Codex Africanus Documentation

Release 0.2.9

Simon Perkins

Dec 15, 2020

Contents:

CHAPTER 1

Codex Africanus

Radio Astronomy Building Blocks

1.1 Documentation

https://codex-africanus.readthedocs.io.

CHAPTER 2

Installation

2.1 Stable release

To install Codex Africanus, run this command in your terminal:

```
$ pip install codex-africanus
```

This is the preferred method to install Codex Africanus, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

By default, Codex Africanus will install with a minimal set of dependencies, numpy and numba.

Further functionality can be enabled by installing extra requirements as follows:

```
$ pip install codex-africanus[dask]
$ pip install codex-africanus[scipy]
$ pip install codex-africanus[astropy]
$ pip install codex-africanus[python-casacore]
```

To install the complete set of dependencies for the CPU:

```
$ pip install codex-africanus[complete]
```

To install the complete set of dependencies including CUDA:

```
$ pip install codex-africanus[complete-cuda]
```

2.2 From sources

The sources for Codex Africanus can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/ska-sa/codex-africanus
```

Or download the tarball:

```
$ curl  -OL https://github.com/ska-sa/codex-africanus/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use Codex Africanus in a project:

```
import africanus
```

CHAPTER 4

Command Line Utilities

The following command line utilities are installed. Run each utility's help for further information.

```
$ utility --help
```

4.1 plot-filter

Plots convolution filters.

4.2 plot-taper

Plots tapers associated with convolution filters.

CHAPTER 5

API

5.1 Radio Interferometer Measurement Equation

Functions used to compute the terms of the Radio Interferometer Measurement Equation (RIME). It describes the response of an interferometer to a sky model.

$$V_{pq} = G_p \left(\sum_s E_{ps} L_p K_{ps} B_s K_{qs}^H L_q^H E_{qs}^H \right) G_q^H$$

where for antenna p and q, and source s:

- G_p represents direction-independent effects.
- E_{ps} represents direction-dependent effects.
- L_p represents the feed rotation.
- K_{ps} represents the phase delay term.
- B_s represents the brightness matrix.

The RIME is more formally described in the following four papers:

- I. A full-sky Jones formalism
- II. Calibration and direction-dependent effects
- III. Addressing direction-dependent effects in 21cm WSRT observations of 3C147
- IV. A generalized tensor formalism

5.1.1 Numpy

predict_vis(time_index, antenna1, antenna2)	Multiply Jones terms together to form model visibilities according to the following formula:
phase_delay(lm, uvw, frequency[, convention])	Computes the phase delay (K) term:
parallactic_angles(times, antenna_positions, ...)	Computes parallactic angles per timestep for the given reference antenna position and field centre.
feed_rotation(parallactic_angles[, feed_type])	Computes the 2x2 feed rotation (L) matrix from the `parallactic_angles`.
transform_sources(lm, parallactic_angles, ...)	Creates beam sampling coordinates suitable for use in *beam_cube_dde()* by:
beam_cube_dde(beam, beam_lm_extents, ...)	Evaluates Direction Dependent Effects along a source's path by interpolating the values of a complex beam cube at the source location.
zernike_dde(coords, coeffs, noll_index)	Computes Direction Dependent Effects by evaluating Zernicke Polynomials defined by coefficients `coeffs` and noll indexes `noll_index` at the specified coordinates `coords`.
wsclean_predict(uvw, lm, source_type, flux, ...)	Predict visibilities from a WSClean sky model.

africanus.rime.**predict_vis**(*time_index*, *antenna1*, *antenna2*, *dde1_jones=None*, *source_coh=None*, *dde2_jones=None*, *die1_jones=None*, *base_vis=None*, *die2_jones=None*)

Multiply Jones terms together to form model visibilities according to the following formula:

$$V_{pq} = G_p \left(B_{pq} + \sum_s E_{ps} X_{pqs} E_{qs}^H \right) G_q^H$$

where for antenna p and q, and source s:

- B_{pq} represent base coherencies.
- E_{ps} represents Direction-Dependent Jones terms.
- X_{pqs} represents a coherency matrix (per-source).
- G_p represents Direction-Independent Jones terms.

Generally, E_{ps}, G_p, X_{pqs} should be formed by using the *RIME API* functions and combining them together with `einsum()`.

Please read the Notes

Parameters **time_index** : `numpy.ndarray`

Time index used to look up the antenna Jones index for a particular baseline with shape `(row,)`. Obtainable via `np.unique(time, return_inverse=True)[1]`.

antenna1 : `numpy.ndarray`

Antenna 1 index used to look up the antenna Jones for a particular baseline. with shape `(row,)`.

antenna2 : `numpy.ndarray`

Antenna 2 index used to look up the antenna Jones for a particular baseline. with shape `(row,)`.

dde1_jones : `numpy.ndarray`, optional

E_{ps} Direction-Dependent Jones terms for the first antenna. shape `(source,time, ant,chan,corr_1,corr_2)`

source_coh : numpy.ndarray, optional

X_{pqs} Direction-Dependent Coherency matrix for the baseline. with shape (source, row, chan, corr_1, corr_2)

dde2_jones : numpy.ndarray, optional

E_{qs} Direction-Dependent Jones terms for the second antenna. This is usually the same array as dde1_jones as this preserves the symmetry of the RIME. predict_vis will perform the conjugate transpose internally. shape (source, time, ant, chan, corr_1, corr_2)

die1_jones : numpy.ndarray, optional

G_{ps} Direction-Independent Jones terms for the first antenna of the baseline. with shape (time, ant, chan, corr_1, corr_2)

base_vis : numpy.ndarray, optional

B_{pq} base coherencies, added to source coherency summation *before* multiplication with *die1_jones* and *die2_jones*. shape (row, chan, corr_1, corr_2).

die2_jones : numpy.ndarray, optional

G_{ps} Direction-Independent Jones terms for the second antenna of the baseline. This is usually the same array as die1_jones as this preserves the symmetry of the RIME. predict_vis will perform the conjugate transpose internally. shape (time, ant, chan, corr_1, corr_2)

Returns **visibilities** : numpy.ndarray

Model visibilities of shape (row, chan, corr_1, corr_2)

Notes

- Direction-Dependent terms (dde{1,2}_jones) and Independent (die{1,2}_jones) are optional, but if one is present, the other must be present.
- The inputs to this function involve row, time and ant (antenna) dimensions.
- Each row is associated with a pair of antenna Jones matrices at a particular timestep via the time_index, antenna1 and antenna2 inputs.
- The row dimension must be an increasing partial order in time.

africanus.rime.**phase_delay**(*lm*, *uvw*, *frequency*, *convention='fourier'*)

Computes the phase delay (K) term:

$$e^{-2\pi i(ul+vm+w(n-1))}$$

where $n = \sqrt{1-l^2-m^2}$

Parameters **lm** : numpy.ndarray

LM coordinates of shape (source, 2) with L and M components in the last dimension.

uvw : numpy.ndarray

UVW coordinates of shape (row, 3) with U, V and W components in the last dimension.

frequency : numpy.ndarray

frequencies of shape `(chan,)`

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

Returns **complex_phase** : `numpy.ndarray`

complex of shape `(source, row, chan)`

Notes

Corresponds to the complex exponential of the Van Cittert-Zernike Theorem.

MeqTrees uses the CASA sign convention.

`africanus.rime.`**`parallactic_angles`**(*times*, *antenna_positions*, *field_centre*, *backend='casa'*)

Computes parallactic angles per timestep for the given reference antenna position and field centre.

Parameters **times** : `numpy.ndarray`

Array of Mean Julian Date times in *seconds* with shape `(time,)`,

antenna_positions : `numpy.ndarray`

Antenna positions of shape `(ant, 3)` in *metres* in the *ITRF* frame.

field_centre : `numpy.ndarray`

Field centre of shape `(2,)` in *radians*

backend : {'casa', 'test'}, optional

Backend to use for calculating the parallactic angles.

- `casa` defers to an implementation depending on `python-casacore`. This backend should be used by default.
- `test` creates parallactic angles by multiplying the `times` and `antenna_position` arrays. It exist solely for testing.

Returns **parallactic_angles** : `numpy.ndarray`

Parallactic angles of shape `(time,ant)`

`africanus.rime.`**`feed_rotation`**(*parallactic_angles*, *feed_type='linear'*)

Computes the 2x2 feed rotation (L) matrix from the `parallactic_angles`.

$$\text{linear}\begin{bmatrix} cos(pa) & sin(pa) \\ -sin(pa) & cos(pa) \end{bmatrix} \qquad \text{circular}\begin{bmatrix} e^{-ipa} & 0 \\ 0 & e^{ipa} \end{bmatrix}$$

Parameters **parallactic_angles** : `numpy.ndarray`

floating point parallactic angles. Of shape `(pa0, pa1, ..., pan)`.

feed_type : {'linear', 'circular'}

The type of feed

Returns **feed_matrix** : `numpy.ndarray`

Feed rotation matrix of shape `(pa0, pa1,...,pan,2,2)`

`africanus.rime.`**`transform_sources`**(*lm*, *parallactic_angles*, *pointing_errors*, *antenna_scaling*, *frequency*, *dtype=None*)

Creates beam sampling coordinates suitable for use in *`beam_cube_dde()`* by:

1. Rotating `lm` coordinates by the `parallactic_angles`
2. Adding `pointing_errors`
3. Scaling by `antenna_scaling`

Parameters **lm** : `numpy.ndarray`

LM coordinates of shape `(src,2)` in radians offset from the phase centre.

parallactic_angles : `numpy.ndarray`

parallactic angles of shape `(time, antenna)` in radians.

pointing_errors : `numpy.ndarray`

LM pointing errors for each antenna at each timestep in radians. Has shape `(time, antenna, 2)`

antenna_scaling : `numpy.ndarray`

antenna scaling factor for each channel and each antenna. Has shape `(antenna, chan)`

frequency : `numpy.ndarray`

frequencies for each channel. Has shape `(chan,)`

dtype : `numpy.dtype`, optional

Numpy dtype of result array. Should be float32 or float64. Defaults to float64

Returns **coords** : `numpy.ndarray`

coordinates of shape `(3, src, time, antenna, chan)` where each coordinate component represents **l**, **m** and **frequency**, respectively.

`africanus.rime.`**`beam_cube_dde`**(*beam*, *beam_lm_extents*, *beam_freq_map*, *lm*, *parallactic_angles*, *point_errors*, *antenna_scaling*, *frequency*)

Evaluates Direction Dependent Effects along a source's path by interpolating the values of a complex beam cube at the source location.

Parameters **beam** : `numpy.ndarray`

Complex beam cube of shape `(beam_lw, beam_mh, beam_nud, corr, corr)`. *beam_lw*, *beam_mh* and *beam_nud* define the size of the cube in the l, m and frequency dimensions, respectively.

beam_lm_extents : `numpy.ndarray`

lm extents of the beam cube of shape `(2, 2)`. `[[lower_l, upper_l], [lower_m, upper_m]]`.

beam_freq_map : `numpy.ndarray`

Beam frequency map of shape `(beam_nud,)`. This array is used to define interpolation along the `(chan,)` dimension.

lm : `numpy.ndarray`

Source lm coordinates of shape `(source, 2)`. These coordinates are:

1. Scaled if the associated frequency lies outside the beam cube.
2. Offset by pointing errors: `point_errors`
3. Rotated by parallactic angles: `parallactic_angles`.

4. Scaled by antenna scaling factors: `antenna_scaling`.

parallactic_angles : `numpy.ndarray`

Parallactic angles of shape `(time, ant)`.

point_errors : `numpy.ndarray`

Pointing errors of shape `(time, ant, chan, 2)`.

antenna_scaling : `numpy.ndarray`

Antenna scaling factors of shape `(ant, chan, 2)`

frequency : `numpy.ndarray`

Frequencies of shape `(chan,)`.

Returns **ddes** : `numpy.ndarray`

Direction Dependent Effects of shape `(source, time, ant, chan, corr, corr)`

Notes

1. Sources are clamped to the provided *beam_lm_extents*.
2. Frequencies outside the cube (i.e. outside beam_freq_map) introduce linear scaling to the lm coordinates of a source.

`africanus.rime.`**`zernike_dde`**(*coords*, *coeffs*, *noll_index*)

Computes Direction Dependent Effects by evaluating Zernicke Polynomials defined by coefficients `coeffs` and noll indexes `noll_index` at the specified coordinates `coords`.

Decomposition of a voxel beam cube into Zernicke polynomial coefficients can be achieved through the use of the eidos package.

Parameters **coords** : `numpy.ndarray`

Float coordinates at which to evaluate the zernike polynomials. Has shape `(3, source, time, ant, chan)`. The three components in the first dimension represent l, m and frequency coordinates, respectively.

coeffs : `numpy.ndarray`

complex Zernicke polynomial coefficients. Has shape `(ant, chan, corr_1, ..., corr_n, poly)` where `poly` is the number of polynomial coefficients and `corr_1, ..., corr_n` are a variable number of correlation dimensions.

noll_index : `numpy.ndarray`

Noll index associated with each polynomial coefficient. Has shape `(ant, chan, corr_1, ..., corr_n, poly)`.

Returns **dde** : `numpy.ndarray`

complex values with shape `(source, time, ant, chan, corr_1, ..., corr_n)`

`africanus.rime.`**`wsclean_predict`**(*uvw*, *lm*, *source_type*, *flux*, *coeffs*, *log_poly*, *ref_freq*, *gauss_shape*, *frequency*)

Predict visibilities from a WSClean sky model.

Parameters **uvw** : `numpy.ndarray`

UVW coordinates of shape `(row, 3)`

lm : `numpy.ndarray`

Source LM coordinates of shape `(source, 2)`, in radians. Derived from the `Ra` and `Dec` fields.

source_type : `numpy.ndarray`

Strings defining the source type of shape `(source,)`. Should be either `"POINT"` or `"GAUSSIAN"`. Contains the `Type` field.

flux : `numpy.ndarray`

Source flux of shape `(source,)`. Contains the `I` field.

coeffs : `numpy.ndarray`

Source Polynomial coefficients of shape `(source, coeffs)`. Contains the `SpectralIndex` field.

log_poly : `numpy.ndarray`

Source polynomial type of shape `(source,)`. If True, logarithmic polynomials are used. If False, standard polynomials are used. Contains the `LogarithmicSI` field.

ref_freq : `numpy.ndarray`

Source Reference frequency of shape `(source,)`. Contains the `ReferenceFrequency` field.

gauss_shape : `numpy.ndarray`

Gaussian shape parameters of shape `(source, 3)` used when the corresponding `source_type` is `"GAUSSIAN"`. The 3 components should contain the `MajorAxis`, `MinorAxis` and `Orientation` fields in radians, respectively.

frequency : `numpy.ndarray`

Frequency of shape `(chan,)`.

Returns **visibilities** : `numpy.ndarray`

Complex visibilities of shape `(row, chan, 1)`

5.1.2 Cuda

predict_vis(time_index, antenna1, antenna2)	Multiply Jones terms together to form model visibilities according to the following formula:
phase_delay(lm, uvw, frequency)	Computes the phase delay (K) term:
feed_rotation(parallactic_angles[, feed_type])	Computes the 2x2 feed rotation (L) matrix from the `parallactic_angles`.
beam_cube_dde(beam, beam_lm_ext, ...)	Evaluates Direction Dependent Effects along a source's path by interpolating the values of a complex beam cube at the source location.

africanus.rime.cuda.**predict_vis**(*time_index*, *antenna1*, *antenna2*, *dde1_jones=None*, *source_coh=None*, *dde2_jones=None*, *die1_jones=None*, *base_vis=None*, *die2_jones=None*)

Multiply Jones terms together to form model visibilities according to the following formula:

$$V_{pq} = G_p \left(B_{pq} + \sum_s E_{ps} X_{pqs} E_{qs}^H \right) G_q^H$$

where for antenna p and q, and source s:

- B_{pq} represent base coherencies.
- E_{ps} represents Direction-Dependent Jones terms.
- X_{pqs} represents a coherency matrix (per-source).
- G_p represents Direction-Independent Jones terms.

Generally, E_{ps}, G_p, X_{pqs} should be formed by using the *RIME API* functions and combining them together with `einsum()`.

Please read the Notes

Parameters **time_index** : `cupy.ndarray`

Time index used to look up the antenna Jones index for a particular baseline with shape `(row,)`. Obtainable via `cp.unique(time, return_inverse=True)[1]`.

antenna1 : `cupy.ndarray`

Antenna 1 index used to look up the antenna Jones for a particular baseline. with shape `(row,)`.

antenna2 : `cupy.ndarray`

Antenna 2 index used to look up the antenna Jones for a particular baseline. with shape `(row,)`.

dde1_jones : `cupy.ndarray`, optional

E_{ps} Direction-Dependent Jones terms for the first antenna. shape `(source,time,ant,chan,corr_1,corr_2)`

source_coh : `cupy.ndarray`, optional

X_{pqs} Direction-Dependent Coherency matrix for the baseline. with shape `(source,row,chan,corr_1,corr_2)`

dde2_jones : `cupy.ndarray`, optional

E_{qs} Direction-Dependent Jones terms for the second antenna. This is usually the same array as `dde1_jones` as this preserves the symmetry of the RIME. `predict_vis` will perform the conjugate transpose internally. shape `(source,time,ant,chan,corr_1,corr_2)`

die1_jones : `cupy.ndarray`, optional

G_{ps} Direction-Independent Jones terms for the first antenna of the baseline. with shape `(time,ant,chan,corr_1,corr_2)`

base_vis : `cupy.ndarray`, optional

B_{pq} base coherencies, added to source coherency summation *before* multiplication with *die1_jones* and *die2_jones*. shape `(row,chan,corr_1,corr_2)`.

die2_jones : `cupy.ndarray`, optional

G_{ps} Direction-Independent Jones terms for the second antenna of the baseline. This is usually the same array as `die1_jones` as this preserves the symmetry of the RIME. `predict_vis` will perform the conjugate transpose internally. shape `(time,ant, chan,corr_1,corr_2)`

Returns **visibilities** : `cupy.ndarray`

Model visibilities of shape `(row,chan,corr_1,corr_2)`

Notes

- Direction-Dependent terms (dde{1,2}_jones) and Independent (die{1,2}_jones) are optional, but if one is present, the other must be present.
- The inputs to this function involve `row`, `time` and `ant` (antenna) dimensions.
- Each `row` is associated with a pair of antenna Jones matrices at a particular timestep via the `time_index`, `antenna1` and `antenna2` inputs.
- The `row` dimension must be an increasing partial order in time.

`africanus.rime.cuda.`**`phase_delay`**(*lm*, *uvw*, *frequency*)

Computes the phase delay (K) term:

$$e^{-2\pi i(ul+vm+w(n-1))}$$

where $n = \sqrt{1-l^2-m^2}$

Parameters **lm** : `cupy.ndarray`

LM coordinates of shape `(source, 2)` with L and M components in the last dimension.

uvw : `cupy.ndarray`

UVW coordinates of shape `(row, 3)` with U, V and W components in the last dimension.

frequency : `cupy.ndarray`

frequencies of shape `(chan,)`

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

Returns **complex_phase** : `cupy.ndarray`

complex of shape `(source, row, chan)`

Notes

Corresponds to the complex exponential of the Van Cittert-Zernike Theorem.

MeqTrees uses the CASA sign convention.

`africanus.rime.cuda.`**`feed_rotation`**(*parallactic_angles*, *feed_type='linear'*)

Computes the 2x2 feed rotation (L) matrix from the `parallactic_angles`.

$$\text{linear}\begin{bmatrix} cos(pa) & sin(pa) \\ -sin(pa) & cos(pa) \end{bmatrix} \qquad \text{circular}\begin{bmatrix} e^{-ipa} & 0 \\ 0 & e^{ipa} \end{bmatrix}$$

Parameters **parallactic_angles** : cupy.ndarray

floating point parallactic angles. Of shape `(pa0, pa1, ..., pan)`.

feed_type : {'linear', 'circular'}

The type of feed

Returns **feed_matrix** : cupy.ndarray

Feed rotation matrix of shape `(pa0, pa1,...,pan,2,2)`

africanus.rime.cuda.**beam_cube_dde**(*beam*, *beam_lm_ext*, *beam_freq_map*, *lm*, *parangles*, *pointing_errors*, *antenna_scaling*, *frequencies*)

Evaluates Direction Dependent Effects along a source's path by interpolating the values of a complex beam cube at the source location.

Parameters **beam** : cupy.ndarray

Complex beam cube of shape `(beam_lw, beam_mh, beam_nud, corr, corr)`. *beam_lw*, *beam_mh* and *beam_nud* define the size of the cube in the l, m and frequency dimensions, respectively.

beam_lm_extents : cupy.ndarray

lm extents of the beam cube of shape `(2, 2)`. `[[lower_l, upper_l], [lower_m, upper_m]]`.

beam_freq_map : cupy.ndarray

Beam frequency map of shape `(beam_nud,)`. This array is used to define interpolation along the `(chan,)` dimension.

lm : cupy.ndarray

Source lm coordinates of shape `(source, 2)`. These coordinates are:

1. Scaled if the associated frequency lies outside the beam cube.
2. Offset by pointing errors: `point_errors`
3. Rotated by parallactic angles: `parallactic_angles`.
4. Scaled by antenna scaling factors: `antenna_scaling`.

parallactic_angles : cupy.ndarray

Parallactic angles of shape `(time, ant)`.

point_errors : cupy.ndarray

Pointing errors of shape `(time, ant, chan, 2)`.

antenna_scaling : cupy.ndarray

Antenna scaling factors of shape `(ant, chan, 2)`

frequency : cupy.ndarray

Frequencies of shape `(chan,)`.

Returns **ddes** : cupy.ndarray

Direction Dependent Effects of shape `(source, time, ant, chan, corr, corr)`

Notes

1. Sources are clamped to the provided *beam_lm_extents*.
2. Frequencies outside the cube (i.e. outside beam_freq_map) introduce linear scaling to the lm coordinates of a source.

5.1.3 Dask

predict_vis(time_index, antenna1, antenna2)	Multiply Jones terms together to form model visibilities according to the following formula:
phase_delay(lm, uvw, frequency[, convention])	Computes the phase delay (K) term:
parallactic_angles(times, antenna_positions, ...)	Computes parallactic angles per timestep for the given reference antenna position and field centre.
feed_rotation(parallactic_angles, feed_type)	Computes the 2x2 feed rotation (L) matrix from the `parallactic_angles`.
transform_sources(lm, parallactic_angles, ...)	Creates beam sampling coordinates suitable for use in *beam_cube_dde()* by:
beam_cube_dde(beam, beam_lm_extents, ...)	Evaluates Direction Dependent Effects along a source's path by interpolating the values of a complex beam cube at the source location.
zernike_dde(coords, coeffs, noll_index)	Computes Direction Dependent Effects by evaluating Zernicke Polynomials defined by coefficients `coeffs` and noll indexes `noll_index` at the specified coordinates `coords`.
wsclean_predict(uvw, lm, source_type, flux, ...)	Predict visibilities from a WSClean sky model.

africanus.rime.dask.**predict_vis**(*time_index*, *antenna1*, *antenna2*, *dde1_jones=None*, *source_coh=None*, *dde2_jones=None*, *die1_jones=None*, *base_vis=None*, *die2_jones=None*, *streams=None*)

Multiply Jones terms together to form model visibilities according to the following formula:

$$V_{pq} = G_p \left(B_{pq} + \sum_s E_{ps} X_{pqs} E_{qs}^H \right) G_q^H$$

where for antenna p and q, and source s:

- B_{pq} represent base coherencies.
- E_{ps} represents Direction-Dependent Jones terms.
- X_{pqs} represents a coherency matrix (per-source).
- G_p represents Direction-Independent Jones terms.

Generally, E_{ps}, G_p, X_{pqs} should be formed by using the *RIME API* functions and combining them together with `einsum()`.

Please read the Notes

Parameters **time_index** : `dask.array.Array`

Time index used to look up the antenna Jones index for a particular baseline with shape `(row,)`. Obtainable via `time.map_blocks(lambda a: np.unique(a, return_inverse=True)[1])`.

antenna1 : dask.array.Array

Antenna 1 index used to look up the antenna Jones for a particular baseline. with shape (row,).

antenna2 : dask.array.Array

Antenna 2 index used to look up the antenna Jones for a particular baseline. with shape (row,).

dde1_jones : dask.array.Array, optional

E_{ps} Direction-Dependent Jones terms for the first antenna. shape (source,time,ant,chan,corr_1,corr_2)

source_coh : dask.array.Array, optional

X_{pqs} Direction-Dependent Coherency matrix for the baseline. with shape (source,row,chan,corr_1,corr_2)

dde2_jones : dask.array.Array, optional

E_{qs} Direction-Dependent Jones terms for the second antenna. This is usually the same array as dde1_jones as this preserves the symmetry of the RIME. predict_vis will perform the conjugate transpose internally. shape (source,time,ant,chan,corr_1,corr_2)

die1_jones : dask.array.Array, optional

G_{ps} Direction-Independent Jones terms for the first antenna of the baseline. with shape (time,ant,chan,corr_1,corr_2)

base_vis : dask.array.Array, optional

B_{pq} base coherencies, added to source coherency summation *before* multiplication with *die1_jones* and *die2_jones*. shape (row,chan,corr_1,corr_2).

die2_jones : dask.array.Array, optional

G_{ps} Direction-Independent Jones terms for the second antenna of the baseline. This is usually the same array as die1_jones as this preserves the symmetry of the RIME. predict_vis will perform the conjugate transpose internally. shape (time,ant,chan,corr_1,corr_2)

streams : {False, True}

If True the coherencies are serially summed in a linear chain. If False, dask uses a tree style reduction algorithm.

Returns **visibilities** : dask.array.Array

Model visibilities of shape (row,chan,corr_1,corr_2)

Notes

- Direction-Dependent terms (dde{1,2}_jones) and Independent (die{1,2}_jones) are optional, but if one is present, the other must be present.
- The inputs to this function involve row, time and ant (antenna) dimensions.
- Each row is associated with a pair of antenna Jones matrices at a particular timestep via the time_index, antenna1 and antenna2 inputs.
- The row dimension must be an increasing partial order in time.

- The `ant` dimension should only contain a single chunk equal to the number of antenna. Since each `row` can contain any antenna, random access must be preserved along this dimension.
- The chunks in the `row` and `time` dimension **must** align. This subtle point **must be understood otherwise invalid results will be produced** by the chunking scheme. In the example below we have four unique time indices `[0,1,2,3]`, and four unique antenna `[0,1,2,3]` indexing `10` rows.

```
#  Row indices into the time/antenna indexed arrays
time_idx = np.asarray([0,0,1,1,2,2,2,2,3,3])
ant1 = np.asarray(    [0,0,0,0,1,1,1,2,2,3]
ant2 = np.asarray(    [0,1,2,3,1,2,3,2,3,3])
```

A reasonable chunking scheme for the `row` and `time` dimension would be `(4,4,2)` and `(2,1,1)` respectively. Another way of explaining this is that the first four rows contain two unique timesteps, the second four rows contain one unique timestep and the last two rows contain one unique timestep.

Some rules of thumb:

1. The number chunks in `row` and `time` must match although the individual chunk sizes need not.
2. Unique timesteps should not be split across row chunks.
3. For a Measurement Set whose rows are ordered on the `TIME` column, the following is a good way of obtaining the row chunking strategy:

```
import numpy as np
import pyrap.tables as pt

ms = pt.table("data.ms")
times = ms.getcol("TIME")
unique_times, chunks = np.unique(times, return_counts=True)
```

4. Use *`aggregate_chunks()`* to aggregate multiple `row` and `time` chunks into chunks large enough such that functions operating on the resulting data can drop the GIL and spend time processing the data. Expanding the previous example:

```
# Aggregate row
utimes = unique_times.size
# Single chunk for each unique time
time_chunks = (1,)*utimes
# Aggregate row chunks into chunks <= 10000
aggregate_chunks((chunks, time_chunks), (10000, utimes))
```

`africanus.rime.dask.`**`phase_delay`**(*lm*, *uvw*, *frequency*, *convention='fourier'*)

Computes the phase delay (K) term:

$$e^{-2\pi i(ul+vm+w(n-1))}$$

where $n = \sqrt{1-l^2-m^2}$

Parameters **lm** : `dask.array.Array`

LM coordinates of shape `(source, 2)` with L and M components in the last dimension.

uvw : `dask.array.Array`

UVW coordinates of shape `(row, 3)` with U, V and W components in the last dimension.

frequency : `dask.array.Array`

frequencies of shape `(chan,)`

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

Returns **complex_phase** : `dask.array.Array`

complex of shape `(source, row, chan)`

Notes

Corresponds to the complex exponential of the Van Cittert-Zernike Theorem.

MeqTrees uses the CASA sign convention.

`africanus.rime.dask.`**`parallactic_angles`**(*times*, *antenna_positions*, *field_centre*, ***kwargs*)

Computes parallactic angles per timestep for the given reference antenna position and field centre.

Parameters **times** : `dask.array.Array`

Array of Mean Julian Date times in *seconds* with shape `(time,)`,

antenna_positions : `dask.array.Array`

Antenna positions of shape `(ant, 3)` in *metres* in the *ITRF* frame.

field_centre : `dask.array.Array`

Field centre of shape `(2,)` in *radians*

backend : {'casa', 'test'}, optional

Backend to use for calculating the parallactic angles.

- `casa` defers to an implementation depending on `python-casacore`. This backend should be used by default.
- `test` creates parallactic angles by multiplying the `times` and `antenna_position` arrays. It exist solely for testing.

Returns **parallactic_angles** : `dask.array.Array`

Parallactic angles of shape `(time,ant)`

`africanus.rime.dask.`**`feed_rotation`**(*parallactic_angles*, *feed_type*)

Computes the 2x2 feed rotation (L) matrix from the `parallactic_angles`.

$$\text{linear}\begin{bmatrix} cos(pa) & sin(pa) \\ -sin(pa) & cos(pa) \end{bmatrix} \qquad \text{circular}\begin{bmatrix} e^{-ipa} & 0 \\ 0 & e^{ipa} \end{bmatrix}$$

Parameters **parallactic_angles** : `numpy.ndarray`

floating point parallactic angles. Of shape `(pa0, pa1, ..., pan)`.

feed_type : {'linear', 'circular'}

The type of feed

Returns **feed_matrix** : `numpy.ndarray`

Feed rotation matrix of shape `(pa0, pa1,...,pan,2,2)`

`africanus.rime.dask.`**`transform_sources`**(*lm*, *parallactic_angles*, *pointing_errors*, *antenna_scaling*, *frequency*, *dtype=None*)

Creates beam sampling coordinates suitable for use in *`beam_cube_dde()`* by:

1. Rotating `lm` coordinates by the `parallactic_angles`
2. Adding `pointing_errors`
3. Scaling by `antenna_scaling`

Parameters **lm** : `dask.array.Array`

LM coordinates of shape `(src,2)` in radians offset from the phase centre.

parallactic_angles : `dask.array.Array`

parallactic angles of shape `(time, antenna)` in radians.

pointing_errors : `dask.array.Array`

LM pointing errors for each antenna at each timestep in radians. Has shape `(time, antenna, 2)`

antenna_scaling : `dask.array.Array`

antenna scaling factor for each channel and each antenna. Has shape `(antenna, chan)`

frequency : `dask.array.Array`

frequencies for each channel. Has shape `(chan,)`

dtype : `numpy.dtype`, optional

Numpy dtype of result array. Should be float32 or float64. Defaults to float64

Returns **coords** : `dask.array.Array`

coordinates of shape `(3, src, time, antenna, chan)` where each coordinate component represents **l**, **m** and **frequency**, respectively.

`africanus.rime.dask.`**`beam_cube_dde`**(*beam*, *beam_lm_extents*, *beam_freq_map*, *lm*, *parallactic_angles*, *point_errors*, *antenna_scaling*, *frequencies*)

Evaluates Direction Dependent Effects along a source's path by interpolating the values of a complex beam cube at the source location.

Parameters **beam** : `dask.array.Array`

Complex beam cube of shape `(beam_lw, beam_mh, beam_nud, corr, corr)`. *beam_lw*, *beam_mh* and *beam_nud* define the size of the cube in the l, m and frequency dimensions, respectively.

beam_lm_extents : `dask.array.Array`

lm extents of the beam cube of shape `(2, 2)`. `[[lower_l, upper_l], [lower_m, upper_m]]`.

beam_freq_map : `dask.array.Array`

Beam frequency map of shape `(beam_nud,)`. This array is used to define interpolation along the `(chan,)` dimension.

lm : `dask.array.Array`

Source lm coordinates of shape `(source, 2)`. These coordinates are:

1. Scaled if the associated frequency lies outside the beam cube.
2. Offset by pointing errors: `point_errors`
3. Rotated by parallactic angles: `parallactic_angles`.

4. Scaled by antenna scaling factors: `antenna_scaling`.

parallactic_angles : `dask.array.Array`

Parallactic angles of shape `(time, ant)`.

point_errors : `dask.array.Array`

Pointing errors of shape `(time, ant, chan, 2)`.

antenna_scaling : `dask.array.Array`

Antenna scaling factors of shape `(ant, chan, 2)`

frequency : `dask.array.Array`

Frequencies of shape `(chan,)`.

Returns **ddes** : `dask.array.Array`

Direction Dependent Effects of shape `(source, time, ant, chan, corr, corr)`

Notes

1. Sources are clamped to the provided *beam_lm_extents*.
2. Frequencies outside the cube (i.e. outside beam_freq_map) introduce linear scaling to the lm coordinates of a source.

`africanus.rime.dask.`**`zernike_dde`**(*coords*, *coeffs*, *noll_index*)

Computes Direction Dependent Effects by evaluating Zernicke Polynomials defined by coefficients `coeffs` and noll indexes `noll_index` at the specified coordinates `coords`.

Decomposition of a voxel beam cube into Zernicke polynomial coefficients can be achieved through the use of the eidos package.

Parameters **coords** : `dask.array.Array`

Float coordinates at which to evaluate the zernike polynomials. Has shape `(3, source, time, ant, chan)`. The three components in the first dimension represent l, m and frequency coordinates, respectively.

coeffs : `dask.array.Array`

complex Zernicke polynomial coefficients. Has shape `(ant, chan, corr_1, ..., corr_n, poly)` where `poly` is the number of polynomial coefficients and `corr_1, ..., corr_n` are a variable number of correlation dimensions.

noll_index : `dask.array.Array`

Noll index associated with each polynomial coefficient. Has shape `(ant, chan, corr_1, ..., corr_n, poly)`.

Returns **dde** : `dask.array.Array`

complex values with shape `(source, time, ant, chan, corr_1, ..., corr_n)`

`africanus.rime.dask.`**`wsclean_predict`**(*uvw*, *lm*, *source_type*, *flux*, *coeffs*, *log_poly*, *ref_freq*, *gauss_shape*, *frequency*)

Predict visibilities from a WSClean sky model.

Parameters **uvw** : `dask.array.Array`

UVW coordinates of shape `(row, 3)`

lm : `dask.array.Array`

Source LM coordinates of shape `(source, 2)`, in radians. Derived from the `Ra` and `Dec` fields.

source_type : `dask.array.Array`

Strings defining the source type of shape `(source,)`. Should be either `"POINT"` or `"GAUSSIAN"`. Contains the `Type` field.

flux : `dask.array.Array`

Source flux of shape `(source,)`. Contains the `I` field.

coeffs : `dask.array.Array`

Source Polynomial coefficients of shape `(source, coeffs)`. Contains the `SpectralIndex` field.

log_poly : `dask.array.Array`

Source polynomial type of shape `(source,)`. If True, logarithmic polynomials are used. If False, standard polynomials are used. Contains the `LogarithmicSI` field.

ref_freq : `dask.array.Array`

Source Reference frequency of shape `(source,)`. Contains the `ReferenceFrequency` field.

gauss_shape : `dask.array.Array`

Gaussian shape parameters of shape `(source, 3)` used when the corresponding `source_type` is `"GAUSSIAN"`. The 3 components should contain the `MajorAxis`, `MinorAxis` and `Orientation` fields in radians, respectively.

frequency : `dask.array.Array`

Frequency of shape `(chan,)`.

Returns **visibilities** : `dask.array.Array`

Complex visibilities of shape `(row, chan, 1)`

5.2 Direct Fourier Transform

Functions used to compute the discretised direct Fourier transform (DFT) for an ideal interferometer. The DFT for an ideal interferometer is defined as

$$V(u,v,w) = \int B(l,m) e^{-2\pi i(ul+vm+w(n-1))} \frac{dldm}{n}$$

where u, v, w are data space coordinates and where visibilities V have been obtained. The l, m, n are signal space coordinates at which we wish to reconstruct the signal B. Note that the signal correspondes to the brightness matrix and not the Stokes parameters. We adopt the convention where we absorb the fixed coordinate n in the denominator into the image. Note that the data space coordinates have an implicit dependence on frequency and time and that the image has an implicit dependence on frequency. The discretised form of the DFT can be written as

$$V(u,v,w) = \sum_s e^{-2\pi i(ul_s+vm_s+w(n_s-1))} \cdot B_s$$

where s labels the source (or pixel) location. If only a single correlation is present $B = I$, this can be cast into a matrix equation as follows

$$V = RI$$

where R is the operator that maps an image to visibility space. This mapping is implemented by the *im_to_vis()* function. If multiple correlations are present then each one is mapped to its corresponding visibility. An imaging algorithm also requires the adjoint denoted $R^\dagger$ which is simply the complex conjugate transpose of R. The dirty image is obtained by applying the adjoint operator to the visibilities

$$I^D = R^\dagger V$$

This is implemented by the *vis_to_im()* function. Note that an imaging algorithm using these operators will actually reconstruct $\frac{I}{n}$ but that it is trivial to obtain I since n is known at each location in the image.

5.2.1 Numpy

im_to_vis(image, uvw, lm, frequency[, ...])	Computes the discrete image to visibility mapping of an ideal interferometer:
vis_to_im(vis, uvw, lm, frequency, flags[, ...])	Computes visibility to image mapping of an ideal interferometer:

africanus.dft.**im_to_vis**(*image*, *uvw*, *lm*, *frequency*, *convention='fourier'*, *dtype=None*)

Computes the discrete image to visibility mapping of an ideal interferometer:

$$\sum_s e^{-2\pi i(ul_s + vm_s + w(n_s - 1))} \cdot I_s$$

Parameters **image** : numpy.ndarray

image of shape `(source, chan, corr)` The brighness matrix in each pixel (flatten 2D array per channel and corr). Note not Stokes terms

uvw : numpy.ndarray

uvw coordinates of shape `(row, 3)` with u, v and w components in the last dimension.

lm : numpy.ndarray

lm coordinates of shape `(source, 2)` with l and m components in the last dimension.

frequency : numpy.ndarray

frequencies of shape `(chan,)`

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

dtype : np.dtype, optional

Datatype of result. Should be either np.complex64 or np.complex128. If `None`, `numpy.result_type()` is used to infer the data type from the inputs.

Returns **visibilties** : numpy.ndarray

complex of shape `(row, chan, corr)`

africanus.dft.**vis_to_im**(*vis*, *uvw*, *lm*, *frequency*, *flags*, *convention='fourier'*, *dtype=None*)

Computes visibility to image mapping of an ideal interferometer:

$$\sum_k e^{2\pi i(u_k l + v_k m + w_k(n-1))} \cdot V_k$$

Parameters **vis** : numpy.ndarray

visibilities of shape `(row, chan, corr)` Visibilities corresponding to brightness terms. Note the dirty images produced do not necessarily correspond to Stokes terms and need to be converted.

uvw : numpy.ndarray

uvw coordinates of shape `(row, 3)` with u, v and w components in the last dimension.

lm : numpy.ndarray

lm coordinates of shape `(source, 2)` with l and m components in the last dimension.

frequency : numpy.ndarray

frequencies of shape `(chan,)`

flags : numpy.ndarray

Boolean array of shape `(row, chan, corr)` Note that if one correlation is flagged we discard all of them otherwise we end up irretrievably mixing Stokes terms.

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

dtype : np.dtype, optional

Datatype of result. Should be either np.float32 or np.float64. If `None`, numpy.result_type() is used to infer the data type from the inputs.

Returns **image** : numpy.ndarray

float of shape `(source, chan, corr)`

5.2.2 Dask

im_to_vis(image, uvw, lm, frequency[, ...])	Computes the discrete image to visibility mapping of an ideal interferometer:
vis_to_im(vis, uvw, lm, frequency, flags[, ...])	Computes visibility to image mapping of an ideal interferometer:

africanus.dft.dask.**im_to_vis**(*image*, *uvw*, *lm*, *frequency*, *convention='fourier'*, *dtype=<MagicMock id='140420490449872'>*)

Computes the discrete image to visibility mapping of an ideal interferometer:

$$\sum_s e^{-2\pi i(ul_s + vm_s + w(n_s-1))} \cdot I_s$$

Parameters **image** : dask.array.Array

image of shape `(source, chan, corr)` The brighness matrix in each pixel (flatten 2D array per channel and corr). Note not Stokes terms

uvw : dask.array.Array

uvw coordinates of shape `(row, 3)` with u, v and w components in the last dimension.

lm : `dask.array.Array`

lm coordinates of shape `(source, 2)` with l and m components in the last dimension.

frequency : `dask.array.Array`

frequencies of shape `(chan,)`

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

dtype : np.dtype, optional

Datatype of result. Should be either np.complex64 or np.complex128. If `None`, `numpy.result_type()` is used to infer the data type from the inputs.

Returns **visibilties** : `dask.array.Array`

complex of shape `(row, chan, corr)`

africanus.dft.dask.**vis_to_im**(*vis*, *uvw*, *lm*, *frequency*, *flags*, *convention='fourier'*, *dtype=<MagicMock id='140204489888912'>*)

Computes visibility to image mapping of an ideal interferometer:

$$\sum_k e^{2\pi i(u_k l + v_k m + w_k(n-1))} \cdot V_k$$

Parameters **vis** : `dask.array.Array`

visibilities of shape `(row, chan, corr)` Visibilities corresponding to brightness terms. Note the dirty images produced do not necessarily correspond to Stokes terms and need to be converted.

uvw : `dask.array.Array`

uvw coordinates of shape `(row, 3)` with u, v and w components in the last dimension.

lm : `dask.array.Array`

lm coordinates of shape `(source, 2)` with l and m components in the last dimension.

frequency : `dask.array.Array`

frequencies of shape `(chan,)`

flags : `dask.array.Array`

Boolean array of shape `(row, chan, corr)` Note that if one correlation is flagged we discard all of them otherwise we end up irretrievably mixing Stokes terms.

convention : {'fourier', 'casa'}

Uses the $e^{-2\pi i}$ sign convention if `fourier` and $e^{2\pi i}$ if `casa`.

dtype : np.dtype, optional

Datatype of result. Should be either np.float32 or np.float64. If `None`, `numpy.result_type()` is used to infer the data type from the inputs.

Returns **image** : `dask.array.Array`

float of shape `(source, chan, corr)`

5.3 Gridding and Degridding

This section contains routines for

1. Gridding complex visibilities onto an image.
2. Degridding complex visibilities from an image.

5.3.1 Nifty

Dask wrappers around Nifty's Gridder.

Dask

grid_config([nx, ny, eps, cell_size_x, ...])	Returns a wrapper around a NIFTY GridderConfiguration object.
grid(vis, uvw, flags, weights, frequencies, ...)	Grids the supplied visibilities in parallel.
dirty(grid, grid_config)	Computes the dirty image from gridded visibilities and the gridding configuration.
degrid(grid, uvw, flags, weights, ...[, ...])	Degrids the visibilities from the supplied grid in parallel.
model(image, grid_config)	Computes model visibilities from an image and a gridding configuration.

africanus.gridding.nifty.dask.**grid_config**(*nx=1024*, *ny=1024*, *eps=2e-13*, *cell_size_x=2.0*, *cell_size_y=2.0*)

Returns a wrapper around a NIFTY GridderConfiguration object.

Parameters **nx** : int, optional

Number of X pixels in the grid. Defaults to 1024.

ny : int, optional

Number of Y pixels in the grid. Defaults to 1024.

cell_size_x : float, optional

Cell size of the X pixel in arcseconds. Defaults to 2.0.

cell_size_y : float, optional

Cell size of the Y pixel in arcseconds. Defaults to 2.0.

eps : float

Gridder accuracy error. Defaults to 2e-13

Returns **grid_config** : `GridderConfigWrapper`

The NIFTY Gridder Configuration

africanus.gridding.nifty.dask.**grid**(*vis*, *uvw*, *flags*, *weights*, *frequencies*, *grid_config*, *wmin=-1e+30*, *wmax=1e+30*, *streams=None*)

Grids the supplied visibilities in parallel. Note that a grid is create for each visibility chunk.

Parameters **vis** : `dask.array.Array`

visibilities of shape `(row, chan, corr)`

uvw : dask.array.Array

uvw coordinates of shape (row, 3)

flags : dask.array.Array

flags of shape (row, chan, corr)

weights : dask.array.Array

weights of shape (row, chan, corr).

frequencies : dask.array.Array

frequencies of shape (chan,)

grid_config : GridderConfigWrapper

Gridding Configuration

wmin : float

Minimum W coordinate to grid. Defaults to -1e30.

wmax : float

Maximum W coordinate to grid. Default to 1e30.

streams : int, optional

Number of parallel gridding operations. Default to None, in which case as many grids as visibility chunks will be created.

Returns **grid** : dask.array.Array

grid of shape (ny, nx, corr)

africanus.gridding.nifty.dask.**dirty**(*grid*, *grid_config*)

Computes the dirty image from gridded visibilities and the gridding configuration.

Parameters **grid** : dask.array.Array

Gridded visibilities of shape (nv, nu, ncorr)

grid_config : GridderConfigWrapper

Gridding configuration

Returns **dirty** : dask.array.Array

dirty image of shape (ny, nx, corr)

africanus.gridding.nifty.dask.**degrid**(*grid*, *uvw*, *flags*, *weights*, *frequencies*, *grid_config*, *wmin=-1e+30*, *wmax=1e+30*)

Degrids the visibilities from the supplied grid in parallel.

Parameters **grid** : dask.array.Array

gridded visibilities of shape (ny, nx, corr)

uvw : dask.array.Array

uvw coordinates of shape (row, 3)

flags : dask.array.Array

flags of shape (row, chan, corr)

weights : dask.array.Array

weights of shape (row, chan, corr). Currently unsupported and ignored.

frequencies : dask.array.Array

frequencies of shape (chan,)

grid_config : GridderConfigWrapper

Gridding Configuration

wmin : float

Minimum W coordinate to grid. Defaults to -1e30.

wmax : float

Maximum W coordinate to grid. Default to 1e30.

Returns **grid** : dask.array.Array

grid of shape (ny, nx, corr)

africanus.gridding.nifty.dask.**model**(*image*, *grid_config*)

Computes model visibilities from an image and a gridding configuration.

Parameters **image** : dask.array.Array

Image of shape (ny, nx, corr).

grid_config : GridderConfigWrapper

nifty gridding configuration object

Returns **model_vis** : dask.array.Array

Model visibilities of shape (nu, nv, corr).

5.3.2 wgridder

Wrappers around ‘ducc.wgridder <https://gitlab.mpcdf.mpg.de/mtr/ducc>‘_.

Numpy

dirty(uvw, freq, vis, freq_bin_idx, . . . [, . . .])	Compute visibility to image mapping using ducc gridder i.e.
model(uvw, freq, image, freq_bin_idx, . . . [, . . .])	Compute image to visibility mapping using ducc degridder i.e.
residual(uvw, freq, image, vis, . . . [, . . .])	Compute residual image given a model and visibilities using ducc degridder i.e.

africanus.gridding.wgridder.**dirty**(*uvw*, *freq*, *vis*, *freq_bin_idx*, *freq_bin_counts*, *nx*, *ny*, *cell*, *weights=None*, *flag=None*, *celly=None*, *epsilon=1e-05*, *nthreads=1*, *do_wstacking=True*)

Compute visibility to image mapping using ducc gridder i.e.

$$I^D = R^\dagger \Sigma^{-1} V$$

where $R^\dagger$ is an implicit gridding operator, V denotes visibilities of shape (row, chan) and I^D is the dirty image of shape (band, nx, ny).

The number of imaging bands (band) must be less than or equal to the number of channels (chan) at which the data were obtained. The mapping from (chan) to (band) is described by freq_bin_idx and freq_bin_counts as described below.

Note that, if self adjoint gridding and degridding operators are required then `weights` should be the square root of what is typically referred to as imaging weights and should also be passed into the degridder. In this case, the data needs to be pre-whitened.

Parameters **uvw** : `numpy.ndarray`

uvw coordinates at which visibilities were obtained with shape `(row, 3)`.

freq : `numpy.ndarray`

Observational frequencies of shape `(chan,)`.

vis : `numpy.ndarray`

Visibilities of shape `(row,chan)`.

freq_bin_idx : `numpy.ndarray`

Starting indices of frequency bins for each imaging band of shape `(band,)`.

freq_bin_counts : `numpy.ndarray`

The number of channels in each imaging band of shape `(band,)`.

cell : float

The cell size of a pixel along the x direction in radians.

weights : `numpy.ndarray`, optional

Imaging weights of shape `(row, chan)`.

flag: :class:'numpy.ndarray', optional

Flags of shape `(row,chan)`. Will only process visibilities for which flag!=0

celly : float, optional

The cell size of a pixel along the y direction in radians. By default same as cell size along x direction.

epsilon : float, optional

The precision of the gridder with respect to the direct Fourier transform. By deafult, this is set to `1e-5` for single precision and `1e-7` for double precision.

nthreads : int, optional

The number of threads to use. Defaults to one. If set to zero will use all available cores.

do_wstacking : bool, optional

Whether to correct for the w-term or not. Defaults to True

Returns **model** : `numpy.ndarray`

Dirty image corresponding to visibilities of shape `(nband, nx, ny)`.

`africanus.gridding.wgridder.`**`model`**(*uvw*, *freq*, *image*, *freq_bin_idx*, *freq_bin_counts*, *cell*, *weights=None*, *flag=None*, *celly=None*, *epsilon=1e-05*, *nthreads=1*, *do_wstacking=True*)

Compute image to visibility mapping using ducc degridder i.e.

$$V = Rx$$

where R is an implicit degridding operator, V denotes visibilities of shape `(row, chan)` and x is the image of shape `(band, nx, ny)`.

The number of imaging bands (band) has to be less than or equal to the number of channels (chan) at which the data were obtained. The mapping from (chan) to (band) is described by freq_bin_idx and freq_bin_counts as described below.

There is an option to provide weights during degridding to cater for self adjoint gridding and degridding operators. In this case weights should actually be the square root of what is typically referred to as imaging weights. In this case the degridder computes the whitened model visibilities i.e.

$$V = \Sigma^{-\frac{1}{2}} Rx$$

where Σ refers to the inverse of the weights (i.e. the data covariance matrix when using natural weighting).

Parameters **uvw** : numpy.ndarray

uvw coordinates at which visibilities were obtained with shape (row, 3).

freq : numpy.ndarray

Observational frequencies of shape (chan,).

model : numpy.ndarray

Model image to degrid of shape (nband, nx, ny).

freq_bin_idx : numpy.ndarray

Starting indices of frequency bins for each imaging band of shape (band,).

freq_bin_counts : numpy.ndarray

The number of channels in each imaging band of shape (band,).

cell : float

The cell size of a pixel along the x direction in radians.

weights : numpy.ndarray, optional

Imaging weights of shape (row, chan).

flag: :class:'numpy.ndarray', optional

Flags of shape (row,chan). Will only process visibilities for which flag!=0

celly : float, optional

The cell size of a pixel along the y direction in radians. By default same as cell size along x direction.

epsilon : float, optional

The precision of the gridder with respect to the direct Fourier transform. By deafult, this is set to 1e-5 for single precision and 1e-7 for double precision.

nthreads : int, optional

The number of threads to use. Defaults to one. If set to zero will use all available cores.

do_wstacking : bool, optional

Whether to correct for the w-term or not. Defaults to True

Returns **vis** : numpy.ndarray

Visibilities corresponding to model of shape (row,chan).

africanus.gridding.wgridder.**residual**(*uvw*, *freq*, *image*, *vis*, *freq_bin_idx*, *freq_bin_counts*, *cell*, *weights=None*, *flag=None*, *celly=None*, *epsilon=1e-05*, *nthreads=1*, *do_wstacking=True*)

Compute residual image given a model and visibilities using ducc degridder i.e.

$$I^R = R^\dagger \Sigma^{-1}(V - Rx)$$

where R is an implicit degridding operator, V denotes visibilities of shape `(row, chan)` and x is the image of shape `(band, nx, ny)`.

The number of imaging bands `(band)` must be less than or equal to the number of channels `(chan)` at which the data were obtained. The mapping from `(chan)` to `(band)` is described by `freq_bin_idx` and `freq_bin_counts` as described below.

Note that, if the gridding and degridding operators both apply the square root of the imaging weights then the visibilities that are passed in should be pre-whitened. In this case the function computes

$$I^R = R^\dagger \Sigma^{-\frac{1}{2}}(\tilde{V} - \Sigma^{-\frac{1}{2}} Rx)$$

which is identical to the above expression if $\tilde{V} = \Sigma^{-\frac{1}{2}} V$.

Parameters **uvw** : numpy.ndarray

uvw coordinates at which visibilities were obtained with shape `(row, 3)`.

freq : numpy.ndarray

Observational frequencies of shape `(chan,)`.

model : numpy.ndarray

Model image to degrid of shape `(band, nx, ny)`.

vis : numpy.ndarray

Visibilities of shape `(row,chan)`.

weights : numpy.ndarray

Imaging weights of shape `(row, chan)`.

freq_bin_idx : numpy.ndarray

Starting indices of frequency bins for each imaging band of shape `(band,)`.

freq_bin_counts : numpy.ndarray

The number of channels in each imaging band of shape `(band,)`.

cell : float

The cell size of a pixel along the x direction in radians.

flag: :class:‘numpy.ndarray‘, optional

Flags of shape `(row,chan)`. Will only process visibilities for which flag!=0

celly : float, optional

The cell size of a pixel along the y direction in radians. By default same as cell size along x direction.

nu : int, optional

The number of pixels in the padded grid along the x direction. Chosen automatically by default.

nv : int, optional

The number of pixels in the padded grid along the y direction. Chosen automatically by default.

epsilon : float, optional

The precision of the gridder with respect to the direct Fourier transform. By deafult, this is set to `1e-5` for single precision and `1e-7` for double precision.

nthreads : int, optional

The number of threads to use. Defaults to one.

do_wstacking : bool, optional

Whether to correct for the w-term or not. Defaults to True

Returns **residual** : `numpy.ndarray`

Residual image corresponding to `model` of shape `(band, nx, ny)`.

Dask

dirty(uvw, freq, vis, freq_bin_idx, ... [, ...])	Compute visibility to image mapping using ducc gridder i.e.
model(uvw, freq, image, freq_bin_idx, ... [, ...])	Compute image to visibility mapping using ducc degridder i.e.
residual(uvw, freq, image, vis, ... [, ...])	Compute residual image given a model and visibilities using ducc degridder i.e.

africanus.gridding.wgridder.dask.**dirty**(*uvw*, *freq*, *vis*, *freq_bin_idx*, *freq_bin_counts*, *nx*, *ny*, *cell*, *weights=None*, *flag=None*, *celly=None*, *epsilon=1e-05*, *nthreads=1*, *do_wstacking=True*)

Compute visibility to image mapping using ducc gridder i.e.

$$I^D = R^\dagger \Sigma^{-1} V$$

where $R^\dagger$ is an implicit gridding operator, V denotes visibilities of shape `(row, chan)` and I^D is the dirty image of shape `(band, nx, ny)`.

The number of imaging bands `(band)` must be less than or equal to the number of channels `(chan)` at which the data were obtained. The mapping from `(chan)` to `(band)` is described by `freq_bin_idx` and `freq_bin_counts` as described below.

Note that, if self adjoint gridding and degridding operators are required then `weights` should be the square root of what is typically referred to as imaging weights and should also be passed into the degridder. In this case, the data needs to be pre-whitened.

Parameters **uvw** : `dask.array.Array`

uvw coordinates at which visibilities were obtained with shape `(row, 3)`.

freq : `dask.array.Array`

Observational frequencies of shape `(chan,)`.

vis : `dask.array.Array`

Visibilities of shape `(row,chan)`.

freq_bin_idx : `dask.array.Array`

Starting indices of frequency bins for each imaging band of shape `(band,)`.

freq_bin_counts : `dask.array.Array`

The number of channels in each imaging band of shape `(band,)`.

cell : float

The cell size of a pixel along the x direction in radians.

weights : `dask.array.Array`, optional

Imaging weights of shape `(row, chan)`.

flag: :class:'dask.array.Array', optional

Flags of shape `(row,chan)`. Will only process visibilities for which flag!=0

celly : float, optional

The cell size of a pixel along the y direction in radians. By default same as cell size along x direction.

epsilon : float, optional

The precision of the gridder with respect to the direct Fourier transform. By deafult, this is set to `1e-5` for single precision and `1e-7` for double precision.

nthreads : int, optional

The number of threads to use. Defaults to one. If set to zero will use all available cores.

do_wstacking : bool, optional

Whether to correct for the w-term or not. Defaults to True

Returns **model** : `dask.array.Array`

Dirty image corresponding to visibilities of shape `(nband, nx, ny)`.

`africanus.gridding.wgridder.dask.`**`model`**(*uvw*, *freq*, *image*, *freq_bin_idx*, *freq_bin_counts*, *cell*, *weights=None*, *flag=None*, *celly=None*, *epsilon=1e-05*, *nthreads=1*, *do_wstacking=True*)

Compute image to visibility mapping using ducc degridder i.e.

$$V = Rx$$

where R is an implicit degridding operator, V denotes visibilities of shape `(row, chan)` and x is the image of shape `(band, nx, ny)`.

The number of imaging bands `(band)` has to be less than or equal to the number of channels `(chan)` at which the data were obtained. The mapping from `(chan)` to `(band)` is described by `freq_bin_idx` and `freq_bin_counts` as described below.

There is an option to provide weights during degridding to cater for self adjoint gridding and degridding operators. In this case `weights` should actually be the square root of what is typically referred to as imaging weights. In this case the degridder computes the whitened model visibilities i.e.

$$V = \Sigma^{-\frac{1}{2}} Rx$$

where Σ refers to the inverse of the weights (i.e. the data covariance matrix when using natural weighting).

Parameters **uvw** : `dask.array.Array`

uvw coordinates at which visibilities were obtained with shape `(row, 3)`.

freq : `dask.array.Array`

Observational frequencies of shape `(chan,)`.

model : `dask.array.Array`

Model image to degrid of shape `(nband, nx, ny)`.

freq_bin_idx : `dask.array.Array`

Starting indices of frequency bins for each imaging band of shape `(band,)`.

freq_bin_counts : `dask.array.Array`

The number of channels in each imaging band of shape `(band,)`.

cell : float

The cell size of a pixel along the x direction in radians.

weights : `dask.array.Array`, optional

Imaging weights of shape `(row, chan)`.

flag: :class:'dask.array.Array', optional

Flags of shape `(row,chan)`. Will only process visibilities for which flag!=0

celly : float, optional

The cell size of a pixel along the y direction in radians. By default same as cell size along x direction.

epsilon : float, optional

The precision of the gridder with respect to the direct Fourier transform. By deafult, this is set to `1e-5` for single precision and `1e-7` for double precision.

nthreads : int, optional

The number of threads to use. Defaults to one. If set to zero will use all available cores.

do_wstacking : bool, optional

Whether to correct for the w-term or not. Defaults to True

Returns **vis** : `dask.array.Array`

Visibilities corresponding to `model` of shape `(row,chan)`.

africanus.gridding.wgridder.dask.**residual**(*uvw*, *freq*, *image*, *vis*, *freq_bin_idx*, *freq_bin_counts*, *cell*, *weights=None*, *flag=None*, *celly=None*, *epsilon=1e-05*, *nthreads=1*, *do_wstacking=True*)

Compute residual image given a model and visibilities using ducc degridder i.e.

$$I^R = R^\dagger \Sigma^{-1} (V - Rx)$$

where R is an implicit degridding operator, V denotes visibilities of shape `(row, chan)` and x is the image of shape `(band, nx, ny)`.

The number of imaging bands `(band)` must be less than or equal to the number of channels `(chan)` at which the data were obtained. The mapping from `(chan)` to `(band)` is described by `freq_bin_idx` and `freq_bin_counts` as described below.

Note that, if the gridding and degridding operators both apply the square root of the imaging weights then the visibilities that are passed in should be pre-whitened. In this case the function computes

$$I^R = R^\dagger \Sigma^{-\frac{1}{2}} (\tilde{V} - \Sigma^{-\frac{1}{2}} Rx)$$

which is identical to the above expression if $\tilde{V} = \Sigma^{-\frac{1}{2}} V$.

Parameters **uvw** : dask.array.Array

uvw coordinates at which visibilities were obtained with shape `(row, 3)`.

freq : dask.array.Array

Observational frequencies of shape `(chan,)`.

model : dask.array.Array

Model image to degrid of shape `(band, nx, ny)`.

vis : dask.array.Array

Visibilities of shape `(row,chan)`.

weights : dask.array.Array

Imaging weights of shape `(row, chan)`.

freq_bin_idx : dask.array.Array

Starting indices of frequency bins for each imaging band of shape `(band,)`.

freq_bin_counts : dask.array.Array

The number of channels in each imaging band of shape `(band,)`.

cell : float

The cell size of a pixel along the x direction in radians.

flag: :class:‘dask.array.Array‘, optional

Flags of shape `(row,chan)`. Will only process visibilities for which flag!=0

celly : float, optional

The cell size of a pixel along the y direction in radians. By default same as cell size along x direction.

nu : int, optional

The number of pixels in the padded grid along the x direction. Chosen automatically by default.

nv : int, optional

The number of pixels in the padded grid along the y direction. Chosen automatically by default.

epsilon : float, optional

The precision of the gridder with respect to the direct Fourier transform. By deafult, this is set to `1e-5` for single precision and `1e-7` for double precision.

nthreads : int, optional

The number of threads to use. Defaults to one.

do_wstacking : bool, optional

Whether to correct for the w-term or not. Defaults to True

Returns **residual** : dask.array.Array

Residual image corresponding to `model` of shape `(band, nx, ny)`.

5.3.3 Utilities

estimate_cell_size(u, v, wavelength[, ...])	Estimate the cell size in arcseconds given baseline u and v coordinates, as well as the wavelengths, λ.

africanus.gridding.util.**estimate_cell_size**(*u*, *v*, *wavelength*, *factor=3.0*, *ny=None*, *nx=None*)

Estimate the cell size in arcseconds given baseline u and v coordinates, as well as the wavelengths, λ.

The cell size is computed as:

$$\Delta u = 1.0 / (2 \times \text{ factor } \times \max(|u|) / \min(\lambda))$$
$$\Delta v = 1.0 / (2 \times \text{ factor } \times \max(|v|) / \min(\lambda))$$

If ny and nx are provided the following checks are performed and exceptions are raised on failure:

$$\Delta u * \text{ ny } \leq \min(\lambda) / \min(|u|)$$
$$\Delta v * \text{ nx } \leq \min(\lambda) / \min(|v|)$$

Parameters **u** : numpy.ndarray or float

Maximum u coordinate in metres.

v : numpy.ndarray or float

Maximum v coordinate in metres.

wavelength : numpy.ndarray or float

Wavelengths, in metres.

factor : float, optional

Scaling factor

ny : int, optional

Grid y dimension

nx : int, optional

Grid x dimension

Returns numpy.ndarray

Cell size of u and v in arcseconds with shape (2,)

Raises **ValueError**

If the cell size criteria are not matched.

5.4 Deconvolution Algorithms

africanus.deconv.hogbom.**hogbom_clean**(*dirty*, *psf*, *gamma=0.1*, *threshold='default'*, *niter='default'*)

Performs Hogbom Clean on the dirty image given the psf.

Parameters **dirty** : np.ndarray

float64 dirty image of shape (ny, nx)

psf : np.ndarray

float64 Point Spread Function of shape (2*ny, 2*nx)

gamma (optional) float

the gain factor (must be less than one)

threshold (optional) : float or str

the threshold to clean to

niter (optional : integer

the maximum number of iterations allowed

Returns np.ndarray

float64 clean image of shape (ny, nx)

np.ndarray

float64 residual image of shape (ny, nx)

5.5 Coordinate Transforms

5.5.1 Numpy

radec_to_lm(radec[, phase_centre])	Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.
radec_to_lmn(radec[, phase_centre])	Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.
lm_to_radec(lm[, phase_centre])	Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.
lmn_to_radec(lmn[, phase_centre])	Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.

africanus.coordinates.**radec_to_lm**(*radec*, *phase_centre=None*)

Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.

$$l = \cos\,\delta \sin\,\Delta\alpha \tag{5.1}$$

$$m = \sin\,\delta \cos\,\delta 0 - \cos\delta \sin\delta 0 \cos\Delta\alpha \tag{5.2}$$

$$n = \sqrt{1 - l^2 - m^2} - 1 \tag{5.3}$$

where $\Delta\alpha = \alpha - \alpha 0$ is the difference between the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **radec** : numpy.ndarray

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

phase_centre : numpy.ndarray, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns numpy.ndarray

lm Direction Cosines of shape `(coord, 2)`

africanus.coordinates.**radec_to_lmn**(*radec*, *phase_centre=None*)

Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.

$$l = \cos\,\delta \sin\,\Delta\alpha \tag{5.4}$$

$$m = \sin\,\delta \cos\,\delta 0 - \cos\delta \sin\delta 0 \cos\Delta\alpha \tag{5.5}$$

$$n = \sqrt{1 - l^2 - m^2} - 1 \tag{5.6}$$

where $\Delta\alpha = \alpha - \alpha 0$ is the difference between the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **radec** : numpy.ndarray

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

phase_centre : numpy.ndarray, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns numpy.ndarray

lm Direction Cosines of shape `(coord, 3)`

africanus.coordinates.**lm_to_radec**(*lm*, *phase_centre=None*)

Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.

$$\delta = \arcsin\left(m\cos\delta 0 + n\sin\delta 0\right) \tag{5.7}$$

$$\alpha = \arctan\left(\frac{l}{n\cos\delta 0 - m\sin\delta 0}\right) \tag{5.8}$$

$$\tag{5.9}$$

where α is the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **lm** : numpy.ndarray

lm Direction Cosines of shape `(coord, 2)`

phase_centre : numpy.ndarray, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns numpy.ndarray

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

africanus.coordinates.**lmn_to_radec**(*lmn*, *phase_centre=None*)

Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.

$$\delta = \arcsin\left(m\cos\delta 0 + n\sin\delta 0\right) \tag{5.10}$$

$$\alpha = \arctan\left(\frac{l}{n\cos\delta 0 - m\sin\delta 0}\right) \tag{5.11}$$

$$\tag{5.12}$$

where α is the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **lmn** : numpy.ndarray

lm Direction Cosines of shape `(coord, 3)`

phase_centre : numpy.ndarray, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns numpy.ndarray

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

5.5.2 Dask

radec_to_lm(radec[, phase_centre])	Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.
radec_to_lmn(radec[, phase_centre])	Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.
lm_to_radec(lm[, phase_centre])	Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.
lmn_to_radec(lmn[, phase_centre])	Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.

africanus.coordinates.dask.**radec_to_lm**(*radec*, *phase_centre=None*)

Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to the Phase Centre.

$$l = \cos\,\delta \sin\,\Delta\alpha \tag{5.13}$$

$$m = \sin\,\delta \cos\,\delta 0 - \cos\delta \sin\delta 0 \cos\Delta\alpha \tag{5.14}$$

$$n = \sqrt{1 - l^2 - m^2} - 1 \tag{5.15}$$

where $\Delta\alpha = \alpha - \alpha 0$ is the difference between the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **radec** : dask.array.Array

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

phase_centre : dask.array.Array, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns dask.array.Array

lm Direction Cosines of shape `(coord, 2)`

africanus.coordinates.dask.**radec_to_lmn**(*radec*, *phase_centre=None*)

Converts Right-Ascension/Declination coordinates in radians to a Direction Cosine lm coordinates, relative to

the Phase Centre.

$$l = \cos\,\delta \sin\,\Delta\alpha \tag{5.16}$$
$$m = \sin\,\delta \cos\,\delta 0 - \cos\delta \sin\delta 0 \cos\Delta\alpha \tag{5.17}$$
$$n = \sqrt{1 - l^2 - m^2} - 1 \tag{5.18}$$

where $\Delta\alpha = \alpha - \alpha 0$ is the difference between the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **radec** : `dask.array.Array`

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

phase_centre : `dask.array.Array`, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns `dask.array.Array`

lm Direction Cosines of shape `(coord, 3)`

`africanus.coordinates.dask.`**`lm_to_radec`**(*lm*, *phase_centre=None*)

Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.

$$\delta = \arcsin\left(m\cos\delta 0 + n\sin\delta 0\right) \tag{5.19}$$
$$\alpha = \arctan\left(\frac{l}{n\cos\delta 0 - m\sin\delta 0}\right) \tag{5.20}$$
$$\tag{5.21}$$

where α is the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **lm** : `dask.array.Array`

lm Direction Cosines of shape `(coord, 2)`

phase_centre : `dask.array.Array`, optional

radec coordinates of the Phase Centre. Shape `(2,)`

Returns `dask.array.Array`

radec coordinates of shape `(coord, 2)` where Right-Ascension and Declination are in the last 2 components, respectively.

`africanus.coordinates.dask.`**`lmn_to_radec`**(*lmn*, *phase_centre=None*)

Convert Direction Cosine lm coordinates to Right Ascension/Declination coordinates in radians, relative to the Phase Centre.

$$\delta = \arcsin\left(m\cos\delta 0 + n\sin\delta 0\right) \tag{5.22}$$
$$\alpha = \arctan\left(\frac{l}{n\cos\delta 0 - m\sin\delta 0}\right) \tag{5.23}$$
$$\tag{5.24}$$

where α is the Right Ascension of each coordinate and the phase centre and $\delta 0$ is the Declination of the phase centre.

Parameters **lmn** : `dask.array.Array`

lm Direction Cosines of shape `(coord, 3)`

phase_centre : dask.array.Array, optional

radec coordinates of the Phase Centre. Shape (2,)

Returns dask.array.Array

radec coordinates of shape (coord, 2) where Right-Ascension and Declination are in the last 2 components, respectively.

5.6 Sky Model

Functionality related to the Sky Model.

5.6.1 Coherency Conversion

Utilities for converting back and forth between stokes parameters and correlations

Numpy

convert(input, input_schema, output_schema)	This function converts forward and backward from stokes I,Q,U,V to both linear XX,XY,YX,YY and circular RR, RL, LR, LL correlations.

africanus.model.coherency.**convert**(*input*, *input_schema*, *output_schema*)

This function converts forward and backward from stokes I,Q,U,V to both linear XX,XY,YX,YY and circular RR, RL, LR, LL correlations.

For example, we can convert from stokes parameters to linear correlations:

```
stokes.shape == (10, 4, 4)
corrs = convert(stokes, ["I", "Q", "U", "V"],
                [['XX', 'XY'], ['YX', 'YY']])

assert corrs.shape == (10, 4, 2, 2)
```

Or circular correlations to stokes:

```
vis.shape == (10, 4, 2, 2)

stokes = convert(vis, [['RR', 'RL'], ['LR', 'LL']],
                      ['I', 'Q', 'U', 'V'])

assert stokes.shape == (10, 4, 4)
```

input can output can be arbitrarily nested or ordered lists, but the appropriate inputs must be present to produce the requested outputs.

The elements of input and output may be strings or integers representing stokes parameters or correlations. See the Notes for a full list.

Parameters **input** : numpy.ndarray

Complex or floating point input data of shape (dim_1, ..., dim_n, icorr_1, ..., icorr_m)

input_schema : list of str or int

A schema describing the `icorr_1, ..., icorr_m` dimension of `input`. Must have the same shape as the last dimensions of `input`.

output_schema : list of str or int

A schema describing the `ocorr_1, ..., ocorr_n` dimension of the return value.

Returns **result** : `numpy.ndarray`

Result of shape `(dim_1, ..., dim_n, ocorr_1, ..., ocorr_m)` The type may be floating point or promoted to complex depending on the combinations in `output`.

Notes

Only stokes parameters, linear and circular correlations are currently handled, but the full list of id's and strings as defined in the CASA documentation is:

```
{{ Undefined: 0, I: 1, Q: 2, U: 3, V: 4, RR: 5, RL: 6, LR: 7, LL: 8,
    XX: 9, XY: 10, YX: 11, YY: 12, RX: 13, RY: 14, LX: 15, LY: 16,
    XR: 17, XL: 18, YR: 19, YL: 20, PP: 21, PQ: 22, QP: 23, QQ:
    24, RCircular: 25, LCircular: 26, Linear: 27, Ptotal: 28,
    Plinear: 29, PFtotal: 30, PFlinear: 31, Pangle: 32 }}
```

Cuda

convert(inputs, input_schema, output_schema)	This function converts forward and backward from stokes `I,Q,U,V` to both linear `XX,XY,YX,YY` and circular `RR, RL, LR, LL` correlations.

africanus.model.coherency.cuda.**convert**(*inputs*, *input_schema*, *output_schema*)

This function converts forward and backward from stokes `I,Q,U,V` to both linear `XX,XY,YX,YY` and circular `RR, RL, LR, LL` correlations.

For example, we can convert from stokes parameters to linear correlations:

```
stokes.shape == (10, 4, 4)
corrs = convert(stokes, ["I", "Q", "U", "V"],
                [['XX', 'XY'], ['YX', 'YY']])

assert corrs.shape == (10, 4, 2, 2)
```

Or circular correlations to stokes:

```
vis.shape == (10, 4, 2, 2)

stokes = convert(vis, [['RR', 'RL'], ['LR', 'LL']],
                      ['I', 'Q', 'U', 'V'])

assert stokes.shape == (10, 4, 4)
```

`input` can `output` can be arbitrarily nested or ordered lists, but the appropriate inputs must be present to produce the requested outputs.

The elements of `input` and `output` may be strings or integers representing stokes parameters or correlations. See the Notes for a full list.

Parameters **input** : `cupy.ndarray`

Complex or floating point input data of shape `(dim_1, ..., dim_n, icorr_1, ..., icorr_m)`

input_schema : list of str or int

A schema describing the `icorr_1, ..., icorr_m` dimension of `input`. Must have the same shape as the last dimensions of `input`.

output_schema : list of str or int

A schema describing the `ocorr_1, ..., ocorr_n` dimension of the return value.

Returns **result** : `cupy.ndarray`

Result of shape `(dim_1, ..., dim_n, ocorr_1, ..., ocorr_m)` The type may be floating point or promoted to complex depending on the combinations in `output`.

Notes

Only stokes parameters, linear and circular correlations are currently handled, but the full list of id's and strings as defined in the CASA documentation is:

```
{{ Undefined: 0, I: 1, Q: 2, U: 3, V: 4, RR: 5, RL: 6, LR: 7, LL: 8,
    XX: 9, XY: 10, YX: 11, YY: 12, RX: 13, RY: 14, LX: 15, LY: 16,
    XR: 17, XL: 18, YR: 19, YL: 20, PP: 21, PQ: 22, QP: 23, QQ:
    24, RCircular: 25, LCircular: 26, Linear: 27, Ptotal: 28,
    Plinear: 29, PFtotal: 30, PFlinear: 31, Pangle: 32 }}
```

Dask

convert(input, input_schema, output_schema)	This function converts forward and backward from stokes `I,Q,U,V` to both linear `XX,XY,YX,YY` and circular `RR, RL, LR, LL` correlations.

africanus.model.coherency.dask.**convert**(*input*, *input_schema*, *output_schema*)

This function converts forward and backward from stokes `I,Q,U,V` to both linear `XX,XY,YX,YY` and circular `RR, RL, LR, LL` correlations.

For example, we can convert from stokes parameters to linear correlations:

```
stokes.shape == (10, 4, 4)
corrs = convert(stokes, ["I", "Q", "U", "V"],
                [['XX', 'XY'], ['YX', 'YY']])

assert corrs.shape == (10, 4, 2, 2)
```

Or circular correlations to stokes:

```
vis.shape == (10, 4, 2, 2)
```

(continues on next page)

(continued from previous page)

```
stokes = convert(vis, [['RR', 'RL'], ['LR', 'LL']],
                      ['I', 'Q', 'U', 'V'])

assert stokes.shape == (10, 4, 4)
```

`input` can `output` can be arbitrarily nested or ordered lists, but the appropriate inputs must be present to produce the requested outputs.

The elements of `input` and `output` may be strings or integers representing stokes parameters or correlations. See the Notes for a full list.

Parameters **input** : `dask.array.Array`

Complex or floating point input data of shape `(dim_1, ..., dim_n, icorr_1, ..., icorr_m)`

input_schema : list of str or int

A schema describing the `icorr_1, ..., icorr_m` dimension of `input`. Must have the same shape as the last dimensions of `input`.

output_schema : list of str or int

A schema describing the `ocorr_1, ..., ocorr_n` dimension of the return value.

Returns **result** : `dask.array.Array`

Result of shape `(dim_1, ..., dim_n, ocorr_1, ..., ocorr_m)` The type may be floating point or promoted to complex depending on the combinations in `output`.

Notes

Only stokes parameters, linear and circular correlations are currently handled, but the full list of id's and strings as defined in the CASA documentation is:

```
{{ Undefined: 0, I: 1, Q: 2, U: 3, V: 4, RR: 5, RL: 6, LR: 7, LL: 8,
    XX: 9, XY: 10, YX: 11, YY: 12, RX: 13, RY: 14, LX: 15, LY: 16,
    XR: 17, XL: 18, YR: 19, YL: 20, PP: 21, PQ: 22, QP: 23, QQ:
    24, RCircular: 25, LCircular: 26, Linear: 27, Ptotal: 28,
    Plinear: 29, PFtotal: 30, PFlinear: 31, Pangle: 32 }}
```

5.6.2 Spectral Model

Functionality for computing a Spectral Model.

Numpy

spectral_model(stokes, spi, ref_freq, frequency)	Compute a spectral model, per polarisation.

africanus.model.spectral.**spectral_model**(*stokes*, *spi*, *ref_freq*, *frequency*, *base=0*)

Compute a spectral model, per polarisation.

$$I(\lambda) = I_0 \prod_{i=1} (\lambda/\lambda_0)^{\alpha_i} \tag{5.25}$$

$$\ln(I(\lambda)) = \sum_{i=0} \alpha_i \ln(\lambda/\lambda_0)^i \text{ where } \alpha_0 = \ln I_0 \tag{5.26}$$

$$\log_{10}(I(\lambda)) = \sum_{i=0} \alpha_i \log_{10}(\lambda/\lambda_0)^i \text{ where } \alpha_0 = \log_{10} I_0 \tag{5.27}$$

$$\tag{5.28}$$

Parameters **stokes** : numpy.ndarray

Stokes parameters of shape `(source,)` or `(source, pol)`. If a `pol` dimension is present, then it must also be present on `spi`.

spi : numpy.ndarray

Spectral index of shape `(source, spi-comps)` or `(source, spi-comps, pol)`.

ref_freq : numpy.ndarray

Reference frequencies of shape `(source,)`

frequencies : numpy.ndarray

Frequencies of shape `(chan,)`

base : {"std", "log", "log10"} or {0, 1, 2} or list.

string or corresponding enumeration specifying the polynomial base. Defaults to 0.

If a list is provided, a polynomial base can be specified for each stokes parameter or polarisation in the `pol` dimension.

string specification of the base is only supported in python 3. while the corresponding integer enumerations are supported on all python versions.

Returns **spectral_model** : numpy.ndarray

Spectral Model of shape `(source, chan)` or `(source, chan, pol)`.

Dask

spectral_model(stokes, spi, ref_freq, ... [, ...])	Compute a spectral model, per polarisation.

africanus.model.spectral.dask.**spectral_model**(*stokes*, *spi*, *ref_freq*, *frequencies*, *base=0*)

Compute a spectral model, per polarisation.

$$I(\lambda) = I_0 \prod_{i=1} (\lambda/\lambda_0)^{\alpha_i} \tag{5.29}$$

$$\ln(I(\lambda)) = \sum_{i=0} \alpha_i \ln(\lambda/\lambda_0)^i \text{ where } \alpha_0 = \ln I_0 \tag{5.30}$$

$$\log_{10}(I(\lambda)) = \sum_{i=0} \alpha_i \log_{10}(\lambda/\lambda_0)^i \text{ where } \alpha_0 = \log_{10} I_0 \tag{5.31}$$

$$\tag{5.32}$$

Parameters **stokes** : dask.array.Array

Stokes parameters of shape `(source,)` or `(source, pol)`. If a `pol` dimension is present, then it must also be present on `spi`.

spi : `dask.array.Array`

Spectral index of shape `(source, spi-comps)` or `(source, spi-comps, pol)`.

ref_freq : `dask.array.Array`

Reference frequencies of shape `(source,)`

frequencies : `dask.array.Array`

Frequencies of shape `(chan,)`

base : {"std", "log", "log10"} or {0, 1, 2} or list.

string or corresponding enumeration specifying the polynomial base. Defaults to 0.

If a list is provided, a polynomial base can be specified for each stokes parameter or polarisation in the `pol` dimension.

string specification of the base is only supported in python 3. while the corresponding integer enumerations are supported on all python versions.

Returns **spectral_model** : `dask.array.Array`

Spectral Model of shape `(source, chan)` or `(source, chan, pol)`.

5.6.3 Spectral Index

Functionality related to the spectral index.

For example, we may want to compute the spectral indices of components in a sky model defined by

$$I(\nu) = I(\nu_0)\left(\frac{\nu}{\nu_0}\right)^{\alpha}$$

where ν are frequencies ay which we want to construct the intensity of a Stokes I image and the ν_0 is the corresponding reference frequency. The spectral index α determines how quickly the intensity grows or decays as a function of frequency. Given a list of model image components (preferably with the residuals added back in) we can recover the corresponding spectral indices and reference intensities using the *`fit_spi_components()`* function. This will also return a lower bound on the associated uncertainties on these components.

Numpy

`fit_spi_components`(data, weights, freqs, freq0)	Computes the spectral indices and the intensity at the reference frequency of a spectral index model:

`africanus.model.spi.`**`fit_spi_components`**(*data*, *weights*, *freqs*, *freq0*, *alphai=None*, *I0i=None*, *tol=0.0001*, *maxiter=100*)

Computes the spectral indices and the intensity at the reference frequency of a spectral index model:

$$I(\nu) = I(\nu_0)\left(\frac{\nu}{\nu_0}\right)^{\alpha}$$

Parameters **data** : `numpy.ndarray`

array of shape `(comps, chan)` The noisy data as a function of frequency.

weights : numpy.ndarray

array of shape (chan,) Inverse of variance on each frequency axis.

freqs : numpy.ndarray

frequencies of shape (chan,)

freq0 : float

Reference frequency

alphai : numpy.ndarray, optional

array of shape (comps,) Initial guess for the alphas. Defaults to -0.7.

I0i : numpy.ndarray, optional

array of shape (comps,) Initial guess for the intensities at the reference frequency. Defaults to 1.0.

tol : float, optional

Solver absolute tolerance (optional). Defaults to 1e-6.

maxiter : int, optional

Solver maximum iterations (optional). Defaults to 100.

dtype : np.dtype, optional

Datatype of result. Should be either np.float32 or np.float64. Defaults to np.float64.

Returns **out** : numpy.ndarray

array of shape (4, comps) The fitted components arranged as [alphas, alphavars, I0s, I0vars]

Dask

fit_spi_components(data, weights, freqs, freq0)	Computes the spectral indices and the intensity at the reference frequency of a spectral index model:

africanus.model.spi.dask.**fit_spi_components**(*data*, *weights*, *freqs*, *freq0*, *alphai=None*, *I0i=None*, *tol=1e-05*, *maxiter=100*)

Computes the spectral indices and the intensity at the reference frequency of a spectral index model:

$$I(\nu) = I(\nu_0) \left(\frac{\nu}{\nu_0} \right)^{\alpha}$$

Parameters **data** : dask.array.Array

array of shape (comps, chan) The noisy data as a function of frequency.

weights : dask.array.Array

array of shape (chan,) Inverse of variance on each frequency axis.

freqs : dask.array.Array

frequencies of shape (chan,)

freq0 : float

Reference frequency

alphai : dask.array.Array, optional

array of shape (comps,) Initial guess for the alphas. Defaults to -0.7.

I0i : dask.array.Array, optional

array of shape (comps,) Initial guess for the intensities at the reference frequency. Defaults to 1.0.

tol : float, optional

Solver absolute tolerance (optional). Defaults to 1e-6.

maxiter : int, optional

Solver maximum iterations (optional). Defaults to 100.

dtype : np.dtype, optional

Datatype of result. Should be either np.float32 or np.float64. Defaults to np.float64.

Returns **out** : dask.array.Array

array of shape (4, comps) The fitted components arranged as [alphas, alphavars, I0s, I0vars]

5.6.4 Source Morphology

Shape functions for different Source Morphologies

Numpy

gaussian(uvw, frequency, shape_params)	Computes the Gaussian Shape Function.

africanus.model.shape.**gaussian**(*uvw*, *frequency*, *shape_params*)

Computes the Gaussian Shape Function.

$$
\begin{aligned}
\lambda' &= 2\lambda\pi \\
r &= \frac{e_{min}}{e_{maj}} \\
u_1 &= (u\, e_{maj}\, cos(\alpha) - v\, e_{maj}\, sin(\alpha))r\lambda' \\
v_1 &= (u\, e_{maj}\, sin(\alpha) - v\, e_{maj}\, cos(\alpha))\lambda' \\
\text{shape} &= e^{(-u_1^2 - v_1^2)}
\end{aligned}
$$

where:

- u and v are the UV coordinates and λ the frequency.
- e_{maj} and e_{min} are the major and minor axes and α the position angle.

Parameters **uvw** : numpy.ndarray

UVW coordinates of shape (row, 3)

frequency : numpy.ndarray

frequencies of shape (chan,)

shape_param : numpy.ndarray

Gaussian Shape Parameters of shape `(source, 3)` where the second dimension contains the *(emajor, eminor, angle)* parameters describing the shape of the Gaussian

Returns **gauss_shape** : `numpy.ndarray`

Shape parameters of shape `(source, row, chan)`

Dask

gaussian(uvw, frequency, shape_params)	Computes the Gaussian Shape Function.

africanus.model.shape.dask.**gaussian**(*uvw*, *frequency*, *shape_params*)

Computes the Gaussian Shape Function.

$$\lambda' = 2\lambda\pi$$
$$r = \frac{e_{min}}{e_{maj}}$$
$$u_1 = (u\, e_{maj}\, cos(\alpha) - v\, e_{maj}\, sin(\alpha))r\lambda'$$
$$v_1 = (u\, e_{maj}\, sin(\alpha) - v\, e_{maj}\, cos(\alpha))\lambda'$$
$$\text{shape} = e^{(-u_1^2 - v_1^2)}$$

where:

- u and v are the UV coordinates and λ the frequency.
- e_{maj} and e_{min} are the major and minor axes and α the position angle.

Parameters **uvw** : `dask.array.Array`

UVW coordinates of shape `(row, 3)`

frequency : `dask.array.Array`

frequencies of shape `(chan,)`

shape_param : `dask.array.Array`

Gaussian Shape Parameters of shape `(source, 3)` where the second dimension contains the *(emajor, eminor, angle)* parameters describing the shape of the Gaussian

Returns **gauss_shape** : `dask.array.Array`

Shape parameters of shape `(source, row, chan)`

5.6.5 WSClean Spectral Model

Utilities for creating a spectral model from a wsclean component file.

Numpy

load(filename)	Loads wsclean component model.
spectra(I, coeffs, log_poly, ref_freq, frequency)	Produces a spectral model from a polynomial expansion of a wsclean file model.

africanus.model.wsclean.**load**(*filename*)

Loads wsclean component model.

```
sources = load("components.txt")
sources = dict(sources)  # Convert to dictionary

I = sources["I"]
ref_freq = sources["ReferenceFrequency"]
```

See the WSClean Component List for further details.

Parameters **filename** : str or iterable

Filename of wsclean model file or iterable producing the lines of the file.

Returns list of (name, list of values) tuples

list of column (name, value) tuples

See also:

africanus.model.wsclean.spectra

africanus.model.wsclean.**spectra**(*I*, *coeffs*, *log_poly*, *ref_freq*, *frequency*)

Produces a spectral model from a polynomial expansion of a wsclean file model. Depending on how *log_poly* is set ordinary or logarithmic polynomials are used to produce the expansion:

$$flux(\lambda) = I_0 + \sum_{c=0} \text{coeffs}(c)(\lambda/\lambda_{ref} - 1)^{c+1}$$

$$flux(\lambda) = \exp\left(\log I_0 + \sum_{c=0} \text{coeffs}(c) \log(\lambda/\lambda_{ref})^{c+1}\right)$$

See the WSClean Component List for further details.

Parameters **I** : numpy.ndarray

flux density in Janskys at the reference frequency of shape (source,)

coeffs : numpy.ndarray

Polynomial coefficients for each source of shape (source, comp)

log_poly : numpy.ndarray or bool

boolean array of shape (source,) indicating whether logarithmic (True) or ordinary (False) polynomials should be used.

ref_freq : numpy.ndarray

Source reference frequencies of shape (source,)

frequency : numpy.ndarray

frequencies of shape (chan,)

Returns **spectral_model** : numpy.ndarray

Spectral Model of shape (source, chan)

See also:

africanus.model.wsclean.load

Dask

spectra(stokes, spi, log_si, ref_freq, frequency)	Produces a spectral model from a polynomial expansion of a wsclean file model.

africanus.model.wsclean.dask.**spectra**(*stokes*, *spi*, *log_si*, *ref_freq*, *frequency*)

Produces a spectral model from a polynomial expansion of a wsclean file model. Depending on how *log_poly* is set ordinary or logarithmic polynomials are used to produce the expansion:

$$flux(\lambda) = I_0 + \sum_{c=0} \text{coeffs}(c)(\lambda/\lambda_{ref} - 1)^{c+1}$$

$$flux(\lambda) = \exp\left(\log I_0 + \sum_{c=0} \text{coeffs}(c)\log(\lambda/\lambda_{ref})^{c+1}\right)$$

See the WSClean Component List for further details.

Parameters **I** : dask.array.Array

flux density in Janskys at the reference frequency of shape (source,)

coeffs : dask.array.Array

Polynomial coefficients for each source of shape (source, comp)

log_poly : dask.array.Array or bool

boolean array of shape (source,) indicating whether logarithmic (True) or ordinary (False) polynomials should be used.

ref_freq : dask.array.Array

Source reference frequencies of shape (source,)

frequency : dask.array.Array

frequencies of shape (chan,)

Returns **spectral_model** : dask.array.Array

Spectral Model of shape (source, chan)

See also:

africanus.model.wsclean.load

5.7 Averaging

Routines for averaging visibility data.

5.7.1 Time and Channel Averaging

The routines in this section average row-based samples by:

1. Averaging samples of consecutive **time** values into bins defined by an period of time_bin_secs seconds.
2. Averaging channel data into equally sized bins of chan_bin_size.

In order to achieve this, a **baseline x time** ordering is established over the input data where **baseline** corresponds to the unique **(ANTENNA1, ANTENNA2)** pairs and **time** corresponds to the unique, monotonically increasing **TIME** values associated with the rows of a Measurement Set.

Baseline	T0	T1	T2	T3	T4
(0, 0)	0.1	0.2	0.3	0.4	0.5
(0, 1)	0.1	0.2	0.3	0.4	0.5
(0, 2)	0.1	0.2	X	0.4	0.5
(1, 1)	0.1	0.2	0.3	0.4	0.5
(1, 2)	0.1	0.2	0.3	0.4	0.5
(2, 2)	0.1	0.2	0.3	0.4	0.5

It is possible for times or baselines to be missing. In the above example, T2 is missing for baseline (0, 2).

Warning: The above requires unique lexicographical combinations of (TIME, ANTENNA1, ANTENNA2). This can usually be achieved by suitably partitioning input data on indexing rows, DATA_DESC_ID and SCAN_NUMBER in particular.

For each baseline, adjacent time's are assigned to a bin if $h_c - h_e/2 - (l_c - l_e/2) < \texttt{time\_bin\_secs}$, where h_c and l_c are the upper and lower time and h_e and l_e are the upper and lower intervals, taken from the **INTERVAL** column. Note that no distinction is made between flagged and unflagged data when establishing the endpoints in the bin.

The reason for this is that the Measurement Set v2.0 Specification specifies that **TIME** and **INTERVAL** columns are defined as containing the *nominal* time and period at which the visibility was sampled. This means that their values includie valid, flagged and missing data. Thus, averaging a regular high-resolution **baseline x htime** grid should produce a regular low-resolution **baseline x ltime** grid (**htime > ltime**) in the presence of bad data

By contrast, other columns such as **TIME_CENTROID** and **EXPOSURE** contain the *effective* time and period as they exclude missing and bad data. Their increased accuracy, and therefore variability means that they are unsuitable for establishing a grid over the data.

To summarise, the averaged times in each bin establish a map:

- from possibly unordered input rows.
- to a reduced set of output rows ordered by averaged `(TIME, ANTENNA1, ANTENNA2)`.

Flagged Data Handling

Both **FLAG_ROW** and **FLAG** columns may be supplied to the averager, but they should be consistent with each other. The averager will throw an exception if this is not the case, rather than making an assumption as to which is correct.

When provided with flags, the averager will output averages for bins that are completely flagged.

Part of the reason for this is that the specifies that the **TIME** and **INTERVAL** columns represent the *nominal* time and interval values. This means that they should represent valid as well as flagged or missing data in their computation.

By contrast, most other columns such as **TIME_CENTROID** and **EXPOSURE**, contain the *effective* values and should only include valid, unflagged data.

To support this:

1. **TIME** and **INTERVAL** are averaged using both flagged and unflagged samples.
2. Other columns, such as **TIME_CENTROID** are handled as follows:

    1. If the bin contains some unflagged data, only this data is used to calculate average.
    2. If the bin is completely flagged, the average of all samples (which are all flagged) will be used.
3. In both cases, a completely flagged bin will have it's flag set.
4. To support the two cases, twice the memory of the output array is required to track both averages, but only one array of merged values is returned.

Guarantees

1. Averaged output data will be lexicographically ordered by `(TIME, ANTENNA1, ANTENNA2)`
2. **TIME** and **INTERVAL** columns always contain the *nominal* average and sum and therefore contain both and missing or unflagged data.
3. Other columns will contain the *effective* average and will contain only valid data *except* when all data in the bin is flagged.
4. Completely flagged bins will be set as flagged in both the *nominal* and *effective* case.
5. Certain columns are averaged, while others are summed, or simply assigned to the last value in the bin in the case of antenna indices.
6. **Visibility data** is averaged by multiplying and dividing by **WEIGHT_SPECTRUM** or **WEIGHT** or natural weighting, in order of priority.

$$\frac{\sum v_i w_i}{\sum w_i}$$

7. **SIGMA_SPECTRUM** is averaged by multiplying and dividing by **WEIGHT_SPECTRUM** or **WEIGHT** or natural weighting, in order of priority and availability.

   SIGMA is only averaged with **WEIGHT** or natural weighting.

$$\sqrt{\frac{\sum w_i^2 \sigma_i^2}{(\sum w_i)^2}}$$

The following table summarizes the handling of each column in the main Measurement Set table:

Column	Unflagged/Flagged sample handling	Aggregation Method	Required
TIME	Nominal	Mean	Yes
INTERVAL	Nominal	Sum	Yes
ANTENNA1	Nominal	Assigned to Last Input	Yes
ANTENNA2	Nominal	Assigned to Last Input	Yes
TIME_CENTROID	Effective	Mean	No
EXPOSURE	Effective	Sum	No
FLAG_ROW	Effective	Set if All Inputs Flagged	No
UVW	Effective	Mean	No
WEIGHT	Effective	Sum	No
SIGMA	Effective	Weighted Mean	No
DATA (vis)	Effective	Weighted Mean	No
FLAG	Effective	Set if All Inputs Flagged	No
WEIGHT_SPECTRUM	Effective	Sum	No
SIGMA_SPECTRUM	Effective	Weighted Mean	No

The following SPECTRAL_WINDOW sub-table columns are averaged as follows:

Column	Aggregation Method
CHAN_FREQ	Mean
CHAN_WIDTH	Sum
EFFECTIVE_BW	Sum
RESOLUTION	Sum

Dask Implementation

The dask implementation chunks data up by row and channel and averages each chunk independently of values in other chunks. This should be kept in mind if one wishes to maintain a particular ordering in the output dask arrays.

Typically, Measurement Set data is monotonically ordered in time. To maintain this guarantee in output dask arrays, the chunks will need to be separated by distinct time values. Practically speaking this means that the first and second chunk should not both contain value time 0.1, for example.

Numpy

time_and_channel(time, interval, antenna1, ...)	Averages in time and channel.

africanus.averaging.**time_and_channel**(*time*, *interval*, *antenna1*, *antenna2*, *time_centroid=None*, *exposure=None*, *flag_row=None*, *uvw=None*, *weight=None*, *sigma=None*, *chan_freq=None*, *chan_width=None*, *effective_bw=None*, *resolution=None*, *vis=None*, *flag=None*, *weight_spectrum=None*, *sigma_spectrum=None*, *time_bin_secs=1.0*, *chan_bin_size=1*)

Averages in time and channel.

Parameters **time** : numpy.ndarray

Time values of shape (row,).

interval : numpy.ndarray

Interval values of shape (row,).

antenna1 : numpy.ndarray

First antenna indices of shape (row,)

antenna2 : numpy.ndarray

Second antenna indices of shape (row,)

time_centroid : numpy.ndarray, optional

Time centroid values of shape (row,)

exposure : numpy.ndarray, optional

Exposure values of shape (row,)

flag_row : numpy.ndarray, optional

Flagged rows of shape (row,).

uvw : numpy.ndarray, optional

UVW coordinates of shape (row, 3).

weight : numpy.ndarray, optional

Weight values of shape (row, corr).

sigma : numpy.ndarray, optional

Sigma values of shape (row, corr).

chan_freq : numpy.ndarray, optional

Channel frequencies of shape (chan,).

chan_width : numpy.ndarray, optional

Channel widths of shape (chan,).

effective_bw : numpy.ndarray, optional

Effective channel bandwidth of shape (chan,).

resolution : numpy.ndarray, optional

Effective channel resolution of shape (chan,).

vis : numpy.ndarray, optional

Visibility data of shape (row, chan, corr).

flag : numpy.ndarray, optional

Flag data of shape (row, chan, corr).

weight_spectrum : numpy.ndarray, optional

Weight spectrum of shape (row, chan, corr).

sigma_spectrum : numpy.ndarray, optional

Sigma spectrum of shape (row, chan, corr).

time_bin_secs : float, optional

Maximum summed interval in seconds to include within a bin. Defaults to 1.0.

chan_bin_size : int, optional

Number of bins to average together. Defaults to 1.

Returns namedtuple

A namedtuple whose entries correspond to the input arrays. Output arrays will be None if the inputs were None.

Notes

The implementation currently requires unique lexicographical combinations of (TIME, ANTENNA1, ANTENNA2). This can usually be achieved by suitably partitioning input data on indexing rows, DATA_DESC_ID and SCAN_NUMBER in particular.

Dask

time_and_channel(time, interval, antenna1, ...)	Averages in time and channel.

africanus.averaging.dask.**time_and_channel**(*time*, *interval*, *antenna1*, *antenna2*, *time_centroid=None*, *exposure=None*, *flag_row=None*, *uvw=None*, *weight=None*, *sigma=None*, *chan_freq=None*, *chan_width=None*, *effective_bw=None*, *resolution=None*, *vis=None*, *flag=None*, *weight_spectrum=None*, *sigma_spectrum=None*, *time_bin_secs=1.0*, *chan_bin_size=1*)

Averages in time and channel.

Parameters **time** : dask.array.Array

Time values of shape (row,).

interval : dask.array.Array

Interval values of shape (row,).

antenna1 : dask.array.Array

First antenna indices of shape (row,)

antenna2 : dask.array.Array

Second antenna indices of shape (row,)

time_centroid : dask.array.Array, optional

Time centroid values of shape (row,)

exposure : dask.array.Array, optional

Exposure values of shape (row,)

flag_row : dask.array.Array, optional

Flagged rows of shape (row,).

uvw : dask.array.Array, optional

UVW coordinates of shape (row, 3).

weight : dask.array.Array, optional

Weight values of shape (row, corr).

sigma : dask.array.Array, optional

Sigma values of shape (row, corr).

chan_freq : dask.array.Array, optional

Channel frequencies of shape (chan,).

chan_width : dask.array.Array, optional

Channel widths of shape (chan,).

effective_bw : dask.array.Array, optional

Effective channel bandwidth of shape (chan,).

resolution : dask.array.Array, optional

Effective channel resolution of shape `(chan,)`.

vis : `dask.array.Array`, optional

Visibility data of shape `(row, chan, corr)`.

flag : `dask.array.Array`, optional

Flag data of shape `(row, chan, corr)`.

weight_spectrum : `dask.array.Array`, optional

Weight spectrum of shape `(row, chan, corr)`.

sigma_spectrum : `dask.array.Array`, optional

Sigma spectrum of shape `(row, chan, corr)`.

time_bin_secs : float, optional

Maximum summed interval in seconds to include within a bin. Defaults to 1.0.

chan_bin_size : int, optional

Number of bins to average together. Defaults to 1.

Returns namedtuple

A namedtuple whose entries correspond to the input arrays. Output arrays will be `None` if the inputs were `None`.

Notes

The implementation currently requires unique lexicographical combinations of (TIME, ANTENNA1, ANTENNA2). This can usually be achieved by suitably partitioning input data on indexing rows, DATA_DESC_ID and SCAN_NUMBER in particular.

5.8 Utilities

5.8.1 Command Line

parse_python_assigns(assign_str)	Parses a string, containing assign statements into a dictionary.

africanus.util.cmdline.**parse_python_assigns**(*assign_str*)

Parses a string, containing assign statements into a dictionary.

```
data = parse_python_assigns("beta=5.6; l=[2,3], s='hello, world'")

assert data == {
    'beta': 5.6,
    'l': [2, 3],
    's': 'hello, world'
}
```

Parameters **assign_str: str**

Assignment string. Should only contain assignment statements assigning python literals

or builtin function calls, to variable names. Multiple assignment statements should be separated by semi-colons.

Returns dict

Dictionary { name: value } containing assignment results.

5.8.2 Requirements Handling

requires_optional(*requirements)	Decorator returning either the original function, or a dummy function raising a `MissingPackageException` when called, depending on whether the supplied `requirements` are present.

africanus.util.requirements.**requires_optional**(**requirements*)

Decorator returning either the original function, or a dummy function raising a `MissingPackageException` when called, depending on whether the supplied `requirements` are present.

If packages are missing and called within a test, the dummy function will call `pytest.skip()`.

Used in the following way:

```
try:
    from scipy import interpolate
except ImportError as e:
    # https://stackoverflow.com/a/29268974/1611416, pep 3110 and 344
    scipy_import_error = e
else:
    scipy_import_error = None

@requires_optional('scipy', scipy_import_error)
def function(*args, **kwargs):
    return interpolate(...)
```

Parameters **requirements** : iterable of string, None or ImportError

Sequence of package names required by the decorated function. ImportError exceptions (or None, indicating their absence) may also be supplied and will be immediately re-raised within the decorator. This is useful for tracking down problems in user import logic.

Returns callable

Either the original function if all `requirements` are available or a dummy function that throws a `MissingPackageException` or skips a pytest.

5.8.3 Shapes

aggregate_chunks(chunks, max_chunks)	Aggregate dask `chunks` together into chunks no larger than `max_chunks`.
corr_shape(ncorr, corr_shape)	Returns the shape of the correlations, given `ncorr` and the type of correlation shape requested

africanus.util.shapes.**aggregate_chunks**(*chunks*, *max_chunks*)

Aggregate dask `chunks` together into chunks no larger than `max_chunks`.

```
chunks, max_c = ((3,4,6,3,6,7),(1,1,1,1,1,1)), (10,3)
expected = ((7,9,6,7), (2,2,1,1))
assert aggregate_chunks(chunks, max_c) == expected
```

Parameters **chunks** : sequence of tuples or tuple

max_chunks : sequence of ints or int

Returns sequence of tuples or tuple

africanus.util.shapes.**corr_shape**(*ncorr*, *corr_shape*)

Returns the shape of the correlations, given `ncorr` and the type of correlation shape requested

Parameters **ncorr** : integer

Number of correlations

corr_shape : {'flat', 'matrix'}

Shape of output correlations

Returns tuple

Shape tuple describing the correlation dimensions

- If `flat` returns `(ncorr,)`
- If `matrix` returns
  - `(1,)` if `ncorr == 1`
  - `(2,)` if `ncorr == 2`
  - `(2,2)` if `ncorr == 4`

5.8.4 Beams

beam_filenames(filename_schema, corr_types)	Returns a dictionary of beam filename pairs, keyed on correlation,from the cartesian product of correlations and real, imaginary pairs
beam_grids(header[, l_axis, m_axis])	Extracts the FITS indices and grids for the beam dimensions in the supplied FITS `header`.

africanus.util.beams.**beam_filenames**(*filename_schema*, *corr_types*)

Returns a dictionary of beam filename pairs, keyed on correlation,from the cartesian product of correlations and real, imaginary pairs

Given `beam_$(corr)_$(reim).fits` returns:

```
{
  'xx' : ['beam_xx_re.fits', 'beam_xx_im.fits'],
  'xy' : ['beam_xy_re.fits', 'beam_xy_im.fits'],
  ...
  'yy' : ['beam_yy_re.fits', 'beam_yy_im.fits'],
}
```

Given `beam_$(CORR)_$(REIM).fits` returns:

```
{
  'xx' : ['beam_XX_RE.fits', 'beam_XX_IM.fits'],
  'xy' : ['beam_XY_RE.fits', 'beam_XY_IM.fits'],
  ...
  'yy' : ['beam_YY_RE.fits', 'beam_YY_IM.fits']),
}
```

Parameters **filename_schema** : str

String containing the filename schema.

corr_types : list of integers

list of integers defining the correlation type.

Returns dict

Dictionary of schema `{correlation : (refile, imfile)}` mapping correlations to real and imaginary filename pairs

`africanus.util.beams.`**`beam_grids`**(*header*, *l_axis=None*, *m_axis=None*)

Extracts the FITS indices and grids for the beam dimensions in the supplied FITS `header`. Specifically the axes specified by

1. `L` or `X` CTYPE
2. `M` or `Y` CTYPE
3. `FREQ` CTYPE

If the first two axes have a negative sign, such as `-L`, the grid will be inverted.

Any grids corresponding to axes with a CUNIT type of `DEG` will be converted to radians.

Parameters **header** : `Header` or dict

FITS header object.

l_axis : str

FITS axis interpreted as the L axis. *L* and *X* are sensible values here. *-L* will invert the coordinate system on that axis.

m_axis : str

FITS axis interpreted as the M axis. *M* and *Y* are sensible values here. *-M* will invert the coordinate system on that axis.

Returns tuple

Returns ((l_axis, l_grid), (m_axis, m_grid), (freq_axis, freq_grid)) where the axis is the FORTRAN indexed FITS axis (1-indexed) and grid contains the values at each pixel along the axis.

5.8.5 Code

format_code(code)	Formats some code with line numbers
memoize_on_key(key_fn)	Memoize based on a key function supplied by the user.

africanus.util.code.**format_code**(*code*)
Formats some code with line numbers

Parameters **code** : str

Code

Returns str

Code prefixed with line numbers

class africanus.util.code.**memoize_on_key**(*key_fn*)
Memoize based on a key function supplied by the user. The key function should return a custom key for memoizing the decorated function, based on the arguments passed to it.

In the following example, the arguments required to generate the *_generate_phase_delay_kernel* function are the types of the *lm*, *uvw* and *frequency* arrays, as well as the number of correlations, *ncorr*.

The supplied `key_fn` produces a unique key based on these types and the number of correlations, which is used to cache the generated function.

```
def key_fn(lm, uvw, frequency, ncorrs=4):
    '''
    Produce a unique key for the arguments of
     _generate_phase_delay_kernel
    '''
    return (lm.dtype, uvw.dtype, frequency.dtype, ncorrs)

_code_template = jinja2.Template('''
#define ncorrs {{ncorrs}}

__global__ void phase_delay(
    const {{lm_type}} * lm,
    const {{uvw_type}} * uvw,
    const {{freq_type}} * frequency,
    {{out_type}} * out)
{
    ...
}
''')

_type_map = {
    np.float32: 'float',
    np.float64: 'double'
}

@memoize_on_key(key_fn)
def _generate_phase_delay_kernel(lm, uvw, frequency, ncorrs=4):
    ''' Generate the phase delay kernel '''
    out_dtype = np.result_type(lm.dtype, uvw.dtype, frequency.dtype)
    code = _code_template.render(lm_type=_type_map[lm.dtype],
                                 uvw_type=_type_map[uvw.dtype],
                                 freq_type=_type_map[frequency.dtype],
                                 ncorrs=ncorrs)
    return cp.RawKernel(code, "phase_delay")
```

Methods

__call__(fn)	Call self as a function.

5.8.6 dask

EstimatingProgressBar([minimum, width, dt, out])	Progress Bar that displays elapsed time as well as an estimate of total time taken.

class africanus.util.dask_util.**EstimatingProgressBar**(*minimum=0*, *width=42*, *dt=1.0*, *out=sys.stdout*)

Progress Bar that displays elapsed time as well as an estimate of total time taken.

When starting a dask computation, the bar examines the graph and determines the number of chunks contained by a dask collection.

During computation the number of completed chunks and their the total time taken to complete them are tracked. The average derived from these numbers are used to estimate total compute time, relative to the current elapsed time.

The bar is not particularly accurate and will underestimate near the beginning of computation and seems to slightly overestimate during the buk of computation. However, it may be more accurate than the default dask task bar which tracks number of tasks completed by total tasks.

Parameters **minimum** : int, optional

Minimum time threshold in seconds before displaying a progress bar. Default is 0 (always display)

width : int, optional

Width of the bar, default is 42 characters.

dt : float, optional

Update resolution in seconds, default is 1.0 seconds.

5.8.7 CUDA

grids(dims, blocks)	Determine the grid size, given space dimensions sizes and blocks

africanus.util.cuda.**grids**(*dims*, *blocks*)

Determine the grid size, given space dimensions sizes and blocks

Parameters **dims** : tuple of ints

(x, y, z) tuple

Returns tuple

(x, y, z) grid size tuple

5.9 Calibration

This module provides basic radio interferometry calibration utilities. Calibration is the process of estimating the 2×2 Jones matrices which describe transformations of the signal as it propagates from source to observer. Currently, all

utilities assume a discretised form of the radio interferometer measurement equation (RIME) as described in *Radio Interferometer Measurement Equation*.

Calibration is usually divided into three phases viz.

- First generation calibration (1GC): using an external calibrator to infer the gains during the target observation. Sometimes also refered to as calibrator transfer
- Second generation calibration (2GC): using a partially incomplete sky model to perform direction independent calibration. Also known as direction independent self-calibration.
- Third generation calibration (3GC): using a partially incomplete sky model to perform direction dependent calibration. Also known as direction dependent self-calibration.

On top of these three phases, there are usually three possible calibration scenarios. The first is when both the Jones terms and the visibilities are assumed to be diagonal. In this case the two correlations can be calibrated separately and it is refered to as `diag-diag` calibration. The second case is when the Jones matrices are assumed to be diagonal but the visibility data are full 2×2 matrices. This is refered to as `diag` calibration. The final scenario is when both the full 2×2 Jones matrices and the full 2×2 visibilities are used for calibration. This is simply refered to as calibration. The specific scenario is determined from the shapes of the input gains and the input data.

This module also provides a number of utilities which are useful for calibration.

5.9.1 Utils

Numpy

corrupt_vis(time_bin_indices, ...)	Corrupts model visibilities with arbitrary Jones terms.
residual_vis(time_bin_indices, ...)	Computes residual visibilities given model visibilities and gains solutions.
correct_vis(time_bin_indices, ...)	Apply inverse of direction independent gains to visibilities to generate corrected visibilities.
compute_and_corrupt_vis(time_bin_indices, ...)	Corrupts time variable component model with arbitrary Jones terms.

africanus.calibration.utils.**corrupt_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *model*)

Corrupts model visibilities with arbitrary Jones terms.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape `(utime)`

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape `(utime)`

antenna1 : numpy.ndarray

First antenna indices of shape `(row,)`.

antenna2 : numpy.ndarray

Second antenna indices of shape `(row,)`

jones : numpy.ndarray

Gains of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

model : numpy.ndarray

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

Returns **vis** : numpy.ndarray

visibilities of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

africanus.calibration.utils.**residual_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *vis*, *flag*, *model*)

Computes residual visibilities given model visibilities and gains solutions.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape `(utime)`

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape `(utime)`

antenna1 : numpy.ndarray

First antenna indices of shape `(row,)`.

antenna2 : numpy.ndarray

Second antenna indices of shape `(row,)`

jones : numpy.ndarray

Gain solutions of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

vis : numpy.ndarray

Data values of shape `(row, chan, corr)`. or `(row, chan, corr, corr)`.

flag : numpy.ndarray

Flag data of shape `(row, chan, corr)` or `(row, chan, corr, corr)`

model : numpy.ndarray

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

Returns **residual** : numpy.ndarray

Residual visibilities of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

africanus.calibration.utils.**correct_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *vis*, *flag*)

Apply inverse of direction independent gains to visibilities to generate corrected visibilities. For a measurement model of the form

$$V_{pq} = G_p X_{pq} G_q^H + n_{pq}$$

the corrected visibilities are defined as

$$C_{pq} = G_p^{-1} V_{pq} G_q^{-H}$$

The corrected visibilities therefore have a non-trivial noise contribution. Note it is only possible to form corrected data from direction independent gains solutions so the `dir` axis on the jones terms should always be one.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape (utime).

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape (utime).

antenna1 : numpy.ndarray

Antenna 1 index used to look up the antenna Jones for a particular baseline with shape (row,).

antenna2 : numpy.ndarray

Antenna 2 index used to look up the antenna Jones for a particular baseline with shape (row,).

jones : numpy.ndarray

Gain solutions of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

vis : numpy.ndarray

Data values of shape (row, chan, corr) or (row, chan, corr, corr).

flag : numpy.ndarray

Flag data of shape (row, chan, corr) or (row, chan, corr, corr).

Returns

——-

corrected_vis : numpy.ndarray

True visibilities of shape (row,chan,corr_1,corr_2)

africanus.calibration.utils.**compute_and_corrupt_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *model*, *uvw*, *freq*, *lm*)

Corrupts time variable component model with arbitrary Jones terms. Currrently only time variable point source models are supported.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape (utime)

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape (utime)

antenna1 : numpy.ndarray

First antenna indices of shape (row,).

antenna2 : numpy.ndarray

Second antenna indices of shape (row,)

jones : numpy.ndarray

Gains of shape (utime, ant, chan, dir, corr) or (utime, ant, chan, dir, corr, corr).

model : numpy.ndarray

Model image as a function of time with shape `(utime, chan, dir, corr)` or `(utime, chan, dir, corr, corr)`.

uvw : `numpy.ndarray`

uvw coordinates of shape `(row, 3)`

lm : `numpy.ndarray`

Source lm coordinates as a function of time `(utime, dir, 2)`

Returns **vis** : `numpy.ndarray`

visibilities of shape `(row, chan, corr)` or `(row, chan, corr, corr)`.

Dask

corrupt_vis(time_bin_indices, ...)	Corrupts model visibilities with arbitrary Jones terms.
residual_vis(time_bin_indices, ...)	Computes residual visibilities given model visibilities and gains solutions.
correct_vis(time_bin_indices, ...)	Apply inverse of direction independent gains to visibilities to generate corrected visibilities.
compute_and_corrupt_vis(time_bin_indices, ...)	Corrupts time variable component model with arbitrary Jones terms.

africanus.calibration.utils.dask.**corrupt_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *model*)

Corrupts model visibilities with arbitrary Jones terms.

Parameters **time_bin_indices** : `dask.array.Array`

The start indices of the time bins of shape `(utime)`

time_bin_counts : `dask.array.Array`

The counts of unique time in each time bin of shape `(utime)`

antenna1 : `dask.array.Array`

First antenna indices of shape `(row,)`.

antenna2 : `dask.array.Array`

Second antenna indices of shape `(row,)`

jones : `dask.array.Array`

Gains of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

model : `dask.array.Array`

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

Returns **vis** : `dask.array.Array`

visibilities of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

africanus.calibration.utils.dask.**residual_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *vis*, *flag*, *model*)

Computes residual visibilities given model visibilities and gains solutions.

Parameters **time_bin_indices** : dask.array.Array

The start indices of the time bins of shape (utime)

time_bin_counts : dask.array.Array

The counts of unique time in each time bin of shape (utime)

antenna1 : dask.array.Array

First antenna indices of shape (row,).

antenna2 : dask.array.Array

Second antenna indices of shape (row,)

jones : dask.array.Array

Gain solutions of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

vis : dask.array.Array

Data values of shape (row, chan, corr). or (row, chan, corr, corr).

flag : dask.array.Array

Flag data of shape (row, chan, corr) or (row, chan, corr, corr)

model : dask.array.Array

Model data values of shape (row, chan, dir, corr) or (row, chan, dir, corr, corr).

Returns **residual** : dask.array.Array

Residual visibilities of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

africanus.calibration.utils.dask.**correct_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *vis*, *flag*)

Apply inverse of direction independent gains to visibilities to generate corrected visibilities. For a measurement model of the form

$$V_{pq} = G_p X_{pq} G_q^H + n_{pq}$$

the corrected visibilities are defined as

$$C_{pq} = G_p^{-1} V_{pq} G_q^{-H}$$

The corrected visibilities therefore have a non-trivial noise contribution. Note it is only possible to form corrected data from direction independent gains solutions so the dir axis on the jones terms should always be one.

Parameters **time_bin_indices** : dask.array.Array

The start indices of the time bins of shape (utime).

time_bin_counts : dask.array.Array

The counts of unique time in each time bin of shape (utime).

antenna1 : dask.array.Array

Antenna 1 index used to look up the antenna Jones for a particular baseline with shape (row,).

antenna2 : dask.array.Array

Antenna 2 index used to look up the antenna Jones for a particular baseline with shape (row,).

jones : dask.array.Array

Gain solutions of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

vis : dask.array.Array

Data values of shape (row, chan, corr) or (row, chan, corr, corr).

flag : dask.array.Array

Flag data of shape (row, chan, corr) or (row, chan, corr, corr).

Returns

——-

corrected_vis : dask.array.Array

True visibilities of shape (row,chan,corr_1,corr_2)

africanus.calibration.utils.dask.**compute_and_corrupt_vis**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *model*, *uvw*, *freq*, *lm*)

Corrupts time variable component model with arbitrary Jones terms. Currrently only time variable point source models are supported.

Parameters **time_bin_indices** : dask.array.Array

The start indices of the time bins of shape (utime)

time_bin_counts : dask.array.Array

The counts of unique time in each time bin of shape (utime)

antenna1 : dask.array.Array

First antenna indices of shape (row,).

antenna2 : dask.array.Array

Second antenna indices of shape (row,)

jones : dask.array.Array

Gains of shape (utime, ant, chan, dir, corr) or (utime, ant, chan, dir, corr, corr).

model : dask.array.Array

Model image as a function of time with shape (utime, chan, dir, corr) or (utime, chan, dir, corr, corr).

uvw : dask.array.Array

uvw coordinates of shape (row, 3)

lm : dask.array.Array

Source lm coordinates as a function of time (utime, dir, 2)

Returns **vis** : dask.array.Array

visibilities of shape `(row, chan, corr)` or `(row, chan, corr, corr)`.

5.9.2 Phase only

Numpy

compute_jhr(time_bin_indices, ...)	Computes the residual projected in to gain space.
compute_jhj(time_bin_indices, ...)	Computes the diagonal of the Hessian required to perform phase-only maximum likelihood calibration.
compute_jhj_and_jhr(time_bin_indices, ...)	Computes the diagonal of the Hessian and the residual locally projected in to gain space.
gauss_newton(time_bin_indices, ... [, tol, ...])	Performs phase-only maximum likelihood calibration using a Gauss-Newton optimisation algorithm.

africanus.calibration.phase_only.**compute_jhr**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *residual*, *model*, *flag*)

Computes the residual projected in to gain space.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape `(utime)`

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape `(utime)`

antenna1 : numpy.ndarray

First antenna indices of shape `(row,)`.

antenna2 : numpy.ndarray

Second antenna indices of shape `(row,)`

jones : numpy.ndarray

Gain solutions of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

residual : numpy.ndarray

Residual values of shape `(row, chan, corr)`. or `(row, chan, corr, corr)`.

model : numpy.ndarray

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

flag : numpy.ndarray

Flag data of shape `(row, chan, corr)` or `(row, chan, corr, corr)`

Returns **jhr** : numpy.ndarray

The residual projected into gain space shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

africanus.calibration.phase_only.**compute_jhj**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *model*, *flag*)

Computes the diagonal of the Hessian required to perform phase-only maximum likelihood calibration. Currently assumes scalar or diagonal inputs.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape (utime)

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape (utime)

antenna1 : numpy.ndarray

First antenna indices of shape (row,).

antenna2 : numpy.ndarray

Second antenna indices of shape (row,)

jones : numpy.ndarray

Gain solutions of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

model : numpy.ndarray

Model data values of shape (row, chan, dir, corr) or (row, chan, dir, corr, corr).

flag : numpy.ndarray

Flag data of shape (row, chan, corr) or (row, chan, corr, corr)

Returns **jhj** : numpy.ndarray

The diagonal of the Hessian of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

africanus.calibration.phase_only.**compute_jhj_and_jhr**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *residual*, *model*, *flag*)

Computes the diagonal of the Hessian and the residual locally projected in to gain space.

Parameters **time_bin_indices** : numpy.ndarray

The start indices of the time bins of shape (utime)

time_bin_counts : numpy.ndarray

The counts of unique time in each time bin of shape (utime)

antenna1 : numpy.ndarray

First antenna indices of shape (row,).

antenna2 : numpy.ndarray

Second antenna indices of shape (row,)

jones : numpy.ndarray

Gain solutions of shape (time, ant, chan, dir, corr) or (time, ant, chan, dir, corr, corr).

residual : numpy.ndarray

Residual values of shape `(row, chan, corr)`. or `(row, chan, corr, corr)`.

model : `numpy.ndarray`

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

flag : `numpy.ndarray`

Flag data of shape `(row, chan, corr)` or `(row, chan, corr, corr)`

Returns **jhj** : `numpy.ndarray`

The diagonal of the Hessian of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

jhr : `numpy.ndarray`

Residuals projected into signal space of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

`africanus.calibration.phase_only.`**`gauss_newton`**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *vis*, *flag*, *model*, *weight*, *tol=0.0001*, *maxiter=100*)

Performs phase-only maximum likelihood calibration using a Gauss-Newton optimisation algorithm. Currently only DIAG mode is supported.

Parameters **time_bin_indices** : `numpy.ndarray`

The start indices of the time bins of shape `(utime)`

time_bin_counts : `numpy.ndarray`

The counts of unique time in each time bin of shape `(utime)`

antenna1 : `numpy.ndarray`

First antenna indices of shape `(row,)`.

antenna2 : `numpy.ndarray`

Second antenna indices of shape `(row,)`.

jones : `numpy.ndarray`

Gain solutions of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

vis : `numpy.ndarray`

Data values of shape `(row, chan, corr)` or `(row, chan, corr, corr)`.

flag : `numpy.ndarray`

Flag data of shape `(row, chan, corr)` or `(row, chan, corr, corr)`.

model : `numpy.ndarray`

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

weight : `numpy.ndarray`

Weight spectrum of shape `(row, chan, corr)`. If the channel axis is missing weights are duplicated for each channel.

tol: float, optional

The tolerance of the solver. Defaults to 1e-4.

maxiter: int, optional

The maximum number of iterations. Defaults to 100.

Returns **gains** : `numpy.ndarray`

Gain solutions of shape `(time, ant, chan, dir, corr)` or shape `(time, ant, chan, dir, corr, corr)`

jhj : `numpy.ndarray`

The diagonal of the Hessian of shape `(time, ant, chan, dir, corr)` or shape `(time, ant, chan, dir, corr, corr)`

jhr : `numpy.ndarray`

Residuals projected into gain space of shape `(time, ant, chan, dir, corr)` or shape `(time, ant, chan, dir, corr, corr)`.

k: int

Number of iterations (will equal maxiter if not converged)

Dask

compute_jhr(time_bin_indices, ...)	Computes the residual projected in to gain space.
compute_jhj(time_bin_indices, ...)	Computes the diagonal of the Hessian required to perform phase-only maximum likelihood calibration.

africanus.calibration.phase_only.dask.**compute_jhr**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *residual*, *model*, *flag*)

Computes the residual projected in to gain space.

Parameters **time_bin_indices** : `dask.array.Array`

The start indices of the time bins of shape `(utime)`

time_bin_counts : `dask.array.Array`

The counts of unique time in each time bin of shape `(utime)`

antenna1 : `dask.array.Array`

First antenna indices of shape `(row,)`.

antenna2 : `dask.array.Array`

Second antenna indices of shape `(row,)`

jones : `dask.array.Array`

Gain solutions of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

residual : `dask.array.Array`

Residual values of shape `(row, chan, corr)`. or `(row, chan, corr, corr)`.

model : `dask.array.Array`

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

flag : `dask.array.Array`

Flag data of shape `(row, chan, corr)` or `(row, chan, corr, corr)`

Returns **jhr** : `dask.array.Array`

The residual projected into gain space shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

africanus.calibration.phase_only.dask.**compute_jhj**(*time_bin_indices*, *time_bin_counts*, *antenna1*, *antenna2*, *jones*, *model*, *flag*)

Computes the diagonal of the Hessian required to perform phase-only maximum likelihood calibration. Currently assumes scalar or diagonal inputs.

Parameters **time_bin_indices** : `dask.array.Array`

The start indices of the time bins of shape `(utime)`

time_bin_counts : `dask.array.Array`

The counts of unique time in each time bin of shape `(utime)`

antenna1 : `dask.array.Array`

First antenna indices of shape `(row,)`.

antenna2 : `dask.array.Array`

Second antenna indices of shape `(row,)`

jones : `dask.array.Array`

Gain solutions of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

model : `dask.array.Array`

Model data values of shape `(row, chan, dir, corr)` or `(row, chan, dir, corr, corr)`.

flag : `dask.array.Array`

Flag data of shape `(row, chan, corr)` or `(row, chan, corr, corr)`

Returns **jhj** : `dask.array.Array`

The diagonal of the Hessian of shape `(time, ant, chan, dir, corr)` or `(time, ant, chan, dir, corr, corr)`.

5.10 Linear Algebra

This module contains specialised linear algebra tools that are not currently available in the `python` standard scientific libraries.

5.10.1 Kronecker tools

A kronecker matrix is matrix that can be written as a kronecker matrix of the individual matrices i.e.

$$K = K_0 \\ otimes K_1 \\ otimes K_2 \\ otimes \\ cdots$$

Matrices which exhibit this structure can exploit properties of the kronecker product to avoid explicitly expanding the matrix K. This module implements some common linear algebra operations which leverages this property for computational gains and a reduced memory footprint.

Numpy

kron_matvec(A, b)	Computes the matrix vector product of a kronecker matrix in linear time.
kron_cholesky(A)	Computes the Cholesky decomposition of a kronecker matrix as a kronecker matrix of Cholesky factors.

africanus.linalg.**kron_matvec**(*A*, *b*)

Computes the matrix vector product of a kronecker matrix in linear time. Assumes A consists of kronecker product of square matrices.

Parameters **A** : numpy.ndarray

An array of arrays holding matrices [K0, K1, ...] where $A = K_0 \otimes K_1 \otimes \cdots$

b : numpy.ndarray

The right hand side vector

Returns **x** : numpy.ndarray

The result of `A.dot(b)`

africanus.linalg.**kron_cholesky**(*A*)

Computes the Cholesky decomposition of a kronecker matrix as a kronecker matrix of Cholesky factors.

Parameters **A** : numpy.ndarray

An array of arrays holding matrices [K0, K1, ...] where $A = K_0 \otimes K_1 \otimes \cdots$

Returns **L** : numpy.ndarray

An array of arrays holding matrices [L0, L1, ...] where $L = L_0 \otimes L_1 \otimes \cdots$ and each `Li = cholesky(Ki)`

5.11 Gaussian processes

This module provides a collection of tools that are useful when performing Gaussian process regression.

5.11.1 Numpy

abs_diff(x, xp)	Gets matrix of differences between D-dimensional vectors x and xp i.e.
exponential_squared(x, xp, sigmaf, l[, pspec])	Create exponential squared covariance function between D dimensional vectors x and x_p i.e.

africanus.gps.**abs_diff**(*x*, *xp*)

Gets matrix of differences between D-dimensional vectors x and xp i.e.

$$X_{ij} = |x_i - x_j|$$

Parameters **x** : numpy.ndarray

Array of inputs of shape `(N, D)`.

xp : numpy.ndarray

Array of inputs of shape `(Np, D)`.

Returns **XX** : numpy.ndarray

Array of differences of shape `(N, Np)`.

africanus.gps.**exponential_squared**(*x*, *xp*, *sigmaf*, *l*, *pspec=False*)

Create exponential squared covariance function between D dimensional vectors x and x_p i.e.

$$k(x, x_p) = \sigma_f^2 \exp\left(-\frac{(x - x_p)^2}{2l^2}\right)$$

Parameters **x** : numpy.ndarray

Array of shape `(N, D)`.

xp : numpy.ndarray

Array of shape `(Np, D)`.

sigmaf : float

The signal variance hyper-parameter

l : float

The length scale hyper-parameter

Returns **K** : numpy.ndarray

Array of shape `(N, Np)`

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/ska-sa/codex-africanus/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

Codex Africanus could always use more documentation, whether as part of the official Codex Africanus docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ska-sa/codex-africanus/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *codex-africanus* for local development.

1. Fork the *codex-africanus* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/codex-africanus.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv codex-africanus
$ cd codex-africanus/
$ pip install -e .
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the test cases, fixup your PEP8 compliance, and check for any code style issues:

   $ py.test -v africanus $ autopep8 -r -i africanus $ flake8 africanus $ pycodestyle africanus

   To get autopep8 and pycodestyle, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in HISTORY.rst.
3. The pull request should work for Python 2.7, 3.5 and 3.6. Check https://travis-ci.org/ska-sa/codex-africanus/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run the tests:

```
$ py.test -vvv africanus/
```

6.5 Deploying

A reminder for the maintainers on how to deploy.

1. Update HISTORY.rst with the intended release number Z.Y.X and commit to git.
2. Bump the version number with bumpversion. This creates a new git commit, as well as an annotated tag Z.Y.X for the release. If your current version is Z.Y.W and the new version is Z.Y.X call:

   ```
   $ python -m pip install bump2version
   $ bump2version --current-version Z.Y.W --new-version Z.Y.X patch
   ```

3. Push the release commit and new tag up:

   ```
   $ git push --follow-tags
   ```

4. Travis should automatically deploy the tagged release to PyPI if the automated tests pass.

CHAPTER 7

Credits

7.1 Development Lead

- Simon Perkins <sperkins@ska.ac.za>

7.2 Contributors

- Landman Bester <lbester@ska.ac.za>
- Benjamin Hugo <bhugo@ska.ac.za>
- Jonathan Kenyon <jkenyon@ska.ac.za>
- Gijs Molenaar <gijs@pythonic.nl>
- Joshua van Staden <joshvstaden14@gmail.com>
- Oleg Smirnov <oms@ska.ac.za, osmirnov@gmail.com>

CHAPTER 8

History

8.1 0.2.9 (2020-12-15)

- Upgrade ducc0 to version 0.7 (GH#232)
- Fix manually specifying wgridder precision (GH#230)

8.2 0.2.8 (2020-10-08)

- Fix NoneType issue in wgridder when weights are None (GH#228)
- Bounding hull geometric and image manipulation routines (GH#192, GH#154)
- Fix row chunk chunking in Perley Polyhedron Degridder Dask Interface

8.3 0.2.7 (2020-09-23)

- Deprecate old gridder and filters (GH#224)
- Upgrade to ducc0 0.6.0 (GH#223)
- Add Perley Polyhedron Faceting Gridder/Degridder (GH#202, GH#215, GH#222)

8.4 0.2.6 (2020-08-07)

- Add wrapper for ducc0.wgridder (GH#204)
- Correct Irregular Grid nesting in BeamAxes (GH#217)

8.5 0.2.5 (2020-07-01)

- Convert WSClean Gaussian arcsecond and degree quantities to radians (GH#206)
- Update classifiers and correct license in setup.py to BSD3 (GH#201)

8.6 0.2.4 (2020-05-29)

- Support overriding the l and m axis sign in beam_grids (GH#199)
- Upgrade to python-casacore 3.3.1 (GH#197)
- Upgrade to jax 0.1.68 and jaxlib 0.1.47 (GH#197)
- Upgrade to scipy 1.4.0 (GH#197)
- Use github workflows (GH#196, GH#197, GH#198, GH#199)
- Make CASA parallactic angles thread-safe (GH#195)
- Fix spectral model documentation (GH#190), to match changes in (GH#189)

8.7 0.2.3 (2020-05-14)

- Fix incorrect SPI calculation and make predict defaults MeqTree equivalent (GH#189)
- Depend on pytest-flake8 >= 1.0.6 (GH#187, GH#188)
- MeqTrees Comparison Script Updates (GH#160)
- Improve requirements handling (GH#187)
- Use python-casacore wheels for travis testing, instead of kernsuite packages (GH#185)

8.8 0.2.2 (2020-04-09)

- Add a dask Estimating Progress Bar (GH#182, GH#183)

8.9 0.2.1 (2020-04-03)

- Update trove to latest master commit (GH#178)
- Added Cubic Spline support (GH#174)
- Depend on python-casacore >= 3.2.0 (GH#172)
- Drop Python 3.5 support and test Python 3.7 (GH#168)
- Implement optimised WSClean predict (GH#166, GH#167, GH#177, GH#179, GH#180, GH#181)
- Simplify dask predict_vis code (GH#164, GH#165)
- Document and check weight shapes in simple gridder and degridder (GH#162, GH#163)
- Restructuring calibration module (GH#127)

- Upgrade to numba 0.46.0, using new inlining functionality in the RIME and averaging code.
- Modified predict to be compatible with eidos fits headers (GH#158)

8.10 0.2.0 (2019-09-30)

- Added standalone SPI fitter (GH#153)
- Fail earlier and explain duplicate averaging rows (GH#155)
- CUDA Beam Implementation (GH#152)
- Fix documentation package versions (GH#151)
- Deprecate experimental w-stacking gridder in favour of nifty gridder (GH#148)
- Expand travis build matrix (GH#147)
- Drop Python 2 support (GH#146, GH#149, GH#150)
- Support the beam in the predict example (GH#145)
- Fix weight indexing in averaging (GH#144)
- Support EFFECTIVE_BW and RESOLUTION in averaging (GH#144)
- Optimise predict_vis jones coherency summation (GH#143)
- Remove use of @wraps (GH#141, GH#142)
- Set row chunks to nan in dask averaging code. (GH#139)
- predict_vis documentation improvements (GH#135, GH#140)
- Upgrade to dask-ms in the examples (GH#134, GH#138)
- Explain how to obtain predict_vis time_index argument (GH#130)
- Update RIME predict example to support Tigger LSM's and Gaussians (GH#129)
- Add dask wrappers for the nifty gridder (GH#116, GH#136, GH#146)
- Testing and requirement updates. (GH#124)
- Upgraded DFT kernels to have a correlation axis and added flags for vis_to_im. Added predict_from_fits example. (GH#122)
- Fixed segfault when using *_unique_internal* on empty ndarrays (GH#123)
- Removed *apply_gains*. Use *africanus.calibration.utils.correct_vis* instead (GH#118)
- Add streams parameter to dask *predict_vis* (GH#118)
- Implement the beam in numba (GH#112)
- Add residual_vis, correct_vis, phase_only_GN (GH#113)

8.11 0.1.8 (2019-05-28)

- Use environment markers in setup.py (GH#110)
- Add *apply_gains*, a wrapper around *predict_vis* (GH#108)
- Fix testing extras_require (GH#107)

- Fix WEIGHT_SPECTRUM averaging and add more averaging tests (GH#106)

8.12 0.1.7 (2019-05-09)

- Even more support for automated travis deploys.

8.13 0.1.6 (2019-05-09)

- Support automated travis deploys.

8.14 0.1.5 (2019-05-09)

- Predict script enhancements (GH#103) and dask channel chunking fix (GH#104).
- Directly jit DFT functions (GH#100, GH#101)
- Spectral Models (GH#86)
- Fix radec sign conversion in wsclean sky model (GH#96)
- Full Time and Channel Averaging Implementation (GH#80, GH#97, GH#98)
- Support integer seconds in wsclean ra and dec columns (GH#91, GH#93)
- Fix ratio computation in Gaussian Shape (GH#89, GH#90)

8.15 0.1.4 (2019-03-11)

- Support *complete* and *complete-cuda* to support non-GPU installs (GH#87)
- Gaussian Shape Parameter Implementation (GH#82, GH#83)
- WSClean Spectral Model (GH#81)
- Compare predict versus MeqTrees (GH#79)
- Time and channel averaging (GH#75)
- cupy implementation of *predict_vis* (GH#73)
- Introduce transpose in second antenna term of predict (GH#72)
- cupy implementation of *feed_rotation* (GH#67)
- cupy implementation of *stokes_convert* kernel (GH#65)
- Add a basic RIME example (GH#64)
- requires_optional accepts ImportError's for a better debugging experience (GH#62, GH#63)
- Added *fit_component_spi* function (GH#61)
- cupy implementation of the *phase_delay* kernel (GH#59)
- Correct *phase_delay* argument ordering (GH#57)
- Support dask for *radec_to_lmn* and *lmn_to_radec*. Also add support for *radec_to_lm* and *lm_to_radec* (GH#56)

- Added test for dft to test if image space covariance is symmetric(GH#55)
- Correct Parallactic Angle Computation (GH#49)
- Enhance visibility predict (GH#50)
- Fix Kaiser Bessel filter and taper (GH#48)
- Stokes/Correlation conversion (GH#41)
- Fix gridding examples (GH#43)
- Add simple dask gridder example (GH#42)
- Implement Kaiser Bessel filter (GH#38)
- Implement W-stacking gridder/degridder (GH#38)
- Use 2D filters by default (GH#37)
- Fixed bug in im_to_vis. Added more tests for im_to_vis. Removed division by n since it is trivial to reinstate after the fact. (GH#34)
- Move numba implementations out of API functions. (GH#33)
- Zernike Polynomial Direction Dependent Effects (GH#18, GH#30)
- Added division by n to DFT. Fixed dask chunking issue. Updated test_vis_to_im_dask (GH#29).
- Implement RIME visibility predict (GH#24, GH#25)
- Direct Fourier Transform (GH#19)
- Parallactic Angle computation (GH#15)
- Implement Feed Rotation term (GH#14)
- Swap gridding correlation dimensions (GH#13)
- Implement Direction Dependent Effect beam cubes (GH#12)
- Implement Brightness Matrix Calculation (GH#9)
- Implement RIME Phase Delay term (GH#8)
- Support user supplied grids (GH#7)
- Add dask wrappers to the gridder and degridder (GH#4)
- Add weights to gridder/degridder and remove PSF function (GH#2)

8.16 0.1.2 (2018-03-28)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index