So you want to fit a model to some ALMA data...

1. Introduction

This document is intended as an introduction to working with visibility data obtained by ALMA in the attempt to fit a model. The power of ALMA is that it has opened up radio astronomy to many people that have never used a radio telescope. The disadvantage is that radio astronomy datasets are not trivial compared to optical/infrared images and there is a bit of a learning curve to work with the visibilities.

This serves as a record of my (Kevin Flaherty) experience working with ALMA data for the first time. It is strongly biased toward the exact dataset for which I have been using (science verification data of the HD 163296 CO J=3-2 line) and the programming environment in which much of this work is done (idl but migrating to python). It passes over much of the functionality within CASA that can perform many of these tasks; there are many more useful documents for using CASA, along with the help desk. It also assumes some familiarity with the basics of radio interferometry. This document starts with a description of the format of an ALMA fits visibility file (in idl then in python) followed by guidelines for using MIRIAD to generate visibilities from a model image.

2. Data Format

The first thing to do is to understand the format of the ALMA data, which includes information about the configuration of the telescope (e.g. the uvw positions of the telescopes) as well as the actual visibility data. There are a number of different formats in which this data can be saved; casa uses something called measurement sets, while miriad uses a completely different format. Here we focus on visibility fits files (e.g. mydata.vis.fits or mydata.uvf). How this data is extracted from the visibility fits file depends on whether or not you are using idl or python. The description below mainly focuses on IDL, with a subsection on python afterwards.

The visibility fits files are stored in a format called random access groups, which is almost exclusively used in radio interferometry. Because of this limited use, they can only be read in by the function mrdfits (e.g. *image = mrdfits('mydata.vis.fits',0,hdr)*). This command tells IDL to load the visibilities into the variable *image* (from extension 0) and to read the header into the variable *hdr*.

The variable *image* is a structure with two parts. The first piece of this structure (*image.params*) contains u,v and w positions of the antennas for each baseline (along with other information). Extracting these (*u=image.params[0]*, *v=im.params[1]*) will give an array with length N$_{bas}$, where N$_{bas}$ is the number of baselines measured during the observations. Originally the uvw values are in units of seconds (as in, the number of seconds for light to travel to this position). Most often we want these antenna positions to be in units of kλ rather than seconds. To convert, simply multiply the uvw values by $\nu/1e3$ where ν is the frequency of the observation in Hz (if you don't know this off the top of your head you can check the header, usually the crval4 keyword, or search for the line in the online tool splatalogue).

Why do we prefer the antenna positions in units of kλ? Think about calculating the spatial resolution of an image. The formula is $\theta = 1.2\lambda/D$ where D is the diameter of the telescope. If D is in units of kλ, then the values of lambda cancel and the spatial resolution is simple to calculate. For example, the spatial resolution of two antennas separated by 400kλ is simply $\theta = 1.2/400e3$ radians.

As a simple excercise, lets plot the baselines on the uv plane.

$$\begin{aligned} &image = mrdfits('HD163296.CO32.vis.fits', 0, hdr) \\ &u = image.array[0] \\ &v = image.array[1] \\ &u* = sxpar(hdr, 'crval4')/1e3 \quad ;convert\ to\ units\ of\ klambda \\ &v* = sxpar(hdr, 'crval4')/1e3 \\ &plot, u, v, psym = 3 \end{aligned} \tag{1}$$

The visbility data itself (*image.array*) is more complicated. At each baseline, and each channel, there will be a visibility measurement. Since this visibility measurement is the fourier transform of the flux on the sky, there will be both a real and imaginary part, along with weights (= the inverse of the variance). For some of the ALMA data, the intensities will be split into XX and YY polarizations. This means our data is contained in an array with dimension 3x2xN$_{chan}$xN$_{bas}$ where 3 covers the real part of the visibilities, the imaginary part of the visibilities and their weights, the 2 covers XX and YY and N$_{chan}$ is the number of channels in our data. For example, to pull out the real part of YY in the 3rd channel on the 137th baseline, we would call *image.array[0,1,2,136]*, while the imaginary part of XX in the 48th channel and the 13047th baseline is *image.array[1,0,47,13046]*

At this point it is a good time to check your weights. Are they entirely 0? If so then there may have been some issue in extracting the weights from CASA. This can be remedied by re-running the CASA task for measuring weights, or by computing them afterwards based on the variance in the data. This is covered in section 2.3 in more detail.

ALMA data (at least the science verification data that I have been working with) comes in two polarizations XX, YY. Linear polarizations along perpendicular axes are straightforward to measure in a radio telescope (the simplest antenna would be a straight wire), which is why the data does not come in Q,U,V. There are also XY, YX polarized visibilities that one could measure, but they are not included. The important thing is that to convert from XX, YY to intensity use the formula I=(XX+YY)/2. Don't forget the factor of two! (e.g. *real=(image.array[0,0,*,*]+image.array[0,1,*,*])/2* and *imag = (image.array[1,0,*,*]+image.array[1,1,*,*])/2.*

If you have converted to intensity correctly then you should have a vector for the real part, and the imaginary part, of the visibility that has dimensions $N_{chan}xN_{bas}$ where N_{chan} is the number of frequency channels in your data. These can be converted to amplitude and phase in the standard way for imaginary numbers (amplitude = $(\text{real}^2+\text{imaginary}^2)^{1/2}$). Roughly speaking, amplitude corresponds to the flux while phase corresponds to position. For example, if you measure a large amplitude on large uv spacings then there is substantial flux on small spatial scales (keeping in mind that as uv spacing increases we are sampling information on smaller spatial scales).

A note about averaging to increase signal to noise. Infrared/optical astronomers are used to seeing an image where we can examine each individual pixel to see if its flux is significant compared to the noise. Do this with an individual visibility point and one might be discouraged, since unless the signal is very strong, the real and imaginary part will appear to be very consistent with noise. But the sheer number of baselines contained within a data set allows us to average over these real and imaginary parts and beat down the noise level.

If one were to plot the measured real (or imaginary) part of the visibility, you would see a near-perfect gaussian whose standard deviation is the uncertainty and a mean close to zero. Close, but not quite. Remember that the uncertainty in the mean is the standard deviation divided by the square root of the number of points; in the ALMA data the number of baselines can easily reach the millions which significantly reduces the uncertainty on the mean. This results in a mean that is statistically significantly higher than zero. This cumulative effect comes through in diagnostics, such as a chi-squared, that compare individual measurements with a model without the need for averaging.

If you did want to average your data to better visualize the amplitude and phase, keep in mind that you must average the real and imaginary parts of the visibility and use these values to calculate the amplitude and phase. Calculating the amplitude and phase at each baseline, and then averaging these values will artificially inflate the amplitude (since the amplitude is always positive while the real and imaginary parts can be both positive and negative).

Within ALMA data, there are also flagged visibilities which are baselines and channels for which there is no data. These will show up in your data as visibilities whose real and imaginary components are identically zero. Often, but not always, they are assigned a weight of zero. When performing calculations, such as averaging of the data or computing a chi-squared, make sure to remove these data points from your analysis or else you will skew your results.

As a simple example, lets calculate the average amplitude in the first channel within the inner

50kλ (assuming we have read in the file using the code snippet above)

$$\begin{array}{l}
real = reform((image.array[0,0,*,*] + image.array[0,1,*,*])/2) \quad ;I + (XX+YY)/2 \\
imag = reform((image.array[1,0,*,*] + image.array[1,1,*,*])/2) \\
w = where(sqrt(u*u + v*v)\ lt\ 50) \\
wuse = where(real[0,w]\ ne\ 0.\ and\ imag[0,w]\ ne\ 0.) \quad ;remove\ flagged\ visibilities \\
avg\_real = mean(real[0,w[wuse]]) \\
avg\_imag = mean(imag[0,w[wuse]]) \\
print,'The\ average\ amplitude\ is\ :', sqrt(avg\_real*avg\_real + avg\_imag*avg\_imag)
\end{array} \tag{2}$$

2.1. Python

Visibility fits files can be read into python, and in some ways this is cleaner, assuming you know the format of the data. The image can be opened in the standard way (*image = fits.open('mydata.vis.fits')*). Once loaded into python the data dn header can be accessed as elements within this image (*data=image[0].data* and *header = image[0].header*). The data array contains the same information as when read into IDL, but in a slightly different format. To access the uvw positions, directly ask for those elements in the *data* array (*u,v,w=data['UU'],data['VV'],data['WW']*). The visibility measurements are labeled 'data' (*vis = data['data']*) and have the same number of visibility measurements as when read into IDL, but the format may be slightly different. Use *data['data'].shape* to check what that format is. If you are curious about the names of the parameters within the data array, use *print data.parnames* toprint them to the screen. If a parameter name shows up twice, this means you are meant to add these two columns. Extracting elements from the header is done in the same manner as with normal fits files (e.g. *header['crval4']* returns the value of 'CRVAL4').

Plotting the uv plane in python:

$$\begin{array}{l}
from\ astropy.io\ import\ fits \\
import\ matplotlib.pyplot\ as\ plt \\
import\ numpy\ as\ np \\
image = fits.open('HD163296.CO32.vis.fits') \\
u = image[0].data['UU'] \\
v = image[0].data['VV'] \\
u* = image[0].header['crval4']/1e3 \\
v* = image[0].header['crval4']/1e3 \\
plt.plot(u,v,'.')
\end{array} \tag{3}$$

And calculating the average amplitude in the first channel among the visibilities within $50k\lambda$:

$$\begin{aligned}
&vis = image[0].data['data'] \\
&real = (vis[:,0,0,:,0,0] + vis[:,0,0,:,1,0])/2. \\
&imag = (vis[:,0,0,:,0,1] + vis[:,0,0,:,1,1])/2. \\
&inner = np.sqrt(u**2 + v**2)\ <\ 50 \\
&use = (real[inner,0]\ != \ 0.)\ \&\ (imag[inner,0]\ != \ 0.) \\
&print\ 'The\ average\ amplitude\ is:\ \%f'\ \% \\
&(np.sqrt(real[inner[use],0].mean()**2 + imag[inner[use],0].mean()**2))
\end{aligned} \tag{4}$$

2.2. Converting from Measurement Set to UV fits file

ALMA data does not originally arrive as a visibility fits file. Instead it is packaged together in what is called a Measurement Set. These files can be easily read and manipulated by the CASA routines[1], but are difficult to parse outside of this context. Luckily CASA has methods for converting a measurement set to a visibility fits file.

The main task for converting a measurement set to a visibility fits file is *exportuvfits*. This takes the name of a measurement set and visibility fits fiel as input and performs the conversion. It also includes an optional parameter for specifying a section of the ALMA data to be dumped into the fits file. ALMA data is taken over four spectral windows and here we can specify a range of channels within one of those spectral windows. This is useful is we only want to extract the spectra of a particular line, rather than the entire spectra. An example of a line of code that performs this task:

$$exportuvfits(vis=' mydata.ms', fitsfile=' mydata.vis.fits', spw=' 0:120\sim 160') \tag{5}$$

This line of code copies the ALMA data located in channels 120-160 of the first spectral window from the file mydata.ms to the visibility fits file mydata.vis.fits

2.3. Estimating weights

The weights, as the inverse of the variance in your data, are important in any method of comparing a model to data. Thus it is important to correctly estimate the weights within the ALMA data. Within casa this is done with the function *statwt*. This function estimates the variance in line-free regions of the spectra and assigns these values to the line-emission region. One example of a call to this function is:

$$statwt(vis=' mydata.ms', fispw=' 0:0\sim 21; 51\sim 74', spw=' 0:22\sim 50') \tag{6}$$

[1]There is extensive documentation covering CASA and its many functions. Here I am only going to cover the small handful of tasks that have come in handy. Refer to the online documentation for more details.

This calculates the variance using channels 0-21 and 51-74 within the first spectral window and applies the results to channels 22-50.

It is important to check that the weights calculated by *statwt* have been correctly propogated to the visbility fits file. There have been instances where this has not occured (see below if you have are plagued by this problem).

I have also had success estimating the weights on my own outside of CASA, by calculating the variance among the N nearest baselines. The value of N is choosen to be large enough to accurately estimate the variance (N>50). If the N points cover too large of an area in the uv plane then you could mistake real variations in the signal with uv spacing, or real variations across the array, for noise. For the ALMA data, these two constraints result in a sweet-spot of N$\sim$70.

To ensure that you have estimated the weights correctly, you can look at the data normalized to the standard deviation (=the inverse square root of the weight). If the weights are correct, the normalized data should be distributed like a gaussian with width of one. Deviations from this indicate that you have mis-estimated the variance at some point[2]. This is another useful point to make sure you have remembered all of your factors of two.

2.4. Exporting weights from CASA

On occasion, it has occured that weights correctly calculated by *statwt* are not propogated into the exported uv fits file. This will be evident when the weights are all identical for every single baseline. While frustrating, there are other ways to extract the weights from the measurements sets. In particular the *table* module allows you to read to weights directly from the measurement set, which then allows you to output them to a text file. An example of how to do this is:

$$\begin{aligned}
&tb.open('mydata.ms') \\
&weight\ =\ tb.getcol('WEIGHT') \\
&target\ =\ open('myweights.dat','w') \\
&for\ i\ in\ range(len(weight)): \\
&\quad target.write(str(weight[0,i])\ +\ '\ '\ +\ str(weight[1,i])\ +\ '\backslash n') \\
&target.close()
\end{aligned} \tag{7}$$

This sequence of commands reads in the measurement set, pulls out the weights, and then writes the weights to a text file. To get a more complete view of the data, use the command *tb.browse()* to browse through the full measurement set. As with other CASA tasks, there is more complete documentation available online.

[2]If the distribution is significantly non-gaussian then you might consider using a lorentzian or a laplacian for your likelihood function instead of a gaussian

The weights are now written into a text file, from which they can be read using your favorite routines. An example set of commands to read them into python is included below.

$$\begin{aligned}
&chain = np.recfromtxt('myweights.dat', names = True)\\
&weight = np.zeros(len(chain))\\
&for\ i\ in\ range(len(chain)):\\
&\quad weight[i] = (chain[i][0] + chain[i][1])/2\\
&weight = weight[:, np.newaxis] * np.ones(nchannels)
\end{aligned} \tag{8}$$

Note that statwt does not calculate frequency dependent weights; there is only one value for each visibility point. The last command in the above sequence is included to convert *weight* with a length N$_{bas}$ to a matrix with dimensions N$_{bas}$xN$_{chan}$, which is the structure of the real and imaginary parts of the data.

3. Generating a model visibility file

Having visibility data from your science target is only half of the equation when it comes to comparing to a model. The other half is the model itself. Presumably you are starting with a model image, maybe at one wavelength or at a series of channels. Turning this into a set of model visibilities, for comparison with the visibilities from the science target. Luckily miriad includes a set of procedures that can accomplish this goal (CASA also has a set of ALMA specific tasks that accomplish the same goal, but they are described in more detail elsewhere). The relevant tasks are:

- *fits* Convert a file back and forth between a fits format and the miriad visibility/image file format
- *gethd, puthd* Two functions that either get a value of a keyword from the header, or put a value into a header.
- *hanning* Perform hanning smoothing of the model image (necessary for spectral observations)
- *uvmodel* Calculate the visibilities for a model image given known antenna baselines. This is your workhorse function.

The *fits* function is both the first and last step in creating a model visibility file. It is used in the first step, with the *op=xyin* keyword set, to convert your model image from a fits image to the native miriad image format. It is used in the last step, with the *op=uvout* keyword set, to convert from the native mariad visibility file format to a fits visibility file.

This miriad format visibility files have all of the keywords that were within the fits file. The function *gethd* gives you the ability to retrieve the values of those keywords, while *puthd* lets you assign those keywords certain values. For example, to get the epoch for an observation use *gethd in=model.im/epoch.* To assign a value to the rest frequency use *puthd in=model.im/restfreq value=345.79599 type=double.* The model image should have keywords for the position of the source, the image size in RA and DEC as well as the rest frequency (if it is a data cube).

The *hanning* function performs hanning smoothing on your data cube. Hanning smoothing arises due to the nature of the measurement of a spectra within the context of radio interferometry. To see this, consider the data directly extracted from antenna A, which is a measure of the voltage as a function of time $V_A(t)$. Similarly the voltage from another antenna B would be called $V_B(t)$. The visibilities are a cross-correlation of these two voltages, $V_A \otimes V_B^*$ where the * refers to the complex conjugate. Taking a fourier transform of this cross-correlation returns the cross-correlation as a function of frequency (instead of time). Introducing time delays into the cross-correlation ($V_A(t) \otimes V_B^*(t-\tau)$), once passed through the fourier transform, correspond to measuring visibility as a function of frequency, ie. measuring the spectra. In practice there is some finite time, T, over which the voltages are measured and $V_A(t)$ is zero for $t<0$ and $t>T$. Taking the fourier transform of function that has sharp edges (in this case at t=0 and t=T) introduces spurious features in the resulting spectra. This will cause, for example, a line in the true spectra to be broadened and introduce oscillations in the spectra emanating outward from the spectral line. Hanning smoothing is introduced as a method to minimize this 'ringing' by introducing a weight on the voltage time series. The upshot of all of this is that your data will have undergone hanning smoothing before it arrives at your desk, and the same process should be applied to your model.

The last, and most important, function is *uvmodel.* This takes in your model images as input, along with a file containing information on the antenna positions, and outputs the visibilities. The file with the antenna positions can simply be your data visibilities; miriad can extract the information it needs from these files. Once this is complete you can use *fits* to convert the visibility output into a fits file and continue with your analysis.

A sample script putting all of this together would look like:

```
fits in = model.fits op = xyin out = model.im
   #Copy the epoch value from the data to the model
puthd in = model.im/epoch value =' gethd in = data.im/epoch'
hanning in = model.im out = model.han.im
uvmodel vis = data.vis model = model.han.im out = model.vis options = replace
fits in = model.vis op = uvout out = model.vis.fits
```

One important note! The function *uvmodel* does not know how to compute XX, YY from intensity data. If you give it a model that has XX, YY, plug a data file that also has XX, YY then it will split things up properly. If your model simpy has I, while your data has XX, YY then uvmodel will simply copy intensity values into the result. If, when running uvmodel, it says 'Polarisations copied: I' then it has created a fits file with dimensions $3 \mathrm{x} 1 \mathrm{x} 1 \mathrm{x} N_{chan} \mathrm{x} (2N_{bas})$ where every other element in the last dimension is a copy of the intensity. You can verify this fact by examining the values of u and v, which will be duplicated. If instead uvmodel returns the message

’Polarisations copied XX,YY then the fits file has dimensions 3x2xN_{chan}x(N_{bas}) but in place of XX and YY, it has used I. Keep this in mind when directly comparing the model and data within idl/python.