Jones Solvers Documentation

Release 0.1.0-beta

Daniel Mitchell

Nov 24, 2022

HOME

A reference library of python-based antenna Jones matrix solvers for radio interferometric calibration.

See the package documentation for usage and performance information.

CHAPTER

ONE

REQUIREMENTS

This package needs to have `Python 3` and `pip` installed, as well as the Radio Astronomy Simulation, Calibration and Imaging Library (RASCIL) and its dependencies. RASCIL requires Python 3.8 or greater.

CHAPTER

TWO

INSTALL

This package can be installed using pip:

```
pip install --extra-index-url=https://artefact.skao.int/repository/pypi-internal/simple␣
↪ska-sdp-jones-solvers
```

Or, once RASCIL has been installed using pip clone then install the package to access the various support scripts:

```
git clone https://gitlab.com/ska-telescope/sdp/ska-sdp-jones-solvers.git
cd ska-sdp-jones-solvers
pip install .
```

CHAPTER

THREE

TESTING

Unit tests can be run to check the installation and dependencies:

```
python tests/processing_components/test_solve_jones.py
```

Example driver scripts are also available:

```
python examples/scripts/run_jones_solvers.py
python examples/scripts/run_AA0.5.py
```

CHAPTER

FOUR

EXAMPLE USAGE

An example script is also available in `jones_solvers/examples/scripts/run_jones_solvers.py`. This simulates a simple observation and calls function `solve_jones()` to find antenna-based Jones matrices.

The package uses the RASCIL classes for visibilities, array metadata generation and polarisation support.

```
import os
import sys
import time
import copy

import numpy as np

from numpy import sin as sin
from numpy import cos as cos

import matplotlib.pyplot as plt

from astropy.coordinates import SkyCoord
from astropy.time import Time
import astropy.units as u
import astropy.constants as consts

from rascil.data_models import PolarisationFrame
from rascil.processing_components import create_named_configuration
from rascil.processing_components import create_blockvisibility
from rascil.processing_components.util.geometry import calculate_azel
from rascil.processing_components.util.coordinate_support import lmn_to_skycoord
from rascil.processing_components.calibration.operations import create_gaintable_from_
↪blockvisibility

from jones_solvers.processing_components import solve_jones

import logging

log = logging.getLogger()
log.setLevel(logging.INFO)
log.addHandler(logging.StreamHandler(sys.stdout))

mpl_logger = logging.getLogger("matplotlib")
mpl_logger.setLevel(logging.WARNING)
```

(continues on next page)

(continued from previous page)

```
np.set_printoptions(linewidth=120)

# ------------------------ #

log.info("Init and predicting blockvisibility")

lowcore = create_named_configuration('LOWBD2-CORE')

# how can we extract these from lowcore?
lon = 116.76444824 * np.pi / 180.
lat = -26.82472208 * np.pi / 180.

vntimes = 1
integration_time = 1.0
times = (np.pi / 43200.0) * np.arange(0,vntimes*integration_time, integration_time)

vnchan = 1
channel_bandwidth = 1.0e6
frequency = np.arange(100.0e6, 100.0e6+vnchan*channel_bandwidth, channel_bandwidth)
channel_bandwidth = np.array(vnchan*[channel_bandwidth])

phasecentre = SkyCoord(ra=+15.0 * u.deg, dec=-45.0 * u.deg, frame='icrs', equinox='J2000
↪')

# create empty blockvis with intrumental polarisation (XX, XY, YX, YY)
xVis = create_blockvisibility(lowcore, times, frequency, channel_bandwidth=channel_
↪bandwidth, phasecentre=phasecentre,
                              integration_time=integration_time, polarisation_
↪frame=PolarisationFrame("linear"),
                              weight=1.0)

assert xVis['vis'].shape[0]   == vntimes, "Shape inconsistent with specified number of␣
↪times"
assert xVis['vis'].shape[2]   == vnchan, "Shape inconsistent with specified number of␣
↪channels"
assert xVis['vis'].shape[3]   == 4, "Shape inconsistent with specified number of␣
↪polarisations"
assert xVis['vis'].shape[0:3] == xVis["uvw_lambda"].data.shape[0:3], "vis & uvw_lambda␣
↪have inconsistent shapes"
assert all(xVis['polarisation'].data == ['XX', 'XY', 'YX', 'YY']), "Polarisations␣
↪inconsistent with expectations"

nvis = xVis["baselines"].shape[0]
```

To help with flexibility in the early stages of development, the package does not yet use RASCIL predict functionality. It also currently just uses a simple short-dipole beam model with a Gaussian taper. These will be replaced with standard sky and beam models in the future.

```
# Generate a sky model from Nsrc point-source components, randomly distributed around␣
↪the phase centre

Nsrc = 10
```

(continues on next page)

(continued from previous page)

```
dist_source_max = 2.5 * np.pi/180.0

# sky model: randomise sources across the field
theta = 2.*np.pi * np.random.rand(Nsrc)
phi = dist_source_max * np.sqrt( np.random.rand(Nsrc) )
l = sin(theta) * sin(phi)
m = cos(theta) * sin(phi)
n = np.sqrt(1-l*l-m*m) - 1
jy = 10 * np.random.rand(Nsrc)
for src in range(0,Nsrc):

    # analytic response of short dipoles aligned NS & EW to sky xy polarisations
    # with an approx Gaussian taper for a 35m station
    srcdir = lmn_to_skycoord(np.array([l[src],m[src],n[src]]), phasecentre)
    ra  = srcdir.ra.value * np.pi / 180.
    dec = srcdir.dec.value * np.pi / 180.
    sep = srcdir.separation(phasecentre).value * np.pi / 180.
    diam = 35.;

    # need to set ha,dec, but need to be in time,freq loop
    for t in range(0,len(xVis['datetime'])):

        utc_time = xVis['datetime'].data[t]
        #azel = calculate_azel(location, utc_time, srcdir);
        lst = Time(utc_time, location=(lon * u.rad, lat * u.rad)).sidereal_time('mean').
↪value * np.pi / 12.
        ha = lst - ra

        J00 =  cos(lat)*cos(dec) + sin(lat)*sin(dec)*cos(ha)
        J01 = -sin(lat)*sin(ha)
        J10 =  sin(dec)*sin(ha)
        J11 =  cos(ha)
        J = np.array([[J00,J01],[J10,J11]], "complex")
        # components are unpolarised, so can form power product now
        JJ = J @ J.conj().T

        for f in range(0,len(xVis['frequency'])):

            wl = consts.c.value / xVis['frequency'].data[f];
            sigma = wl/diam / 2.355;
            gain = np.exp( -sep*sep/(2*sigma*sigma) );

            srcbeam = JJ * gain
            log.debug(src,sep*180./np.pi,l[src]*180./np.pi,m[src]*180./np.pi,n[src],np.
↪array(srcbeam).flatten())

            # vis (time, baselines, frequency, polarisation) complex128

            uvw = xVis['uvw_lambda'].data[t,:,f,:]
            phaser = 0.5*jy[src] * np.exp( 2j*np.pi * (uvw[:,0]*l[src] + uvw[:,1]*m[src]␣
↪+ uvw[:,2]*n[src]) )
```

(continues on next page)

(continued from previous page)

```
            assert all(xVis['polarisation'].data == ['XX', 'XY', 'YX', 'YY']), xVis[
↪'polarisation'].data

            xVis['vis'].data[t,:,f,0] += phaser * srcbeam[0,0]
            xVis['vis'].data[t,:,f,1] += phaser * srcbeam[0,1]
            xVis['vis'].data[t,:,f,2] += phaser * srcbeam[1,0]
            xVis['vis'].data[t,:,f,3] += phaser * srcbeam[1,1]
```

Apply calibration factors to the visibilities and add noise

```
stations = lowcore["stations"]
nstations = stations.shape[0]

stn1 = xVis["antenna1"].data
stn2 = xVis["antenna2"].data

# set up station-based Jones matrices (gains and leakages)
#  - will change to gaintable data model
Jsigma = 0.1

# generate a gaintable with a single timeslice (is in sec, so should be > 43200 for a 12␣
↪hr observation)
#  - could alternatively just use the first time step in the call
#  - using Jones type "G" because type "P" is unknown
gt_true = create_gaintable_from_blockvisibility(xVis, timeslice=1e6, jones_type="G")
gt_fit  = create_gaintable_from_blockvisibility(xVis, timeslice=1e6, jones_type="G")

# set up references to the data
Jt = gt_true["gain"].data
Jm = gt_fit["gain"].data

for stn in range(0,nstations):

    # generate the starting model station gain error matrices
    Jm[0,stn,0,:,:] = np.eye(2, dtype=complex)

    # generate the true station gain error matrices. Set to model matrices plus some␣
↪Gaussian distortions
    Jt[0,stn,0,:,:] = Jm[0,stn,0,:,:] + Jsigma * ( np.random.randn(2,2) + 1j*np.random.
↪randn(2,2) )

# Make copies of the visibilties and multiply in the new Jones matrices
#  - assuming that unknown calibration errors are constant over time and frequency␣
↪samples (i.e. is a snapshot)

# Make copies of the vis and apply calibration factors and noise
modelVis     = xVis.copy(deep=True)
noiselessVis = xVis.copy(deep=True)

for t in range(0,len(xVis['datetime'])):
    for f in range(0,len(xVis['frequency'])):
```

(continues on next page)

(continued from previous page)

```
        # set up references to the data
        modelTmp     = modelVis['vis'].data[t,:,f,:]
        noiselessTmp = noiselessVis['vis'].data[t,:,f,:]

        for k in range(0,nvis):

            vis_in = np.reshape(modelTmp[k,:],(2,2))
            vis_out = Jm[0,stn1[k],0] @ vis_in @ Jm[0,stn2[k],0].conj().T
            modelTmp[k,:] = np.reshape(np.array(vis_out),(4))

            vis_in = np.reshape(noiselessTmp[k,:],(2,2))
            vis_out = Jt[0,stn1[k],0] @ vis_in @ Jt[0,stn2[k],0].conj().T
            noiselessTmp[k,:] = np.reshape(np.array(vis_out),(4))

# Add noise to a visibility
#  - will change to RASCIL addnoise_visibility, but for now just add gaussian noise so␣
↪there is more control over SNR

observedVis = noiselessVis.copy(deep=True)

sigma = 0.01
shape = observedVis['vis'].shape
assert len(shape) == 4, "require 4 dimensions for blockvisibilty"

observedVis['vis'].data += sigma * ( np.random.randn(shape[0],shape[1],shape[2],
↪shape[3]) +
                                     np.random.randn(shape[0],shape[1],shape[2],
↪shape[3]) * 1j )

if sigma > 0: observedVis['weight'].data *= np.ones(shape) / (sigma * sigma)
```

And then the jones_solvers package is used to solve for the Jones matrices:

```
gt1 = gt_fit.copy(deep=True)
modelVis1 = modelVis.copy(deep=True)
chisq1 = solve_jones(observedVis, modelVis1, gt1, testvis=noiselessVis, algorithm=1)
```

```
gt2 = gt_fit.copy(deep=True)
modelVis2 = modelVis.copy(deep=True)
chisq2 = solve_jones(observedVis, modelVis2, gt2, testvis=noiselessVis, accum_opt=2)
```

```
ax = plt.subplot(111)
ax.set_yscale('log')
plt.plot( chisq1, label="RTS" )
plt.plot( chisq2, label="Yanda" )
plt.xlabel("iteration")
plt.ylabel("chisq error")
plt.legend(loc=1, fontsize=11, frameon=False)
plt.grid()
plt.show()
```

CHAPTER

FIVE

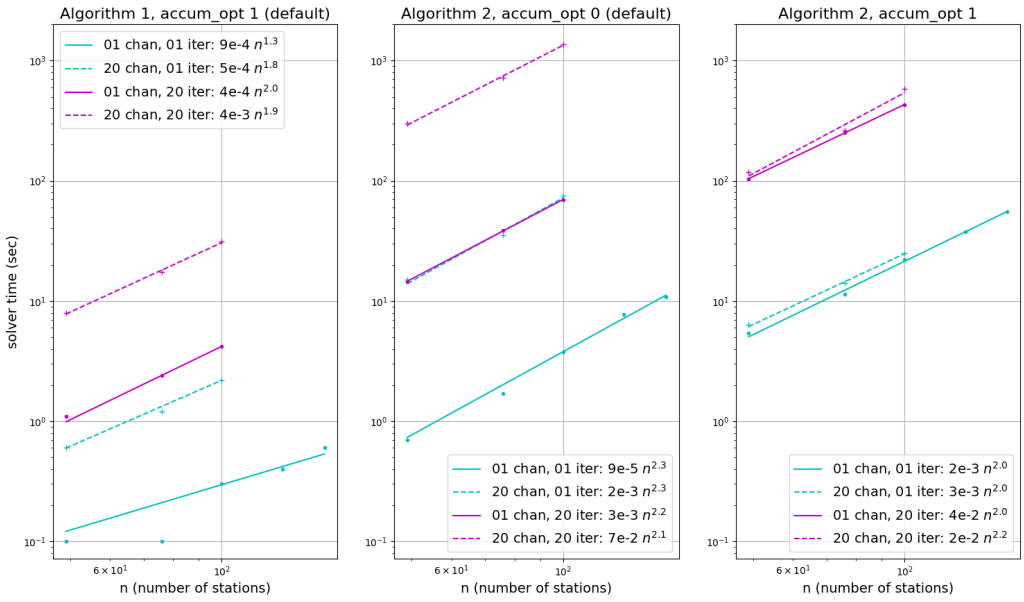
PERFORMANCE

5.1 Scaling

Algorithm 2 alternates between forming linear normal matrices for unknown, residual errors in the calibration model (station-based Jones matrices), and solving the normal equations to update the calibration model.

Algorithm 1 uses the fact that, for a large number of free parameters, the normal matrix is approximately block-diagonal with separate blocks for separate stations. As such, an approximation of the linear update at each iteration can be carried out separately for each station, with far fewer operations than algorithm 2. While algorithm 1 may take more iterations to converge, each one is faster and the overall runtime can be much lower. Particularly when the number of free parameters is large.

The figure below demostrates runtime for three solver options, as implemented in this library. Algorithm 1 runtimes with default options are given in the left-hand panel, algorithm 2 runtimes with default options are given in the centre, and algorithm 2 runtimes with accum_opt 1 are given on the right. The tables below the figure show runtimes in finer detail as the number of iterations is increased from one. Typical solutions may take 10 - 20 iterations, unless starting from existing values such as those from a previous time step or frequency.



- Algorithm 1, accum_opt 1, shows the shortest solver times, which scale with the number of free parameters squared. Runtime is dominated by a pass over the dataset during each iteration, and as such scales approximately

linearly with the number of time and frequency samples, and with the number of iterations. Each pass over the data could be made parallel in various ways (such as time, frequency, antenna), but once per iteration the various 2x2 accumulation matrices need to be gathered to a central solver process for each time-frequency solution interval. There is also scope for avoiding full passes over the dataset during each iteration, in a similar manner as accum_opt 1 of algorithm 2, depending on the commutation properties of the Jones matrices (for instance, depending on whether the calibration errors occur on the left or right of the primary beam Jones matrix). This option may be added in the future.

- Algorithm 2, accum_opt 0, forms the design matrix, A, and data vector, $r = vis - model$, once per iteration, and so, like algorithm 1, shows linear scaling with the number of time and frequency samples and the number of iterations. Scaling with the number of free parameters is more complicated. The accumulation of normal equations should scale roughly with the number of parameters squared – with various parallelism options, as in algorithm 1 – however solving the system of equations scales more rapidly and is less trivial to parallelise.
- Algorithm 2, accum_opt 1, accumulates the $A^H A$ and $A^H r$ matrices directly, but factorises them into products with coefficients that can be pre-summed before the iteration. While it can be seen that this is more time consuming than accum_opt 0 when the number of time and frequency samples is small – as may be the case for real-time calibation – runtime is much more stable as the number of time and frequency samples grows, thanks to the pre-summing. Like accum_opt 0, the accumulation process has various parallelism options. In Yandasoft, separate equations are formed in parallel across frequency and spectral Taylor term, before merging to a joint solver.

Table 1: nchan = 1

	solver 1, accum_opt 1	solver 2, accum_opt 0	solver 2, accum_opt 1
niter = 1	0.1 sec	1.7 sec	11.4 sec
niter = 2	0.2 sec	3.6 sec	22.7 sec
niter = 3	0.3 sec	5.5 sec	35.6 sec

Table 2: nchan = 20

	solver 1, accum_opt 1	solver 2, accum_opt 0	solver 2, accum_opt 1
niter = 1	1.2 sec	35.2 sec	14.0 sec
niter = 2	2.0 sec	73.4 sec	26.9 sec
niter = 3	2.8 sec	102.0 sec	37.7 sec

Table 3: nchan = 40

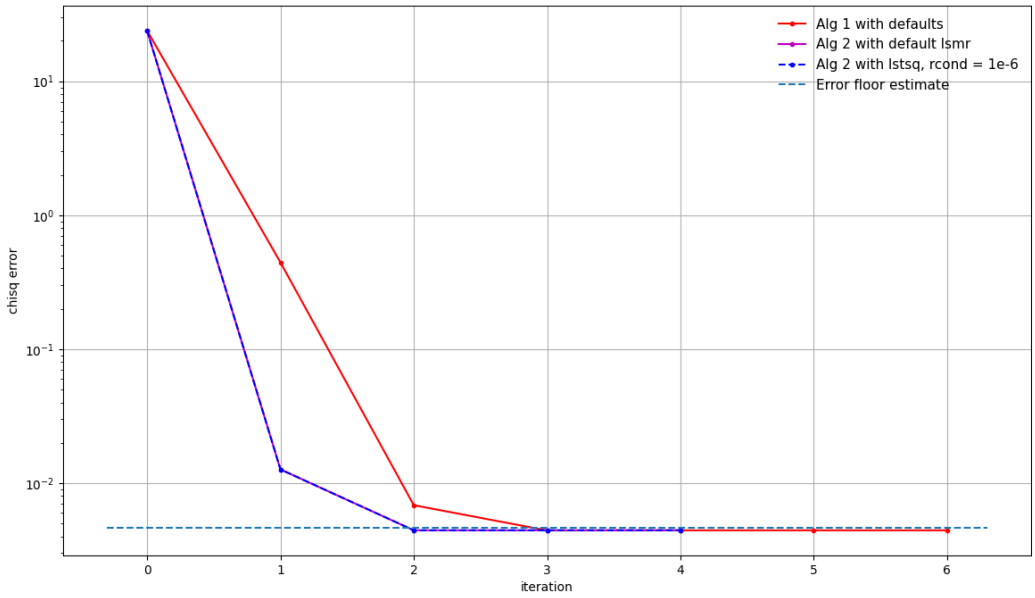
	solver 1, accum_opt 1	solver 2, accum_opt 0	solver 2, accum_opt 1
niter = 1	2.3 sec	69.0 sec	16.4 sec
niter = 2	3.9 sec	141.9 sec	31.7 sec
niter = 3	5.3 sec	201.2 sec	42.6 sec

5.2 Solutions

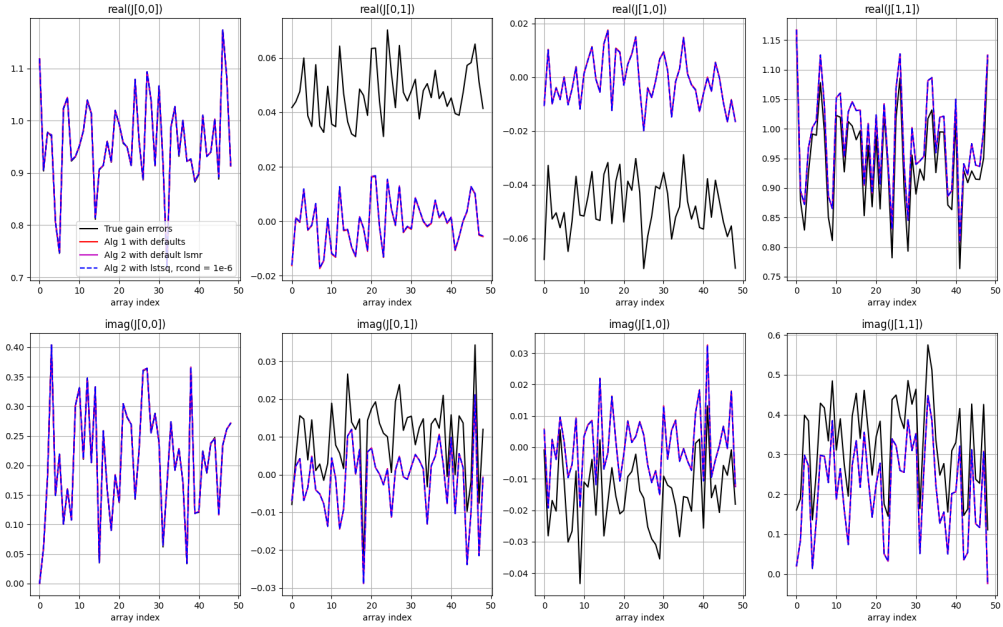
Here, algorithm 1, most suitable for RCAL, is compared with Algorithm 2. For this the SKA-LOW sensitivity calculator described in Sokolowski et al. 2022, PASA 39 was used to calculate the visibility noise level for the MWA EoR0 field at RA = 0 hrs, Dec = -27 deg. All tests were carried out using modified versions of example script `examples/scripts/run_AA0.5.py`.

Creating a sky model from a single, strong (100 Jy) point source in the centre of the field, all solvers and seen to generate equivalent solutions. The following figure shows the χ^2-error for three different solver options, with an array formed from 50 random stations from SKA-LOW and a simple Gaussian beam model. The calibration solution interval was

set to 10 seconds and 10 MHz, which may be reasonable for real-time calibration of LOW during early array releases. The reason for including two versions of algorithm 2 will become apparent shortly. While it takes algorithm 1 more iterations to reach the expected noise level, the runtime was 1-2 seconds, rather than 30.

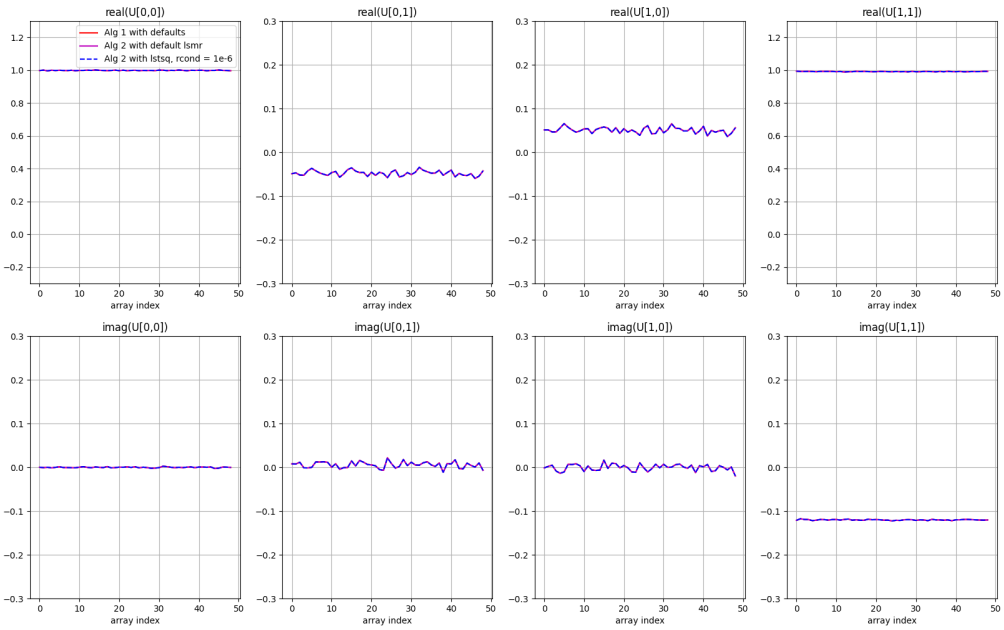


The next figure shows the solutions as coloured curves, relative to the input unknown Jones matrices in black. Each panel shows a different Jones matrix element, top panels real, bottom panels imaginary, with station index along the x axis. All data points have been phase referenced to the X polarisation receptor diagonal element of station 0.

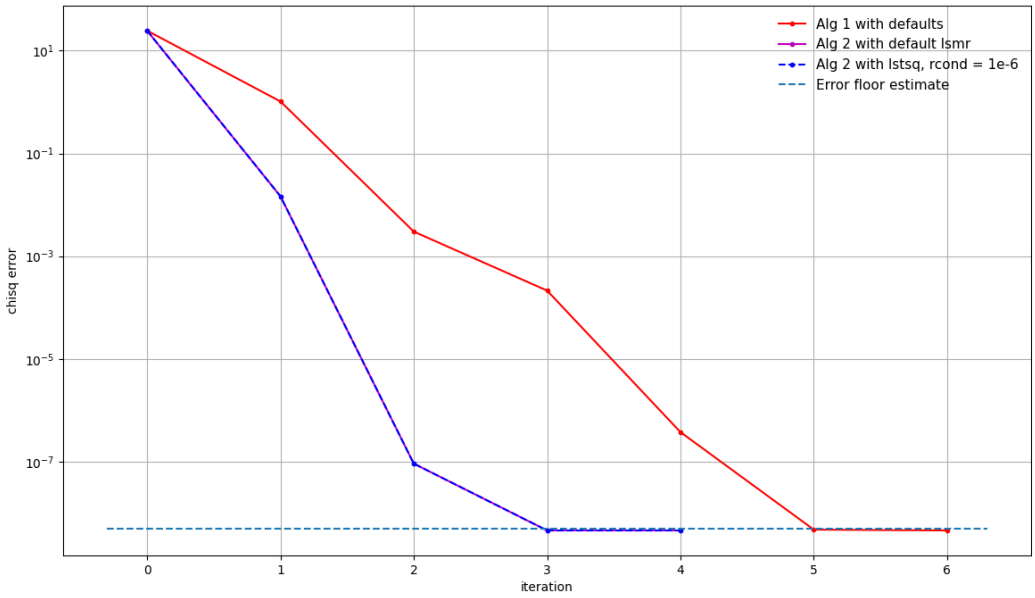


A few points to note. First, all three solvers result in the same solutions. However, there are offsets relative to the true values. This is more evident in the next figure, in which each solution matrix has been multiplied with the inverse of the true matrix. For ideal calibration the result would be 2x2 identity matrices, plus some noise. However a single, unpolarised cailbrator cannot constrain all of the unknowns, and ambiguities remain.

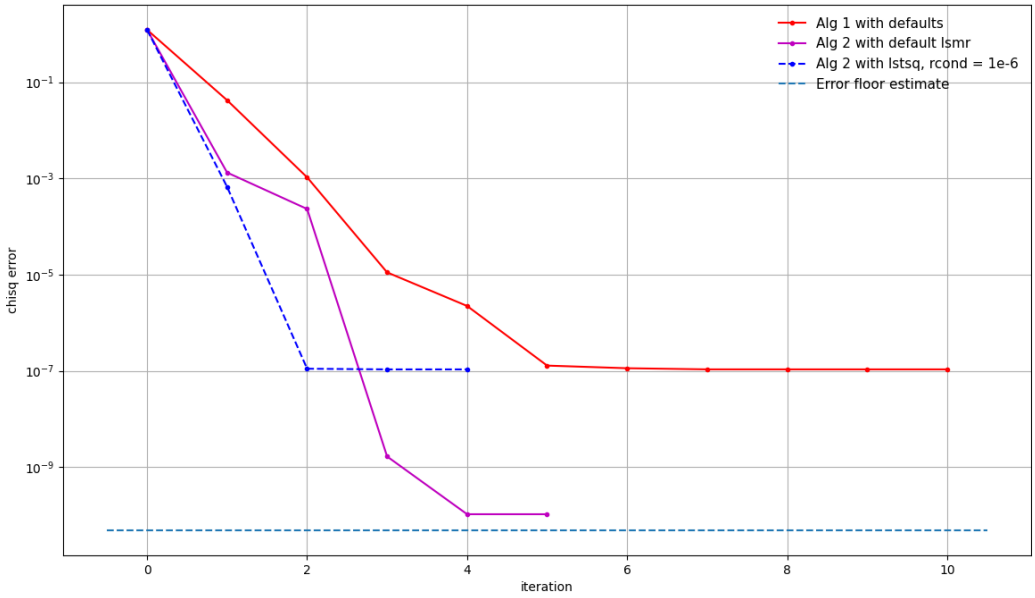
In the simulation, the input unknown Jones matrices comprised complex Gaussian random errors plus a systematic offset between the phase of the X and Y polarised receptors that was common to all stations, and systematic leakage between the receptors that was also common to all stations. The effect of XY-phase is seen as an offset from zero for the imaginary part of the Y diagonal term (noting that the phases have been referenced against the X receptor), and the effect of the leakage is seen as an offset from zero for the real part of the cross terms. The solutions have the expected form, and the separate polarisations are seen to be well calibrated.



If the noise is lowered, by adding more time or frequency samples, or more stations (or in this case by fudging the noise level in the simulation), the χ^2-error improves, but these ambiguities remain.

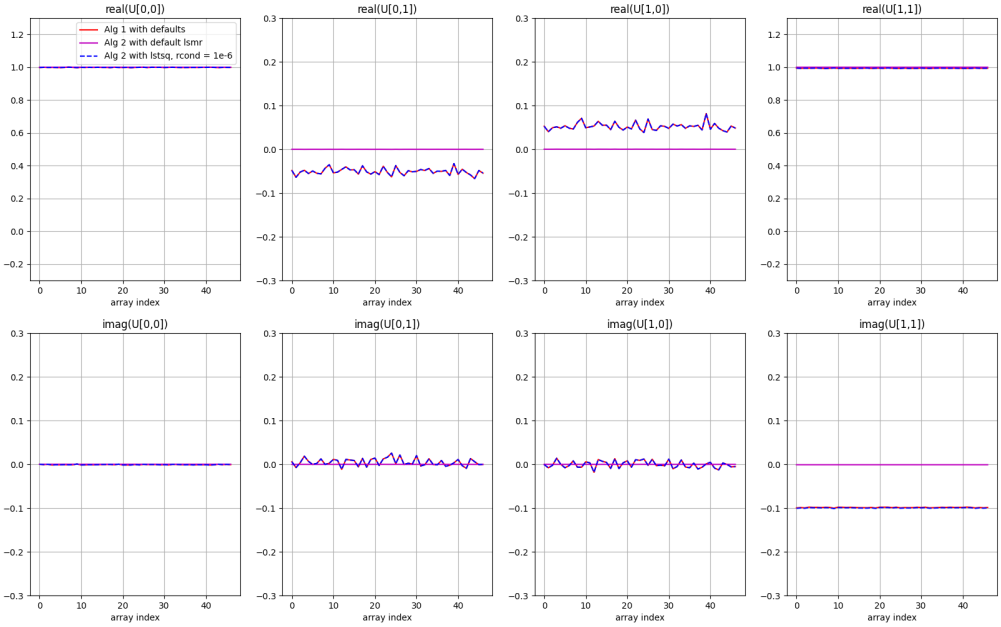


If we now replace the sky model with something more complex, such as the strongest 300 sources within 15 degrees of the EoR0 field, things look a bit different. Processing again with the high signal-to-noise ratio and the complete sky model, we see that the LSMR solver without the rcond cutoff performs better, while the other solvers hit a limit above the expected error level.



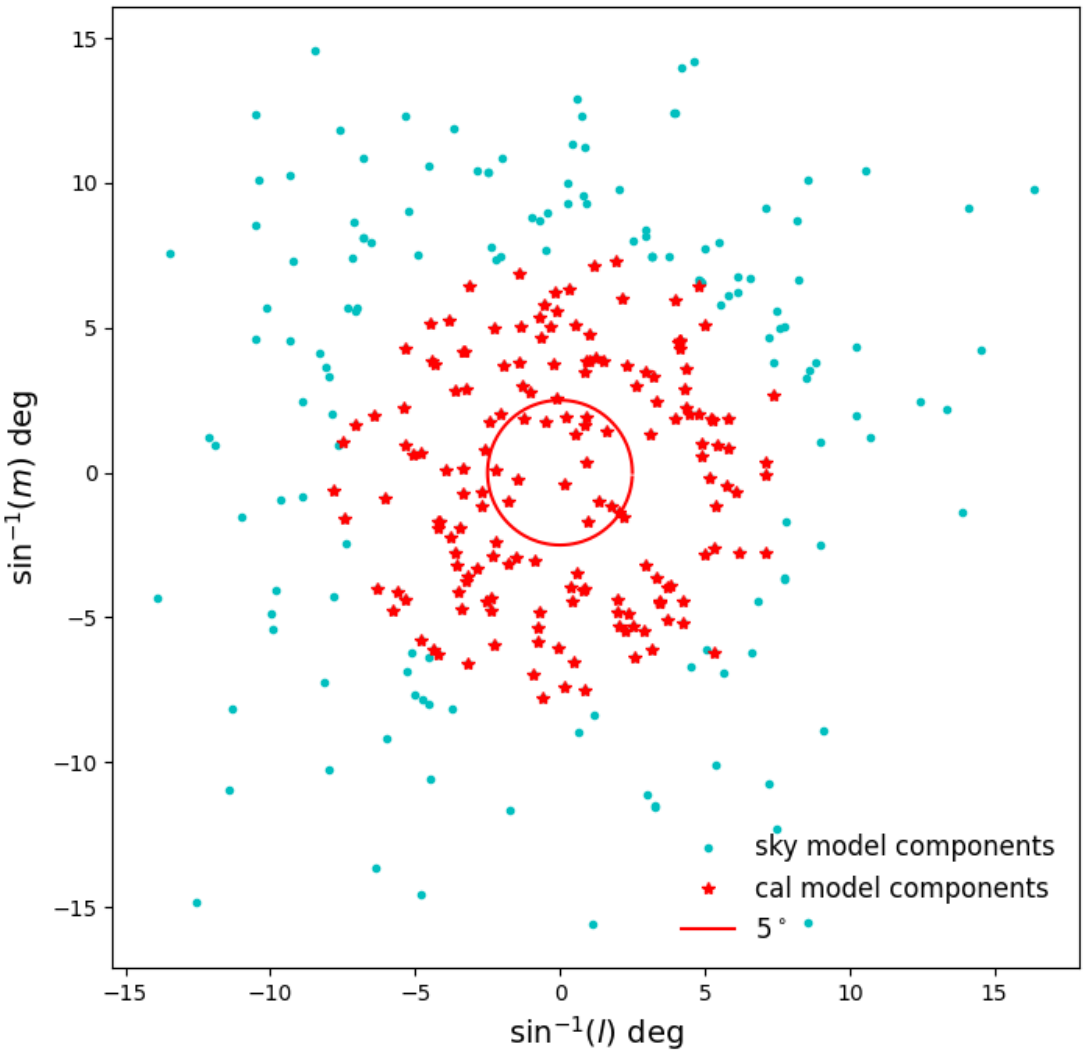
Looking at the calibration solutions, we can see that not only are the separate polarisations calibrated to a high accuracy, the ambiguities are also reduced. This is because of the extra polarisation information contained in the calibration model, coming from the instrument. It should be noted that it only performs at this level in high signal-to-noise

situations, where factors like singular-value cutoffs can be set very low. If the noise level is high or the calibration model is incomplete, ambiguities start to grow. This aspect of the solvers is under active investigation.

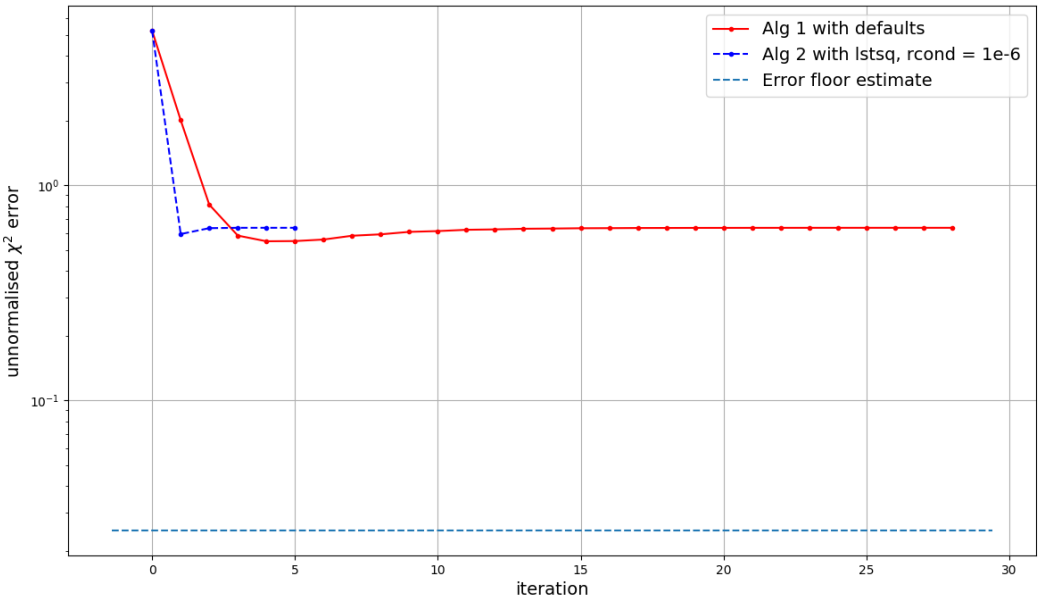


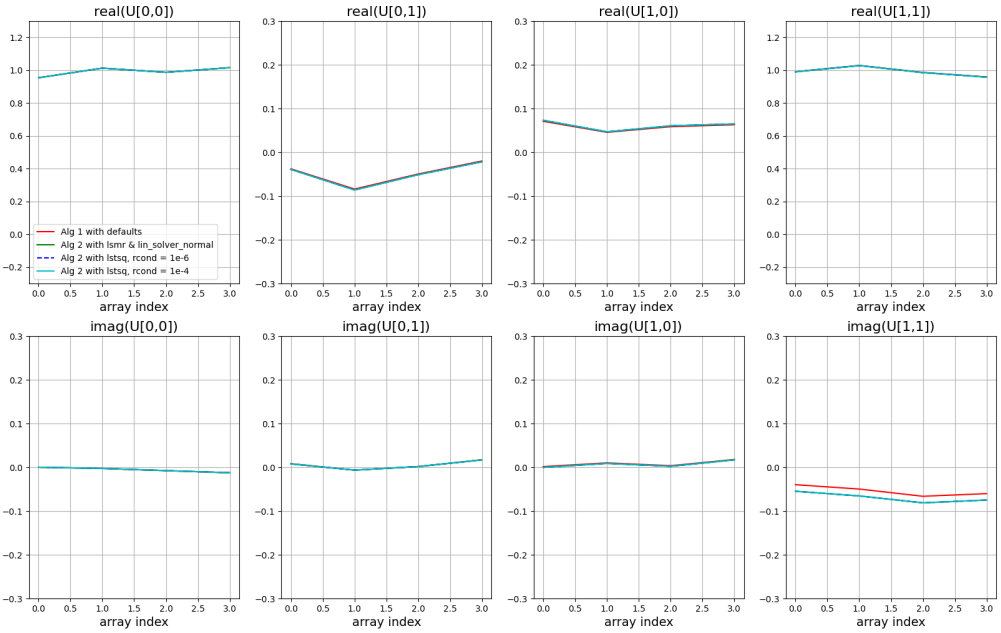
5.3 Small Arrays

A similar simulation was run for a four-station AA0.5 array and the initial realistic noise levels, with a slightly longer calibration solution interval of 30 seconds to decrease the noise level a little. Again the EoR0 field with 300 sources was used in the simulation, but to generate some sky model errors, only components with an apparent flux density of 1 mJy were included in the calibration model.



The χ^2-error shows some convergence, but the solutions hit a local minimum before reaching the expected error level. Nevertheless, the solutions are improved and coherent calibration is achieved for this quiet field without a dominant calibrator.





Todo:

- Fix up linting
- Time and frequency calibration intervals?
- Add single-polarisation gain solvers for comparison?

- Add pre-averaging for algorithm 1?
- Use names rather than indices for algorithms?
- Change name of the python package from jones_solvers to ska-sdp-jones-solvers?

CHAPTER

SIX

JONES SOLVERS DOCUMENTATION

6.1 Solvers

At present, a single set of matrices is found for all times and frequencies, as might be required for real-time calibration. For more general-purpose operation the solutions will need to be split into time and frequency intervals.

6.1.1 RTS-Style Solver

Algorithm 1 of `solve_jones()` iteratively updates the set of Jones matrices, but within each iteration performs an independent least-squares optimisation for each matrix without considering how the others are updating. This is equivalent to forming a normal matrix and setting the off-diagonal terms to zero. While it does not converge as fast as the normal-equation approach, it scales well in terms of operations and memory, and allows the free parameter for each antenna/station to be the 2x2 Jones matrix. It is based on the equivalent solver in the MWA RealTime System (Mitchell et at., 2008, IEEE JSTSP, 2, JSTSP.2008.2005327).

The equation to be minimised is similar to the scalar equation from RASCIL, but with 2x2 Jones matrices, *J*, instead of scalar gain terms and 2x2 coherency matrices, *V* instead of scalar visibilties:

$$S = \sum_{t,f} \sum_{i,j} w_{t,f,i,j} \left|\left| V^{\text{obs}}_{t,f,i,j} - J_i V^{\text{mod}}_{t,f,i,j} J^*_j \right|\right|^2_F$$

where $||A||^2_F$ the squared Frobenius norm of a matrix A, equal to the trace of AA^H.

6.1.2 Yandasoft-Style Solver

Algorithm 2 of `solve_jones()` is a full normal-equation based linear least-squares algorithm. It is based on the approach in Yandasoft. Normal equations can be generated via initial creation of a design matrix, which may be more efficient when calibrating on short time and frequency intervals, or normal-equation element products can be accumulated directly, allowing for an initial accumulation over time and frequency. In Yandasoft the latter was implemented by Max Voronkov using an automatic differentiation class for complex data and parameters. See Max's CalIm presentation for details.

Public API Documentation

6.1.3 Functions

A collection of solvers for antenna-based Jones matrices

Algorithm 1 of solve_jones iteratively updates the set of Jones matrices, but within each iteration performs an independent least-squares optimisation for each matrix without considering how the others are updating. This is equivalent to forming a normal matrix and setting the off-diagonal terms to zero. While it does not converge as fast as the normal-equation approach, it scales well in terms of operations and memory, and allows the free parameter for each antenna/station to be the 2x2 Jones matrix. It is based on the equivalent solver in the MWA RealTime System (Mitchell et at., 2008, IEEE JSTSP, 2, JSTSP.2008.2005327).:

```
Accumulation option 0: easy to read
Accumulation option 1: a bit faster (default)
```

Algorithm 2 is a full normal-equation based linear least-squares algorithm. It is based on the approach in Yandasoft.:

```
Accumulation option 0: via design matrix (default)
Accumulation option 1: direct accumulation of normal matrix with pre-summing of normal␣
↪equation products in time and frequency.
```

For example:

```
solve_jones(vis, modelvis, jones, niter=25)
```

jones_solvers.processing_components.solve_jones.**solve_jones**(*vis: rascil.data_models.memory_data_models.BlockVisibility*, *modelvis: rascil.data_models.memory_data_models.BlockVisibility*, *jones: rascil.data_models.memory_data_models.GainTable*, *niter=30*, *nu=None*, *tol=1e-06*, *algorithm=1*, *accum_opt=None*, *lin_solver='lsmr'*, *lin_solver_normal=None*, *lin_solver_rcond=1e-06*, *testvis: Optional[rascil.data_models.memory_data_models.BlockVi = None*)

Solve Jones matrices by fitting observed visibilities to model visibilities

A single set of matrices is found for all times and frequencies.

Parameters

- **vis** – BlockVisibility containing the observed visibility data
- **modelvis** – BlockVisibility containing the predicted data_model (updated on exit)
- **jones** – Existing GainTable containing the Jones matrices (updated on exit)
- **niter** – Number of iterations (default 30)
- **nu** – iterative adaptation update factor (default variable)
- **tol** – Convergence fractional-change tolerance (default 1e-6)
- **algorithm** – Solver algorithm used (default 1)

- **accum_opt** – Accumulation option (1 for algorithm 1, 0 for algorithm 2)
- **lin_solver** – linear solver used in each iteration of algorithm 2: "lsmr" (scipy, default), "lsqr" (scipy), "lstsq" (numpy) or "svd" (numpy)
- **lin_solver_normal** – whether or not to form normal equations before calling lin_solver (default False where allowed: accum_opt 0 with lsmr or lsqr). Otherwise True
- **lin_solver_rcond** – cutoff ratio used when lin_solver = "svd" or "lstsq" (default 1e-6)
- **testvis** – Optional BlockVisibility for comparisons (e.g. noiseless simulated data). Generates matplotlib output

Returns

numpy array containing chisq values as a function of iteration

CHAPTER

SEVEN

JONES SOLVERS

PYTHON MODULE INDEX

j

INDEX

J

M

S