Chapter 1

EGN (...) Toolkit

EGN () is a toolkit to help you reducing a large set of similar CARMA data. This particular manual was written with the (Fall 2011) EGNoG project (c0834) in mind.[1]

...

It is obvious that some added infrastructure on top of the usual plain vanilla MIRIAD package would be useful to organize the datareduction. Particularly to make it easy to (re-)reduce each project ("track") with slightly different parameters, forcing us to keep all scripts identical. For this we have assembled a very small scripting layer, which we call EGN, with added pipeline capabilities as well as storing and maintaining a global parameter database that can be shared by all project members. The EGN package is currently maintained via CVS. Details on our project page `http://carma.astro.umd.edu/wiki/index.php/EGNoG`

1.1 outline

A single observing script that performs single dish and interferometric observations of a large mosaic field was used. The two types of data were independantly calibrated and processed to maps, after which a joint deconvolution was performed to create final maps.

1.1.1 Interferometry (UV)

The calibration of the UV data follows the standard procedure. A passband source is observed first (usually 3C84, sometimes Uranus to bootstrap the always present 3C84). In our case, 3C84 was also used as phase/gain calibrator.

[1] Draft version Sep 15

Chapter 2

EGN Pipeline

ABSTRACT

We describe 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. Several implementions exist, the first version was called `drPACS`, followed by `MIS` for the N1333 single dish-interferometric mapping project.

2.1 Introduction

To process a large number of datasets in a very similar way is a common theme, particularly for the type of 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 EGN, which was derived from '**drpacs**'[1] and '**mis**[2]

Technically **egn** 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 **egn** was written for a large set of EGNoG data, you can use it and construct your very own pipeline. An example is given later in this manual.

A summary of typical EGN pipeline commands is:

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

2.2 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

[1] Dalton and Roger were its first users for a PACS project, and helped developing it
[2] MIRIAD Interferometry Singledish toolkit

would manually first run a command as "`step2 foo=bar`", the next time you would run just "`step2`", 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 can 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 four 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 par] [-v par] [-e par] [-z par] [-a] [par=val ...]

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

2.3 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`. The current pipeline has no dependancy for this yet.

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

3. Rerun piece of the pipeline in each of a set of directories. In cumbersome `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 type, 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 $EGN/cat/pipeline.002  > Pipefile
pipe clean all
```

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

7. Run the EGN 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 EGN 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 `egn.def` do the following

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

2.4 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: `egn.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 `egn.def` file)
5. task can now compute
6. task should write out all parameter back to the global database

2.4.1 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).

2.4.2 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('egn.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 `$EGN/templates/pipeline` directory. There are no examples in C or Fortran yet, but are relatively easy to implement and add to your library.

2.5 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 (`egn.def`) in an agreed upon location, currently the **project** directory within the `$EGN_DEF` (if present, or else the default `$EGN/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. **egn** 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
```

In the case that you need multiple calls to set a series of `pipefile`'s, the `-a` flag to `pipepar` is needed, but to ease persistence, it is recommended you sort them using the `-z` flag before saving the files, viz.

```
pipepar -a pipefile=uvflag.egn
...
pipepar -a pipefile=NOTES
...

pipepar -z pipefile
pipesave
```

2.6 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 `$EGN/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.

2.7 EGN Pipeline Command Summary

Although the current `$EGN/bin` directory also contains many EGN 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
```

Chapter 3

Installation

Here we discuss the installation of **egn** and comment on some related packages if you need them.

3.1 egn

We use the same CVS repository as MIRIAD and NEMO, and the installation is very similar. The module is called `mis` and once installed it will need MIRIAD, python + scipy/matplotlib and NEMO (at least for the actual MIS 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 egn
% cd egn
% ./configure
% source egn_start.csh
```

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

3.2 MIRIAD

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

A few reminders on updating MIRIAD programs. Lets take an example where one subroutine from the library was updated (fitsio.for in this example) and the fits program was updated as well.

```
cd $MIR
cvs -nq update
cvs update
mirboss
mir.subs  fitsio
mir.prog  fits
```

but these steps assume you have write permission inside the MIRIAD tree. If you do not, and if it just a program , for most programs there is an easy patch. Again, for the fits program you would do:

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

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

```
cd $EGN
mkdir miriad
cp $MIRPROG/convert/fits.for .
cp $MIRPROG/convert/fits.h   .
mirmake fits
mv fits $MIRBIN
rehash
which fits
```

and now you see the EGN/bin version of the fitsio program, assuming your PATH was set with EGN before MIR.

3.3 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
```

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`.

3.4 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 -
```

If you have write permission, and need to install a new version of a task, this should do it:

```
% cd $NEMO
% cvs -nq update
% cvs update
% (cd src; make install)
% mknemo tabplot
```

Chapter 4

EGN

This Appendix describes some MIS specific aspects of the EGN pipeline.

4.1 EGN parameters

The global pipeline parameters are stored in ASCII format in a small text file, by default this is called `egn.def`. For our EGN 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 egn.def files>
	pipefile	optional comma separated list of to-be-saved filenames
	step	most pipeline scripts store their last successful step name here
getdata	rawdata	where tar.gz or miriad/mir files live [/n/algol2/mpound/data/carma/CARMA23/rawdata]
	project	e.g. cx323.1E_89NGC133.20
	ary	e.g. E or D
	trial	e.g. 20
	link	normally 1, if symlinks used, use 0 if you want a local copy
	scp	if used, it will use scp with 'user@host' style syntax
	cvis	derived from project or ary/trial (e.g. cx323.1E_89NGC133.20.miriad)
report	-	create form log files from listobs, uvlist, uvindex
do_uvflag	flagfile	generic flagger using uvflag and a flagfile
	vis	would override the default cvis
fixOff	offname	for single dish only, creates 'sddata'
do_reduceSD	badants	list of bad antennas (orthogonal to egn.uvflag file)
	npoly	order of polynomial for sinpoly
	sleep	normally unset, because it will prompt for next plot, use 0 in batch
	device	normally /xs for interactive work,use /null in batch
	goto	ieck, Start (default), Inspect, SinPoly, or Maps
map_inttime	source	for selected source, create xyt.tab files dra,ddec,inttime see mk_map_inttime.csh for an example post-processing
do_uvcat1		uvcat to trim interferometry data to our 4 USB windows, creates 'uvdata'
do_inspect1		inspect your interferometry data
do_cal0	linecal	optional calibrations (linecal, antpos)

```
               antpos

do_cal1        -              standard gain and passband calibration

do_map0        -              placeholder for now

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

4.1.1 Data products, Diagnostics

```
On top level:

caldir/                          symbolic link to a location (could be itself) where cx* projects are
cx323.1<ary>_89NGC133.<trial>    project name for given <ary> and <trial>  (we have 22 accepted now)
flux.{tab,ps}                    flux of 3C84 from mfcal (i.e. before uranus fluxcal)
<ant>.{tab,ps}                   antenna based 3c84 gains from selfcal
combined.<mol>_<ary>_<ants>      combined simple 4 channel cube

sd.<mol>.median.cube             single dish cube


On project level:
def/                             symbol link to $EGN/def/$project
SD/                              antenna based SD data
egn.def                          all global MIS pipeline variables
egn.uvflag                       MIS uvflags for do_uvflag
xyt.tab                          pointing and integration times for this trial

On SD level:
MOL.ANT.cube                     cube for given MOL and ANT
MOL.ANT.cube.res
MOL.ANT.map                      sample map for given MOL and ANT
MOL.ANT.map.res
MOL.ANT.ps                       spectrum at 0,0 ?
rms.resid.ps                     RMS
```

4.1.2 Jy/K scaling

Jy/K $4108/D^2$, where D is given in m, is the scaling for a perfect radio telescope, independant of observing frequency. For example, for a 10.2m OVRO dish this would be 38.0, for a 6.2m BIMA dish at 106.9 and the 3.5m SZA dish comes in at 335.3. Aperture efficiency will increase these values.

4.1.3 A benchmark case

A benchmark dataset is available seperately, and should be run to confirm the software works properly.

```
% cd $EGN
% wget ftp://ftp.astro.umd.edu:/pub/carma/egndata.tar.gz
% tar zxf egndata.tar.gz
% rm  egndata.tar.gz
% mkdir tmp; cd tmp
% ../egndata/test-all
```

4.2 EGN ISSUES

4.2.1 WIKI

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

BAD:

-once par has been set, hard to set another value

-one single pipeline.... but what if, as here, we have 2

-easy way to set a global def file to be read for each project?