jacal Documentation

YANDA Team

Oct 12, 2022

CONTENTS:

1 Introduction **3**
- 1.1 DALIuGE apps . . . 3
- 1.2 Using Yandasoft in DALiuGE . . . 3

2 Installation **5**
- 2.1 Dependencies . . . 5
- 2.2 Docker Image Installation . . . 5
- 2.3 Bare-metal Installation . . . 6

3 Example Usage with EAGLE **9**
- 3.1 Outline . . . 9
- 3.2 Preparing the Graph . . . 9
- 3.3 Inspecting and Editing the Graph . . . 10
- 3.4 Translating . . . 12
- 3.5 Deploying . . . 12

4 Example Usage from the Command Line **13**
- 4.1 Outline . . . 13
- 4.2 Preparing the Graph . . . 13
- 4.3 Starting DALiuGE . . . 14
- 4.4 Running . . . 14

5 Imaging Overview **17**
- 5.1 JACAL Architecture . . . 17
- 5.2 Yandasoft Data Classes . . . 19
- 5.3 JACAL Data Classes . . . 19
- 5.4 JACAL Interfaces . . . 21

6 API **25**
- 6.1 Available applications . . . 25
- 6.2 Others . . . 31

7 Indices and tables **33**

Index **35**

Joint **A**stronomy **CAL**ibration and imaging software

CHAPTER

ONE

INTRODUCTION

JACAL integrates Yandasoft (previously known as *ASKAPSoft*) and the execution framework DALiuGE. A shared library offers a calling convention supported by DALiuGE and internally links and reuses Yandasoft code. JACAL is freely available in GitLab under a variation of the open source BSD 3-Clause [License](LICENSE). The repository contains the following:

- The C/C++ code of the shared library `libjacal.so` described above.
- A number of tests running the different components inside DALiuGE graphs.
- A standalone utility for library testing independent of DALiuGE.

1.1 DALIuGE apps

The way JACAL integrates Yandasoft into DALiuGE is by wrapping individual pieces of functionality into DALiuGE-compatible applications that can then be deployed on a DALiuGE graph.

DALiuGE is an execution framework where programs are expressed as directed acyclic graphs, with nodes representing not only the different computations performed on the data as it flows through the graph, but also the data itself. Both types of nodes are termed *drops*. Computation drops (in DALiuGE, *application drops*) read or receive data from their input data drops, and write the results into their output data drops. Data drops on the other hand are storage-agnostic and host-agnostic, meaning that regardless of underlying storage and location application drops can work with their inputs and outputs in the same way.

Although application drops can be implemented in many ways, DALiuGE offers out-of-the-box support for certain type of applications. Among those, *shared libraries* can be written by users to implement application drops. This capability allows reusing code written in C, C++ or other low-level languages to work as application drops in a DALiuGE graph.

1.2 Using Yandasoft in DALiuGE

Before JACAL, the only way to use the Yandasoft functionality was to invoke the binaries it generates (e.g., `cimager`, `cbpcalibrator`, etc.); composition was only possible by arranging pipelines using shell scripts and similar techniques, and with data having to touch disk between each invocation of the binaries.

JACAL on the other hand implements a shared library (i.e., `libjacal.so`) wrapping different parts of Yandasoft as DALiuGE-ready application drops. This makes it possible to reuse finer-grained pieces of functionality from the Yandasoft code base, and with data not having to be necessarily written to disk between these steps.

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

JACAL has two main dependencies (which in turn might require a lot more):

- The DALiuGE execution framework, and
- The Yandasoft libraries

Installation for both dependencies is covered below.

2.2 Docker Image Installation

The following installation instructions are recommended for deployment on a laptop or workstation.

2.2.1 Building the Images

DALiuGE is available from the ICRAR github repo. There are three packages:

- *daliuge-common* – the base image containing the basic DALiuGE libraries and dependencies.
- *daliuge-translator* – the DALiuGE translator, built on top of the base image. This converts Logical Graphs into Physical Graphs.
- *daliuge-engine* – the DALiuGE execution engine, built on top of the base image.

```
cd <install dir>
git clone https://github.com/ICRAR/daliuge.git
cd daliuge/daliuge-common; ./build_common.sh dev
cd ../daliuge-translator; ./build_translator.sh dev
cd ../daliuge-engine; ./build_engine.sh dev
```

In JACAL, the standard daliuge-engine is replaced with a JACAL version that is based on both the DALiuGE base image and the Yandasoft image.

```
cd <install dir>
git clone https://gitlab.com/ska-telescope/ska-sdp-jacal
cd ska-sdp-jacal; ./build_engine.sh jacal
```

EAGLE is a web-based visual workspace for generating Logical Graphs.

```
cd <install dir>
git clone https://github.com/ICRAR/EAGLE.git
cd EAGLE; ./build_eagle.sh dev
```

2.2.2 Running the Images

Now that the images have been built, they need to be run. After this they can be graphed and deployed via EAGLE.

```
cd <install dir>
cd daliuge/daliuge-translator; ./run_translator.sh dev
```

```
cd <install dir>
cd ska-sdp-jacal; ./run_engine.sh jacal
```

```
cd <install dir>
cd EAGLE; ./run_eagle.sh dev
```

2.3 Bare-metal Installation

Note: For most use cases the docker installation described above is recommended.

2.3.1 DALiuGE

See DALiuGE documentation for up-to-date installation and usage information.

2.3.2 Yandasoft

See Yandasoft documentation for up-to-date installation and usage information. Note that Yandasoft has a list of dependencies on its own, including casacore, wcslib, cfitsio, fftw, boost, log4cxx and gsl.

2.3.3 JACAL

Once DALiuGE and Yandasoft are installed, JACAL itself can be built. JACAL uses the CMake build system, hence the build instructions are those one would expect:

```
cd <install dir>
git clone https://gitlab.com/ska-telescope/ska-sdp-jacal
cd ska-sdp-jacal
mkdir build
cd build
cmake ..
make
```

This process should generate a `libjacal.so` shared library which one can use within DALiuGE's `DynlibApp` components. Stand-alone executables are also produced under `test`, which are used for testing the code outside the context of DALiuGE.

CHAPTER

THREE

EXAMPLE USAGE WITH EAGLE

In this page we briefly describe how to use JACAL in a DALiuGE graph. This assumes you already *built* JACAL.

3.1 Outline

In this example we will replicate one of the unit tests run in the GitLab CI pipeline, namely `test_basic_imaging`. This test performs basic imaging on an input MeasurementSet using the *CalcNE* and *SolveNE* JACAL components, via the EAGLE web interface.

In DALiuGE a program is expressed as a *graph*, with nodes listing applications, and the data flowing through them. Graphs come in two flavours: *logical*, expressing the logical constructs used in the program (including loops, gather and scatter components), and *physical*, which is the fully-fledged version of a logical graph after expanding all the logical constructs.

This test is expressed as a *logical graph*. After translation into a *physical graph* it is submitted for execution to the DALiuGE *managers*, which were started by running the docker containers during *installation*. During execution one can monitor the progress of the program. This is all handled via a browser using EAGLE.

3.2 Preparing the Graph

For this test you will need to download the JSON logical graph. and the input dataset. When running Yandasoft applications, one typically provides a text file known as a *parset* containing configuration options. However this graph uses a DALiuGE drop to provide the configuration options. First, download the graph to a local directory:

```
$> wget https://gitlab.com/ska-telescope/ska-sdp-jacal/-/raw/master/jacal/test/daliuge/
↪test_basic_imaging_parset.json
```

Running image `icrar/daliuge-engine:jacal` generates a temporary work directory that is accessible both inside and outside the docker container:

```
$> ls -l /tmp/.dlg
code/
logs/
testdata/
workspace/
```

The graph assumes that the dataset is located in the testdata directory, so place it there:

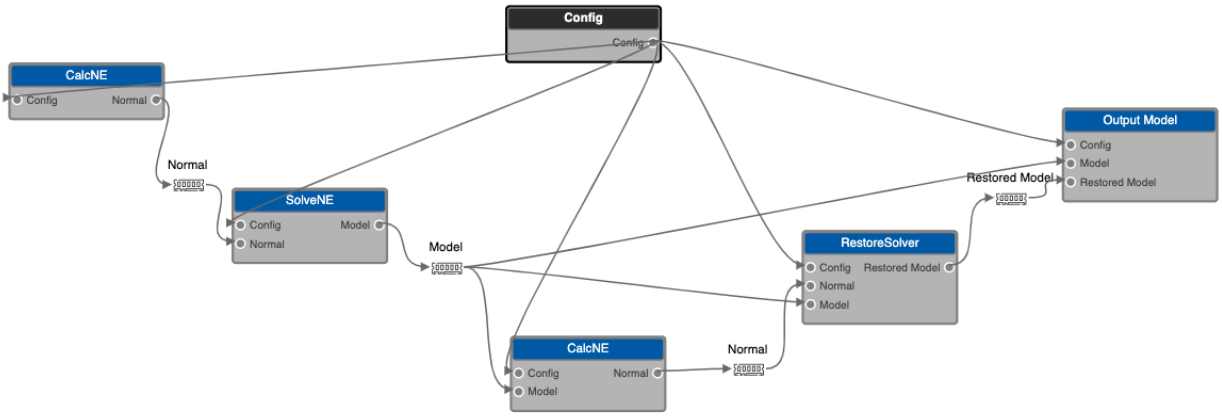
```
$> cd /tmp/.dlg/testdata
$> wget https://gitlab.com/ska-telescope/ska-sdp-jacal/-/raw/master/data/chan_1.ms.tar.gz
$> tar xf chan_1.ms.tar.gz
```

The final assumption in the graph is the specificatin of the JACAL library within the deocker container, which is set to `/usr/local/lib/libjacal.so`.

3.3 Inspecting and Editing the Graph

3.3.1 Deploying a Graph Locally

After *installation* on the local host, the DALiuGE translator will be connected to port 8084 and EAGLE to port 8888. Open a browser and go to `http://localhost:8888/`. This will bring up the EAGLE visual workspace. From the *Graph* drop-down menu, select *Local Storage* and then *Load Graph*. Navigate to test_basic_imaging_parset.json and load it into the browser.

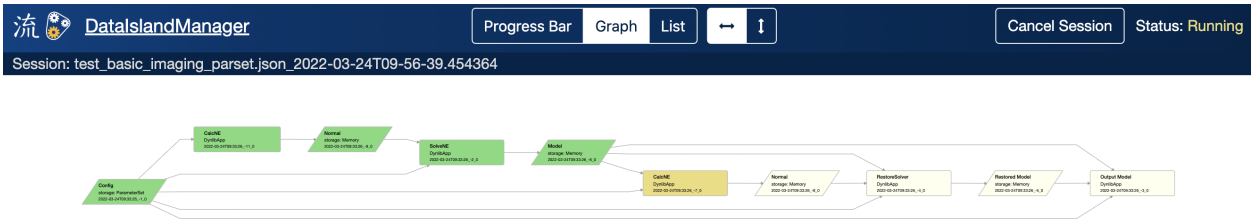


This graph performs imaging with a single major cycle of deconvolution. The graph is just for demonstration and the cleaning is not very deep. See the *imaging overview* for more details. Clicking on each component brings up an *Inspector* in the right-hand panel that can be used to read and edit run-time parameters. Of particular interest are *Component Parameters* and *I/O Ports*. The *Config* component generates imaging parset, and imaging parameters can found by inspecting its *Component Parameters*.

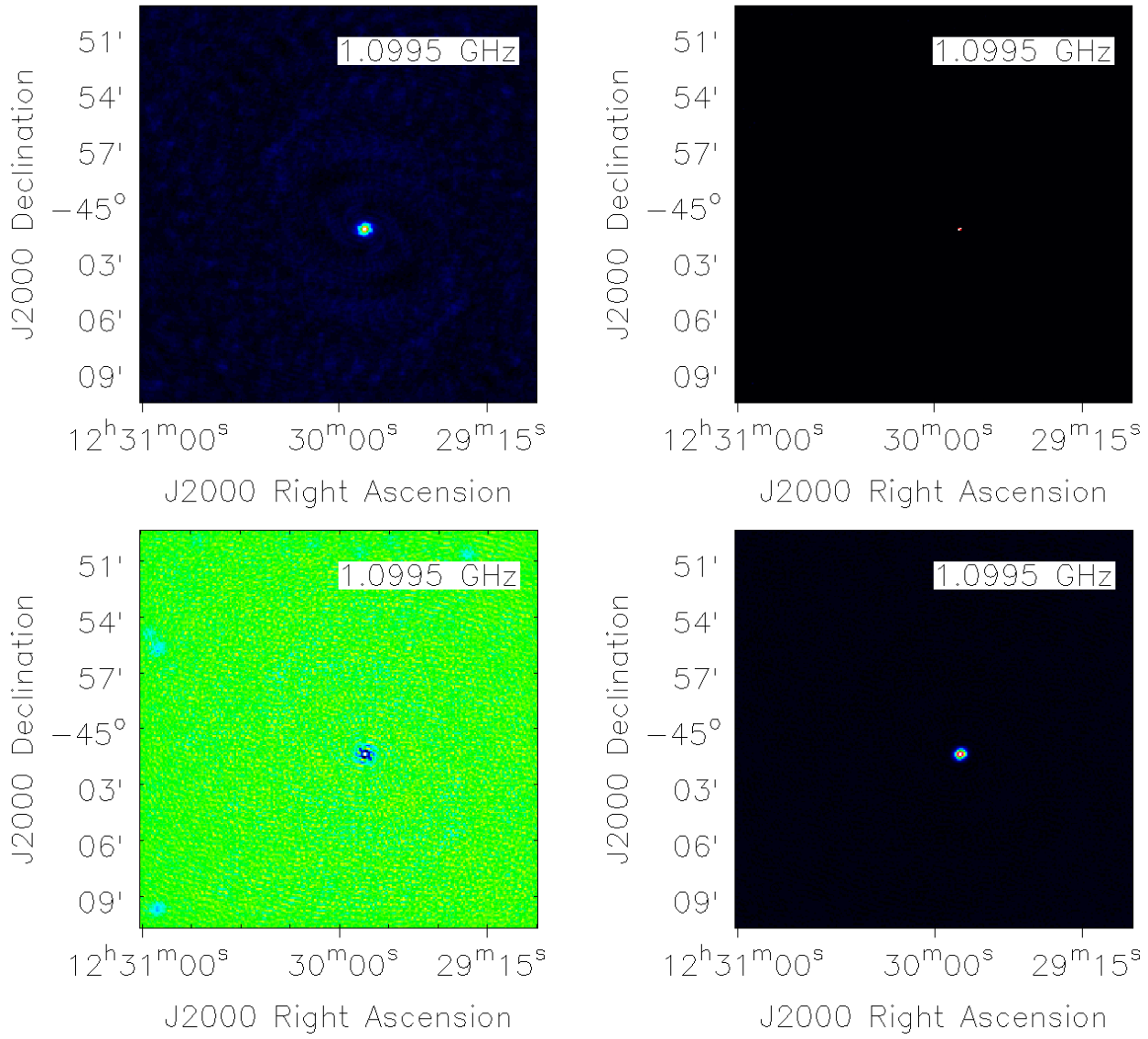
Note: To edit *Component Parameters* some of the *Advanced Editing* setting may need to be altered via the cog in the top right corner.

Before translating and deploying the graph, a few options need to be configured. The most important is the location of the EAGLE translator. The cog in the top right corner brings up EAGLE configuration settings, and under *External Services* is the *Translator URL* box. Set this to `http://localhost:8084/gen_pgt`, the local translator running in the container with port 8084 forwarded from the localhost. Setting GitHub and GitLab *Access Tokens* can also be done under *External Services*, allowing EAGLE to access various repositories.

The logical graph is now ready to be partitioned into a physical graph. From the *Translation* tab in the right-hand panel, select *Generate PGT* to generate a Physical Graph Template that can be mapped to compute nodes. This should open the DALiuGE translator interface in a new browser tab. Click on the Translator settings cog and check the *DALiuGE Manager URL*. This should be set to port 8001 of the actual IP address, e.g. `http://130.155.199.71:8001/`. Now select *Deploy* to generate and deploy the physical graph. This brings up a new browser tab which displays the progress of the graph.



When the graph has completed, the resulting images can be found in `/tmp/.dlg/workspace`. These are shown in the following figure. From the top left is the initial dirty image, the clean component image, the residual image and the restored image.



Runtime logs can be displayed via docker:

```
$> docker logs daliuge-engine
```

3.3.2 Deploying a Graph Remotely

Same as above but point to hostname or IP address rather than localhost.

3.3.3 Deploying a Graph on a Cluster

Todo

3.4 Translating

3.5 Deploying

CHAPTER

FOUR

EXAMPLE USAGE FROM THE COMMAND LINE

In this page we briefly describe how to use JACAL in a DALiuGE graph. This assumes you already *built* JACAL.

4.1 Outline

In this example we will replicate one of the unit tests run in the GitLab CI pipeline, namely `test_basic_imaging`. This test performs basic imaging on an input MeasurementSet using the *CalcNE* and *SolveNE* JACAL components. The other unit tests work similarly, exercising different JACAL components in different modes of operation.

In DALiuGE a program is expressed as a *graph*, with nodes listing applications, and the data flowing through them. Graphs come in two flavours: *logical*, expressing the logical constructs used in the program (including loops, gather and scatter components), and *physical*, which is the fully-fledged version of a logical graph after expanding all the logical constructs.

This test is expressed as a *logical graph*. After translation into a *physical graph* it is submitted for execution to the DALiuGE *managers*, which need to be started beforehand. During execution one can monitor the progress of the program via a browser.

4.2 Preparing the Graph

This test needs a few inputs:

- The logical graph.
- A parset (parsets are text files containing configuration options, and are the configuration mechanism used throughout Yandasoft).
- Some input data.

Put all three files above in a new directory, and then decompress the input data:

```
$> mkdir tmp
$> cd tmp
$> export TEST_WORKING_DIR=$PWD
$> wget https://gitlab.com/ska-telescope/ska-sdp-jacal/-/raw/master/jacal/test/daliuge/
↪test_basic_imaging.json?inline=false
$> wget https://gitlab.com/ska-telescope/ska-sdp-jacal/-/raw/master/jacal/test/daliuge/
↪test_basic_imaging.in?inline=false
$> wget https://gitlab.com/ska-telescope/ska-sdp-jacal/-/raw/master/data/chan_1.ms.tar.
↪gz?inline=false
```

(continues on next page)

(continued from previous page)

```
$> tar xf chan_1.ms.tar.gz
$> PARSET=$PWD/test_basic_imaging.in
```

Next, some adjustments will need to be made to the graph so that the JACAL shared library can be found, and the parset is correctly read at runtime:

```
$> sed -i "s|%JACAL_SO%|$PATH_TO_JACAL_SO|g; s|%PARSET%|$PARSET|g" test_basic_imaging.
↪json
```

4.3 Starting DALiuGE

Firstly, one needs to start the DALiuGE *managers*, the runtime entities in charge of executing graphs. We will start two: the *Node Manager* (NM), in charge of executing the graph, and a *Data Island Manager* (DIM), in charge of managing one or more NMs. Note that starting the DIM is not strictly required, but is done for completeness.

Start the managers each on a different terminal so you can see their outputs independently. Also, to make the test simpler, start both in the same directory where the downloaded files are placed:

```
$> cd $TEST_WORKING_DIR
$> dlg nm -v
$> dlg dim -N 127.0.0.1 -v
```

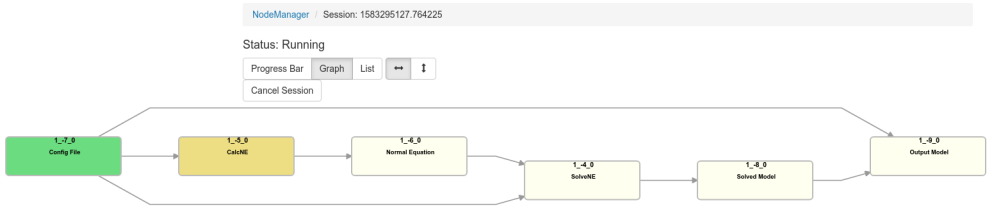
4.4 Running

To execute a graph we submit it to one of the DALiuGE managers (in our case, the DIM). Also, because we are starting from a logical graph, we need to transform it into a physical graph that can be run on the deployed managers.

This can be done as follows:

```
$> cd $TEST_WORKING_DIR
$> cat test_basic_imaging.in \
   | dlg unroll-and-partition `# Logical -> Physical translation` \
   | dlg map                  `# Assign nodes to drops (i.e., schedule the graph)` \
   | dlg submit -w            `# Submit and wait until execution finishes`
```

Finally, connect to `127.0.0.1:8000` to see the graph running:



Note that *CalcNE* now supports new gridders with more flexible data partitioning. This can be enabled with JACAL-specific parset parameter *Cimager.gridder.dataaccess=datapartitions* (set to *yandasoft* or leave unset to use the Yandasoft data iterators and gridders). The type of partitioning is set with JACAL-specific parset parameter *Cimager.gridder.partitiontype*:

```
GitHub: ICRAR/EAGLE-graph-repo (master): examples/jacal_CalcNElight.graph
```

CHAPTER

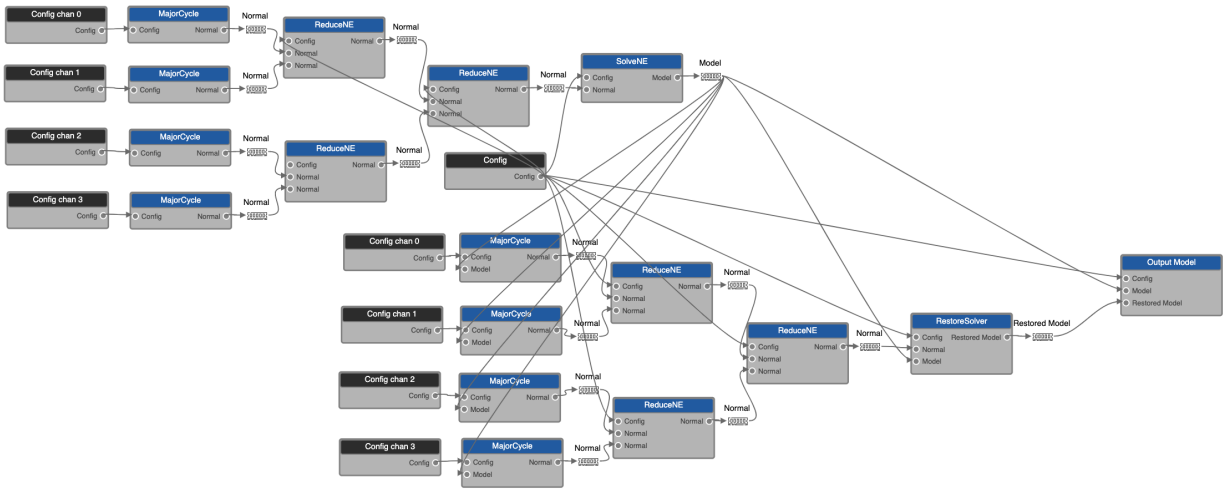
FIVE

IMAGING OVERVIEW

5.1 JACAL Architecture

JACAL is a package in which elements of Yandasoft have been extracted from their MPI-based framework and reset within DALiuGE. The current level of granularity of imported code is one level down from the Yandasoft imaging applications, with separate DALiuGE drops for major cycles (visibility inversion and prediction) and minor cycles (image based deconvolution). At lower levels the Yandasoft libraries are used, and at higher levels DALiuGE is used. As described below, recent updates have added an extra level of control at the major cycle level, with separate drops for visibility ingest, inversion and prediction. This will become the standard level used in JACAL going forward.

Standard Yandasoft continuum imaging has visibility inversion and prediction tasks partitioned across a cluster, with different MPI ranks handling one or more spectral channels and/or Taylor terms. A range of gridders and degridders are available for these tasks. After prediction, subtraction and inversion, the intermediate images are merged to a single MPI rank for joint image-based preconditioning and deconvolution. Such a setup can also be achieved in JACAL by expanding the example graph from the *EAGLE Usage* section. For example, the following graph generates dirty and PSF images separately for four spectral channels, which are then reduced (merged) into a single set that are passed to the solver for deconvolution. The output clean-component sky model is distributed to another set of major cycle components for visibiltiy prediction, subtraction and inversion, and the resulting residual images are again reduced to a single image for restoring and then output.



The images being reduced are shown as *Normal* data drops in the graph, which, as described below, are BlobString copies of Yandasoft *ImagingNormalEquations* objects. These objects contain sets of imaging products, including the main dirty/residual image hypercube (polarisation, frequency and direction axes), the PSF hypercube, an alternative PSF hypercube used for preconditioning if required, and a weights hypercube containing primary beam pixel weights if an appropriate gridder was used. As the frequencies all have the same image coordinates, the merging is a simple

pixel-wise accumulation. Sky models sent for prediction and restoring are BlobString copies of Yandasoft *Params* objects. These are used throughout Yandasoft calibration and imaging, but in this context contain just the solved imaging parameters (a clean component hypercube for each Taylor term) and a small amount of associated metadata. Both of these classes are defined in the Yandasoft `base-scimath` library.

On the DALiuGE side, arbitrary parallelism can be achieved using *Scatter* drops, and multiple major cycle / minor cycle iterations can be realised using a *Loop* drop. The one feature missing at present is the tree reduction merge, which is implemented in the graph as a series of parallel binary reductions. For N parallel major cycle components, this restricts the depth of reduction to $\log_2 N$ rather than N sucessive merges. A binary tree reduction version of the DALiuGE *Gather* drop has been earmarked for development in the near future.

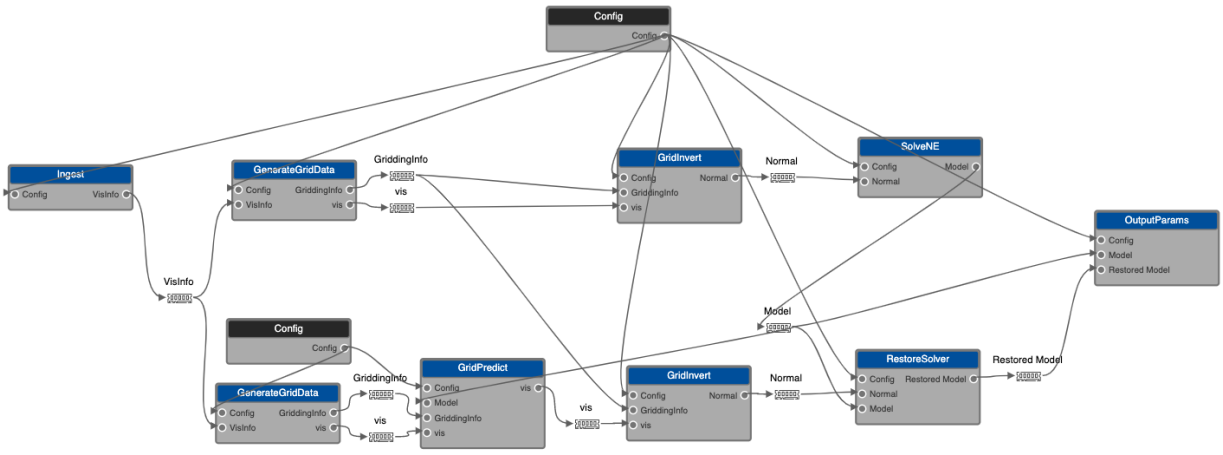
One difference between Yandasoft MPI-based parallelism and that of DALiuGE is the independence or otherwise of processes. In Yandasoft, cycling over major and minor cycles is achieved by alternating between parallel major cycle tasks, running on the majority of processes, and the minor cycle tasks running on a single process. Each MPI rank deals with the same set of channels and Taylor terms throughout the imaging process, making it simple and efficient to cache things like visibilities and gridding kernels. However a DALiuGE scatter is set up differently. DALiuGE drops running on a given node either all use the same process space or all use different ones. This makes it hard to cache data between cycles, however it is the more general and flexible approach from a large-scale HPC perspective, where fixing data to specific nodes may not be the optimal solution. An approach better suited to DALiuGE would be to use data drops for persistant data, with the framework deciding what can be cached and what needs to be moved between compute nodes. The partitioning and interfaces described on this page will evolve as JACAL moves in this direction.

5.1.1 Separate Major Cycle Drops

The main components of the major cycle have been separated out as separate drops.

- *Ingest*: Read data from measurement set and fill a VisInfo partition, including the generation of a visibility data cache.
- *GenerateGridData*: Convert VisInfo partition into a GriddingInfo partition, including the generation of gridding kernel cache. Can be distinct for different types of gridders (e.g. gridders in GridInvert and degridders in GridPredict).
- *GridInvert*: Grid data and weights to form dirty and PSF images.
- *GridPredict*: Degrid model images to form model visibilities, which are subtracted from input visibilities to form output residual visibilities.

An expanded version of the example graph from the *EAGLE Usage* section is given below.



5.2 Yandasoft Data Classes

5.2.1 ImagingNormalEquations

Yandasoft `base-scimath` *ImagingNormalEquations* contain the products generated in the imaging process. A given object can contain multiple sets of products, which may be related to one another (such as Taylor term images) or may be entirely indpendent (although multiple independent imaging paths are not supported in JACAL). The names of the equations determine the relationship. Each equation contains the main dirty image hypercube, which may have additional axes for polarisation and frequency, the PSF hypercube, an alternative PSF hypercube used for preconditioning if required, and a weights hypercube containing primary beam pixel weights if an appropriate gridder was used. Depending on the imaging parameters, the shape, coordinates and reference pixel may also be stored.

The *ImagingNormalEquations* class also contains a number of functions for manipulating data. Of most relevance here is the `merge()` function. When one set of equations is merged into another, any equation with a distinct name is stored separately, however any equations with the same name are combined. If they have the same shape and coordinates, the various cubes are simply added together. If the shape or coordinates are different, as they would be for multi-beaming or facets, accumulation occurs after the image cubes have been weighted by the weights cubes and reprojected to common coordaintes.

5.2.2 Params

Yandasoft `base-scimath` *Params* are used throughout Yandasoft calibration and imaging as arbitrary containers for both fixed and free parameters. In the context of imaging, they contain the imaging parameters (i.e. images or hypercubes) that can be read in from a FITS or CASA image and/or generated from clean components during deconvolution. In continuum imaging there is a separate parameter for each Taylor term. *Params* also contain a small amount of metadata for each parameter, and a number of functions for manipulating the metadata. These give a minimal description of the axes of the parameter (type and extent) and whether or not it is a free parameter. The latter is used, for example, in auto-differentiation when setting up normal equations.

5.3 JACAL Data Classes

A set of new data classes have been added to JACAL to give more flexibility to how data are gridded and degridded. As such, a number of the Yandasoft gridding classes have been copied into JACAL so they can interface with this class and make use of it. It also provides a lightweight way of swapping from the current underlying Yandasoft data accessors to others, such as Apache Plasma.

5.3.1 GriddingData

This is a simple container to hold one or more VisInfo and/or GriddingInfo objects.

5.3.2 VisInfo

The visibility data ingested by an *Ingest* process are stored in one or more data partitions of type *VisInfo*. Current partitioning options are time or w-value, with frequency partitioning handled by the underlying data accessors. However other options can be added in a straightforward manner. The visibilities and metadata are stored as flat vectors and the partitioning tasks deal with any algorithmic complexities. Current vectors are:

```
int itsNumberOfSamples; // total number of baselines & frequencies in each partition
int itsNumberOfPols; // number of polarisations
int itsNumberOfChannels; // number of frequencies (needed to handle Taylor term␣
↪weighting)

bool itsConvertedToImagePol; // have the vis and weights been converted to the imaging␣
↪polarisation frame?

/// Pointing metadata
casacore::MVDirection itsImageCentre;
casacore::MVDirection itsTangentPoint;

/// Vectors of length itsNumberOfSamples
std::vector<double> itsU; // u coordinate in wavelengths. Not used after init.
std::vector<double> itsV; // v coordinate in wavelengths. Not used after init.
std::vector<double> itsW; // w coordinate in wavelengths. Not used after init.
std::vector<std::complex<float> > itsPhasor; // phase shift for each visibility
std::vector<int> itsChannel; // frequency channel, needed to handle Taylor term weighting
std::vector<bool> itsFlag; // flagging state

/// A-projection vectors of length itsNumberOfSamples
std::vector<casacore::MVDirection> itsPointingDir1; // Not used after init.
std::vector<casacore::MVDirection> itsPointingDir2; // Not used after init.
std::vector<float> itFeed1PA; // not used after init.
std::vector<float> itFeed2PA; // not used after init.

/// Vectors of length itsNumberOfPols
casacore::Vector<casacore::Stokes::StokesTypes> itsStokes;

/// Vectors of length itsNumberOfChannels
std::vector<double> itsFrequencyList;

/// Nested vectors of length itsNumberOfSamples,itsNumberOfPols
std::vector<std::vector<float> > itsWeight; // vis sample weight.
std::vector<std::vector<float> > itsNoise; // sample RMS. Not used after init.
std::vector<std::vector<std::complex<float> > > itsSample; // vis samples
```

5.3.3 GriddingInfo

The *GenerateGridData* application converts a set of *VisInfo* partitions to an equal number of *GriddingInfo* partitions. These metadata are also stored as flat vectors. The gridders are set up to simply grid or degrid the elements of the vectors stored in this class. All data required by the Yandasoft gridders are currently passed in this class, but the gridders and the interface are expected to evolve towards the interface used in the SKA Processing Function Library. The *GenerateGridData* application also generates the cache of gridding kernels. Current vectors are:

```
int itsNumberOfSamples; // total number of baselines & frequencies in each partition
int itsNumberOfPols; // number of polarisations
int itsNumberOfChannels; // number of frequencies (needed to handle Taylor term␣
↪weighting)

/// Vectors of length itsNumberOfSamples
std::vector<std::complex<float> > itsPhasor; // phase shift for each visibility
std::vector<int> itsChannel; // frequency channel, needed to handle Taylor term weighting
std::vector<bool> itsFlag; // flagging state
std::vector<int> itsGridIndexU; // u index in grid
std::vector<int> itsGridIndexV; // v index in grid
std::vector<int> itsKernelIndex; // index in kernel grid, including oversampling planes

/// Vectors of length itsNumberOfChannels
std::vector<double> itsFrequencyList;

/// Nested vectors of length itsNumberOfSamples,itsNumberOfPols
std::vector<std::vector<float> > itsWeight; // vis sample weight.

/// Convolution kernels
int nPlanes;
std::vector<int> cSize; // Vector of kernel sizes (length nPlanes)
std::vector<std::vector<casacore::Complex> > itsConvFunc; // Nested vector of kernels␣
↪([nPlanes][cSize*cSize])
std::vector<std::vector<int> > itsConvFuncOffsets; // Nested vector of kernel offsets␣
↪([nPlanes][2])
```

5.4 JACAL Interfaces

Interfaces between JACAL imaging components are at present data drops formed from Yandasoft and JACAL classes that have been converted to BlobStrings.

5.4.1 ImagingNormalEquations

```
/// @brief write the object to a blob stream
void ImagingNormalEquations::writeToBlob(LOFAR::BlobOStream& os) const
{
  os << itsNormalMatrixSlice // PSF cubes; dirty image cubes; std::map<std::string,␣
↪casacore::Vector<imtype> >
     << itsNormalMatrixDiagonal // weights cubes; dirty image cubes; std::map
↪<std::string, casacore::Vector<imtype> >
     << itsPreconditionerSlice // dirty image cubes; std::map<std::string,␣
```

(continues on next page)

(continued from previous page)

```
↪casacore::Vector<imtype> >
     << itsShape // shape of the cubes; std::map<std::string, casacore::IPosition>
     << itsReference // reference pixel of images; std::map<std::string,␣
↪casacore::IPosition>
     << itsCoordSys // coordinate system of the images; std::map<std::string,␣
↪casacore::CoordinateSystem>
     << itsDataVector; // dirty image cubes; std::map<std::string, casacore::Vector
↪<imtype> >
}

/// @brief read the object from a blob stream
void ImagingNormalEquations::readFromBlob(LOFAR::BlobIStream& is)
{
  is >> itsNormalMatrixSlice
     >> itsNormalMatrixDiagonal
     >> itsPreconditionerSlice
     >> itsShape
     >> itsReference
     >> itsCoordSys
     >> itsDataVector;
}
```

Type *imtype* is either float or double.

5.4.2 Params

```
/// @brief write the object to a blob stream
LOFAR::BlobOStream& operator<<(LOFAR::BlobOStream& os, const Params& par)
{
    os.putStart("Params",BLOBVERSION);
    os << par.itsUseFloat // use float or double for the Arrays; bool
       << par.itsArrays // image data; std::map<std::string, casacore::Array<double> >
       << par.itsArraysF // image data; std::map<std::string, casacore::Array<float> >
       << par.itsAxes // image axes; std::map<std::string, Axes>
       << par.itsFree; // the free/fixed status of the parameter; std::map<std::string,␣
↪bool>
    os.putEnd();
    return os;
}

/// @brief read the object from a blob stream
LOFAR::BlobIStream& operator>>(LOFAR::BlobIStream& is, Params& par)
{
    const int version = is.getStart("Params");
    ASKAPCHECK(version == BLOBVERSION, "Attempting to read from a blob stream of the␣
↪wrong version");
    is >> par.itsUseFloat
       >> par.itsArrays
       >> par.itsArraysF
       >> par.itsAxes
       >> par.itsFree;
```

(continues on next page)

(continued from previous page)

```
    is.getEnd();
    // as the object has been updated one needs to obtain new change monitor
    par.itsChangeMonitors.clear();
    return is;
}
```

Type *Axes* is also defined in the Yandasoft `base-scimath` library. It contains the names (e.g. "RA_LIN", "FREQ") and extrema (start and end values as doubles) of a set of axes, using standard casacore types such as casacore::DirectionCoordinate and casacore::Stokes::StokesTypes.

5.4.3 GriddingInfo

```
/// @brief write the object to a blob stream
LOFAR::BlobOStream& operator<<(LOFAR::BlobOStream& os, const GriddingInfo& info)
{
    // Copy the cache of gridding kernels to a suitable format.
    // This will be removed once a final cache format has been chosen.
    const int nPlanes = info.itsConvFunc.size();
    std::vector<int> cSize(nPlanes);
    std::vector<std::vector<casacore::Complex> > tmpConvFunc(nPlanes);
    for (uint plane = 0; plane < nPlanes; ++plane) {
        cSize[plane] = info.itsConvFunc[plane].nrow();
        tmpConvFunc[plane].resize(cSize[plane]*cSize[plane]);
        for (uint j = 0; j < cSize[plane]; ++j) {
            for (uint i = 0; i < cSize[plane]; ++i) {
                tmpConvFunc[plane][j*cSize[plane]+i] = info.itsConvFunc[plane](i,j);
            }
        }
    }
    os.putStart("GriddingInfo",BLOBVERSION);
    os << info.itsNumberOfSamples
       << info.itsNumberOfChannels
       << info.itsNumberOfPols
       << info.itsFrequencyList
       << info.itsGridIndexU
       << info.itsGridIndexV
       << info.itsKernelIndex
       << info.itsPhasor
       << info.itsChannel
       << info.itsFlag
       << info.itsWeight
       << nPlanes
       << cSize
       << tmpConvFunc;
    os.putEnd();
    return os;
}
```

```
/// @brief read the object from a blob stream
LOFAR::BlobIStream& operator>>(LOFAR::BlobIStream& is, GriddingInfo& info)
```

(continues on next page)

(continued from previous page)

```
{
    const int version = is.getStart("GriddingInfo");
    ASKAPCHECK(version == BLOBVERSION, "Attempting to read from a blob stream of the␣
↪wrong version");
    // first get size parameters and resize the vectors
    is >> info.itsNumberOfSamples
       >> info.itsNumberOfChannels
       >> info.itsNumberOfPols;
    is >> info.itsFrequencyList
       >> info.itsGridIndexU
       >> info.itsGridIndexV
       >> info.itsKernelIndex
       >> info.itsPhasor
       >> info.itsChannel
       >> info.itsFlag
       >> info.itsWeight;
    int nPlanes;
    is >> nPlanes;
    std::vector<int> cSize(nPlanes);
    is >> cSize;
    std::vector<std::vector<casacore::Complex> > tmpConvFunc(nPlanes);
    for (uint plane = 0; plane < nPlanes; ++plane) {
        tmpConvFunc[plane].resize(cSize[plane]*cSize[plane]);
    }
    is >> tmpConvFunc;
    // Copy the cache of gridding kernels back to the required format.
    // This will be removed once a final cache format has been chosen.
    info.itsConvFunc.resize(nPlanes);
    for (uint plane = 0; plane < nPlanes; ++plane) {
        info.itsConvFunc[plane].resize(cSize[plane],cSize[plane]);
        for (uint j = 0; j < cSize[plane]; ++j) {
            for (uint i = 0; i < cSize[plane]; ++i) {
                info.itsConvFunc[plane](i,j) = tmpConvFunc[plane][j*cSize[plane]+i];
            }
        }
    }
    is.getEnd();
    return is;
}
```

5.4.4 vis data

The *GenerateGridData* application extracts the visibility data and passes them as a separate blob drop. The current format, which is expected to change, is a simple nested vector of complex values:

```
std::vector<std::vector<std::complex<float> > > vis;
vis.resize(itsNumberOfSamples);
for (uint i=0; i<itsNumberOfSamples; ++i) {
    vis[i].resize(itsNumberOfPols);
}
```

CHAPTER

SIX

API

6.1 Available applications

class **CalcNE** : public askap::*DaliugeApplication*

CalcNE.

Calculates the Normal Equations

This class encorporates all of the tasks needed to form imaging Normal Equations: read from a measurement set; degrid model visibilities; subtract model visibilities; grid residual visibilities and FFT the grid

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL librarc

Param param/Arg01
[in] Arg01/name=CalcNE/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ ParameterSet descriptor for the image solver

Param port/Model
[in] Model/scimath::Params/ Params of solved normal equations

Param port/Normal
[out] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to solve

class **InitSpectralCube** : public askap::*DaliugeApplication*

InitSpectralCube.

Build the output image cube

This class builds the output cube in the format specified by the parset.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=InitSpectralCube/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file

class **LoadNE** : public askap::*DaliugeApplication*

LoadNE.

Example class that simply loads Normal Equations from a drop

Implements a test method that uses the contents of the the parset to load in a measurement set and print a summary of its contents. We will simply load in a NormalEquation from a daliuge drop and output the image. This simply tests the NE interface to the daliuge memory drop.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=LoadNE/String/readonly/

Param port/Normal
[in] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to solve

class **LoadParset** : public askap::*DaliugeApplication*

LoadParset.

Load a LOFAR Parameter Set in the *DaliugeApplication* Framework

Loads a configuration from a file drop and generates a LOFAR::ParameterSet The first ASKAP example in the Daliuge framework that actually performs an ASKAP related task. We load a parset into memory from either a file or another Daliuge drop_status

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=LoadParset/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ ParameterSet descriptor for the image solver

Param port/Config
[out] Config/LOFAR::ParameterSet/

class **LoadVis** : public askap::*DaliugeApplication*

LoadVis.

Loads a visibility set, grids it onto the UV plane and FFTs the grid

Loads a configuration from a file drop and a visibility set from a casacore::Measurement Set

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=LoadVis/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file Params of solved normal equations

Param port/Normal
[out] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to solve

class **MajorCycle** : public askap::*DaliugeApplication*

MajorCycle.

Loads a visibility set, grids it onto the UV plane and FFTs the grid

Loads a configuration from a file drop and a visibility set from a casacore::Measurement Set

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=MajorCycle/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file

Param port/Cube
[in] Cube/Cube

Param port/Normal
[out] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to solve

class **NESpectralCube** : public askap::*DaliugeApplication*

NESpectralCube.

Build an output image cube from input NormalEquations

This class builds the output cube is whatever format specified by the parset. Generates a cube of NormalEquation slices.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=NESpectralCube/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ ParameterSet descriptor for the image solver

Param port/Normal
[in] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to solve

class **OutputParams** : public askap::*DaliugeApplication*

OutputParams.

Solves an Normal Equation provided by a Daliuge Drop. Outputs the Params class as images.

Implements an ASKAPSoft solver. This essentially takes a NormalEquation and generates a a set of "params" usually via a minor cycle deconvolution. We will simply load in a NormalEquation from a daliuge drop and solve it via a minor cycle deconvolution. This drop actually generates the output images based upon the contents of the Params object.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=OutputParams/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ ParameterSet descriptor for the image solver

Param port/Model
[in] Model/scimath::Params/

Param port/Restored Model
[in] Restored Model/scimath::Params/

class **ReduceNE** : public askap::*DaliugeApplication*

ReduceNE.

Merge two Normal Equation objects

Use askap::scimath::ImagingNormalEquations::merge() to merge two NEs

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=ReduceNE/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file ImagingNormalEquations to merge

Param port/Normal
[in] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to merge

Param port/Normal
[out] Normal/scimath::ImagingNormalEquations/ Merged ImagingNormalEquations to merged further or solve

class **RestoreSolver** : public askap::*DaliugeApplication*

RestoreSolver.

Implements an ASKAPSoft Restore solver. This essentially takes a NormalEquation and a set of "params" and creates a restored image.

This takes a configuration and a set of normal equations and uses the Solver requested in in the ParameterSet to produce an ouput model.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=RestoreSolver/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file Params of solved normal equations

Param port/Normal
[in] Normal/scimath::ImagingNormalEquations/ ImagingNormalEquations to solve

Param port/Restored Model
[out] Restored Model/scimath::Params/

class **SolveNE** : public askap::*DaliugeApplication*

SolveNE.

Implements an ASKAPSoft solver. This essentially takes a NormalEquation and generates a set of params usually via a minor cycle deconvolution.

This takes a configuration and a set of normal equations and uses the Solver requested in in the ParameterSet to produce an ouput model.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=SolveNE/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file ImagingNormalEquations to solve

Param port/Model
[out] Model/scimath::Params/ Params of solved normal equations

class **SpectralCube** : public askap::*DaliugeApplication*

SpectralCube.

Build the output image cube

This class builds the output cube is whatever format specified by the parset.

EAGLE_START

EAGLE_END

Param category
DynlibApp

Param param/libpath
[in] Library Path/"%JACAL_SO%"/String/readonly/ The path to the JACAL library

Param param/Arg01
[in] Arg01/name=SpectralCube/String/readonly/

Param port/Config
[in] Config/LOFAR::ParameterSet/ The Config file Params of solved normal equations

Param port/Cube
[out] Cube/Cube/

6.2 Others

class **DaliugeApplication**

Daliuge application class.

This class encapsulates the functions required of a daliuge application as specified in dlg_app.h then exposes them as C functions

Subclassed by *askap::CalcNE*, askap::GenerateGridData, askap::GridInvert, askap::GridPredict, askap::Ingest, *askap::InitSpectralCube*, askap::JacalBPCalibrator, *askap::LoadNE*, *askap::LoadParset*, *askap::LoadVis*, *askap::MajorCycle*, *askap::NESpectralCube*, *askap::OutputParams*, *askap::ReduceNE*, *askap::RestoreSolver*, *askap::SolveNE*, *askap::SpectralCube*

class **DaliugeApplicationFactory**

Factory class that registers and manages the different possible instances of of a *DaliugeApplication*. .

Contains a list of all applications and creates/instantiates the correct one based upon the "name" of the Daliuge DynLib drop. Maintains a registry of possible applications and selects - based upon a name which one will be instantiated.

class **NEUtils**

set of static utility functions for the NE manipulation

These are just a set of static functions I use more than once

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

askap::CalcNE (*C++ class*), 25
askap::DaliugeApplication (*C++ class*), 31
askap::DaliugeApplicationFactory (*C++ class*), 31
askap::InitSpectralCube (*C++ class*), 25
askap::LoadNE (*C++ class*), 26
askap::LoadParset (*C++ class*), 26
askap::LoadVis (*C++ class*), 27
askap::MajorCycle (*C++ class*), 27
askap::NESpectralCube (*C++ class*), 27
askap::NEUtils (*C++ class*), 31
askap::OutputParams (*C++ class*), 28
askap::ReduceNE (*C++ class*), 28
askap::RestoreSolver (*C++ class*), 29
askap::SolveNE (*C++ class*), 29
askap::SpectralCube (*C++ class*), 30