# AIPY Documentation

***Release 1.0.1***

**Aaron Parsons**

May 11, 2015

AIPY (Astronomical Interferometry in PYthon) collects together tools for radio astronomical interferometry. In addition to pure-python phasing, calibration, imaging, and deconvolution code, this package includes interfaces to MIRIAD (a Fortran interferometry package) and HEALPix (a package for representing spherical data sets), and some math/fitting routines from SciPy.

CHAPTER

ONE

# CONTENTS

## 1.1 Installation

### 1.1.1 Requirements

**The requirements for AIPY are:**

- numpy >= 1.2
- pyephem >= 3.7.2.3
- pyfits >= 1.1
- matplotlib >= 0.98 [1]
- matplotlib-basemap >= 0.99 [1]

### 1.1.2 Installing

Install AIPY by running:

```
python setup.py install [--prefix=<prefix>|--user]
```

If the '–prefix' option is not provided, then the installation tree root directory is the same as for the python interpreter used to run setup.py. For instance, if the python interpreter is in '/usr/local/bin/python', then <prefix> will be set to '/usr/local'. Otherwise, the explicit <prefix> value is taken from the command line option. The package will install files in the following locations:

- <prefix>/bin
- <prefix>/lib/python2.6/site-packages
- <prefix>/share/doc
- <prefix>/share/install

If an alternate <prefix> value is provided, you should set the PATH environment to include directory '<prefix>/bin' and the PYTHONPATH environment to include directory '<prefix>/lib/python2.6/site-packages'.

If the '–user' option is provided, then then installation tree root directory will be in the current user's home directory.

[1] Required for some fo the included scripts

### 1.1.3 Obtaining Catalogs

Most of the catalogs that AIPY provides an interface to, e.g. the Third Cambridge Catalog, needed to be downloaded from VizieR as tab-separated files and placed in the AIPY *_src* dirctory before they can be used. See the documentation for the `aipy.src` module for details.

## 1.2 General Utililties

### 1.2.1 Physical Constants

Module for keeping track of physical constants. All constants in cgs units. See description() for a dictionary of constants and their descriptions.

### 1.2.2 1-D Spline Interpolation

A pure-python, spline-like implementation of 1D interpolation. Uses an FIR filter (spline) to interpolate, and a polynomial to extrapolate boundaries.

Author: Aaron Parsons Date: 12/12/07 Revisions:

`aipy.interp.`**`interpolate`**(*ys*, *factor*, *filter=<function default_filter at 0x11233d848>*, *order=4*)
Oversample ys by the specified factor using a filter function to interpolated between samples. The filter function will be used to construct an FIR filter for x values [-order,order] in steps of 1/order. Order should be even to make this an averaging filter. Internally, ys is extended by 2nd degree polynomial in both directions to attempt a smooth boundary transition.

`aipy.interp.`**`default_filter`**(*xs*, *freq=0.25*)
A basic smoothing filter using a Hamming window and a sinc function of the specified frequency. Input a sample grid [-x,x] to get the FIR coefficients used for smoothing.

### 1.2.3 Coordinate System Conversions

An astronomy library for precessing coordinates between epochs and converting between topocentric (z = up, x = east), ecliptic (heliocentric), equatorial (celestial), and galactic coordinate systems. Vectors are 3 dimensional with magnitude 1, representing a point on the unit sphere. Includes generic 3 vector rotation code.

`aipy.coord.`**`azalt2top`**(*az_alt*)
Convert angles az (clockwise around z = up, 0 at x axis = north), alt (from horizon) into topocentric xyz vectors (x,y,z along first axis).

`aipy.coord.`**`convert`**(*crd*, *isys*, *osys*, *iepoch=36525.0*, *oepoch=36525.0*)
Convert 'crd' from coordinate system isys to osys, including epoch precession. Valid coordinate systems are 'ec' (Ecliptic), 'eq' (Equatorial), and 'ga' (Galactic). Epochs may be date strings, or numerical ephemeris times.

`aipy.coord.`**`convert_m`**(*isys*, *osys*, *iepoch=36525.0*, *oepoch=36525.0*)
Return the 3x3 matrix corresponding to a coordinate system/precssion transformation (see 'convert').

`aipy.coord.`**`eq2radec`**(*xyz*)
Convert equatorial xyz vectors (x,y,z along first axis) into angles ra (counter-clockwise around z = north, 0 at x axis), dec (from equator).

`aipy.coord.`**`eq2top_m`**(*ha*, *dec*)
Return the 3x3 matrix converting equatorial coordinates to topocentric at the given hour angle (ha) and declination (dec).

aipy.coord.**latlong2xyz**(*lat_long*)
Convert angles lat (from equator), long (counter-clockwise around z = north, 0 at x axis) into xyz vectors (x,y,z along first axis).

aipy.coord.**radec2eq**(*ra_dec*)
Convert angles ra (counter-clockwise around z = north, 0 at x axis), dec (from equator) into equatorial xyz vectors (x,y,z along first axis).

aipy.coord.**rot_m**(*ang*, *vec*)
Return 3x3 matrix defined by rotation by 'ang' around the axis 'vec', according to the right-hand rule. Both can be vectors, returning a vector of rotation matrices. Rotation matrix will have a scaling of |vec| (i.e. normalize |vec|=1 for a pure rotation).

aipy.coord.**thphi2xyz**(*th_phi*)
Convert angles theta (from z axis), phi (counter-clockwise around z, 0 at x axis) into xyz vectors (x,y,z along first axis).

aipy.coord.**top2azalt**(*xyz*)
Convert topocentric xyz vectors (x,y,z along first axis) into angles az (clockwise around z = up, 0 at x axis = north), alt (from horizon).

aipy.coord.**top2eq_m**(*ha*, *dec*)
Return the 3x3 matrix converting topocentric coordinates to equatorial at the given hour angle (ha) and declination (dec).

aipy.coord.**xyz2thphi**(*xyz*)
Convert xyz vectors (x,y,z along first axis) into angles theta (from z axis), phi (counter-clockwise around z, 0 at x axis).

### 1.2.4 Source Catalogs

This module provides a front-end interface for accessing sources in all catalogs in the _src module of AIPY.

aipy.src.**get_catalog**(*srcs=None, cutoff=None, catalogs=['helm', 'misc']*)
Return a source catalog created out of the sources listed in 'srcs', or with fluxes above the specified (jy_cutoff,freq_ghz) in 'cutoff'. Searches catalogs listed in 'catalogs'.

## 1.3 Interferometry Utilities

### 1.3.1 Array Geometry and Phasing

Module for representing antenna array geometry and for generating phasing information.

**class** aipy.phs.**Antenna**(*x, y, z, beam, phsoff=[0.0, 0.0], **kwargs*)
Representation of physical attributes of individual antenna.

**select_chans**(*active_chans=None*)
Select only the specified channels for use in future calculations.

**class** aipy.phs.**AntennaArray**(*location*, *ants*, ***kwargs*)
A collection of antennas, their spacings, and location/time of observations.

**bl2ij**(*bl*)
Convert Miriad's (i+1) << 8 | (j+1) baseline indexing scheme to i,j (0 indexed)

**bl_indices**(*auto=True*, *cross=True*)
Return bl indices for baselines in the array.

**gen_phs**(*src*, *i*, *j*, *mfreq=0.15*, *ionref=None*, *srcshape=None*, *resolve_src=False*)
Return phasing that is multiplied to data to point to src.

**gen_uvw**(*i*, *j*, *src='z'*, *w_only=False*)
Compute uvw coordinates of baseline relative to provided RadioBody, or 'z' for zenith uvw coordinates. If w_only is True, only w (instead of (u,v,w) will be returned).

**get_afreqs**()
Return array of frequencies that are active for simulation.

**get_baseline**(*i*, *j*, *src='z'*)
Return the baseline corresponding to i,j in various coordinate projections: src='e' for current equatorial, 'z' for zenith topocentric, 'r' for unrotated equatorial, or a RadioBody for projection toward that source.

**get_phs_offset**(*i*, *j*)
Return the frequency-dependent phase offset of baseline i,j.

**ij2bl**(*i*, *j*)
Convert baseline i,j (0 indexed) to Miriad's (i+1) << 8 | (j+1) indexing scheme.

**phs2src**(*data*, *src*, *i*, *j*, *mfreq=0.15*, *ionref=None*, *srcshape=None*)
Apply phasing to zenith-phased data to point to src.

**refract**(*u_sf*, *v_sf*, *mfreq=0.15*, *ionref=(0.0, 0.0)*)
Calibrate a frequency-dependent source offset by scaling measured offsets at a given frequency. Generates dw, a change in the projection of a baseline towards that source, which can be used to fix the computed phase of that source. ionref = (dra, ddec) where dra, ddec are angle offsets (in radians)

of sources along ra/dec axes at the specified mfreq.

**u_sf,v_sf = u,v components of baseline, used to compute the** change in w given angle offsets and the small angle approx. Should be numpy arrays with sources (s) along the 1st axis and freqs (f) along the 2nd.

**resolve_src**(*u*, *v*, *srcshape=(0, 0, 0)*)
Adjust amplitudes to reflect resolution effects for a uniform elliptical disk characterized by srcshape: srcshape = (a1,a2,th) where a1,a2 are angular sizes along the

semimajor, semiminor axes, and th is the angle (in radians) of the semimajor axis from E.

**select_chans**(*active_chans=None*)
Select which channels are used in computations. Default is all.

**unphs2src**(*data*, *src*, *i*, *j*, *mfreq=0.15*, *ionref=None*, *srcshape=None*)
Remove phasing from src-phased data to point to zenith.

**class** aipy.phs.**ArrayLocation**(*location*)
The location and time of an observation.

**get_jultime**()
Get current time as a Julian date.

**set_ephemtime**(*t=None*)
Set current time as derived from the ephem package. Recalculates matrix for projecting baselines into current positions.

**set_jultime**(*t=None*)
Set current time as a Julian date.

**class** aipy.phs.**Beam**(*freqs*, ***kwargs*)
Template for representing antenna beam pattern. Beams also hold info about which frequencies are active (i.e. Antennas and AntennaArrays access frequencies through Beam).

**select_chans**(*active_chans=None*)
Select only enumerated channels to use for future calculations.

**exception** aipy.phs.**PointingError**(*value*)
An error to throw if a source is below the horizon.

**class** aipy.phs.**RadioBody**(*name*, *mfreq*, *ionref*, *srcshape*)
The location of a celestial source.

**compute**(*observer*)
Update coordinates relative to the provided observer. Must be called at each time step before accessing information.

**get_crds**(*crdsys*, *ncrd=3*)
Return the coordinates of this location in the desired coordinate system ('eq','top') in the current epoch. If ncrd=2, angular coordinates (ra/dec or az/alt) are returned, and if ncrd=3, xyz coordinates are returned.

**class** aipy.phs.**RadioFixedBody**(*ra*, *dec*, *mfreq=0.15*, *name=''*, *epoch=36525.0*, *ionref=(0.0, 0.0)*, *srcshape=(0.0, 0.0, 0.0)*, ***kwargs*)
A source at fixed RA,DEC. Combines ephem.FixedBody with RadioBody.

**class** aipy.phs.**RadioSpecial**(*name*, *mfreq=0.15*, *ionref=(0.0, 0.0)*, *srcshape=(0.0, 0.0, 0.0)*, ***kwargs*)
A moving source (Sun,Moon,planets). Combines ephem versions of these objects with RadioBody.

**class** aipy.phs.**SrcCatalog**(**srcs*, ***kwargs*)
A catalog of celestial sources. Can be initialized with a list of src objects, of as an empty catalog.

**add_srcs**(**srcs*)
Add src object(s) (RadioFixedBody,RadioSpecial) to catalog.

**compute**(*observer*)
Call compute method of all objects in catalog.

**get**(*attribute*, *srcs=None*)
Return the specified source attribute (e.g. "mfreq" for src.mfreq) in an array for all src names in 'srcs'. If not provided, defaults to all srcs in catalog.

**get_crds**(*crdsys*, *ncrd=3*, *srcs=None*)
Return coordinates of all objects in catalog.

**get_srcs**(**srcs*)
Return list of all src objects in catalog.

aipy.phs.**ephem2juldate**(*num*)
Convert ephem date (measured from noon, Dec. 31, 1899) to Julian date.

aipy.phs.**juldate2ephem**(*num*)
Convert Julian date to ephem date, measured from noon, Dec. 31, 1899.

### 1.3.2 Simulation Support

Module adding simulation support to RadioBodys and AntennaArrays. Mostly, this means adding gain/amplitude information vs. frequency.

**class** aipy.amp.**Antenna**(*x, y, z, beam, phsoff=[0.0, 0.0], bp_r=array([1]), bp_i=array([0]), amp=1, pointing=(0.0, 1.5707963267948966, 0), **kwargs*)
Bases: aipy.phs.Antenna

Representation of physical location and beam pattern of individual antenna in array.

**bm_response**(*top*, *pol='x'*)
Return response of beam for specified polarization.

**set_pointing**(*az=0*, *alt=1.5707963267948966*, *twist=0*)
Set the antenna beam to point at (az, alt) with specified right-hand twist to polarizations. Polarization y is assumed to be +pi/2 azimuth from pol x.

**class** aipy.amp.**AntennaArray**(**args*, ***kwargs*)
Bases: aipy.phs.AntennaArray

Representation of location and time of observation, and response of array of antennas as function of pointing and frequency.

**bm_response**(*i*, *j*)
Return the beam response towards the cached source positions for baseline i,j with the specified polarization.

**passband**(*i*, *j*)
Return the passband response of baseline i,j.

**set_jultime**(*t=None*)
Set current time as a Julian date.

**sim**(*i*, *j*)
Simulate visibilites for the specified (i,j) baseline and polarization (set with AntennaArray.set_active_pol). sim_cache() must be called at each time step before this returns valid results.

**sim_cache**(*s_eqs*, *jys=array([ 1.])*, *mfreqs=0.15*, *ionrefs=(0.0*, *0.0)*, *srcshapes=(0*, *0*, *0)*)
Cache intermediate computations given catalog information to speed simulation for multiple baselines. For efficiency, should only be called once per time setting. MUST be called before sim(). s_eqs = array of equatorial vectors for all celestial sources jys = array of janskies vs. freq for all celestial sources mfreqs = array of frequencies where ionrefs were measured ionrefs = (dra,ddec), angular offsets in radians for ra/dec at the

frequency 'mfreq'.

**srcshapes = (a1,a2,th) where a1,a2 are angular sizes along the** semimajor, semiminor axes, and th is the angle (in radians) of the semimajor axis from E.

**class** aipy.amp.**Beam**(*freqs*, ***kwargs*)
Bases: aipy.phs.Beam

Representation of a flat (gain=1) antenna beam pattern.

**response**(*xyz*)
Return the (unity) beam response as a function of position.

**class** aipy.amp.**Beam2DGaussian**(*freqs*, *xwidth=inf*, *ywidth=inf*)
Bases: aipy.phs.Beam

Representation of a 2D Gaussian beam pattern, with default setting for a flat beam.

**response**(*xyz*)
Return beam response across active band for specified topocentric coordinates: (x=E,y=N,z=UP). x,y,z may be arrays of multiple coordinates. Returns 'x' linear polarization (rotate pi/2 for 'y').

**class** aipy.amp.**BeamAlm**(*freqs*, *lmax=8*, *mmax=8*, *deg=7*, *nside=64*, *coeffs={}*)
Bases: aipy.phs.Beam

Representation of a beam model where each pointing has a response defined as a polynomial in frequency, and the spatial distributions of these coefficients decomposed into spherical harmonics.

**response**(*top*)
Return beam response across active band for specified topocentric coordinates (x=E,y=N,z=UP). x,y,z may be multiple coordinates. Returns 'x' pol (rotate pi/2 for 'y').

**update**()
Update beam model using new set of coefficients. coeffs = dictionary of polynomial term (integer) and corresponding Alm coefficients (see healpix.py doc).

**class** aipy.amp.**BeamPolynomial**(*freqs*, *poly_azfreq=array([[ 0.5]])*)
Bases: aipy.phs.Beam

Representation of a gaussian beam model whose width varies with azimuth angle and with frequency.

**response**(*top*)
Return beam response across active band for specified topocentric coordinates (x=E,y=N,z=UP). x,y,z may be multiple coordinates. Returns 'x' pol (rotate pi/2 for 'y').

**select_chans**(*active_chans*)
Select only enumerated channels to use for future calculations.

**class** aipy.amp.**RadioBody**(*jys*, *index*)
Class defining flux and spectral index of a celestial source.

**get_jys**()
Return the fluxes vs. freq that should be used for simulation.

**update_jys**(*afreqs*)
Update fluxes relative to the provided observer. Must be called at each time step before accessing information.

**class** aipy.amp.**RadioFixedBody**(*ra*, *dec*, *name=''*, *epoch=36525.0*, *jys=0.0*, *index=-1*, *mfreq=0.15*, *ionref=(0.0, 0.0)*, *srcshape=(0.0, 0.0, 0.0)*, ***kwargs*)
Bases: aipy.phs.RadioFixedBody, aipy.amp.RadioBody

Class representing a source at fixed RA,DEC. Adds flux information to phs.RadioFixedBody.

**class** aipy.amp.**RadioSpecial**(*name*, *jys=0.0*, *index=-1.0*, *mfreq=0.15*, *ionref=(0.0, 0.0)*, *srcshape=(0.0, 0.0, 0.0)*, ***kwargs*)
Bases: aipy.phs.RadioSpecial, aipy.amp.RadioBody

Class representing moving sources (Sun,Moon,planets). Adds flux information to phs.RadioSpecial.

**class** aipy.amp.**SrcCatalog**(**srcs*, ***kwargs*)
Bases: aipy.phs.SrcCatalog

Class for holding a catalog of celestial sources.

**get_jys**(*srcs=None*)
Return list of fluxes of all src objects in catalog.

### 1.3.3 Simulation Fitting

Module for reading and setting parameters in components of an AntennaArray simulation for purpose of fitting.

**class** aipy.fit.**Antenna**(*x, y, z, beam, phsoff=[0.0, 0.0], bp_r=array([1]), bp_i=array([0]), amp=1, pointing=(0.0, 1.5707963267948966, 0), **kwargs*)
Bases: aipy.amp.Antenna

Representation of physical location and beam pattern of individual antenna in array. Adds get_params() and set_params() to amp.Antenna.

**get_params**(*prm_list=['*']*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**AntennaArray**(**args*, ***kwargs*)
Bases: aipy.amp.AntennaArray

Representation of location and time of observation, and response of array of antennas as function of pointing and frequency. Adds get_params() and set_params() to amp.AntennaArray.

**get_params**(*ant_prms={'*': '*'}*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**Beam**(*freqs*, ***kwargs*)
Bases: aipy.amp.Beam

Representation of a flat (gain=1) antenna beam pattern.

**class** aipy.fit.**Beam2DGaussian**(*freqs*, *xwidth=inf*, *ywidth=inf*)
Bases: aipy.amp.Beam2DGaussian

Representation of a 2D Gaussian beam pattern, with default setting for a flat beam.

**get_params**(*prm_list=['*']*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**BeamAlm**(*freqs*, *lmax=8*, *mmax=8*, *deg=7*, *nside=64*, *coeffs={}*)
Bases: aipy.amp.BeamAlm

Representation of a beam model where each pointing has a response defined as a polynomial in frequency, and the spatial distributions of these coefficients decomposed into spherical harmonics.

**get_params**(*prm_list=['*']*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**BeamPolynomial**(*freqs*, *poly_azfreq=array([[ 0.5]])*)
Bases: aipy.amp.BeamPolynomial

Representation of a gaussian beam model whose width varies with azimuth angle and with frequency.

**get_params**(*prm_list=['*']*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**RadioFixedBody**(*ra*, *dec*, *name=''*, *epoch=36525.0*, *jys=0.0*, *index=-1*, *mfreq=0.15*, *ionref=(0.0, 0.0)*, *srcshape=(0.0, 0.0, 0.0)*, ***kwargs*)
Bases: aipy.amp.RadioFixedBody

Class representing a source at fixed RA,DEC. Adds get_params() and set_params() to amp.RadioFixedBody.

**get_params**(*prm_list=['*']*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**RadioSpecial**(*name*, *jys=0.0*, *index=-1.0*, *mfreq=0.15*, *ionref=(0.0, 0.0)*, *srcshape=(0.0, 0.0, 0.0)*, ***kwargs*)
Bases: aipy.amp.RadioSpecial

Class representing moving sources (Sun,Moon,planets). Adds get_params() and set_params() to amp.RadioSpecial.

**get_params**(*prm_list=['*']*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

**class** aipy.fit.**SrcCatalog**(**srcs*, ***kwargs*)
Bases: aipy.amp.SrcCatalog

Class for holding a catalog of celestial sources. Adds get_params() and set_params() to amp.SrcCatalog.

**get_params**(*src_prms={'*': '*'}*)
Return all fitable parameters in a dictionary.

**set_params**(*prms*)
Set all parameters from a dictionary.

aipy.fit.**flatten_prms**(*prms*, *prm_list=None*)
Generate list of parameters suitable for passing to fitting algorithm from heirarchical parameter dictionary 'prms', along with 'key_list' information for reconstructing such a dictionary from a list. 'prm_list' is only for recursion.

aipy.fit.**print_params**(*prms*, *indent=''*, *grad=None*)
Print nice looking representation of a parameter dictionary.

aipy.fit.**reconstruct_prms**(*prm_list*, *key_list*)
Generate a heirarchical parameter dictionary from parameter list (prm_list) and 'key_list' information from flatten_prms.

### 1.3.4 RFI Detection and Flagging

Module for detecting and flaging RFI related effects.

aipy.rfi.**fit_gaussian**(*xs*, *ys*)
Fit a gaussian (returning amplitude, sigma, offset) to a set of x values and their corresponding y values.

aipy.rfi.**flag_by_int**(*preflagged_auto*, *nsig=1*, *raw=False*)
Flag rfi for an autocorrelation. Iteratively removes outliers and fits a smooth passband, then uses smooth passband to remove outliers (both positive and negative). If 'raw' is specified, no smooth function is removed.

aipy.rfi.**gaussian**(*amp*, *sig*, *off*, *x*)
Generate gaussian value at x given amplitude, sigma, and x offset.

aipy.rfi.**gen_rfi_thresh**(*data*, *nsig=2*, *cnt_per_bin=1000*)
Generate a threshold at nsig times the standard deviation (above,below) the mean, outside of which data is considered rfi.

aipy.rfi.**remove_spikes**(*data*, *mask=None*, *order=6*, *iter=3*, *return_poly=False*)
Iteratively fits a smooth function by removing outliers and fitting a polynomial, then using the polynomial to remove other outliers.

## 1.4 Imaging

### 1.4.1 Gridding and Imaging

Module for gridding UVW data (including W projection), forming images, and combining (mosaicing) images into spherical maps.

**class** aipy.img.**Img**(*size=100*, *res=1*, *mf_order=0*)

Class for gridding uv data, recording the synthesized beam profile, and performing transforms into image domain.

**append_hermitian**((*u*, *v*, *w)*, *data*, *wgts=None*)

Append to (uvw, data, [wgts]) the points (-uvw, conj(data), [wgts]). This is standard practice to get a real-valued image.

**bm_image**(*center=(0*, *0)*, *term=None*)

Return the inverse FFT of the sample weightings (for all mf_order terms, or the specified term if supplied), with the 0,0 point moved to 'center'. Tranposes to put up=North, right=East.

**get**((*u*, *v*, *w)*, *uv=None*, *bm=None*)

Generate data as would be observed at the provided (u,v,w) based on this Img's current uv data. Phase due to 'w' will be applied to data before returning.

**get_LM**(*center=(0*, *0)*)

Get the (l,m) image coordinates for an inverted UV matrix.

**get_eq**(*ra=0*, *dec=0*, *center=(0*, *0)*)

Return the equatorial coordinates of each pixel in the image, assuming the image is centered on the provided ra, dec (in radians).

**get_indices**(*u*, *v*)

Get the pixel indices corresponding to the provided uv coordinates.

**get_top**(*center=(0*, *0)*)

Return the topocentric coordinates of each pixel in the image.

**get_uv**()

Return the u,v indices of the pixels in the uv matrix.

**image**(*center=(0*, *0)*)

Return the inverse FFT of the UV matrix, with the 0,0 point moved to 'center'. Tranposes to put up=North, right=East.

**put**((*u*, *v*, *w)*, *data*, *wgts=None*, *apply=True*)

Grid uv data (w is ignored) onto a UV plane. Data should already have the phase due to w removed. Assumes the Hermitian conjugate data is in uvw already (i.e. the conjugate points are not placed for you). If wgts are not supplied, default is 1 (normal weighting). If apply is false, returns uv and bm data without applying it do the internally stored matrices.

**class** aipy.img.**ImgW**(*size=100*, *res=1*, *wres=0.5*, *mf_order=0*)

A subclass of Img adding W projection functionality (see Cornwell et al. 2005 "Widefield Imaging Problems in Radio Astronomy").

**conv_invker**(*u*, *v*, *w*)

Generates the W projection kernel (a function of u,v) for the supplied value of w. See Cornwell et al. 2005 "Widefield Imaging Problems in Radio Astronomy" for discussion. This implementation uses a numerically evaluated Fresnel kernel, rather than the small-angle approximated one given in the literature.

**put** (*(u, v, w)*, *data*, *wgts=None*, *invker2=None*)
Same as Img.put, only now the w component is projected to the w=0 plane before applying the data to the UV matrix.

aipy.img.**convolve2d** (*a*, *b*)
Convolve a and b by multiplying in Fourier domain. Must be same size.

aipy.img.**find_axis** (*phdu*, *name*)
Find the axis number for RA

aipy.img.**from_fits** (*filename*)
Read (data,kwds) from a FITS file. Matches to_fits() above. Attempts to deduce each keyword listed in to_fits() from the FITS header, but is accepting of differences. Returns values in "kwds" dictionary.

aipy.img.**from_fits_to_fits** (*infile*, *outfile*, *data*, *kwds*, *history=None*)
Create a fits file in outfile with data using header from infile using kwds to override values in the header. See img.to_fits for typical header variables.

aipy.img.**gaussian_beam** (*sigma*, *shape=0*, *amp=1.0*, *center=(0, 0)*)
Return a 2D gaussian. Normalized to area under curve = 'amp'. Down by 1/e at distance 'sigma' from 'center'.

aipy.img.**recenter** (*a*, *c*)
Slide the (0,0) point of matrix a to a new location tuple c. This is useful for making an image centered on your screen after performing an inverse fft of uv data.

aipy.img.**to_fits** (*filename*, *data*, *clobber=False*, *axes=('ra–sin', 'dec–sin')*, *object=''*, *telescope=''*, *instrument=''*, *observer=''*, *origin='AIPY'*, *obs_date='05/11/15'*, *cur_date='05/11/15'*, *ra=0*, *dec=0*, *d_ra=0*, *d_dec=0*, *epoch=2000.0*, *freq=0*, *d_freq=0*, *bscale=0*, *bzero=0*, *history=''*)
Write image data to a FITS file. Follows convention of VLA image headers. "axes" describes dimensions of "data" provided. (ra,dec) are the degree coordinates of image center in the specified "epoch". (d_ra,d_dec) are approximate pixel-deltas for ra,dec (approximate because if sine projections of these coordinates are specified– e.g. "ra—sin"–then the deltas change away from the image center). If a "freq" axis is specified, then "freq" is the frequency of the first entry (in Hz), and "d_freq" is the width of the channel. The rest are pretty self-explanitory/can be used however you want.

aipy.img.**word_wrap** (*string*, *width=80*, *ind1=0*, *ind2=0*, *prefix=''*)

**word wrapping function.** string: the string to wrap width: the column number to wrap at prefix: prefix of each line with thie string (goes before any indentation) ind1: number of characters to indent the first line ind2: number of characters to indent the rest of the lines

### 1.4.2 Deconvolution

A module implementing various techniques for deconvolving an image by a kernel. Currently implemented are Clean, Least-Squares, Maximum Entropy, and Annealing. Standard parameters to these functions are: im = image to be deconvolved. ker = kernel to deconvolve by (must be same size as im). mdl = a priori model of what the deconvolved image should look like. maxiter = maximum number of iterations performed before terminating. tol = termination criterion, lower being more optimized. verbose = print info on how things are progressing. lower = lower bound of pixel values in deconvolved image upper = upper bound of pixel values in deconvolved image

aipy.deconv.**anneal** (*im*, *ker*, *mdl=None*, *maxiter=1000*, *lower=2.2250738585072014e-308*, *upper=inf*, *cooling=<function <lambda> at 0x1117125f0>*, *verbose=False*)
Annealing takes a non-deterministic approach to deconvolution by randomly perturbing the model and selecting perturbations that improve the residual. By slowly reducing the temperature of the perturbations, annealing attempts to settle into a global minimum. Annealing is slower than lsq for a known gradient, but is less sensitive to gradient errors (it can solve for wider kernels). Faster cooling speeds terminate more quickly, but are less likely

to find the global minimum. This implementation assigns a temperature to each pixel proportional to the magnitude of the residual in that pixel and the global cooling speed. cooling: A function accepting (iteration,residuals) that returns a

vector of standard deviation for noise in the respective pixels. Picking the scaling of this function correctly is vital for annealing to work.

aipy.deconv.**clean**(*im*, *ker*, *mdl=None*, *area=None*, *gain=0.1*, *maxiter=10000*, *tol=0.001*, *stop_if_div=True*, *verbose=False*, *pos_def=False*)

This standard Hoegbom clean deconvolution algorithm operates on the assumption that the image is composed of point sources. This makes it a poor choice for images with distributed flux. In each iteration, a point is added to the model at the location of the maximum residual, with a fraction (specified by 'gain') of the magnitude. The convolution of that point is removed from the residual, and the process repeats. Termination happens after 'maxiter' iterations, or when the clean loops starts increasing the magnitude of the residual. This implementation can handle 1 and 2 dimensional data that is real valued or complex. gain: The fraction of a residual used in each iteration. If this is too

low, clean takes unnecessarily long. If it is too high, clean does a poor job of deconvolving.

aipy.deconv.**lsq**(*im*, *ker*, *mdl=None*, *area=None*, *gain=0.1*, *tol=0.001*, *maxiter=200*, *lower=2.2250738585072014e-308*, *upper=inf*, *verbose=False*)

This simple least-square fitting procedure for deconvolving an image saves computing by assuming a diagonal pixel-pixel gradient of the fit. In essence, this assumes that the convolution kernel is a delta-function. This works for small kernels, but not so well for large ones. See Cornwell and Evans, 1984 "A Simple Maximum Entropy Deconvolution Algorithm" for more information about this approximation. Unlike maximum entropy, lsq makes no promises about maximizing smoothness, but needs no information about noise levels. Structure can be introduced for which there is no evidence in the original image. Termination happens when the fractional score change is less than 'tol' between iterations. gain: The fraction of the step size (calculated from the gradient) taken

in each iteration. If this is too low, the fit takes unnecessarily long. If it is too high, the fit process can oscillate.

aipy.deconv.**maxent**(*im*, *ker*, *var0*, *mdl=None*, *gain=0.1*, *tol=0.001*, *maxiter=200*, *lower=2.2250738585072014e-308*, *upper=inf*, *verbose=False*)

Maximum entropy deconvolution (MEM) (see Cornwell and Evans 1984 "A Simple Maximum Entropy Deconvolution Algorithm" and Sault 1990 "A Modification of the Cornwell and Evans Maximum Entropy Algorithm") is similar to lsq, but the fit is only optimized to within the specified variance (var0) and then "smoothness" is maximized. This has several desirable effects including uniqueness of solution, equal weighting of Fourier components, and absence of spurious structure. The same delta-kernel approximation (see lsq) is made here. var0: The estimated variance (noise power) in the image. If none is

provided, a quick lsq is used to estimate the variance of the residual.

**gain: The fraction of the step size (calculated from the gradient) taken** in each iteration. If this is too low, the fit takes unnecessarily long. If it is too high, the fit process can oscillate.

aipy.deconv.**maxent_findvar**(*im*, *ker*, *var=None*, *f_var0=0.6*, *mdl=None*, *gain=0.1*, *tol=0.001*, *maxiter=200*, *lower=2.2250738585072014e-308*, *upper=inf*, *verbose=False*, *maxiterok=False*)

This frontend to maxent tries to find a variance for which maxent will converge. If the starting variance (var) is not specified, it will be estimated as a fraction (f_var0) of the variance of the residual of a lsq deconvolution, and then a search algorithm tests an ever-widening range around that value. This function will search until it succeeds.

aipy.deconv.**recenter**(*a*, *c*)

Slide the (0,0) point of matrix a to a new location tuple c.

### 1.4.3 Full Sky Mapping

Module for mapping and modeling the entire sky.

aipy.map.**facet_centers**(*N*, *ncrd=2*)
Return the coordinates of N points equally spaced around the sphere. Will return xyz or ra,dec depending on ncrd. Shuffles the order of the pointing centers wrt pack_sphere so that widely spaced points are done first, and then the gaps between them, and then the gaps between those..

aipy.map.**pack_sphere**(*N*)
Clever formula for putting N points nearly equally spaced on the sphere. Return xyz coordinates of each point in order from S to N.

## 1.5 Miscellaneous

### 1.5.1 MIRIAD Support

A package for interfacing to Miriad: the Multichannel Image Reconstruction Image Analysis and Display package for reducing interferometric data for radio telescopes.

**class** aipy.miriad.**UV**(*filename*, *status='old'*, *corrmode='r'*)
Top-level interface to a Miriad UV data set.

**add_var**(*name*, *type*)
Add a variable of the specified type to a UV file.

**all**(*raw=False*)
Provide an iterator over preamble, data. Allows constructs like: for preamble, data in uv.all(): ...

**init_from_uv**(*uv*, *override={}*, *exclude=[]*)
Initialize header items and variables from another UV. Those in override will be overwritten by override[k], and tracking will be turned off (meaning they will not be updated in pipe()). Those in exclude are omitted completely.

**items**()
Return a list of available header items.

**pipe**(*uv*, *mfunc=<function echo at 0x111700d70>*, *append2hist=''*, *raw=False*)
Pipe in data from another UV through the function mfunc(uv,preamble,data), which should return (preamble,data). If mfunc is not provided, the dataset will just be cloned, and if the returned data is None, it will be omitted. The string 'append2hist' will be appended to history.

**read**(*raw=False*)
Return the next data record. Calling this function causes vars to change to reflect the record which this function returns. 'raw' causes data and flags to be returned seperately.

**select**(*name*, *n1*, *n2*, *include=1*)
Choose which data are returned by read(). name This can be: 'decimate','time','antennae','visibility',

'uvrange','pointing','amplitude','window','or','dra', 'ddec','uvnrange','increment','ra','dec','and', 'clear', 'on','polarization','shadow','auto','dazim','delev'

**n1,n2 Generally this is the range of values to select. For** 'antennae', this is the two antennae pair to select (indexed from 0); a -1 indicates 'all antennae'. For 'decimate', n1 is every Nth integration to use, and n2 is which integration within a block of N to use. For 'shadow', a zero indicates use 'antdiam' variable. For 'on','window','polarization','increment','shadow' only p1 is used. For 'and','or','clear','auto' p1 and p2 are ignored.

**include If true, the data is selected. If false, the data is** discarded. Ignored for 'and','or','clear'.

**vars**()
Return a list of available variables.

**write**(*preamble*, *data*, *flags=None*)
Write the next data record. data must be a complex, masked array. preamble must be (uvw, t, (i,j)), where uvw is an array of u,v,w, t is the Julian date, and (i,j) is an antenna pair.

### 1.5.2 HEALpix Support

Provides interfaces to Healpix_cxx, which was developed at the Max-Planck-Institut fuer Astrophysik and financially supported by the Deutsches Zentrum fuer Luft- und Raumfahrt (DLR). Adds data to the HealpixBase class using numpy arrays, and interfaces to FITS files using pyfits.

**class** aipy.healpix.**HealpixMap**(**args*, ***kwargs*)
Collection of utilities for mapping data on a sphere. Adds a data map to the infrastructure in _healpix.HealpixBase.

**change_scheme**(*scheme*)
Reorder the pixels in map to be "RING" or "NEST" ordering.

**from_alm**(*alm*)
Set data to the map generated by the spherical harmonic coefficients contained in alm.

**from_fits**(*filename*, *hdunum=1*, *colnum=0*)
Read a HealpixMap from the specified location in a fits file.

**from_hpm**(*hpm*)
Initialize this HealpixMap with data from another. Takes care of upgrading or downgrading the resolution, and swaps ordering scheme if necessary.

**get_map**()
Return Healpix data as a 1 dimensional numpy array.

**set_interpol**(*onoff*)
Choose whether __getitem___ (i.e. HealpixMap[crd]) returns an interpolated value or just nearest pixel. Default upon creating a HealpixMap is to not interpolate.

**set_map**(*data*, *scheme='RING'*)
Assign data to HealpixMap.map. Infers Nside from # of pixels via Npix = 12 * Nside**2.

**to_alm**(*lmax*, *mmax*, *iter=1*)
Return an Alm object containing the spherical harmonic components of a map (in RING mode) up to the specified lmax,mmax. Greater accuracy can be achieved by increasing iter.

**to_fits**(*filename*, *format=None*, *clobber=True*)
Write a HealpixMap to a fits file in the fits format specified by 'format'. Default uses mapping of numpy types to fits format types stored in default_fits_format_codes.

### 1.5.3 Scripting Support

Module containing utilities (like parsing of certain command-line arguments) for writing scripts.

aipy.scripting.**add_standard_options**(*optparser*, *ant=False*, *pol=False*, *chan=False*, *cal=False*, *src=False*, *prms=False*, *dec=False*, *cmap=False*, *max=False*, *drng=False*)
Add standard command-line options to an optparse.OptionParser() on an opt in basis (i.e. specify =True for

each option to be added).

aipy.scripting.**parse_ants**(*ant_str*, *nants*)
Generate list of (baseline, include, pol) tuples based on parsing of the string associated with the 'ants' command-line option.

aipy.scripting.**parse_chans**(*chan_str*, *nchan*, *concat=True*)
Return array of active channels based on number of channels and string argument for chans (all, 20_30, or 55,56,57, or 20_30,31,32). Channel ranges include endpoints (i.e. 20_30 includes both 20 and 30).

aipy.scripting.**parse_prms**(*prm_str*)
Return a dict of the form: {'obj': {'prm':(val,sig),...}...} where val is a starting value and sig is a known error associated with that start value. Both default to None if a value is not provided. The string to be parsed can be: "obj=prm", "obj=prm/val", "obj=prm/val/sig", "(obj1/obj2)=prm/(val1/val2)/sig", "obj=(prm1/prm2)/val/(sig1/sig2)", comma separated versions of the above, and so on.

aipy.scripting.**parse_srcs**(*src_str*, *cat_str*)
Return (src_list,flux_cutoff,catalogs) based on string argument for src and cat. Can be "all", "<src_name1>,...", "<ra XX[:XX:xx]>_<dec XX[:XX:xx]>", or "val/freq" (sources with Jy flux density above val at freq in GHz).

aipy.scripting.**uv_selector**(*uv*, *ants=-1*, *pol_str=-1*)
Call uv.select with appropriate options based on string argument for antennas (can be 'all', 'auto', 'cross', '0,1,2', or '0_1,0_2') and string for polarization ('xx','yy','xy','yx').

## 1.6 Included Scripts

### 1.6.1 General Utilities

**lst**

**Description** Output LST, Julian date, and Sun position at the provided location

**Usage** lst jd1 [jd2 [...]]

**Options** None

**compress_uv.py**

**Description** Tarball and compress (using bz2) Miriad UV files.

**Usage** compress_uv.py [options] file.uv

**Options**

| | |
|---|---|
| **-h, --help** | show this help message and exit |
| **-d, --delete** | Delete a uv file after compressing it |
| **-x, --expand** | Inflate tar.bz2 files |

**combine_freqs.py**

**Description** A script for reducing the number of channels in a UV data set by coherently adding adjacent channels together.

**Usage** combine_freqs.py [options] file.uv

**Options**

| | |
|---|---|
| **-h, --help** | Show this help message and exit |

**-n NCHAN, --nchan=NCHAN** Reduce the number of channels in a spectrum to this number.

**-c, --careful_flag** Flag resultant bin if any component bins are flagged (otherwise, flags only when there are more flagged bins than unflagged bins).

**-d, --dont_flag** Only flag resultant bin if every component bin is flagged (otherwise, uses any data available).

**-u, --unify** Output to a single UV file.

### 1.6.2 Calibration

**apply_bp.py**

**Description** Apply the bandpass function in a UV file to the raw data, and then write to a new UV file which will not have a bandpass file. Has the (recommended) option of linearizing for quantization gain effects. For now, these are applied only to the auto-correlations. Cross-correlation quantization gain is less sensitive to level, so linearization will come later.

**Usage** apply_bp.py [options] file.uv

**Options**

**-h, --help** show this help message and exit

**-l LINEARIZATION, --linearization=LINEARIZATION** Apply the specified quantization linearization function to raw correlator values before applying bandpass. Options are null, digi, full, and comb. Default is comb

**-s SCALE, --scale=SCALE** An additional numerical scaling to apply to the data. Default: 12250000.

**flux_cal.py**

**Description** A script for dividing out the passband, primary beam, and/or source spectrum scaling. When dividing by a primary beam or source spectrum, it is recommended a single source have been isolated in the data set.

**Usage** flux_cal.py [options] file.uv

**Options**

**-h, --help** show this help message and exit

**-C CAL, --cal=CAL** Use specified <cal>.py for calibration information.

**-s SRC, --src=SRC** Phase centers/source catalog entries to use. Options are “all”, “<src_name1>,...”, or “<ra XX[:XX:xx]>_<dec XX[:XX:xx]>”.

**--cat=CAT** A comma-delimited list of catalogs from which sources are to be drawn. Default is “helm,misc”. Other available catalogs are listed under aipy._src. Some catalogs may require a separate data file to be downloaded and installed.

**-b, --beam** Normalize by the primary beam response in the direction of the specified source.

**-p, --passband** Normalize by the passband response.

**-f, --srcflux** Normalize by the spectrum of the specified source.

### 1.6.3 Modeling

**mdlvis.py**

**Description** Models visibilities for various catalog sources and creates a new Miriad UV file containing either the simulated data, or the residual when the model is removed from measured data.

**Usage** mdlvis.py [options] file.uv

**Options**

**-h, --help** show this help message and exit

**-C CAL, --cal=CAL** Use specified <cal>.py for calibration information.

**-s SRC, --src=SRC** Phase centers/source catalog entries to use. Options are "all", "<src_name1>,...", or "<ra XX[:XX:xx]>_<dec XX[:XX:xx]>".

**--cat=CAT** A comma-delimited list of catalogs from which sources are to be drawn. Default is "helm,misc". Other available catalogs are listed under aipy._src. Some catalogs may require a separate data file to be downloaded and installed.

**-m MODE, --mode=MODE** Operation mode. Can be "sim" (output simulated data), "sub" (subtract from input data), or "add" (add to input data). Default is "sim"

**-f, --flag** If outputting a simulated data set, mimic the data flagging of the original dataset.

**-n NOISELEV, --noiselev=NOISELEV** RMS amplitude of noise (Jy) added to each UV sample of simulation.

**--nchan=NCHAN** Number of channels in simulated data if no input data to mimic. Default is 256

**--sfreq=SFREQ** Start frequency (GHz) in simulated data if no input data to mimic. Default is 0.075

**--sdf=SDF** Channel spacing (GHz) in simulated data if no input data to mimic. Default is .150/256

**--inttime=INTTIME** Integration time (s) in simulated data if no input data to mimic. Default is 10

**--startjd=STARTJD** Julian Date to start observation if no input data to mimic. Default is 2454600

**--endjd=ENDJD** Julian Date to end observation if no input data to mimic. Default is 2454601

**--pol=POL** Polarizations to simulate (xx,yy,xy,yx) if starting file from scratch.

**filter_src.py**

**Description** A script for filtering using a delay/delay-rate transform. If a source is specified, will remove/extract that source. If none is specified, will filter/extract in absolute terms.

**Usage** filter_src.py [options] file.uv

**Options**

| | |
|---|---|
| **-h, --help** | show this help message and exit |
| **-C CAL, --cal=CAL** | Use specified <cal>.py for calibration information. |
| **-s SRC, --src=SRC** | Phase centers/source catalog entries to use. Options are "all", "<src_name1>,...", or "<ra XX[:XX:xx]>_<dec XX[:XX:xx]>". |
| **--cat=CAT** | A comma-delimited list of catalogs from which sources are to be drawn. Default is "helm,misc". Other available catalogs are listed under aipy._src. Some catalogs may require a separate data file to be downloaded and installed. |
| **-r DRW, --drw=DRW** | The number of delay-rate bins to null. Default is -1 = no fringe filtering. |
| **-d DW, --dw=DW** | The number of delay bins to null. If -1, uses baseline lengths to generate a sky-pass filter. |
| **-p, --passband** | Divide by the model passband before transforming. |
| **-e, --extract** | Extract the source instead of removing it. |
| **--clean=CLEAN** | Deconvolve delay-domain data by the response that results from flagged data. Specify a tolerance for termination (usually 1e-2 or 1e-3). |

**fitmdl.py**

**Description** A script for fitting parameters of a measurement equation given starting parameters in a cal file and a list of sources. The fitter used here is a steepest-decent filter and does not make use of priors.

**Usage** fitmdl.py [options] file.uv

**Options**

| | |
|---|---|
| **-h, --help** | show this help message and exit |
| **-a ANT, --ant=ANT** | Select ants/baselines to include. Examples: all (all baselines) auto (of active baselines, only i=j) cross (only i!=j) 0,1,2 (any baseline involving listed ants) 0_2,0_3 (only listed baselines) "(0,1)_(2,3)" (same as 0_2,0_3,1_2,1_3. Quotes help bash deal with parentheses) "(-0,1)_(2,-3)" (exclude 0_2,0_3,1_3 include 1_2). Default is "cross". |
| **-p POL, --pol=POL** | Choose polarization (xx, yy, xy, yx) to include. |
| **-c CHAN, --chan=CHAN** | Select channels (after any delay/delay-rate transforms) to include. Examples: all (all channels), 0_10 (channels from 0 to 10, including 0 and 10) 0_10_2 (channels from 0 to 10, counting by 2), 0,10,20_30 (mix of individual channels and ranges). Default is "all". |
| **-C CAL, --cal=CAL** | Use specified <cal>.py for calibration information. |
| **-s SRC, --src=SRC** | Phase centers/source catalog entries to use. Options are "all", "<src_name1>,...", or "<ra XX[:XX:xx]>_<dec XX[:XX:xx]>". |

**--cat=CAT** A comma-delimited list of catalogs from which sources are to be drawn. Default is "helm,misc". Other available catalogs are listed under aipy._src. Some catalogs may require a separate data file to be downloaded and installed.

**-P PRMS, --prms=PRMS** Parameters (for fitting, usually), can be specified as can be: "obj=prm", "obj=prm/val", "obj=prm/val/sig", "(obj1/obj2)=prm/(val1/val2)/sig", "obj=(prm1/prm2)/val/(sig1/sig2)", comma separated versions of the above, and so on.

**-x DECIMATE, --decimate=DECIMATE** Use only every Nth integration. Default is 1.

**--dphs=DECPHS** Offset to use when decimating (i.e. start counting integrations at this number for the purpose of decimation). Default is 0.

**-S SHPRMS, --shared_prms=SHPRMS** Parameter listing with the same syntax as "-P/–prms" except that all objects listed for a parameter will share an instance of that parameter.

**--snap** Snapshot mode. Fits parameters separately for each integration.

**-q, --quiet** Be less verbose.

**--maxiter=MAXITER** Maximum # of iterations to run. Default is infinite.

**--xtol=XTOL** Fractional change sought in it parameters before convergence. Default 1e-10.

**--ftol=FTOL** Fractional tolerance sought in score before convergence. Default 1e-10.

**--remem** Remember values from last fit when fitting in snapshot mode.

**--baseport=BASEPORT** Base port # to use for tx/rx. Each daemon adds it's daemon id to this to determine the actual port used for TCP transactions.

**--daemon=DAEMON** Operate in daemon mode, opening a TCP Server to handle requests on the specified increment to the base port.

**--master=MASTER** Operate in master mode, employing daemon-mode servers to do the work and collecting the results. Should be a comma delimited list of host:daemonid pairs to contact. Daemon ID will be added to baseport to determine actual port used for TCP transactions.

**--sim_autos** Use auto-correlations in fitting. Default is to use only cross-correlations.

### 1.6.4 Imaging

**mk_imag.py**

**Description** This is a general-purpose script for making images from MIRIAD UV files. Data (optionally selected for baseline, channel) are read from the file, phased to a provided position, normalized for passband/primary beam effects, gridded to a UV matrix, and imaged.

**Usage** mk_img.py [options] file.uv

**Options**

**-h, --help** show this help message and exit

**-a ANT, --ant=ANT** Select ants/baselines to include. Examples: all (all baselines) auto (of active baselines, only i=j) cross (only i!=j) 0,1,2 (any baseline involving listed ants) 0_2,0_3 (only listed baselines) "(0,1)_(2,3)" (same as 0_2,0_3,1_2,1_3. Quotes help bash deal with parentheses) "(-0,1)_(2,-3)" (exclude 0_2,0_3,1_3 include 1_2). Default is "cross".

**-p POL, --pol=POL** Choose polarization (xx, yy, xy, yx) to include.

**-c CHAN, --chan=CHAN** Select channels (after any delay/delay-rate transforms) to include. Examples: all (all channels), 0_10 (channels from 0 to 10, including 0 and 10) 0_10_2 (channels from 0 to 10, counting by 2), 0,10,20_30 (mix of individual channels and ranges). Default is "all".

**-C CAL, --cal=CAL** Use specified <cal>.py for calibration information.

**-s SRC, --src=SRC** Phase centers/source catalog entries to use. Options are "all", "<src_name1>,...", or "<ra XX[:XX:xx]>_<dec XX[:XX:xx]>".

**--cat=CAT** A comma-delimited list of catalogs from which sources are to be drawn. Default is "helm,misc". Other available catalogs are listed under aipy._src. Some catalogs may require a separate data file to be downloaded and installed.

**-x DECIMATE, --decimate=DECIMATE** Use only every Nth integration. Default is 1.

**--dphs=DECPHS** Offset to use when decimating (i.e. start counting integrations at this number for the purpose of decimation). Default is 0.

**-o OUTPUT, --output=OUTPUT** Comma delimited list of data to generate FITS files for. Can be: dim (dirty image), dbm (dirty beam), uvs (uv sampling), or bms (beam sampling). Default is dim,dbm.

**--list_facets** List the coordinates of all the pointings that will be used.

**--facets=FACETS** If no src is provided, facet the sphere into this many pointings for making a map. Default 200.

**--snap=SNAP** Number of integrations to use in "snapshot" images. Default is to not do snapshoting (i.e. all integrations go into one image).

**--cnt=CNT** Start counting output images from this number. Default 0.

**--fmt=FMT** A format string for counting successive images written to files. Default is im%04d (i.e. im0001).

**--skip_phs** Do not phase visibilities before gridding.

**--skip_amp** Do not use amplitude information to normalize visibilities.

**--skip_bm** Do not weight visibilities by the strength of the primary beam.

**--skip=SKIP** Skip this many pointings before starting. Useful in conjungtion with –cnt for resuming.

**--size=SIZE** Size of maximum UV baseline.

**--res=RES** Resolution of UV matrix.

**--no_w** Don't use W projection.

**--altmin=ALTMIN** Minimum allowed altitude for pointing, in degrees. When the phase center is lower than this altitude, data is omitted. Default is 0.

**--minuv=MINUV** Minimum distance from the origin in the UV plane (in wavelengths) for a baseline to be included. Default is 0.

**--buf_thresh=BUF_THRESH** Maximum amount of data to buffer before gridding. Excessive gridding takes performance hit, but if buffer exceeds memory available... ouch

**cl_img.py**

**Description** This is a general-purpose script for deconvolving dirty images by a corresponding PSF to produce a clean image.

**Usage** cl_img.py [options] file.dim.fits file.dbm.fits

**Options**

**-h, --help** show this help message and exit

**-d DECONV, --deconv=DECONV** Attempt to deconvolve the dirty image by the dirty beam using the specified deconvolver (none,mem,lsq,cln,ann).

**-o OUTPUT, --output=OUTPUT** Comma delimited list of data to generate FITS files for. Can be: cim (clean image), rim (residual image), or bim (best image = clean + residuals). Default is bim.

**--var=VAR** Starting guess for variance in maximum entropy fit (defaults to variance of dirty image.

**--tol=TOL** Tolerance for successful deconvolution. For annealing, interpreted as cooling speed.

**--div** Allow clean to diverge (i.e. allow residual score to increase)

**-r REWGT, --rewgt=REWGT** Reweighting to apply to dim/dbm data before cleaning. Options are: natural, uniform(LEVEL), or radial, where LEVEL is the fractional cutoff for using uniform weighting (recommended range .01 to .1). Default is natural.

**--maxiter=MAXITER** Number of allowable iterations per deconvolve attempt.

CHAPTER

TWO

# TUTORIAL

## 2.1 Miriad UV Files

### 2.1.1 Reading an Existing File

A high-level interface to Miriad UV files is available through the `aipy.miriad.UV` class. We'll dive right in to reading a file, but first, so that we're working on the same UV file, run the following from bash (1 MB download):

```
wget http://setiathome.berkeley.edu/~aparsons/aipy/test.uv.tar.bz2
compress_uv.py -x test.uv.tar.bz2
```

You should now have a *test.uv* file in your current directory. Try the following from Python:

```
>>> import aipy
>>> uv = aipy.miriad.UV('test.uv')
>>> print uv.items()
['vartable', 'obstype', 'history']
>>> print uv['history']
C2M (Python): Version=0.1.1.Fixed bandpass inversion & ordering, and pol
label.APPLY_BP: version=0.0.1, corr type = combXRFI: version 0.0.2XTALK2:
version 0.0.1 Miniaturized...
>>> print uv.vars()
['latitud', 'npol', 'nspect', 'obsdec', 'vsource', 'ischan', 'operator',
'nants', 'baseline', 'sfreq', 'inttime', 'source', 'epoch', 'version',
'ra', 'restfreq', 'nschan', 'sdf', 'corr', 'freq', 'longitu', 'nchan',
'tscale', 'antpos', 'telescop', 'pol', 'coord', 'veldop', 'lst', 'time',
'dec', 'obsra']
>>> print uv['nchan']
64
>>> print uv['antpos']
[ -8.48 205.47 187.1 -262.7 455.28 319.53 -352.95 -219.07 9.82
-251.71 -232.59 318.7 ]
```

First, we made an instance of a UV file (which defaulted to read an "old" file). We could then ask for a list of (header) items. Each of these header items can then be accessed just as if uv were a dictionary holding them. The details of data types are taken care of automagically; strings return as Python strings, shorts and ints return as Python integers, floats and doubles return as Python floats. Similarly, we can ask for a list of variables (vars), and these are accessible as if uv were a dictionary. As you can see, when we are reading, there's no real need to differentiate between a header item and a variable, and so both are accessible via the same interface. When we access "nchan", which has a single value, we get that value. When we access "antpos", which has multiple values, we get an array of the values.

**Warning:** Do not access the "corr" variable. It is evil, and I don't know why Miriad makes it available.

Let's continue:

```
>>> preamble, data = uv.read()
>>> print preamble
(array([ 0., 0., 0.]), 2454302.8700115741, (0, 0))
>>> print data
[(3.55898427963+0j) (5.16037225723+0j) (7.65382957458+0j)
(11.5349502563+0j) (17.6214637756+0j) (26.8085384369+0j)
(40.0749702454+0j) (56.860118866+0j) (74.8811569214+0j) (89.6064910889+0j)
(98.601524353+0j) (101.491455078+0j) (100.617973328+0j) (98.0315933228+0j)
(95.0735092163+0j) (92.583152771+0j) (90.0556259155+0j) (88.2838745117+0j)
(86.3324737549+0j) (84.3934631348+0j) (82.3522338867+0j) --
(77.4334640503+0j) (74.7851333618+0j) (71.8084716797+0j)
(68.7729568481+0j) (65.6971817017+0j) (62.5315704346+0j) (59.719078064+0j)
(56.9530410767+0j) (54.4193191528+0j) (52.1953392029+0j)
(50.2718162537+0j) (48.5867958069+0j) (47.1000137329+0j)
(45.9260749817+0j) (44.9503746033+0j) (44.1512298584+0j) (43.609172821+0j)
(43.2684516907+0j) (43.1135787964+0j) (42.8874664307+0j)
(42.9587059021+0j) (43.0020713806+0j) (43.1228713989+0j)
(43.1600418091+0j) (43.1321640015+0j) (43.1135787964+0j)
(43.0020713806+0j) (42.726398468+0j) (42.5312576294+0j) (42.1409759521+0j)
(41.6794548035+0j) (41.0073051453+0j) (40.369228363+0j) (39.5948638916+0j)
(38.8019142151+0j) (38.0523262024+0j) (37.1168937683+0j)
(36.1814575195+0j) (35.2924880981+0j) (34.3105926514+0j)
(33.4278144836+0j) (32.3839683533+0j)]
>>> print uv['pol'], aipy.miriad.pol2str[uv['pol']]
-6 yy
>>> preamble, data = uv.read()
>>> print preamble
(array([-538.6, 298.79613781, -674.73816035]), 2454302.8700115741, (1, 3))
>>> preamble, data, flags = uv.read(raw=True)
```

We read a preamble/data pair. The preamble contained (uvw, time, (i,j)), and the data was a masked, complex Numpy array. One of the entries was masked (see the "-"), and did not print. If you count, there are 64 entries, just like "nchan" suggested. This was an autocorrelation (0,0), so the data has no complex component, and the uvw corrdinates are 0. Next, we showed how to access and convert the polarization codes. Following another read(), we find that we have a new preamble for a different baseline, but for the same time. Finally, we demonstrate the option "raw" for uv.read, which returns data and flags as separate arrays, and executes substantially faster.

Now for some more advanced stuff:

```
>>> uv.rewind()
>>> uv.select('antennae', 0, 1, include=True)
>>> for preamble, data in uv.all():
... uvw, t, (i,j) = preamble
... print i, j, t
...
0 1 2454302.87001
0 1 2454302.87009
0 1 2454302.87017
0 1 2454302.87025
[snip]
0 1 2454302.91135
0 1 2454302.91144
0 1 2454302.91152
0 1 2454302.9116
```

First, we rewound the uv file so that we could start from the beginning again. Next, we demonstrated one usage of select()-a function that allows you to choose which data you receive via read(). In this case, we've selected to only

include data that involve antennae 0 and 1 together. We set up a loop over all the data in uv, split up the preamble into its components, and print the antennae in the baseline and the time of integration. Note that (*i,j*) == (0; 1), thanks to select(), and we step through all the integrations in the file. Inside the loop, you can access variables as they change. Just like uv.read(), uv.all() has a "raw" operative that returns data and flags separately.

### 2.1.2 Initializing and Piping From Another UV File

Quite often, you will find yourself wanting to read in a UV file, operate on the data, and then write the data out to a new file. Building a UV file from scratch is a pain, but copying one shouldn't be. Starting with a fresh console, here's how to do it:

```
>>> import aipy
>>> uvi = aipy.miriad.UV('test.uv')
>>> uvo = aipy.miriad.UV('new1.uv', status='new')
>>> uvo.init_from_uv(uvi)
>>> def conjugate_01(uv, preamble, data):
... uvw, t, (i,j) = preamble
... if i == 0 and j == 1: return preamble, data.conjugate()
... else: return preamble, data
...
>>> uvo.pipe(uvi, mfunc=conjugate_01, append2hist="Conjugated (0,1)\n")
>>> del(uvo)
```

We opened up test.uv, and started a new file new.uv. Then we initialized the new file with the old one (all the header items and initial variables got copied across) with init from uv(). Next, we defined a "mapping function" or "mfunc". This is a function that accepts a (uv, preamble, data), and returns a (preamble, data). The idea is that this function operates on the preamble and data from the old file (along with a reference to the old file so you can access any variables and items you need), and returns the preamble and data for the new file. In our case, we have defined a function that conjugates the (0,1) baseline. We piped the data from uvi through mfunc to uvo, and append a string onto the history of the file. Just like read() and all(), pipe() accepts the keyword "raw" to pass data and flags as separate arrays into mfunc. In this case, your mfunc should be defined to accept arguments (uv, preamble, data, flags).

At the end of the above code snippet, we deleted uvo. This is normally not necessary, but some interactive consoles to not properly destruct their variables, and thus improperly close a new UV file. When you are writing scripts, there is no need to delete.

Now suppose you want a new file that relabels the "pol" variable for all integrations, and removes the variables "ra" and "lst" (which happen to have incorrect values in this file). Continuing where we left off:

```
>>> uvi = aipy.miriad.UV('new1.uv')
>>> uvo = aipy.miriad.UV('new2.uv', status='new')
>>> uvo.init_from_uv(uvi, override={'pol':-7}, exclude=['ra','lst'])
>>> uvo.pipe(uvi)
>>> del(uvo)
```

This time, we passed override and exclude parameters to init from uv(). "Override" is a dictionary of variables (or items) and new values, and variables (or items) in "exclude" are omitted from the output file. If you wish to rewrite a variable on an integration-by- integration basis, place that variable with some value in override, and then use uvo to set the variable inside a mapping function:

```
>>> uvi = aipy.miriad.UV('new2.uv')
>>> uvo = aipy.miriad.UV('new3.uv', status='new')
>>> def change_pol(uv, p, d):
... uvw, t, (i,j) = p
... if i == j: uvo['pol'] = -5
... else: uvo['pol'] = -6
... return p, d
```

```
...
>>> uvo.init_from_uv(uvi, override={'pol':-7})
>>> uvo.pipe(uvi, mfunc=change_pol)
>>> del(uvo)
```

By placing the variable in override, it is prevented from being automatically updated every time its value changes in the input file. We are then free to set that variable from the mapping function and have the changes stick. Based on the data type of the variable you are writing, you are expected to provide an appropriately typed Python variable. An error will get thrown when you don't.

### 2.1.3 Writing a UV File from Scratch

So far, we've been able to sweep details about data types under the rug. This is because, for a written UV file, we can infer the data types of all the items (which are statically typed in the Miriad User Guide) and variables (which are specified in the vartable header item). However, when we are writing a file from scratch, we can't do this for variables anymore. Because the data types of header items are spelled out ahead of time (in `aipy.miriad.itemtable`, if you're wondering), we only need to do this for variables.

Miriad Data Types:

| | |
|---|---|
| a | ascii (NULL terminated) |
| r | real (32 bin IEEE) |
| d | double (64 bit) |
| c | complex (2 * 32 bit IEEE) |
| i | integer (32 bit two's complement) |
| j | short (16 bit two's complement) |

Python wrapper routines use strings to code data types, according to the Miriad convention (see above, or `aipy.miriad.data` types). The following illustrates how to write a file from scratch:

```
>>> import aipy, numpy
>>> uv = aipy.miriad.UV('newest.uv', 'new')
>>> uv['history'] = 'Made this file from scratch.\n'
>>> uv.add_var('nchan', 'i')
>>> uv.add_var('pol', 'i')
>>> uv['nchan'] = 4
>>> uv['pol'] = -5
>>> uvw = numpy.array([1,2,3], dtype=numpy.double)
>>> preamble = (uvw, 12345.6789, (0,1))
>>> data = numpy.ma.array([1j,2,3j,4], mask=[0,0,1,0], dtype=numpy.complex64)
>>> uv.write(preamble,data)
>>> uv['pol'] = -6
>>> uv.write(preamble,data)
>>> del(uv)
>>> uv = aipy.miriad.UV('newest.uv')
>>> for p, d in uv.all():
... print p, uv['pol']
... print d
...
(array([ 1., 2., 3.]), 12345.678900000001, (0, 1)) -5
[1j (2+0j) -- (4+0j)]
(array([ 1., 2., 3.]), 12345.678900000001, (0, 1)) -6
[1j (2+0j) -- (4+0j)]
```

After creating a new file and demonstrating that we can immediately write to a header item like "history", we added 2 new variables: "nchan" and "pol". Strictly speaking, a UV file doesn't have to have "nchan", but the Python wrapper uses this value to effciently create an array exactly large enough to hold the data, so it's a good variable to include. Both

"nchan" and "pol" are integers. Although you are free to make any variable any type you want, there are conventions to follow if you want to use routines that are part of the Miriad package. These conventions are in the back of the Users Guide mentioned in x3. After writing the first values of these variables, we construct a preamble and a datum (paying careful attention to the types needed for each). Between spectra, we flipped "pol". After finishing the file, we close it and the open it for reading. Looks like it works!

## 2.2 Working with Antenna Arrays

`aipy.phs.AntennaArray` inherits from `aipy.phs.ArrayLocation` so that youcan pass it to the compute() of a `aipy.phs.RadioBody` or `aipy.phs.SrcCatalog`. Secondly, an AntennaArray is initialized with a list of `aipy.phs.Antenna` instances, and those contain `aipy.phs.Beam` instances, so an AntennaArray has all the information needed to figure out phasing. Let's learn by example, starting from scratch:

```
>>> import aipy, numpy
>>> freqs = numpy.array([.150,.160,.170])
>>> beam = aipy.phs.Beam(freqs)
>>> ants = []
>>> ants.append(aipy.phs.Antenna(0,0,0,beam,delay=0,offset=0))
>>> ants.append(aipy.phs.Antenna(0,100,0,beam,delay=1))
>>> ants.append(aipy.phs.Antenna(100,0,0,beam,offset=.5))
>>> aa = aipy.phs.AntennaArray(ants=ants,location=("18:20:39","-66:45:10"))
>>> print aa.get_baseline(0,2,'r'), aa.get_phs_offset(0,2)
[ 100. 0. 0.] [ 0.  0.  0.]
>>> aa.set_jultime(2454447.37472)
>>> srcs = []
>>> srcs.append(aipy.phs.RadioSpecial("Sun"))
>>> srcs.append(aipy.phs.RadioSpecial("Venus"))
>>> cat = aipy.phs.SrcCatalog(srcs)
>>> cat.compute(aa) # REMEMBER to call this before projecting!
>>> print aa.get_baseline(0,1,src=cat['Sun'])
[ 34.6664193  -36.79755778 -86.27965644]
>>> print aa.get_baseline(0,1,src=cat['Venus'])
aipy.phs.PointingError: Venus below horizon
```

We made a Beam with frequency information, created 3 Antennas using the same Beam, and then made an AntennaArray at Arecibo with those Antennas. We showed how we can access baselines and offsets, obeying sign conventions depending whether you specify (0,1) or (1,0). We made a SrcCatalog containing the Sun and Venus, and then we computed their locations relative to the AntennaArray. It is important to always call compute() before proceeding to other processing. At best you will get an error. At worst, you could end up with old positions. Finally, we retrieve baseline (0,1) projected towards the Sun, and then try to do the same towards Venus. However, Venus is below the horizon, and rather than let you use a projection that will give incorrect results, AIPY throws a PointingError. If you want, you can catch this exception (`aipy.phs.PointingError`) and recover.

---

**Note:** The coordinate returned here are still in nanoseconds and are not yet proper uvw coordinates.

---

Let's continue:

```
>>> print aa.gen_phs(cat['Sun'], 0, 1)
[ 0.93421349-0.3567144j   0.33723017-0.94142223j -0.49523164-0.86876097j]
>>> data = aa.phs2src(numpy.array([.5,1,0j]),cat['Sun'],1,2)
>>> print data
[-0.07802227+0.49387501j  0.83315419+0.55304077j  0.00000000+0.j        ]
>>> uvw = aa.gen_uvw(1,2,cat['Sun'])
>>> print uvw
[[[  8.86987669   9.4612018   10.05252691]]
```

```
 [[  7.55959227   8.06356508   8.5675379 ]]
 [[ 17.72506283  18.90673368  20.08840454]]]
>>> print aa.unphs2src(data,cat['Sun'],1,2)
[ 0.5+0.j 1.0+0.j 0.0+0.j]
```

Using the AntennaArray and SrcCatalog we created earlier, we can now use gen_phs() to return the phases that, when multiplied by data for the speci ed baseline, will flatten the fringes of that source. Note that 3 values are returned here for the 3 active frequencies (we set them in the Beam). We could apply these phases to the data ourselves (or take the complex conjugate and call it simulated data), but phs2src() does that for us. We can also get the corresponding uvw coordinates projected toward this source using gen_uvw(). Note again the 3 entries in uvw for the 3 active frequencies. To undo what phs2src() just did, there is unphs2src().

## 2.3 Imaging

An `aipy.img.Img` instance is responsible for storing the UV matrix and the beam matrix that can be inverted, using a fast Fourier transform, to get the dirty image and the dirty beam, respectively. You start by specifying the size of your two matrices in UV coordinates by providing a size (number of wavelengths across your matrix) and a resolution (number of pixels per wavelength). The size of your UV matrix determines the resolution of your image (image resolution = 1/UV size) and the resolution of your UV matrix determines the field-of-view of your image on the sky (image size in *l,m* = 1/UV resolution).

### 2.3.1 Coordinate Systems

```
>>> import aipy, pylab
>>> im = aipy.img.Img(size=200, res=0.5)
>>> L,M = im.get_LM(center=(200,200))
>>> pylab.subplot(121); pylab.imshow(L); pylab.colorbar(shrink=.7)
>>> pylab.subplot(122); pylab.imshow(M); pylab.colorbar(shrink=.7)
>>> pylab.show()
```

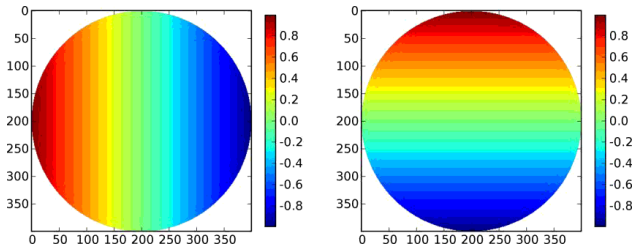


Figure 2.1: Plotted are the *lm,* coordinates for an Img with size=200, res=0.5. Note that *l* measures +E -W, and *m* measures +N -S. AIPY follows the convention that for images, coordinates increase right to left and bottom to top (assuming the origin is placed in the top-left corner). This presents a geocentric view of the sky.

In the above snippet, we've defined a matrix 200 wavelengths across, with a resolution of 0.5 wavelengths. In image domain, this generates an image with a resolution of $\sim 0.28^\circ$ near image center (and decreasing resolution away from center), and a range of $-90^\circ$ to $+90^\circ$ since *l,m* range from -1 to 1.

---

**Note:** This generates sky coordinates which are outside the range of physical possibility. These coordinates are masked.

---

Internally to Img, UV and image centers are at (0,0). To recenter for plotting, where it is preferable that image center is in the middle of the screen, most functions that return a matrix accept a "center" argument with a pixel offset.

AIPY follows the convention that *l* increases right to left, and *m* increases bottom to top, assuming the origin is placed in the top-left corner. This gives us a geocentric view of the sky. Before we get into actual imaging, lets explore coordinates a little more. Continuing from above:

```
>>> xyz = im.get_top(center=(200,200))
>>> az,alt = aipy.coord.top2azalt(xyz)
>>> pylab.subplot(221); pylab.imshow(az)
>>> pylab.subplot(222); pylab.imshow(alt)
>>> xyz = im.get_eq(ra=0, dec=3.14/4, center=(200,200))
>>> ra,dec = aipy.coord.eq2radec(xyz)
>>> pylab.subplot(223); pylab.imshow(ra)
>>> pylab.subplot(224); pylab.imshow(dec)
>>> pylab.show()
```

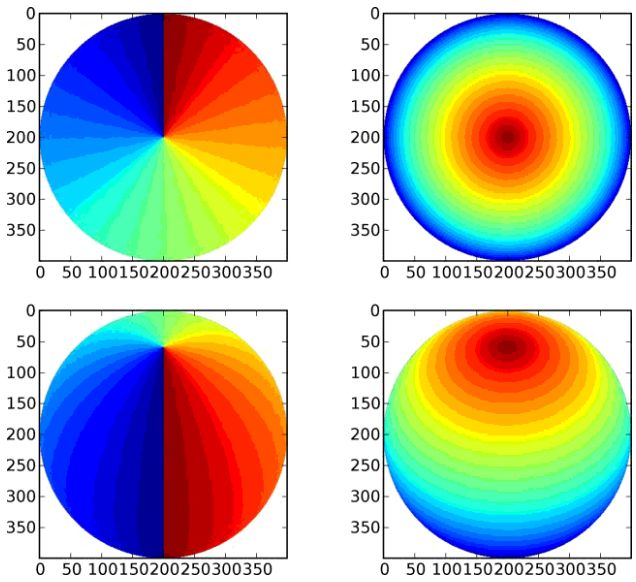


Figure 2.2: On the top row are plotted topocentric coordinates for an Img in azimuth (angle clockwise from N) and altitude (angle from horizon). The second row shows right-ascension and declination coordinates in the current epoch for an observer at $+45°$ latitude with 0:00 right ascension overhead. Coordinate transformations are provided by the `aipy.coord` module.

The above example illustrates how topocentric coordinates (useful for calculating beam patterns) and equatorial coordinates (useful for calculated source positions) at the current epoch are both available through an Img.

---

**Note:** The `aipy.coord` module also provides all functionality for converting between coordinate systems and precessing to various epochs; the several coordinate systems available through Img are just for convenience.

---

### 2.3.2 Gridding and Imaging

It is finally time to generate a dirty image. To do this, we will gather visibility samples, calculate the uvw coordinates where they were measured, and then grid the samples and weights onto a matrix using Img.put(). Finally, we will use an inverse Fourier transform to generate a dirty image and matching dirty beam. For data, we will use the same test.uv file as in previous sections, with associated parameters recorded in `aipy.cal` under "pwa303". These files are attainable via:

```
wget http://setiathome.berkeley.edu/~aparsons/aipy/test.uv.tar.bz2
wget http://fornax.phys.unm.edu/aipy-1.0.x/pwa303.py
```

```
compress_uv.py -x test.uv.tar.bz2
```

Starting with a fresh Python interpreter, we will open our Miriad UV file and prepare an AntennaArray with antenna positions derived from the "pwa303" location key in `aipy.cal`, and frequency channels matching the data in the UV file (sdf, sfreq, nchan). We will also choose a source (in this case, "vir" indicates Virgo A from `aipy.src`) to phase our data to, putting it at the center of our image:

```
>>> import aipy, pylab, numpy as n
>>> uv = aipy.miriad.UV('test.uv')
>>> aa = aipy.cal.get_aa('pwa303', uv['sdf'], uv['sfreq'], uv['nchan'])
>>> srcs = aipy._src.misc.get_srcs(srcs=['vir'])
>>> src = srcs[0]
```

Next, we will gather visibility data from the UV file and calculate the corresponding uvw coordinates using our AntennaArray and celestial source. We will not include auto-correlation data (uv.select), we will skip data where Virgo is below the horizon (the PointingError try-except clause), and we will throw out data that is agged as bad in the data mask (the compress/compressed functions). For more signal-to-noise, we're including all channels-all data and coordinates are a function of frequency, making this a multi-frequency map:

```
>>> data, uvw, wgts = [], [], []
>>> uv.select('auto', 0, 0, include=False)
>>> for (crd,t,(i,j)),d in uv.all():
...     aa.set_jultime(t)
...     src.compute(aa)
...     try:
...         d = aa.phs2src(d, src, i, j)
...         crd = aa.gen_uvw(i, j, src=src)
...     except aipy.phs.PointingError:
...         continue
...     uvw.append(n.squeeze(crd.compress(n.logical_not(d.mask), axis=2)))
...     data.append(d.compressed())
...     wgts.append(n.array([1.] * len(data[-1])))
...
>>> data = n.concatenate(data)
>>> uvw = n.concatenate(uvw, axis=1)
>>> wgts = n.concatenate(wgts)
```

The above also illustrates how different sample weights can be specified for each data, although in this case we equally weight each sample. Now that we've gathered up all our visibility data with uvw coordinate and sample weights, we are ready to make an image:

```
>>> im = aipy.img.Img(size=200, res=0.5)
>>> uvw, data, wgts = im.append_hermitian(uvw, data, wgts=wgts)
>>> im.put(uvw, data, wgts=wgts)
>>> pylab.subplot(221); pylab.imshow(n.abs(im.uv))
>>> pylab.subplot(222); pylab.imshow(n.abs(im.bm))
>>> pylab.subplot(223); pylab.imshow(n.log10(im.image(center=(200,200))))
>>> pylab.subplot(224); pylab.imshow(n.log10(im.bm_image(center=(200,200))))
>>> pylab.show()
```

We have specified an Img that is larger than our maximum baseline to be able to hold all UV data, and we have specifed a resolution that should cover horizon to horizon. We used the append hermitian() command to generate the complex-conjugate data at -uvw that complements every (i,j) baseline with the equivalent (j,i) measurement and ensures that a real-valued sky image is generated. Finally, we use put() to grid the data. Do not forget that the Img, as opposed to the ImgW, ignores the w component of each uvw coordinate.

It is also worth pointing out that the sidelobes of Virgo in our dirty image wrap around the edges and alias back into our image. This is an effect related to the eld of view (FoV) of our image, or equivalently, the resolution of our UV matrix.

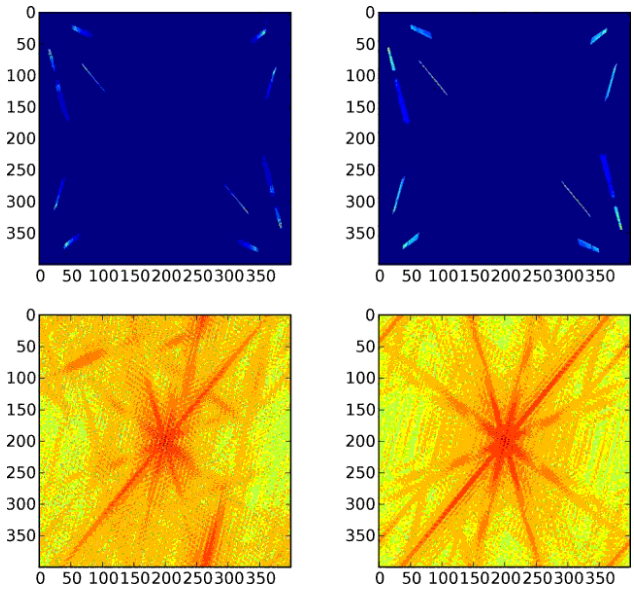


Figure 2.3: The upper plots illustrate matrices containing gridded visibility data (left) and gridded sample weights (right). Below them are the inverse Fourier transforms of these matrices, illustrating a dirty image of Virgo A (left) and the corresponding dirty beam (right).

Wrapping effects can be combatted by choosing a resolution corresponding to a larger FoV than we are interested in, or by more carefully gridding our samples with a convolution kernel that filters near the edge of the image. For simplicity, AIPY's Img currently only supports the former solution.

CHAPTER

THREE

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

## a

## A



## B



## C



## D



## E



## F



## G

## H



## I



## J



## L



## M



## P



## R



## S

## T



## U



## V



## W



## X