sfreq; used in unwrap
                unwrap          none, antenna, baseline (only one) are optional [none]

plotdata        -

pipesave        -               (any parameters in pipesetup also apply here)
```

C.2 PACS ISSUES

C.2.1 WIKI

See also our wiki: http://carma.astro.umd.edu/wiki/index.php/PACS

```
Version 1.1 issues (17 Dec and after)

    * now have properly dimensioned systemp, though some data don't have systemp (e.g. pacs 19)
    * some data create an extra hierarchy data/eml/mirData/ in the current directory. Simple manually move it.

[edit] Version 1.1 issues (before Dec 17)

    * systemp is erroneously written with the wrong dimension. New version of listobs (V1.26) has been patched
      to look at wsystemp in this case.
    * There are no linelength data (phasem1), but this is a known issue and won't be added for a while.

[edit] Version 1.0 issues (before Dec 11)

    * Data not always time sorted (solved in 1.1). If the data is not time sorted, mfcal won't work.
      Rest should be ok. See comments on 3QSO below.
    * The antpos uv variable is not correct. (fixed in 1.1)
    * The freq variable is missing, listobs now works around that (fixed in 1.1)
    * There is no version keyword in the data. (fixed in 1.1)
    * Only wide band data are flagged (fixed in 1.1), you must run

      uvwide vis=SZA narrow=t

    * missing NOISE source data.(zrcx252.Arp220_12CO.1.mir.tar)
    * (now resolved) prior to about Dec 3, data had the wrong sign in the U-V coordinates.
      You can solve that by using scale < 1 in gpbuddy (V1.9). To check if your data has the
      wrong sign of the phases, check a similarly oriented antenna pair using UV list and see
      in which quadrant the UV coordinates are, viz.

 uvlist vis=SZA select='ant(21)(23),time(15:00,15:01),-source(noise)'
 uvlist vis=CAR select='ant(2)(4),time(15:00,15:01),-source(noise)'

    * systemp's are constant, but this is a feature. Need to double check if the array is properly
      written as systemp(nants,nwide), and not the other way around.

    *  3QSO track (zrct015.3QSO_3C273.3mm.1.mir).
          o no wide band data
          o first few integrations after a source change are bad.

There are actually two versions of the 3QSO dataset (zrct015.3QSO_3c273.3mm.1.mir.tar.gz and
zrct015.3QSO_3C273.3mm.1.mir.tar.gz) that have a 30MB difference in size. The latter (upper case C)
one is smaller and sorted in time, but has lost the wide band data (a feature of uvsort).
In the following procedure

  uvwide vis=zrct015.3QSO_3c273.3mm.1.mir narrow=t
  uvsort vis=zrct015.3QSO_3c273.3mm.1.mir out=sza1
  uvwide vis=sza1 out=sza2 nwide=16

all flags are correct, including the bad data after a source change.
```

C.2.2 NEW

1. listobs output and other tests showed that time in CARMA is off by inttime/2, see BUGZILLA #786. Solution:

   ```
   uvedit options=time
   ```

   for data prior August 2009. (but after Feb 2006 ?) . CARMA data version 1.0.3 or earlier has this issue. To find out data version, e.g. do this:

```
% uvio c0184.3B_108PG2130.13.miriad | grep version
0x000009a8 SIZE: version   Count=5,Type=a
0x000009b0 DATA: version   1.0.3
```

2. `select=time(t0,t1)` is of course subject to rounding, but for a given time 't' in the vis data, the record selection is not dependant on inttime, but purely on the values of t_0 and t_1 in the following way: $t_0 \leq t < t_1$.

3. selfcal and mselfcal assign a timestamp to an interval that can be confusing if 'interval' larger but close to 'inttime'. Also recall that the "*validity time*" for a gain table entry is "`interval`" on either side the timestamp.

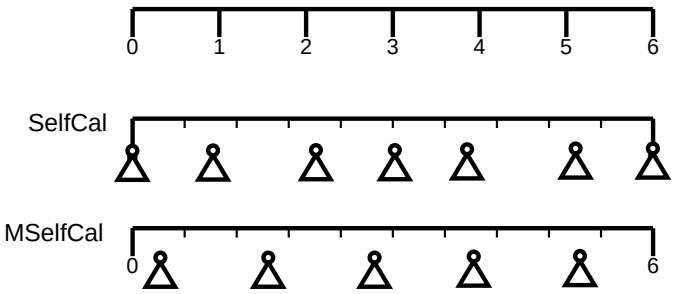

Figure C.1: Timestamps: with an selfcal gain interval=1 (top bar) and integration time of 0.6 minutes, selfcal (middle bar) and mselfcal (bottom bar) compute their timestamps differently, as depicted by the triangles.

Given that we could clearly see a 2 second shift on a 4 second integration time in 3C454.3, getting the times right on this level is important.

The following table was created with `uvgen`[1]:

vis time	selfcal time	mselfcal time
00:00.0	00:00.0	
00:36.0		00:18.0
01:12.0	00:54.0	
01:48.0		01:30.0
02:24.0	02:06.0	
03:00.0	03:00.0	02:42.0
03:36.0		
04:12.0	03:54.0	03:54.0
04:48.0		
05:24.0	05:06.0	05:06.0
06:00.0	06:00.0	

4. The interpolation formulae[2] that computes a baseline based gain at time t, from antenna based gains g_A and g_B that had been computed at times t_1 and t_2 is given by the following expression:

$$G = g_{A,2}(1 + (\left|\frac{g_{A,1}}{g_{A,2}}\right| - 1)\epsilon) \left(\frac{g_{A,1}}{g_{A,2}}\right)^{\epsilon} g^*_{B,2}(1 + (\left|\frac{g_{B,1}}{g_{B,2}}\right| - 1)\epsilon) \left(\frac{g_{B,1}}{g_{B,2}}\right)^{-\epsilon}$$

[1] `harange=-6,6,0.01 options=slip` is needed to get the times rounded a little nicer
[2] See also `$MIRSUBS/uvgn.for::uvGnFac()`

and for a phase-only selfcal solution, where $|g| = 1$, this simplifies to:

$$G = g_{A,2} g^*_{B,2} \left(\frac{g_{A,1}}{g_{A,2}}\right)^{\epsilon} \left(\frac{g_{B,1}}{g_{B,2}}\right)^{-\epsilon}$$

where

$$\epsilon = \frac{t_2 - t}{t_2 - t_1}$$

5. SZA data can have varying inttime's, by fractions of the frames
6. some SZA data have no narrowband data flagged:

```
% uvwide vis=c0184.3B_108PG2130.13.miriad narrow=t
```

   will flag the wide-band based on narrow band. Also, bands 1 and 2 are always flagged due to missing correlator boards at that time (winter 2008/2009)

7. linreg on all the phase correlation data, csPhaseDiff.txt, gives a sigma in the intercept, that's pretty good way to measure goodness of fit? Or should be we use sigma(offset)/slope ?
8. If RA/DEC for all sources the same, szaPatchSource can be set on a tuple of sourcename,ra,dec, e.g. `szaPatchSource=3c111,12:11:10.2,-20:12.02.3,3c454,22:20:10.2,35:30:21.6`
9. wrapping is still bad in the pipeline,here and there
10. if an ant is missing, pipeline breaks in places
11. use of proper antpos file vs. output of listobs, not for SZA yet, since we don't have antpos files for SZA yet.