ALMA Memo ???
Version 1.2 WVR Simulation Code User's Manual

M.A. Holdaway
NRAO/Tucson

Alison Stirling
Cambridge University

August 24, 2004

Abstract

In order to investigate the performance of water vapor radiometry (WVR) as a phase calibration technique, and the interface between fast switching and WVR, we have built a simulation package in AIPS++. This package integrates standard AIPS++ tools and data structures with special-purpose code to produce a powerful, flexible, and extensible package. This memo documents the functionality of this simulation package.

1 System Overview

The heart of the simulation system is the `wvrsim` tool. At the heart of this tool is a 3-D atmospheric model, consisting of arrays of temperature, pressure, and water vapor pressure, along with associated functions which can manipulate and integrate through the model atmosphere. The `wvrsim` tool attaches to an existing `measurement set` (ie, a data object of an inteferometric observation), a gain table which stores the calculated phase corruptions, and a special purpose gain table which stores the calculated WVR response to the atmosphere. The WVR channel brightness is calculated with Scott Paine's *am* atmospheric model program (ie, see `http://sma-www.harvard.edu/private/memos/tech_no.html`, memo 152), and the delay (which is converted into a phase for the cal and target observing frequencies) is calculated in our own code by integrating the Smith-Weintraub equation (ie, our atmosphere is non-dispersive).

After the data has been simulated, the data can be reduced using the `wvrsimreduce` tool. The public functions of these tools are explained in the appendices at the end of this manual.

These tools and their functions have been written both to provide fundamentally new functionality within AIPS++, as well as to steer the user through existing AIPS++ functionality. In addition to the straight and narrow path defined by these tools and their functions, the user also has access to the full range of AIPS++ tools, including manipulation of the measurement set and table access/viewing, as well as command line manipulation of glish arrays and data structures. This is especially powerful as the scripting language is the command line language, so any part of a script can be entered at the command line. Additional functionality can easily be written as glish functions.

2 Data Flow and Hand Holding

In this section, I try to explain where the data comes from and where it goes, and parallel to that process, I take the user step by step through the process of running a simulation. Between the two intertwined threads, hopefully the user understands what is going on with the simulations and how to make them happen.

2.1 Required Starting Data

Before we start the atmospheric simulations, we need a model atmosphere. This is provided by Alison Sterling's atmospheric simulation code. These turbulent hydrodynamical simulations take over a week to produce the 3-D arrays which represent temperature, pressure, and water vapor pressure, so we'll just assume you've got one of these laying about on disk. The ASCII format for these files is not explained here, an example can be found in `TEST_ATMOS/`, which holds the files for the example atmosphere included in the tar file of this package.

2.2 Starting AIPS++

If you don't have AIPS++ installed, go to the AIPS++ web site `ftp://ftp.cv.nrao.edu/aips++/end-user/application/rh9` and download several required RPM's. There are about 10 RPM's here. The first two AIPS++ RPM's should have 19.615-26ds in their names. You probably need to get the all of these RPM's, including the g77, gcc, and pgplot RPM's. After downloading the RPM's, login with root permission, and do a `rpm -i *.rpm` to install AIPS++.

To begin, go to the work directory and type `aips++`. A few windows may pop up – minimize them, you may need the Log Messages window to decipher an error, but you will likely not need these windows. We are running AIPS++ from the command line and with glish scripts. At times, we will invoke functions from the command line which pop up a viewer window or a pgplot window. When you are done with these, you can `dismiss` or `done` them. It is assumed that you, the user, will soon be able to write your own glish scripts to drive and evaluate entire sets of complex simulations.

The first actions to do in AIPS++ are to load in the scripts which define our simulation tools and functions:

```
include 'wvrsim.g'
```

An advanced user might follow by including their own function definitions. An example of such a function is `wvrtest2()` which is defined at the end of `wvrsim.g`. This runs a simulation with the test amtmosphere, simulating a 10 antenna array for 50 s, and will take about 10 minutes to run. Do this now to test that everything is fine. Feel free to create a new "*.g" file and create your own functions based on the examples provided in `wvrsim.g`. The individual lines in a script can also by typed directly into the AIPS++ command line interface, or they can be "dragged and dropped" from a text editor showing such a file, into the terminal window running AIPS++.

2.3 The MeasurementSet

One of the first things we need to do is to simulate a Measurement Set (ie, the AIPS++ visibility data set) to provide the coordinates onto which we will be simulating.

The `createms()` function drives the AIPS++ simulator to produce a fast switching measurementset (also `ms`, the AIPS++ name for a visibility data set)[1]. The simulator requires an antenna station file to define the array geometry in local coordinates, an integration time, the number of integrations on the calibrator and target source for each cycle, and the total observing time – these define both the `ms` and most of the coordinates of the observations. However, the sky coordinates of the cal and target sources are dummy values at this point – they can be updated later in the simulation process to more simply reflect the AZ-EL geometry we will be interested in when we are doing the atmospheric simulations.

2.4 wvrsim

Next, we create a wvrsim tool with

```
mywvrsim := wvrsim()
```

This wvrsim tool is the central hub for the WVR *simulations*. The wvrreduce tool will be the central hub for processing the WVR data after the atmospheric data has been simulated. In AIPS++, a *tool* has associated data and defines functions which act on this data. When we create a tool, we give it a name, because we could have more than one instantiation of a given type of tool. This instantiation of the `wvrsim` tool is called `mywvrsim`. The syntax for invoking the functions that are defined for this tool is *toolname.functionname(parametername=value)*. As each instantiation of a tool uses up computer resources (ie, memory and diskspace), it is wise to free up those resources when that tool has fulfilled its purpose. Every AIPS++ tool has a "done" function to free those resources:

```
mywvrsim.done()
```

Oops! We didn't mean to end the party so soon. If you DID done(), create mywvrsim again before proceeding.

The first wvrsim functions we need to use are the ones that grab the main data objects, the model atmosphere and the MeasurementSet.

```
mywvrsim.readatmosascii(basename='r0103.dg0007', tablename='atmos.tab')
```

will convert the ASCII format provided by Alison Stirling, and will write out an AIPS++ table. After writing the table, we can just read in the AIPS++ table without ever referring to the ASCII:

```
mywvrsim.readatmos('atmos.tab');
```

The MeasurementSet which we made with `createms()` can be attached to the wvrsim tool by typing

```
mywvrsim.setdata(msname='TEST1.MS', atmogainname='TEST1.GAINS',
                 wvrgainname='TEST1.WVR');
```

[1]Note that the AIPS++ simulator tool is undergoing a major rewrite which will alter the interface to that tool. When the new simulator is finished, we will provide a function `createnewms`. The new simulator will have significant improvements, and will permit the dual-band fast switching to be simulated

In addition to attaching the preexisting `ms` to the wvrsim tool, `setdata()` also creates a blank gain table to hold the atmospheric phase corruptions integrated along the line of sight, and a blank gain table to hold the WVR channel data. (The WVR channel parameters and weights are hardwired into the `wvrsim` code, but these values could be editted.)

Now that we have attached the model atmosphere and the MeasurementSet to the `wvrsim` tool, we need to align their coordinates, basically by shifting the atmosphere to some place on the earth, in the antenna's coordinate system, which will permit all lines of sight from the antennas to travel through the atmosphere and reach the top (rather than pop out the sides or miss the atmosphere altogether). First, we need to set the AZ and EL angles for the target source and calibrator source. (Of course, we could have set the cal and target sources' celestial positions in `createms()`, but setting the AZ and EL is actually closer to the questions which researchers will be asking of the simulations.)

```
mywvrsim.setobs(targetaz=90, targetel=70, calaz=88, calel=71);
```

does this. Note that all angles specified through the `wvrsim` user interface are in degrees. Also note that we can create a single `ms` and copy it for repeated use, just changing various parameters such as the elevation angle of the observations.

Finally, to connect the coordinates of the `ms| and the model atmosphere, we set the atmospheric velocity, reference position, and reference pixel. To prevent user burnout in calculation these, we provide the simple method`

```
mywvrsim.suggestrefpos(velocity=[10,0,0], refpix=[1,1,1], adopt=T)
```

The code will basically scope out the region of space required in the atmosphere for the observations based on the antenna locations, the length of the observations, the atmospheric velocity, and the directions of the observations, then determine if the observations will fit, and then fix the reference position of the given reference pixel such that the observations are "centered" in the atmosphere, or return F if the observations are too big for the atmosphere (in which case, we need a bigger atmosphere, or a smaller array configuration, a higher elevation, or a shorter observation).

The user does have complete control over the referencing of the atmosphere to the array configuration coordinates through `setatmocoordinates()`, and can then check that the referencing is fine with `checkrefpos()`; these will be described later.

Now we have all the required data, and the atmosphere and antennas are correctly referenced to each other. We can now perform the simulations, calculating the atmospheric delay (phase) and WVR channel data for each antenna at each integration time while blowing the atmosphere over the array. The function

```
mywvrsim.letitroll();
```

oversees these operations. As these calculations are very time consuming, you may want to run it as

```
mywvrsim.letitroll(verifyonly=T);
```

which will do a dry run without calculations, just to make sure that all lines of sight do indeed stay within the atmosphere! Note that `letitroll()` enters values for the antenna phases, the WVR channel brightnesses, and the weighted WVR brightness, into gain tables and that the MeasurementSet is not corrupted.

To add thermal noise to the WVR channel data, invoke

```
mywvrsim.addwvrnoise(noisescale=1.0);
```

The thermal noise is that given in Richard Hills' memo (ALMA Memo 495) (0.313, 0.144, 0.112, and 0.079 K in 1 s for the four channels), but the thermal noise can be adjusted with the noisescale parameter. 1/f gain fluctuations can be added with addwvrdrift (asee Appendix).

We apply the atmospheric phase fluctuations to the MeasurementSet via

```
mywvrsim.corruptvis();
```

After the antenna-based atmospheric phase errors are applied to the visibilities, we must add some thermal noise to the visibilities. If we don't do so, not only will we get unrealistically accurate gain solutions in the reduction stage, but the gain solution process will be likely to fail as well (it does not work well with perfect data). This noise is a "per visibility, per integration" quantity. Noise as small as 0.000001 Jy is sufficient to stabilize the gain solutions; one may want to add minimal noise when debugging the system.

```
amywvrsim.ddnoisetovis(noise='0.01Jy');
```

By the way, estimates of the noise performance for ALMA antennas can be made by the `almasensitivity` tool:

```
include 'almasensitivity.g'
mysens := almasensitivity();
mysens.usejanskies();
mysens.setinstrument(nants=2, diam=12);
mysens.setatmos(pwv=1.3446, airmass=1.3);
# one baseline, 1~s observation, 8GHz bandwidth, 230 GHz obs freq
mysens.setobservation(npol=2, bandwidth=8, freq=230, dt=1);
noise := mysens.calcsensitivity1('obs');
mysens.done();
```

We have completed the simulation stage of our work. From here on out, we will use a `wvrreduce` tool. Before we free the resources taken up by the `wvrsim` tool, we will capture some of the information which we need to create the reduction tool in a record called `myinfo`:

```
myinfo := mywvrsim.getinfo()
mywvrsim.done();
```

And thus we bid farewell to our trusty servent the `wvrsim` tool.

2.5 A few notes on Tables

Thus far in the process, we have created a MeasurementSet (actually a collection of tables), a regular antenna-based gain table, and a WVR data table. These are stored on disk in a binary format specific to AIPS++, but they can be viewed very easily by creating a table tool:

```
mytab := table('TEST1.MS')
mytab.browse();
```

We can see all of the data in these tables. Once we see what the columns are named, we can fetch out the values from these tables as vectors:

```
mytime := mytab.getcol('TIME');
nn := len(mytime)
for (i in [1:nn]) { print 'The time is now', mytime[i] };
```

Some of the table cells are themselves arrays, so fetching a column will result in a higher dimension array:

```
mytab := table('WVR.GAIN')
wvrdat:= mytab.getcol('WVR-DATA');
wvrshape := shape(wvrdat);
for (i in [1:wvrshape[2]]) {
     print 'First two channels are:',wvrdat[1,i], wvrdat[2,i]  };
```

If you want to modify table data and put it back, you need to create the table object (here we name it `mytab`) as writeable:

```
mytab := table('WVR.GAIN', readonly=F)
wvrdat:= mytab.getcol('WVR-DATA');
wvrdat := wvrdat + 0.15;
mytab.putcol('WVR-DATA', wvrdat);
mytab.done();
```

If you do modify a table with `putcol()`, the new results will not show up in the browser – you actually need to `done()` the table, reopen it, and `browse()` again to see the modifications.

For more information on playing with tables, see the AIPS++ User Reference Manual at `http://aips2.nrao.edu/docs/user/Refman/Refman.html`. Of course, they've also got the documentation on how to use the other standard AIPS++ tools too: the MeasurementSet, the simulator, the calibrater, mathematics objects, random number generators, fitters, plotters; its all there – EXCEPT for the user generated tools that we are trying to document here, like `wvrsim` and `wvrreduce`.

2.6 wvrreduce

Up until this point, we have simulated a fake MeasurementSet and then simulated the phase errors and WVR channel emission which a model atmosphere would have produced. Creating the MeasurementSet was performed by a stand alone function `createms()`, but the rest of the work was accomplished using the `wvrsim` tool. There is something of a firewall between the tool to simulate the affects of trhe atmosphere and the tool to process the data and attempt to correct for the atmospheric affects. The tool which processes the simulated data is called `wvrreduce`.

We create the `wvrreduce` tool and initialize it with a bit of data from the old `wvrsim` tool:

```
mywvrreduce := wvrreduce();
mywvrreduce.setdat( myinfo );
```

(The record `myinfo` contains the name of the MeasurementSet and the WVR channel data tables, along with the elevation angles of the calibrator and target sources, and resulted from invoking `mywvrsim.getinfo()`).

The phase errors in the target source visibilities can be exlored as a function of baseline length via a struction function analysis:

```
  sf := mywvrreduce.structurefunctionfromms(doplot=T);
  print 'RMS phase(r) = ', sf.amp, ' * r ^(', sf.exp, ')';
```

To proceed with the WVR reduction, we must solve for the atmospheric gains on the calibrator source:

```
mywvrreduce.solveatmogains(newgtname='CAL.GAINS');
```

At this point, we are on the verge of doing the cool stuff, working out observing/calibration algorithms for combining fast switching and WVR. Unfortunately, the code for this has not been written. SO, the following calls are *dreamware*, but could be completed in a few days.

One thing we can do is just look at how well fast switching works. The function `interpolatefs()` has been written, but we don't yet have `applyfs()`. When we do, it will look like

```
(dreamware)
mywvrreduce.applyfs(newms='TEST1.MS.FS', method='2pt')
```

This will copy the MeasurementSet to a new one (we don't want to ruin all the work we did to make that corrupted MS!), interpolate the gain solutions we got on the calibrator source onto the target source time grid, and then correct the MS for these gain fluctuations. (AIPS++ is writing the real interpolation code at this moment, but we can't wait, so we've got our own interpolation code.) If we are studying fast switching alone, we'd want to use '2pt' interpolation, but if we are looking into WVR, we should just take the 'nearest' interpolation method – by doing '2pt' interpolation of the gain solution, we would be messing up the WVR correction.

Or we could utilize the CAL source gain solutions to calibrate the WVR data, and apply the WVR variations to correct for the short term atmospheric fluctuations which spoil the coherence in between fast switching calibrations:

```
(dreamware)
mywvrreduce.applyfs(newms='TEST1.MS.FS', method='nearest')
mywvrreduce.calibratewvr('WVR.SCALING');
mywvrreduce.applywvr(oldms='TEST1.MS.FS', newms='TEST1.MS.WVR');
```

After various phase correction methods have been applied, we can investigate the nature of the residual phase fluctuations via a structure function analysis, or via a "time since last calibration" analysis:

```
(dreamware)
phasevtime := mywvrreduce.phasevtime(doplot=T);
for (i in ind(phasevtime.time)) {
       print phasevtime.time[i], phasevtime.phase[i];
}
```

As the application of WVR data to the visibilities is a topic of research, we expect the available methods to change and grow. For now, thats about it, so we can close the WVR reduction tool:

```
mywvrreduce.done();
```

Note that `mywvrsim` and `mywvrreduce` are instances of software tools which perform operations on the data. The basic data such as the MeasurementSet, gain tables, and atmospheric models, reside on disk in permanent tables, and have persistence even after the `wvrsim` or `wvrreduce` tools have gone away. We can create other `wvrsim` or `wvrreduce` tools to perform more operations if required, or we can access the data in the tables apart from these tools.

Appendices

A AIPS++ and WVR Simulation Package Installation

You need to install a recent version of AIPS++, and you need to download the WVR package.

Old versions of AIPS++, or very new versions of AIPS++ (ie, if you have a weekly inhale) will probably break the WVR simulation package. I've used the most recent stable end-user release. To get it, go to the AIPS++ web site `ftp://ftp.cv.nrao.edu/aips++/end-user/application/rh9` and download the required RPM's. There are about 10 RPM's there. The first two AIPS++ RPM's should have 19.615-26ds in their names. You probably need to get all of these RPM's, including the g77, gcc, and pgplot RPM's. After downloading the RPM's, login with root permission and do a `rpm -i *.rpm` to install AIPS++.

Next, fetch the WVR simulation code via anonymous ftp. You might want to do something along these lines:

```
mkdir wvrsims
cd wvrsims
ftp  ftp.tuc.nrao.edu
cd mholdawa
bin
get WVR.RELEASE.1.1.TAR.gz
quit
gunzip WVR.RELEASE.1.1.TAR.gz
tar -xvf WVR.RELEASE.1.1.TAR
aips++
include 'wvrsim.g'
wvrtest2()
```

B All the Glish Syntax You'll Need To Know

The Glish command line interpreter understands many kinds of statements, but we will only describe what we need to run the WVR simuations. Assignments of variables are designated by "`:=`". Most mathematical operations and typical mathematical functions are defined in Glish. There is an optional "`;`" at the end of each statement. Statements can continue onto subsequent lines if each line is ended with a symbol which implies more text is coming, such as a coma "`,`".

Probably the most useful sort of Glish statement is a function call. There are lots of predefined functions in Glish/AIPS++, and then we have to go and define some more. The general syntax is *functionname(parname=value)*. Multiple comma-separated parameters may be entered if they are defined. If a parameter is missing, the default value for that parameter will be used. Parameters need not be named if they are passed in order. The return value of the function will show up on the screen (often T or F). Alternatively, the return value could be captured in a variable. By the way, if you attempt to invoke a function with an improper parameter name, Glish will knock your socks off by filling the screen with what appears to be garbage – but upon closer examination, you can figure out from this garbage the correct parameter names for the attempted function.

A tool is a software construct or *object* with associated data and functions. In general, we can't see the data inside, but we can invoke the functions, which sometimes return bits of data that we might need to see. We create a tool of some sort by assigning the return value of its constructor to a variable name of our choosing. For example, we could create a `wvrsim` tool called `mywvrsim`:

```
mywvrsim := wvrsim();
```

After the wvrsim tool is created, we access its functions with the syntax *toolname.functionname(parname=value)*.

Glish is a loosely-typed language, and we usually don't have to declare variables to be of one type or another. In this package, parameter values can have types float (we need not distinguish between floats and ints), strings (delimited by a pair of single quotes), boolean (T or F), vectors of floats, or a flexible composite data structure called a record. Vectors are respresented as

```
vec1 := [2.5, 5, 10]
```

A vector with sequential values can be created as

```
vec2 := [1:10]
```

A vector's values (or any other variable's values) can be printed out at the command line just by typing the name of the variable, or with a `print` statement:

```
print 'Here are the vectors:', vec1, vec2
```

A multidimensional array is created and accessed as

```
initvalue := 0.0; nx := 4; ny := 4;
myarray := array(initvalue, nx, ny);
myarray[1,3] := 5
```

Records are defined by

```
myrec := [=]
```

The record can then be labeled and stuffed with data of varying types:

```
myrec.mstable := 'TEST1.MS';
myrec.elevation := 65;
myrec.array  := [1,2,3,10];
```

The values in a record can be accessed by label:

```
print 'The MeasurementSet named', myrec.mstable,
      'was observed at elevation', myrec.elevation;
```

You can make a `for` loop:

```
for (in in [1:10]) { print vec2[i]; }
```

or

```
for (in in ind(vec2)) { print vec2[i]; }
```

Two mathematical opperations that are a bit different from other languages are raising a number to a power and the square root. Raising an expression to a power is accomplished via

```
anumber := x^2
```

Taking a square root can be done as

```
bnumber := sqrt( x^2 + y^2 );
```

For a more comprehensive guide to Glish syntax and its built-in functions, browse through the *.g files included in the WVR simulation package, or go to the Glish user's manual at `http://aips2.nrao.edu/docs/reference/Glish`.

C Stand-alone Function `createms()`

A stand-alone function `createms()` is defined in the glish script `wvrsim.g`. The full list of parameters and their default values are shown here.

```
createms(antfile='ants/test_10.stn',  # input antenna configuration file
         msname='TEST.MS',  # name of output MeasurementSet table
         calflux=0.1,       # calibrator flux is low to have realistic gain errors
         targetflux=10,     # target flux is bright so we see above the noise
         tint='1s',         # integration time
         totaltime='300s',  # total integration time
         tcal=1,            # number of integrations on the cal source
         ttarget=20,        # 20 1s integrations on target
         freq='300GHz')     # observing frequency
```

`createms()` doesn't know anything about any atmospheres or anything, but it does create a MeasurementSet which contains observations of two different sources which we will call the cal and the target source. We control how long we sit on each one, and we go back and forth observing the two until the totaltime is exhausted. The integration time `tint` and the `totaltime` are given as strings with units (s, m, or h all work). Frequency is also a string. The fluxes are in units of Jy. Later on, we will add phase errors and thermal noise to the visibilities. As the thermal noise will limit our ability to clearly see the phase errors, we need to pick the fluxes of the cal and target sources wisely. The `calflux` needs to be realistic for a nearby calibrator so we can see the realistic effects of noise on the gain solutions. However, as we don't use the target source to calculate any gains, we are free to give it any flux we like. To make matters simple for viewing the phase errors, we make the target source a bright point source so that the atmospheric phase errors rise above the the thermal noise.

D Public Functions of the `wvrsim` Tool

The `wvrsim` tool is defined in the glish script `wvrsim.g`, which must be explicitly included. We list the complete set of `wvrsim` public functions, their parameter names, and the default values for the paramaters, along with some explanation about what these function do.

```
mywvrsim := wvrsim()
```

The constructor of the wvrsim tool. Returns a tool which can operate on data and has an internal state.

```
mywvrsim.type()
```

Returns the tool type ('wvrsim').

```
mywvrsim.getprivate()
```

This returns a reference to the private part of the wvrsim tool, and can be useful in debugging. Assuming all is fine, this tool shouldn't be used.

```
mywvrsim.done()
```

Closes the wvrsim tool.

```
mywvrsim.state(level=0)
```

Prints state information on the tool. This is useful when you want to understand more about how the tool works or what you've actually done, and may be useful for debugging. `level=0` provides the least information, `level=2` provides the most.

```
mywvrsim.getinfo()
```

Returns a record with the information that a `wvrreduce` tool will require to get going: rec.msname, rec.wvrgains, rec.targetel, rec.calel.

```
mywvrsim.setdata(msname='',
                 atmogainname='',
                 wvrgainname='WVR.GAINS')
```

Set up the MS table, ATMO gain table, and WVR gain table. The MS table must exist prior to invoking setdata(), but the atmogain and wvrgain tables are created by setdata().

```
mywvrsim.setobs(targetaz=60, targetel=80, calaz=62, calel=61)
```

Most of the observing details are set in the MS creation. However, we permit fudging on CAL, TARGET source AZ, EL. ALL ANGLES in DEGREES, but Glish trigonometric functions assume radians (if you use them).

```
mywvrsim.readatmosascii(basename='TEST_ATMOS/r0103.dg0007',
                        tablename='atmos.tab')
```

Reads in the ASCII files for the atmosphere (all ASCII files begin with "basename", and end with "_p3d" or "_x"....), and converts them to an AIPS++ table that can be more quickly read in the future (ie, invoke this function once, and then you can use `readatmostable()` from then on).

```
mywvrsim.readatmostable(tablename='', nx=1, ny=1)
```

After you have read in the atmospheric model files with `readatmosascii()` and written out the arrays as an AIPS++ table, you can read in the table to reconstitute the atmosphere. The parameters `nx` and `ny` permit you to "wallpaper" the model atmosphere as these arrays should have circular boundary conditions (ie, continuity from one side of the array to the other).

```
mywvrsim.rescaleatmos(pwv=1.0,  # zenith PWV in mm
                              stiphase=1.0)  # phase flucs, in degrees,
                                             # of site testing interferometer:
                                             # 300m baseline, 29 deg el, 11.2 GHz
```

This function scales the water vapor and the water vapor fluctuations so we can simulate a variety of atmospheric opacity and stability conditions from the same model atmosphere.

```
mywvrsim.squash(display=F)
```

This takes the 3D water vapor distribution and integrates it down to 2D and optionally displays the 2D image. Return the statistics of the integrated PWV as a record.

```
mywvrsim.pwvprofile(display=F, dopwv=T)
```

Calculate the mean PWV profile with height, show it if you want it (this mean profile is used by rescaleatmos()). If dopwv is T, we show the PWV integrated through each cell; if F, we divide by CELL HEIGHT and show Q in kg/m$\hat{3}$.

```
mywvrsim.sf2d()
```

Calculate a structure function from 2d array and return the data and power law fit in a record.

```
mywvrsim.displayatmos()
```

Display the 3-D atmosphere image. This will show one (x,y) plane at a time, but you can use the arrow buttons in the right to move through the planes or even automatically scan through all planes. As the different planes have different mean amounts of water vapor and the differences within each plane are small, you will need to adjust the display with the controls on the left, or with the "adjust" menu which is activated by the button on the lower left.

```
mywvrsim.checkatmos()
```

Check that the atmospheric model has been read into memory and all the arrays have the right size.

```
mywvrsim.suggestrefpos(velocity=[10,0,0], # atmos velocity vector [m/s]
                       refpix=[1,1,1],    # pixel at which to apply refpos
                       adopt=T)           # take the calculated refpos?
```

Our problem here: we've got a model atmosphere which is being blown over the array with some velocity, and we've got an antenna array on the ground, looking up at an angle through the atmosphere. We need to shift the coordinates of the atmosphere model so that all lines of sight from all antennas at all times during the observation stay within the atmospheric model. This function figures out how to shift the atmospheric coordinates.

The USER provides a `velocity` (in m/s) and `refpix` for the atmosphere, and `suggestrefpos()` will calculate a reference position such that the observations will fit, or will return F if the array or time interval are too big to fit. The `velocity` is important. The `refpix` is almost irrelevant (ie, using [1,1,1] will be fine essentially all the time). If `adopt` is T, the calculated reference position and the given `refpix` will be adopted.

```
mywvrsim.setatmocoordinates(velocity=[12,0,0],
                            refpix=[1,1,1],
                            refpos=[0,0,0])
```

Set the atmosphere's velocity and the coordinates of the reference pixel. `suggestrefpos()` is preferred because it is simple, but this gives the user more control.

If you need to use `setatmocoordinates()`, you should understand the details about the reference pixel and the reference position. The coordinates of a pixel (ix, iy) in the atmosphere will be given by

$$x = refpos[1] + (ix - refpix[1]) * imdelta - velocity[1] * time$$

$$y = refpos[2] + (iy - refpix[2]) * imdelta - velocity[2] * time,$$

where imdelta is the x (or y) size of each atmospheric cell (the cells are assumed to be square in (x,y); however, the fact that the z dimension of the cells changes with height makes it difficult to write down a similar expression for the z coordinate). Currently, the convention for the velocity is that a positive velocity blows from positive to negative (ie, East to West) – oops! Meant to do that the other way around! The values (x, y) can be directly compared to the antenna coordinates. In order to find which pixel of the atmosphere is above an antenna at some time, we invert these equations for (ix, iy).

```
mywvrsim.checkrefpos()
```

If the user provides the refpos and refpix via `setatmocoordinates()`, they should use this routine to quickly check that the observations will fit within the atmosphere.

```
mywvrsim.figurepath( antpos=[], # [x,y,z] of antenna
                     az=60,     # az [deg] of line of sight
                     el=60,     # el [deg] of line of sight
                     tt=0.0)    # Time [s] since start of observations
# This function returns a record with the path of the line of sight:
# rec.nn
# rec.pathindex == array(0, nn, 3);  i, ix, iy, iz
# rec.pathlength == array(0, nn) -- length through each cell
```

Figures out the path through the atmosphere that the line of sight takes. Generally, the user will not use this function, as `letitroll()` is the preferred method, calling `figurepath()` zillions of times.

```
mywvrsim.verify(pathrec=[=] )
```

We need to verify if the full pathrecord is within the model atmosphere. Returns a record which indicates what the offences are, and return.isok will be T or F as a summary judgement.

```
mywvrsim.integratepath1(pathrec=[=])
```

`integratepath1()` takes a pathrecord as calculated by `figurepath()` and then sends out the order to the atmospheric transmission program *am* to integrate the water vapor along the path to get the opacity and line brightness spectrum. This function returns the channel brightness temperatures as a vector: [ch1, ch2, ch3, ch4]. Generally, the user will not use this function, as `letitroll()` is the preferred method, calling `integratepath1()` zillions of times.

```
mywvrsim.integratepath2(pathrec=[=])
```

`integratepath2()` takes a pathrecord as calculated by `figurepath()` and integrates the Smith-Weintraub equation in house (as opposed to sending the job to *am*) to calculate the wet and dry delays along this path. Generally, the user will not use this function, as `letitroll()` is the preferred method, calling `integratepath2()` zillions of times.

```
mywvrsim.letitroll(verifyonly=F,  # ie, no actual calcs
                            dowvr=T,           # do the WVR calcs too?
                            delaytype='total') # ['wet'|'dry'|'total']
```

`letitroll()` blows the atmosphere over the site and calculates what the atmosphere is doing above each antenna for each integration. This function does not apply the atmospheric affects to the MeasurementSet data, but rather fills in the gain tables for the atmospheric delay and the WVR channel data. A weighted sum of the WVR channel data is also created, using weights from Richard Hills' memo (ALMA MEMO 495), taking the mean PWV of the atmosphere times the airmass to determine exactly which weights are to be used. As the weights are only tabulated for certain values of PWV, the weights are interpolated and renormalized. A single set of weights is used for the entire simulated observation. 'wet' and 'dry' delays can be studied separately or together via 'total'.

```
mywvrsim.addwvrnoise(noisescale=1.0)
```

The thermal noise is that given in Richard Hills' memo (ALMA Memo 495) (0.313, 0.144, 0.112, and 0.079 K in 1 s for the four channels), but the thermal noise can be adjusted with the noisescale parameter.

```
  mywvrsim.addwvrdrift(gainfluc=0.0002,   # fractional gain drift
                       gaintime=300,  # time scale for gain drift
      seed=12345,  # Random Number seed
                       fraccorr=0.0);   # fraction correlated for 4 chans[0-1]
```

This will modify the amplitude of the WVR channel data by a gain which is fluctuating about 1.0 with an rms of the fluctuating part given by `gainfluc` over a time `gaintime` seconds. If `fraccorr=0`, then the four WVR channels have uncorrelated gain fluctuations. If `fraccorr=1`, all four WVR channels will have the same 1/f gain fluctuations. Intermediate values will result in partial correlation:

$$G_i(t) = 1 + \sqrt{f}(g_0(t) - 1) + \sqrt{1-f}(g_i(t) - 1),$$

where $g_0(t)$ is a gain time series common to all channels, $g_i(t)$ is the gain time series of channel i which we would have used if there were no correlation, $G_i(t)$ is the gain time series we used, and f is the correlation fraction. This method maintains the mean and dispersion of the gain time series.

```
mywvrsim.corruptvis(override=F)
```

After we have calculated the atmospheric delay (or phase) above each antenna via `letitroll()`, `corruptvis()` corrupts the visibility phases by these phase errors. There is a state variable which prevents corrupting twice, unless override=T.

```
mywvrsim.addnoisetovis(noise='0.1Jy')
```

Add thermal noise to the visibilities in the MeasurementSet. Thermal noise will limit the accuracy of the gain solutions obtained from the calibrator. However, due to shortcomings in the gain solution algorithm, the algorithms sometimes fails to converge for noise-free visibilities. Noise as low as 0.000001 Jy effectively solves this problem.

E Public Functions of the `wvrreduce` Tool

The `wvrreduce` tool is defined in `wvrreduce.g`, which is automatically included by `wvrsim.g`.

```
mywvrreduce := wvrreduce()
```

This is how we construct a `wvrreduce` tool, assigning it to the variable mywvrreduce.

```
mywvrreduce.type()
```

This function provides the type, "wvrreduce".

```
mywvrreduce.getprivate()
```

A debugging tool which returns the private part of the `wvrreduce` tool so the private data can be seen and the private functions can be invoked.

```
mywvrreduce.done()
```

Close the `wvrreduce` tool and recycle the used resources.

```
mywvrreduce.state(level=0)
```

Prints state information on the tool. This is useful when you want to understand more about how the tool works or what you've actually done, and may be useful for debugging. Currently, `level` makes no difference, all info is printed out no matter what.

```
myrec := [=];
myrec.msname := 'TEST1.MS';
myrec.wvrgains := 'WVR.GAINS';
myrec.targetel := 65.0;
myrec.calel    := 66.5;
mywvrreduce.setdata(rec=myrec)
```

Set the data and basic information required to run the `wvrreduce` tool. The parameters are passed via a record because the function `wvrsim.getinfo()` returns an appropriate record which can be passed into `setdata()`.

```
mywvrreduce.addphase(tabname='')
```

The phases are usually hidden in the MeasurementsSet, as the visibilities are stored as a 4-element Stokes vector of complex numbers. This function will extract the phase information and put it in a column of the Table named 'PHASE'. Similarly, the phase is hidden in a gain table as the gains for each antenna are represented as a 2x2 complex Jones matrix. If the tabname refers to a gain table, the phase is similarly extracted and placed in a PHASE column. This is very important for debugging.

```
mywvrreduce.solveatmosgains(solvedgains='')
```

This function solves for the antenna phases on the CAL observations only, and puts those gain solutions into a table named `solvedgains`.

```
mywvrreduce.structurefunctionfromms(doplot=F, wtoo=F)
```

This function calculates the phase structure function from the MeasurementSet visibility phases and baseline lengths. If wtoo is T, then the full length of the physical baseline is used, otherwise the projected baseline is used. This function returns a record which contains the baseline length and phase vectors, and also the fit amplitude and exponent for

$$\sigma_\phi(\rho) = A * \rho^\alpha$$

where A is the amplitude and α is the exponent.

```
mywvrreduce.interpolatefs(gaintabout='',   # new gaintab
                    dtnew=1,          # time interval [s] for new gain tab
                    method='2pt'      # ['nearest'|'2pt']
```

This function interpolates the gain table for the cal source so that it can be applied to the target source. AIPS++ is writing similar code as we speak, but we couldn't wait for them, so we wrote it ourselves. NOTE that WE still need to apply these new gains to the MeasurementSet.

```
mywvrreduce.calibratewvr(outtab='TEST.WVRCAL')
```

This function compares the WVR data taken on the cal source with the phase data from the cal source each time we perform a cal observation, and solves for a scaling parameter $A_{i,j}(t)$:

$$-(\phi_j(t) - \phi_i(t)) = A_{i,j}(t)(T_j(t) - T_i(t))$$

where $\phi_j(t)$ is the solved gain phase for antenna j at time t, and $T_j(t)$ is the weighted WVR channel brightness temperature for antenna j at time t. The leading negative sign just indicates that the definition of the gain solutions is the inverse of the atmospheric corruption. By constructing the baseline phase from the phase solutions, we win significantly in SNR as compared to using the visibility phases. One issue which we need to address: if the phase on a particular baseline is very close to zero, then it is dominated by noise, and the scaling term $A_{i,j}$ ends up being greatly in error.

Hey, looking at this approach, I am realizing it may be the wrong way to proceed, as we'ce really like to have antenna-based scaling parameters:

$$-(\phi_j(t) - \phi_i(t)) = B_j(t) * T_j(t) - B_i(t) * T_i(t)$$

Well, thats why Alison is coming to visit Tucson in August.