Chapter 1

MIS (Miriad Interferometry Singledish) Toolkit

MIS (Miriad Interferometry Singledish) is a toolkit to help you reducing mosaiced CARMA-23 single dish and interferometric data, and combine them. This particular manual was written with the (Spring 2011) NGC 1333 project (cx323) in mind, but is also being used for the CARMA's CLASSy key project (c0924)(Spring/Summer 2012) and followups (c1186, Fall 2013) [1]

To give you an idea of the multiplicity and complexity of this kind of project for CARMA, consider the following Drake-like formula for these data:

$$NCLOUD * NMOL * NCH * NAC * NTR * NP * NAB$$

where, in the example of N1333:

NCLOUD	Number of clouds (5: SVS13 (n1333), B1S, L1451, SERPM, SERPS)
NMOL	Number of molecules (3: HCN, 88.631847 GHz, N_2H^+, 93.173505 GHz, and HCO^+, 89.188518 GHz, plus an additional continuum band).
NCH	Number of channels (159 for line, 47 for continuum)
NAC	Number of array configurations (2: E and D)
NTR	Number of trials (10-20)
NP	Number of pointings (527), 30 arcsec grid spacing in an 8´ by 11´ region for SVS13. Similar for others.
NAB	Number of antennas (23) or baselines (253), depending on data reduction

where first the single dish data with NAB=23 has to be reduced, followed by NAB=253 interferometric baselines. In the end, after combining the single dish and interferometry data, we will wind up with NMOL (3) cubes of NCH (159) channels each, and with probably around 512 x 512 spatial pixels. In addition, there will also be a single continuum map of the same region. In the particular case of N1333 this resulted in 23 datasets, taking up about 16 GB of raw MIRIAD data.

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

[1] Draft version January, 2014

small scripting layer, which we call MIS, with added pipeline capabilities as well as storing and maintaining a global parameter database that can be shared by all project members. The MIS package is currently maintained via CVS. Details on our project page `http://carma.astro.umd.edu/wiki/index.php/NGC1333`

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 Single Dish (SD)

The reduction of SD data is somewhat unusual. First of all, we have 3 different types of dishes, each with roughly the same, but a different size of their primary beam. But as we shall see, each dish has a different response (modeled by a simple linear scaling) to the sky signal.

We used a position switching technique to a piece of the sky (about XX arcmin to the east of SVS13) where no signal was known. This only works for the molecular line windows, not the continuum:

$$T = \frac{ON - OFF}{OFF} T_{sys}$$

The miriad task `sinbad` does this step.

Next a linear baseline was removed. One has to be careful with outflows, which in certain positions can occupy a significant part of the window. The miriad task `sinpoly` does this step.

Finally, the pointings are interpolated into a single map (cube really). The miriad task `varmaps` does this.

1.1.2 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, because of its close proximity (10 degrees) from the source.

1.1.3 some more SD notes

You need a clean, emission-free OFF position as close to your source as possible, less than half a degree if possible. Create a separate source in your catalog for this position. Make sure you set 'doppler' in your script to your main source.

You need to integrate on the OFF about every 2 minutes. To do this set up the sources and mosaic sections of your script to visit the OFF at this interval and to change the "pointStatus" value depending on the source. Below is the section from the NGC1333 script. N1333OFF is the OFF position and SVS13. We did each position for 15 seconds and switched to the calibrator every 8 cycles.

We have developed new scripts and miriad tasks (and upgraded old ones) to reduce the data. See http://carma.astro.umd.edu/wiki/index.php/NGC1333 Some of the scripts are still N1333 specific (e.g. they assume our correlator setup), so examine them closely.

CAVEATS:

We found that doing singledish in this way increases the observing overhead to 50-60%. This affects not only your rms sensitivity but your UV coverage. We had a very large mosaic so updated our mosaic starting position every day to move the uv coverage around.

Flagging is very important. Make sure you flag the autocorrelation data where you flag the crosscorrelation data (do_uvflag script does this).

Note we did NGC1333 in CARMA-23 mode, so got some of the short spacings from the 3.5m for overlap with the 10-m autocorrs. If you are doing CARMA15, the joint deconvolution may not work as well.

Chapter 2

MIS 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.

2.1 Introduction

To process a large number of datasets in a very similar way is a common theme, particularly for the type of N1333 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 MIS, which was derived from '**drpacs**'[1].

Technically **mis** 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 **mis** was written for a large set of N1333 data, you can use it and construct your very own pipeline. An example is given later in this manual.

A summary of typical MIS pipeline commands is:

```
pipeline $MIS/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 would manually first run a command as "`step2 foo=bar`", the next time you would run just "`step2`", it

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

[2]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 [-1|2] [-h] [-c] [-l] [-f parfile_name] [-s shell] [-p fmt] [-d par] [-v par] [-e par] [-z par] [-a] [par=val ...

 -h     help
 -1         Use single quotes (default)
 -2         Use double quotes
 -c     create empty mis.def (parfile_name)
 -l         use spaces when printing out (-v) comma separated keyword values
 -f file    use another parfile_name from the default mis.def (not recommended)
 -s shell   output for given shell (csh and bash are currently known)
 -p fmt     C-style format (e.g. %10.2f) used in printing (-v) values
 -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
 -a         append a value (in a comma separated way) to the existing value(s)
 par=val    assign (new) value to a parameter  (multiple allowed, -a will append)
```

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 $MIS/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 MIS 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] [-v] [-w] dirs.txt cmd [args]
   -h          this help
   -n #procs   parallel processing using #procs processors
   -o logfile  output log
   -c          create directories
   -v          verbose (debug)
   -w          wait for RETURN after each project
   dirs.txt    text file with directory names
   cmd         unix command to run
   args        arguments to unix command (inlcuding ; cmd2 args2...)
   ```

8. Re-Run the MIS 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 Ĉ, it will instead work on the next project directory. A better way to halt the series, is to issue Ẑ, 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 `mis.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: `mis.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 `mis.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('mis.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 `$MIS/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 (`mis.def`) in an agreed upon location, currently the **project** directory within the `$MIS_DEF` (if present, or else the default `$MIS/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. **mis** 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.mis
...
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 `$MIS/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 MIS Pipeline Command Summary

Although the current `$MIS/bin` directory also contains many MIS 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 MIS and comment on some related packages if you need them.

3.1 MIS

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 mis
  % cd mis
  % ./configure
  % source mis_start.csh
```

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

3.2 MIRIAD

For the actual MIS 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 (`fits.for`) 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 $MIS
  mkdir miriad
  cp $MIRPROG/convert/fits.for .
  cp $MIRPROG/convert/fits.h   .
  mirmake fits
  mv fits $MISBIN
  rehash
  which fits
```

and now you see the MIS/bin version of the fitsio program, assuming your PATH was set with MIS 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

MIS

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

4.1 MIS parameters

The global pipeline parameters are stored in ASCII format in a small text file, by default this is called `mis.def`. For our MIS 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 mis.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 mis.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 $MIS/def/$project
SD/                             antenna based SD data
mis.def                         all global MIS pipeline variables
mis.uvflag                      MIS uvflags for do_uvflag
MOL.median.cube                 for given molecule, median cube from all ants
MOL.resid.cube                  residual, but all 23 ants in here
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 $\approx 4108/D^2$, 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 $MIS
% wget ftp://ftp.astro.umd.edu:/pub/carma/misdata.tar.gz
% tar zxf misdata.tar.gz
% rm  misdata.tar.gz
% mkdir tmp; cd tmp
% ../misdata/test-all
```

4.2 MIS example data ingestion

In this Section we are show how a just newly observed project is entered to MIS. Usually an email come in, which we refer to as the “endtrack” email, which contains two attachments: the `scriptlog` and the observing file, e.g. `c1186_1E_87Serpen.obs`, for a given track, e.g. `c1186.1E_87Serpen.SL.10`. This is entered to MIS, and lets say this is on computer `A`, given by the `A%` prompt:

```
A% mis_new c1186.1E_87Serpen.SL.10 ~/Downloads/scriptlog.txt ~/Downloads/c1186_1E_87Serpen.obs
```

we then move to the data reduction computer (B) and assuming the new dataset, `c1186.1E_87Serpen.SL.10.miriad.tar.gz`, had been downloaded to the appropriate place, MIS can be updated :

```
B% mis -u
B% cd $MIS/data/CLASSy_reduction
B% set p=c1186.1E_87Serpen.SL.10
B% mkdir $p
B% cd $p
B% pipesetup project=$p
B% getdata
B% report
```

inspecting a section in the `listobs.log` file

```
--------------------------------------------------------------------------------
            Observed Sources Coordinates and Corr Freqs
Source         Purpose    RA          Decl         Vlsr            Corfs in MHz
NOISE              B      12 56 11.17 -5 47 21.52     8.00E+00     0.0
1751+096           B      17 51 32.82 9 39 00.73      8.00E+00     0.0
MWC349             F      20 32 45.53 40 39 36.63     8.00E+00     0.0
1743-038           G      17 43 58.86 -3 50 04.61     8.00E+00     0.0
SERPM              S      18 29 49.66 1 14 55.81      8.00E+00     0.0
--------------------------------------------------------------------------------
```

we can then assign the calibrators (in theory this step could be automated, see examples in the quality script)

```
B% pipepar fcalname=MWC349 pcalname=1751+096 gcalname=1743-038 calname=1743-038 srcname=SERPM
B% cp $MIS/cat/mis.uvflag .
B% do_uvcat1
B% do_uvcat3

# project                size   bmaj   bmin  bpa tint    rmscal      rmsres    res/cal
c1186.1E_87Serpen.SL.10  10m  7.669  6.166 42.5 0.68 3.834E-03 1.86878E-03  0.4874230 1743-038
c1186.1E_87Serpen.SL.10   6m 30.136 13.585 -3.2 0.68 8.361E-03 6.30106E-03  0.7536252 1743-038
c1186.1E_87Serpen.SL.10 3.5m 30.734 20.250  6.2 0.68 1.903E-02 1.18977E-02  0.6252076 1743-038

B% do_uvcat2
B% do_uvcatSD
B% pipesave
B% (cd def; cvs add mis.def    ; cvs ci -m new mis.def)
B% (cd def; cvs add mis.uvflag ; cvs ci -m new mis.uvflag)
```

and now the project has been ingested. The CVS repository now contains reasonable `mis.def` and a default empty `mis.uvflag` file. Team members can now grab their version and fine tune the flagging.

4.3 MIS ISSUES

4.3.1 WIKI

See also our N1333 wiki: `http://carma.astro.umd.edu/wiki/index.php/NGC1333`

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?

4.4 OVERHEADS

For the 2011 data (N1333) we used a semi-optimized observing procedure, which still leaves rather large overheads. Here are two examples for a typical 8 hour observing session in the N1333 project: (`inttime` is the true on-source integration time, whereas `winttime` includes the wait until that source is on target, so it includes slewing to that source.

```
         inttime    winttime %inttime   %winttime
          (hr)      (hr)

w/SD, OFF position is 19 arcmin to the east       cx323.1E_89NGC133.11

Total               7.490
3C84     0.467      0.719    0.062      0.096
N1333OFF 0.329      1.505    0.044      0.201
NOISE    0.008      0.900    0.001      0.120
SVS13    2.283      4.366    0.305      0.583

wo/SD                                             cx323.1D_89NGC133.12

Total               8.434
3C84     0.729      1.142    0.086      0.135
NOISE    0.013      1.329    0.002      0.158
SVS13    3.725      5.963    0.442      0.707
```

It should be noted that a NOISE scan was often done before a SYSTEMP calculation, thus showing rather large fractions (nearly 16 and 12% in the two cases) for `winttime`. None-the-less, on-source fractions for the non-SD projects of 44% is not very impressive.

4.5 NOTES

Here we gather some notes on the different objects we have mapped and are available for reduction in MIS

4.5.1 Sources

The following sources have been part of the Perseus and Serpens molecular SERPM/c1186 18 29 49.66 1 14 55.81 8.0cloud surveys:

```
srcname/project       RA          DEC         VLSR

SERPM/c0924     18 30 01.32    1 12 38.11      8.0
SERPM/c1186     18 29 49.66    1 14 55.81      8.0
SERPS/c1186     18 29 52.50   -1 56 36.48      8.0
SERPS/c0924     18 29 52.50   -1 56 36.47      8.0
SERPS/c1186     18 29 52.50   -1 56 36.47      8.0
SVS13/cx323      3 29 03.30   31 16 00.01      8.6
B1S/c0924        3 33 20.08   31 08 45.33      6.5
L1451/c0924      3 25 17.00   30 21 22.74      4.0   (aka B1E)
```

In `c1186` new isotopics were introduced: $H^{13}CN$, 86.34 GHz, HNC, 90.66356 GHz, and $H^{13}CO^+$, 86.34 GHz.

4.5.2 cx323: IRC10216

First test-object we observed. IRC 10216 is an evolved star, has been well studied before and has a range of spatial frequencies that allowed us to test the larger N1333 survey. (HCN, 88.631847 GHz, HC_3N, 90.978993 GHz(no detections), and HCO^+, 89.188518 GHz, plus an additional 500 MHz continuum band). There are 3 decent tracks in both E and D array. These also used different molecules from the ones we used in subsequent observations (N1333 and others)

4.5.3 cx323: N1333

Observed spring 2011 in E and D array. 8 x 11 arcmin, 527 pointings. Last three tracks in D we deemed we had enough Single Dish, and improved observing efficiency by removing the SD off-source scans. This increased the on-science time from XX% to YY%.

E array: 13 tracks, of which 12 were run, and 10 had useful data.

D array: 22 tracks, of which 18 were run, and 13 had useful data.

As an example of archive access, all the N1333 data were refilled and downloaded in September 2012. Refillling took about an hour, downloading UIUC to UMD at a typical rate of 10MB/sec. Filling speed is about 1.6 MB/sec. The total disk space is 14.1 GB (compressed tar).

4.5.4 c0924: Serpens: SERPM and SERPS

One of the two cloud complexes for the c0924 key project, called CLASSy. Observed Spring 2012 - Winter 2012. For consortium sharing reasons the projects were labeled c0924, c0924I and c0924V in the CARMA data archive.

4.5.5 c0924: Perseus: B1S and L1451

One of the two cloud complexes for the c0924 key project, called CLASSy. Observed Spring 2012 - Winter 2012. For consortium sharing reasons the projects were labeled c0924, c0924I and c0924V in the CARMA data archive.

4.5.6 c1186: Serpens: SERPM and SERPS

Observe 3 new molecules : $H^{13}CN$, 86.34 GHz, HNC, 90.66356 GHz, and $H^{13}CO^+$, 86.34 GHzon a smaller grid (20 pointings for both SERPM and SERPS)

4.6 CARMA Auto-Correllations: Single Dish Map Making

This section describes some of the practical aspects of the data reduction of CARMA auto-correlation (AC) data into a single dish map. A convenient way to visualize the AC's is as a (real) matrix $A(c, t)$, where c denotes channel (159 in our case) and t denotes time[1] See Figure 4.1. In a typical track of 6 hours we have close to 1000 integrations of 15 seconds each: the AC's are typically organized in two paired OFF integrations, followed by about 12-15 ON integrations (of course the cross-correlations are taking place as well). Also note for each track we obtain many independant single dish maps, from each antennae, although for the current analysis we only use the 6 more sensitive 10m dishes of the array.

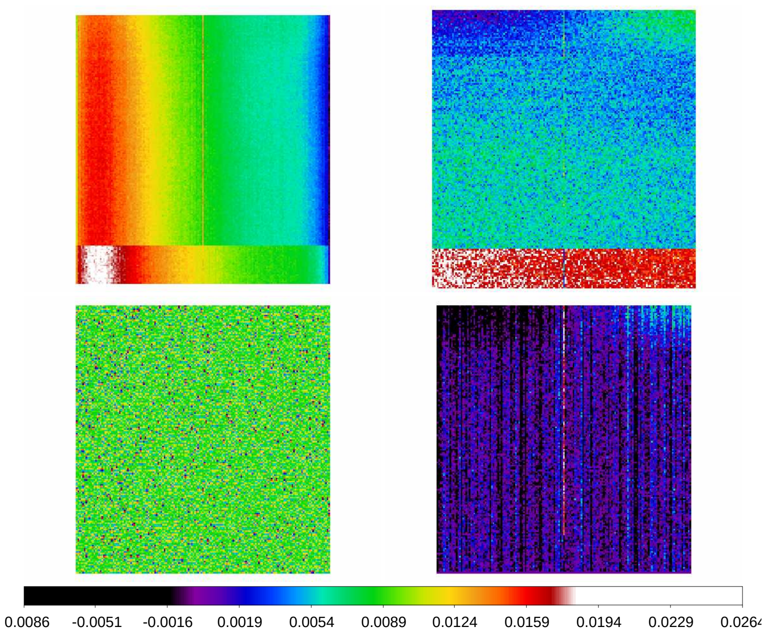

Figure 4.1: Auto Correlation (AC) maps for the OFF positions (c0924I.1D_88B1S.1) of the HCO^+ window 15 for Antenna 1. Horizontal are 159 channels, vertical are 166 times, between 20:56:29 and 02:56:27.0. Top left are the raw data, notice the slight jump of almost 15mU (or about 1.5% of autocorrelation level) near the start of the observation. Top right panel shows the raw data where the average of each column has been subtracted, showing longer term trends. In the bottom left differences between two adjancent OFF rows are plotted, showing short term time variations. The bottom right panel shows when the first OFF observation was used to normalize (diff?) and subtract all other observations. Also notice the middle ("channel 80") spike. Created with the `do_statsSD normalize=0` script.

We construct a single dish map (in K) using the usual

$$T_A = T_{sys}(\frac{ON}{OFF} - 1)$$

Where the Jy/K scaling still needs to be applied.

We first note the following about these data:

- The AC's are dimensionless and about unity. We simply refer to them as "U", by lack of a better term. Typical variations are just a few "mU", but multiplying the differences between ON and OFF by a sizeable system temperature (typically 300K) still results in 0.1K noise in T_A. Most of

[1] MIS script `do_statsSD` computes most of the images and numbers quoted in this section

our signal is just a few K. After averaging 20 tracks, the final noise in the cubes winds up between 10 and 30 mK, but with some baseline issues in places.

- At certain moments in time, luckily always right before an OFF integration (if they occur), the AC's seem to get re-normalized a bit. This is typically 5-10%. There appears to be no correlation with either window or antennae, but in a single observation, if they do happen, they happen at a given set of discrete times, but not for all antennae/windows pairs, neither for all windows or antennas. This prevents us from averaging and interpolating accross OFF positions, at least without normalization (see below).
- Short term time variations of the (normalized) autocorrelations are very well behaved, do not depend on either channel or time, and are about 0.879 mU (±0.002), measured from the "D" maps. Since this measurement was done on "(even-odd)/2", the actual variation on the autocorrellation is $\sqrt{2}$ more, i.e. 1.243 mU (±0.003). Differencing measurements accross the ON's (where the two OFF's are separated by about 3-4 minutes) the same answer.
- Raw data have short term RMS of about 0.94 mU (variance 0.09) normalized data have a smaller RMS, 0.879 mU.
- Longer term time variations are more like 1.28 mU (±0.05), as judged on the "S" maps (such as renormalized maps). With the $\sqrt{2}$ this is actually nearly identical what was measured in the "D" maps (1.243). This suggests it is safe to box-car average a number of passbands to achieve a better OFF division. If the OFF position would be known to high precision, the noise is only improved by $\sqrt{2}$.
- A single OFF position was done after every 7 scans for SVS13 (N1333), each integration was 15 second. For all other regions (B1S, L1451, SERPM and SERPS), two OFF positions were done after 9 scans. Also 15 seconds per integration. Stability suggest we may not even need two.
- system temperatures at CARMA are measured per spectral window, and are instantaneous.
- Using the OFF positions from blanksky data we can study the noise characteristics as function of bandwidth. So far it looks like $\frac{1}{\sqrt{BW}}$ is only obeyed from 8 to 125 MHz, but the 250 and 500 MHz bands seem to have an increased noise, a factor ≈ 2.

Here's an outline how the data reduction takes place and some caveats and comments. There are 4 steps to get to a final single dish map:

1 **sinbad**: Spectrum construction using ON and OFF. First the AC's are normalized, to account for the occasional slight jumps. This allows for a wider averaging of OFF's, instead of just using the last OFF. If just the last OFF was used, this is strictly not needed. However, by averaging OFF's we can reduce the channel to channel variations in the source data by up to $\sqrt{2}$.

2 **sinpoly**: Baseline subtraction. This is fairly straightforward. For each pointing a channel range is defined where a first order polynomial is fitted to the background, and subtracted. The data is now ready for gridding.

3 **varmaps**: Gridding. This is done by a weighted average using a gaussian convolution of all the nearby (as defined by size=size1,size2) points. Optionally weighted using the systemp temperatures.

4a **ORIMSR**: Optionally we can use ORIMSR data to predict the scaling between antennae. Previously it had been determined that Jy/K $\approx$ 65 for the 10m dishes.

4b **imstack**: After `varmaps` has made (six) SD maps for each antenna, it turns out these maps are not at the same scale. `imcmp` clearly shows that antennas can vary as much as 20%. `imstack` will allow a set of cubes to be mean/median "averaged", and a residual cube for each antennae from this "averaged" cube be derived for inspection. Using C4 as reference antenna, the scaling factors for the other antennas happen to be all larger than 1 and are: C1: 1.26 (0.03), C2: 1.48 (0.09), C3: 1.28 (0.08), C5: 1.51 (0.10), C6: 1.16 (0.05), with variances listed in parenthesis.

4c **smooth**: For stability of the joint deconvolution, the single dish maps have a rather sharp edge as seen by 6m and 3.5m dishes. The gridded map needs to be gently extended to roughly match the expected dropoff of the signal from the smaller dishes. We achieve this by iteratively smoothing this extended map, but continuesly replacing the inner well defined higher resolution maps at each iteration. Thus one arrives at a hybrid resolution map where the resolution in the inner and outer region is

$$E_i^2 = O^2 + B^2 + S^2$$

and

$$E_o^2 = O^2 + B^2 + nS^2$$

Some unresolved lower priority issues, some perhaps useful for future expansion:

- no rigorous flagging has been done, previously we use the interferometric flags in `mis.uvflag`, but now we enforce `mis.sdflag`. Most are empty. There's some indication some scan-based striping can occur from times when the systemp temps were not behaving. However, they tend to average out over many tracks.
- In plots comparing the pixel-by-pixel flux (`imcmp in1= in2=`), at the high end it seems that not all tracks were equally well calibrated. 2-4% variations are visible.
- The code in `imstack` that compute the scaling factor(s) via a mean OLS does not report a formal error. With the previous point it would be good to understand the magnitudes of both effects.
- no good comparison done between the simple sinbad approach, and the more sophisticated `oaver=8,8 normalize=true repair=80,239,398` version.
- proper softened edges out of varmaps, these are still far from ideal, as we have to cheat here and fight reality. Now solved by iterative smoothing? See `do_mapSDmedian4`.
- weighting in varmaps by systemp. Although implemented by default, not well tested if it makes a difference.
- no rigurous writeup of the effect of a gaussian interpolation of a signal sampled with another gaussian beam. The resulting beam has been assumed to be the sum of the two beams (in the usual quadrature).
- In addition to this: cone shaped interpolations in varmaps (`mode=1`) looks pretty horrible. Is that really what it should look like.
- In June 2012, the primary beam of 10m was redefined from 1.073 to 1.210 arcmin if measured at 100 GHz. (see also CARMA memo 52). MIRIAD software was updated at that time to take advantage of this.

Remaining Oddities

- cx323.1D_89NGC133.1: sinbad normalize=true does not seem to remove. high systemp flagging was needed.

	HCN, 88.631847 GHz	N_2H^+, 93.173505 GHz	HCO^+, 89.188518 GHz
B1S	1.38	2.74	1.54
SERPM	3.89	3.22	8.16
L1451 (B1E)	1.31	0.96	2.61
SYS13 (N1333)	4.13	3.49	5.62
SERPS	1.39	2.90	1.01

Table 4.1: Single Dish cloud properties: peak temperatures in the cube. The noise in these cubes ranges from about 10-30 mK.