TKP Documentation

Release 2.1.0

LOFAR Transients Key Science Project

July 17, 2015

CHAPTER 1

Version 2.1.0

1.1 Introduction

The LOFAR Transients Pipeline ("TraP") provides a means of searching a stream of N-dimensional (two spatial, frequency, polarization) image "cubes" for transient astronomical sources. The pipeline is developed specifically to address data produced by the LOFAR Transients Key Science Project, but may also be applicable to other instruments or use cases.

The TraP codebase provides the pipeline definition itself, as well as a number of supporting routines for source finding, measurement, characterization, and so on. Some of these routines are also available as *stand-alone tools*.

1.1.1 High-level overview

The TraP consists of a tightly-coupled combination of a "pipeline definition" – effectively a Python script that marshals the flow of data through the system – with a library of analysis routines written in Python and a database, which not only contains results but also performs a key role in data processing.

Broadly speaking, as images are ingested by the TraP, a Python-based source-finding routine scans them, identifying and measuring all point-like sources. Those sources are ingested by the database, which *associates* them with previous measurements (both from earlier images processed by the TraP and from other catalogues) to form a lightcurve. Measurements are then performed at the locations of sources which were expected to be seen in this image but which were *not* detected. A series of statistical analyses are performed on the lightcurves constructed in this way, enabling the quick and easy identification of potential transients. This process results in two key data products: an *archival database* containing the lightcurves of all point-sources included in the dataset being processed, and *community alerts* of all transients which have been identified.

Exploiting the results of the TraP involves understanding and analysing the resulting lightcurve database. The TraP itself provides no tools directly aimed at this. Instead, the Transients Key Science Project has developed the Banana web interface to the database, which is maintained separately from the TraP. The database may also be interrogated by end-user developed tools using SQL.

1.1.2 Documentation layout

The documentation is split into four broad sections:

Getting Started Provides a guide to installing the TraP and its supporting libraries on common platforms and some basic information to help get up and running quickly.

User's Reference Here we provide a complete description of all the functionality available in the TraP and describe the various configuration and setup options available to the end user.

Developer's Reference A guide to the structure of the codebase, the development methodologies, and the functionality available in the supporting libraries. This is of interest both to developers within the project and to those who want to build upon TraP functionality for their own purposes.

Stand-alone Tools Some functionality developed for the TraP is also available in these simple, end-user focused tools.

This documentation focuses on the technical aspects of using the TraP: all the pipeline components are described, together with their user-configurable parameters and the systems which have been developed for connecting them together to form a pipeline. However, it does not provide detailed rationale for all of the scientific choices made in the pipeline design. It is the position of the author that achieving high quality results requires understanding *both* the technical and the scientific choices made. For help with the latter, the reader is referred to *Swinbank et al*.

1.2 Getting Started

1.2.1 Installation

To install the TraP you must:

1. Install and configure a suitable back-end database;
2. Optionally, install and configure MongoDB as a *pixel store*;
3. Install the core pipeline dependencies (casacore etc);
4. Install the TraP itself, via the 'tkp' Python package.

Some details on each of these steps is provided below.

Note, though, that the overall procedure is complex, and can be difficult if you've not had prior experience with e.g. database configuration. It is possible instead to use Vagrant to quickly and easily set up a virtual machine which provides a fully configured and working ready-to-go pipeline and supporting tools. This is a quick and easy way to get up and running for testing purposes or when simplicity is preferable to ultimately high performance. Refer to the Vagrant TraP repository for details and instructions.

Back-end Database

The TraP supports two database management systems for use as the *pipeline database*: MonetDB and PostgreSQL. Both are available for common operating systems and package managers: pick one and install it.

A complete description of configuring your database management system is beyond the scope of this guide: refer to its documentation for details. Some brief notes on things to look out for follow.

PostgreSQL

Ensure that the access rights to your server are set appropriately, for example to trust connections from whichever machine(s) will be used to run the TraP. This is done by editing `pg_hba.conf`, and can be verified by connecting with the command line tool `psql`.

MonetDB

To be able to administer MonetDB databases, you need to be a member of the `monetdb` group.

To issue remote management commands, such as database creation, you need to both enable this functionality and set a passphrase:

```
monetdbd set control=yes ${dbfarm}
monetdbd set passphrase=${myphassphrase} ${dbfarm}
```

Pixel Store

Optionally, the pixel contents of all images processed (but not the metadata) can be saved to a MongoDB database for future reference (e.g. via the Banana web interface). This naturally requires that a MongoDB daemon is installed and configured to accept requests from TraP clients.

Core Dependencies

TraP mostly depends on standard packages which you should be able to find in your system's package manager (e.g. apt, yum, etc). To install the TraP, you will need the following:

- C++ and Fortran compilers (tested with GCC)
- GNU Make
- Python (2.7.x series, including header files)
- Boost Python
- WCSLIB

TraP also has a number of Python-package dependencies. The install process will attempt to download and install these as necessary, but you may wish to pre-install system packages for some of the following, in order to save time recompiling them from source:

- NumPy (at least version 1.3.0)
- SciPy (at least version 0.7.0)
- python-dateutil (at least version 1.4.1)
- python-psycopg2 (for PostgreSQL)
- python-monetdb (for MonetDB)

To work with the *pixel store* you will also need:

- PyMongo

Finally, TraP also requires the 'casacore' library, which is not yet widely available as a system package:

- casacore (including measures data)

Casacore can be compiled from source, or users of Ubuntu-based distributions might find the Radio Astronomy Launchpad page useful.

Warning: See also the note on *casacore measures data*, which can often cause confusing errors if out-of-date or incorrectly configured.

Installation

Once all dependencies have been satisfied, installation should be straightforward. You can either install from source:

```
$ git clone --branch release2.1 https://github.com/transientskp/tkp.git
$ cd tkp
$ python setup.py install
```

Or you can install directly from the Python Package Index (PyPI), e.g. using pip):

```
$ pip install tkp==2.1
```

Note that if you want to make use of the *pixel store* functionality, then:

```
$ pip install tkp[pixelstore]==2.1
```

will install the required libaries, similarly:

```
$ pip install tkp[monetdb]==2.1
```

will ensure installation of the python-monetdb interface package.

Following installation, including setting up and configuring the database, follow the *test procedure* to ensure that everything is working and ready for use.

Distributed processing via Celery

If you wish to run a TraP job across multiple machines, you may optionally also install a *Celery* broker (at least version 3.0); see the Celery website for further details of the Celery package.

Multiple different options for Celery brokers are available; refer to the Celery documentation for details. We have had success with RabbitMQ.

This functionality is currently not well supported and should be considered experimental.

1.2.2 Configuring the casacore ephemeris

Note that as part of the installation you (or your system administrator) will have installed the casacore "measures data". This includes information essential to carrying out astronomical calculations, such as a list of leap seconds and a set of solar system ephemerides (which specify the positions of the planets at any given time). Data for the ephemerides are ultimately supplied by NASA JPL; they have been converted into a format that casacore can use. Any given ephemeris is only valid for a limited time (usually on the order of centuries), determined by the accuracy with which it was calculated.

By default, casacore will use the `DE 200` ephemeris. Although the version of `DE 200` supplied by JPL is valid until 2169, *some versions converted for use with casacore are not*, and may not provide coverage of the dates of your observation. A simple Python script can be used to check:

```
$ cat check_ephemeris.py
import sys
from casacore.measures import measures
dm = measures()
dm.do_frame(dm.epoch('UTC', sys.argv[1]))
dm.separation(dm.direction('SUN'), dm.direction('SUN'))
$ python check_ephemeris.py 1990/01/01
$ python check_ephemeris.py 2015/01/01
WARN    MeasTable::Planetary(MeasTable::Types, Double)
  (file /build/buildd/casacore-1.7.0/measures/Measures/MeasTable.cc, line 4056)
  Cannot find the planetary data for MeasJPL object number 3 at UT day 57023 in
  table DE200
```

If no warning is printed, there is no problem; otherwise, you should use a different ephemeris. For example, the `DE 405` ephemeris should be valid until at least early 2015:

```
$ cat > ~/.casarc
measures.jpl.ephemeris: DE405
$ python check_ephemeris.py 2015/01/01
# No warnings
```

An alternative issue sometimes encountered is that of measures data which is simply outdated. Should you see an error along the lines of

```
SEVERE  gaincal::MeasTable::dUTC(Double) (file measures/Measures/MeasTable.cc, line 6307    Leap seco
SEVERE  gaincal::MeasTable::dUTC(Double) (file measures/Measures/MeasTable.cc, line 6307)+  Until tab
SEVERE  gaincal::MeasTable::dUTC(Double) (file measures/Measures/MeasTable.cc, line 6307)+  derived f
```

Then you might try to update your measures data via Rsync, as described in this NRAO helpdesk article

1.2.3 Concepts

Here we describe some basic concepts that are important to understanding TraP operation.

Pipeline Database

The TraP consists of a very tightly-coupled set of logic implemented partial in Python code and partially in a relational database. The database contains measurements made of sources being processed by the TraP, as well as information about the images being processed, the regions of the sky being surveyed, and so on. The database is fundamental to the operation of the TraP: except for a few *standalone tools*, use of a database is absolutely required.

The tight coupling between the Python code and the database implies that the version of the *database schema* in use must match the version in the of the code. Note that the schema version referred to in this document is 34.

In general, a single database management system (RDBMS) can support more than one independent database. It is suggested that each coherent "project" processed through the TraP be given an independent database. For example, a project in this sense might include all the data from a particular survey, or all the data processed by a particular user. The resulting data can then be archived as a coherent unit, while other projects continue to use the same RDBMS undisturbed.

Within the context of the TraP, we support two different RDBMSs: MonetDB and PostgreSQL. All TraP functionality is available whichever database you choose: it is suggested you experiment to determine which provides the best combination of usability and performance for your particular usage.

See the *relevant section* of the documentation for much more information about configuring and operating the database as well as understanding its contents.

Finally, it is important to note that the pipeline database does *not* contain any image pixel data: it stored metadata and derived products only. It is possible to store pixel data as part of a pipeline run, but that is a separate subsystem: see *MongoDB*.

MongoDB

In general, image pixel data is not regarded as a standard TraP *data product*: stated policy is that archiving the large amount of pixel data generated by the sort of surveys that TraP is designed to process is simply impractical.

However, particularly when processing smaller volumes of data, or for pipeline testing and commissioning, it is convenient to be able to store a copy of the pixels that have been processed. The TraP therefore supports storing image data to a MongoDB database. MongoDB is a "document-oriented database", which provides a convenient, low-overhead way to store and retrieve large volumes of image data.

It is worth noting that the data stored to MongoDB is not a bit-for-bit copy of the input images. We store only the pixel data together with world coordinate system information embedded in a FITS container; other metadata is stripped.

The use of MongoDB within the TraP is completely optional: it is perfectly possibly to run a complete survey without installing it. However, if storing image data is required, you will need to install MongoDB along with the TraP and configure the *persistence step* appropriately.

1.2.4 Tutorial Overview

This page walks you through setting up a complete, stand-alone TraP environment running on your own system.

Install the software

First of all, the software should be installed on your system. See *the installation manual*.

Note: Issues with the casacore installation can occasionally require a per-user fix (placing a small config file in your home directory). If you see errors along the lines of:

```
WARN    MeasTable::Planetary(MeasTable::Types, Double)
(file /build/buildd/casacore-1.7.0/measures/Measures/MeasTable.cc, line 4056)
Cannot find the planetary data for MeasJPL object number 3 at UT day 57023 in
table DE200
```

or:

```
SEVERE  gaincal::MeasTable::dUTC(Double) ...
Leap second table TAI_UTC seems out-of-date.
```

Then consult your sysadmin or see *this note*.

`trap-manage.py`

The main tool for configuring and running TraP is *trap-manage.py*, which should be available to you as a command-line utility after installing the TraP. You may need to consult your sysadmin for details of how to access your local TraP installation. You can remind yourself of the options available to you by running:

```
$ trap-manage.py -h
```

Or you can consult the *documentation* for details.

Create a pipeline project directory

To get started using TraP, you should first create a project directory: this will contain your pipeline settings and job directories. To create a project folder in your current working directory, type:

```
$ trap-manage.py initproject <projectname>
```

(substituting `<projectname>` for your chosen directory name).

Create a database

The pipeline requires a database for storing data, which needs to be created manually. The database then needs to be initialised with the TRAP database schema before it can be used.

Depending on your site configuration, creating a database may require sysadmin rights. Refer to the relevant documentation for your installed database engine on creating a database:

- MonetDB online documentation
- PostgreSQL online documentation

Initialize a database

To initialise a database the TraP manage `initdb` subcommand can be used. Set the details of the database you have created in the `database` section of your *pipeline config file*. These include the host and port number of the system running the database management system, the name of the database, and the username and password. Then, from the project directory, type:

```
$ trap-manage.py initdb
```

Resetting a TraP database

You may wish to reset a previously used TraP database to an empty state.

Warning: As you might expect, this may incur irreversible data loss. Be careful!

PostgreSQL For PostgreSQL there is the optional **-d** flag for the `initdb` subcommand, which removes all content before populating the database.

MonetDB In the case of MonetDB you need to do this manually. You can do this with the following commands, where **${dbname}** should be replaced with the database name:

```
monetdb stop ${dbname}
monetdb destroy -f ${dbname}
monetdb create ${dbname}
monetdb start ${dbname}
monetdb release ${dbname}
```

For security reasons you should change the default password:

```
mclient -d ${dbname} -s"ALTER USER \"monetdb\" RENAME TO \"${username}\";
ALTER USER SET PASSWORD '${password}' USING OLD PASSWORD 'monetdb';"
```

Create and configure a job

Your pipeline project directory can contain multiple jobs, each represented by a subdirectory. Job directories contain a list of files to process, and config file that can be used to define various properties used during processing. To initialise a job directory run:

```
$ trap-manage.py initjob <jobname>
```

This will create a job subdirectory within your pipeline directory. This directory contains three files:

`images_to_process.py` This is a Python script that is used to generate a list of paths to images. You will need to adjust this to point to your data files.

`job_params.cfg` The *parameters configuration file* for this job.

`inject.cfg` Configuration for *image metadata injection*.

Run the pipeline

To start processing your data run (from your pipeline directory):

```
$ trap-manage.py run <jobname>
```

1.3 User's Reference Guide

1.3.1 trap-manage.py

`trap-manage.py` is the command-line tool for initialising and running the TraP pipeline. Different tasks are handled via the use of 'subcommands' as detailed below.

When the TraP is correctly installed on the system you can issue the `trap-manage.py` command. Documentation of subcommands is also available on the command line. You can use the `--help` flag (also per subcommand) to explore all possible options.

A tool for managing TKP projects.

Use 'initproject' to create a project directory. Other Subcommands should be run from within a project directory.

NB: To overwrite the database settings in pipeline.cfg you can use these environment variables to configure the connection:

- TKP_DBENGINE
- TKP_DBNAME
- TKP_DBUSER
- TKP_DBPASSWORD
- TKP_DBHOST
- TKP_DBPORT

(This is useful for setting up test databases, etc.)

```
usage: trap-manage.py [-h] {initproject,initjob,run,initdb,celery} ...
```

Sub-commands:

initproject

Initialize a pipeline project directory, complete with config files which you can use to configure your pipeline.

```
usage: trap-manage.py initproject [-h] [-t TARGET] name
```

Positional arguments:

name	project folder name

Options:

-t, --target	location of new TKP project

initjob

Create a job folder, complete with job-specific config files which you will need to modify.

```
usage: trap-manage.py initjob [-h] name
```

Positional arguments:

name	Name of new job

run Run a job by specifying the name of the job folder.

```
usage: trap-manage.py run [-h] [-m MONITOR_COORDS] [-l MONITOR_LIST] name
```

Positional arguments:

name	Name of job to run

Options:

-m, --monitor-coords	a list of RA,DEC coordinates to monitor in JSON format, example: [[5, 6], [7, 8]]
-l, --monitor-list	Specify a file containing a list of RA,DEC

initdb Initialize a database with the TKP schema.

```
usage: trap-manage.py initdb [-h] [-y] [-d]
```

Options:

-y=False, --yes=False	don't ask for confirmation
-d=False, --destroy=False	remove all tables before population(only works with Postgres backend)

celery Shortcut for access to celery sub commands

```
usage: trap-manage.py celery [-h] ...
```

Positional arguments:

rest	A celery subcommand

1.3.2 Pipeline Configuration

The TraP has a multi-level configuration system which can be complex to get to grips with. Here, we cover the logic behind the system and describe how users can best customize to meet their needs.

Configuration System Overview

Preamble

Users organize their TraP runs into *projects*, which define the basic operational TraP parameters, for example which *database* to use. All the configuration and log files relating to a particular project are stored in the same directory hierarchy.

Within a project, the user is able to configure multiple pipeline runs, which are referred to as *jobs*. For example, the user could use a single project to repeatedly re-analyse a particular set of images with a range of different sourcefinder settings, with each analysis constituting a particular job. Alternatively, a project might be dedicated to a multi-epoch survey, with each job corresponding to a different epoch. The fundamental point is that the basic control structures such as the database and task distribution system remain the same, but, within those limits, the user is free to organize their work as they please.

Creating a Project Directory

Use *trap-manage.py* to create a project directory:

```
$ trap-manage.py initproject <directory-name>
```

The created directory will contain the following configuration files:

`pipeline.cfg` The overall *pipeline configuration file*. This configures the database which will be used for this project, as well as specifying where and how to store log files. *See here* for a full description.

`celeryconfig.py` Configuration of the *Celery* task distribution system. *See here* for a full description.

Creating a Job

From within a project directory, use the `trap-manage.py` script to create a job:

```
$ cd ./<projectname>
$ trap-manage.py initjob <jobname>
```

This creates a job directory as a subdirectory of your project directory, named according to the job name. Within that directory, there are three files:

`images_to_process.py` This defines the list of images which will be processed by this job. It is a Python file, which, when loaded, provide at module scope iterable named `images` which contains the full path to each image to be processed.

In other words, the file could be as simple as:

```
    images = ["/path/to/image1", "/path/to/image2", ...]
```

Or could contain an elaborate set of globbing and pathname expansion as required.

The end user *will* need to customize to properly specify their requirements.

`inject.cfg` Configuration for the *metadata injection tool*.

`job_params.cfg` Configuration for each stage of the pipeline run represented by this job. This contains all the end-user tunable parameters which are relevant to TraP operation. *See here* for details.

Running a Job

Once fully configured, you will want to run a TraP job to process your data. From within a project directory, you can start a job using:

```
$ trap-manage.py run <jobname>
```

Configuration file syntax

Several of the TraP's configuration files – *pipeline.cfg*, *inject.cfg*, *job_params.cfg* – use the Python `ConfigParser` file format. This is defined by the Python standard library, and you should refer to its documentation for a comprehensive reference. However, it is worth noting a few salient points that may be of relevance to the TraP user.

These files are divided into named sections: the name comes at the top of the section, surrounded by square brackets (`[` and `]`). Within a section, a simple `name = value` syntax is used. `;` indicates a comment (`#` may also be used for commenting, but only at the start of a line).

Variable substiution is performed using the notation `%(name)s`: this will be expanded into the value of the variable `name` when the file is read. Variables used in expansion are taken either from the same section of the file, or from the special `DEFAULT` section. For example:

```
[DEFAULT]
a = 1

[section_name]
b = 2
c = %(a)s
d = %(b)s
```

Would set the values of `a` and `c` to `1`, and `b` and `d` to `2`. In some cases, the TraP provides additional variables which may be used in expansions in a particular file: these are noted in the documentation for that file.

`pipeline.cfg` - Project Configuration File

The project configuration file provides a common configuration to all pipeline runs which are part of a particular *project*. Through this file, it is possible to configure the database used for pipeline runs, the location in which jobs are stored, and the amount and storage location for logging.

The default `pipeline.cfg` file is as follows:

```
[DEFAULT]
runtime_directory = %(cwd)s
job_directory = %(runtime_directory)s/%(job_name)s

[logging]
#log_dir contains output log, plus a copy of the config files used.
log_dir = %(job_directory)s/logs/%(start_time)s
debug = False

[database]
engine = ;(monetdb or postgresql)
database = "" ; e.g. '{% user_name %}'
user =  "" ; e.g. '{% user_name %}', or 'postgres'
password = "" ; e.g. '{% user_name %}'
host = "localhost"
port =
passphrase =
dump_backup_copy = False

[image_cache]
copy_images = True
mongo_host = "localhost"
mongo_port = 27017
mongo_db = "tkp"


[parallelise]
method = "multiproc"  ; or celery, or serial
cores = 0  ; the number of cores to use. Set to 0 for autodetect
```

The file follows the *standard ConfigParser syntax*. Three special variables which may be used in expansions are provided provided by the TraP: `cwd`, the current working directory, `start_time`, the time at which the current pipeline job is started and `job_name`, the name of the job currently being executed.

`DEFAULT` section

The `DEFAULT` section provides a location for defining parameters which may be referred to be other sections. The following parameters may be defined:

`runtime_directory` This is the root directory for the project. The default value, `%(cwd)s`, means that the pipeline.cfg refers to the project in the directory in which it is stored: this is almost always correct.

`job_directory` This is the directory under which new jobs will be created. The default is to create a directory named after the job as a subdirectory of the project directory. This is almost always correct.

`logging` section

`log_dir` The full path to a directory into which the pipeline will write logging information as it progresses, and also make a record of the parameters used for a job. The log file provides a record of pipeline activity, and, in particular, any errors or problems encountered, while the parameter files record the configuration that produced these results. This folder is therefore important for reproducibility and debugging purposes.

`debug` A boolean (True or False) value. If True, extra information will be written to the log file, which might be helpful in diagnosing hard-to-find problems.

`database` Section

Note: The database config settings can be over-ridden using environment variables, e.g. for configuring a unit-testing environment. See *`tkp.config.get_database_config()`* for details.

`engine` The database engine to use. Two engines are supported: `postgresql` and `monetdb`. See the *introductory material on databases* for details.

`host`, `port` The host and port on the network at which the database server is listening.

`database`, `user`, `password` The name of the database to use, and the username and password required to connect to it.

`passphrase` A passphrase which provides administrative access to the database server. Only applicable to the `monetdb` engine. This is not required for normal operation, but enables the user to (for example) create and destroy databases.

`dump_backup_copy` A boolean value. If True, a copy of the configured database will be dumped to disk at the beginning of each pipeline run. This is not recommended in regular use, but can be useful if encountering intermittent database errors, both for recovering a working database, and diagnosing how errors occur. The dump is made to the job directory in a file named according to the pattern `<database host>_<database name>_<current time>.dump`.

`image_cache` Section

This section configures the *image caching or 'pixel store'* functionality.

See also: the 'optional dependencies' section of your relevant *installation* guide.

`copy_images` Boolean. If `True`, image pixel data will be stored to a MongoDB database.

mongo_host, mongo_port String, integer. Network hostname and port to use to connect to MongoDB. Only used if `copy_images` is `True`.

mongo_db String. Name of MongoDB database in which to store image pixel data. Only used if `copy_images` is `True`.

`parallelise` Section

method Determines whether the TraP is run in single-process, multi-process, or *distributed* mode. `"multiproc"` should be suitable for most users.

cores Determines the number of cores to use in multi-process mode. `0` will attempt to autodetect (and use all available cores).

`job_params.cfg` - Job Parameters Configuration

The job parameters file provides the detailed, scientifically-motivated settings for each pipeline step. Providing the appropriate configuration here is essential for achieving scientifically valid results.

The default `job_params.cfg` file is as follows:

```
[persistence]
description = "TRAP dataset"
dataset_id = -1
#Sigma value used for iterative clipping in RMS estimation:
rms_est_sigma = 4
#Determines size of image subsection used for RMS estimation:
rms_est_fraction = 8

[quality_lofar]
low_bound = 1           ; multiplied with noise to define lower threshold
high_bound = 80         ; multiplied with noise to define upper threshold
oversampled_x = 30      ; threshold for oversampled check
elliptical_x = 2.0      ; threshold for elliptical check
min_separation = 10     ; minimum distance to a bright source (in degrees)

[source_extraction]
# extraction threshold (S/N)
detection_threshold = 8
analysis_threshold = 3
back_size_x = 50
back_size_y = 50
margin = 10
deblend_nthresh = 0 ; Number of subthresholds for deblending; 0 disables
extraction_radius_pix = 250
force_beam = False
box_in_beampix = 10
# ew/ns_sys_err: Systematic errors on ra & decl (units in arcsec)
# See Dario Carbone's presentation at TKP Meeting 2012/12/04
ew_sys_err = 10
ns_sys_err = 10

[association]
deruiter_radius = 5.68
beamwidths_limit =  1.0
```

```
[transient_search]
new_source_sigma_margin = 3
```

The file follows the *standard ConfigParser syntax*.

The parameters in this file are defined as follows:

`persistence` Section

(See also the *Persistence stage*.)

`dataset_id` Integer. Specifies the unique ID of a dataset to which the current pipeline run should be appended. If `-1`, a new dataset is created. If you specify a specific data set ID the configuration of your job is retrieved from the database. This will override your job configuration.

`description` String. The name under which the database will be stored in the database. This value is only used if a new dataset is constructed (see `dataset_id`, below).

`rms_est_sigma` Float. Sigma value used for iterative clipping.

`rms_est_fraction` Integer. Determines the size of the subsection used for RMS measurement: the central `1/f` of the image will be used (where f=rms_est_fraction).

`quality_lofar` Section

These are the quality-checking parameters applied if the ingested data is from LOFAR. See also *Quality check stage*.

`low_bound` Float. Reject the image if the measured RMS is less than `low_bound` times the theoretical noise.

`high_bound` Float. Reject the image if the measured RMS is greater than `high_bound` times the theoretical noise.

`oversampled_x` The maximum length of a beam axis.

`elliptical_x` The maximum ratio of major to minor axis length.

`min_separation` The minimum allowed distance from the image centre to a bright radio source in degrees.

`source_extraction` Section

Parameters used in source extraction. See also *"Blind" source extraction stage* and *Forced source-fitting stage*.

`detection_threshold` Float. The detection threshold as a multiple of the RMS noise.

`analysis_threshold` Float. The analysis threshold as a multiple of the RMS noise.

`back_size_x, back_size_y` Integers. The size of the background grid parallel to the X and Y axes of the pixel grid.

`margin` Integer. Pixel data within `margin` pixels of the edge of the image will be excluded from the analysis.

`extraction_radius_pix` Integer. Pixel data more than `extraction_radius_pix` pixels from the centre of the image will be excluded from the analysis.

`deblend_nthresh` Integer. The number of subthresholds to use for deblending. Set to `0` to disable deblending.

`force_beam` Boolean. If `True`, all detected sources are assumed to have the size and shape of the restoring beam (ie, to be unresolved point sources), and these parameters are held constant during fitting. If `False`, all parameters are allowed to vary freely.

box_in_beampix The size of the masking aperture which determines which pixels are used for forced fitting, as a multiple of the beam major axis length. See `tkp.sourcefinder.image.ImageData.fit_to_point()` for details.

ew_sys_err, ns_sys_err Floats. Systematic errors in units of arcseconds which augment the sourcefinder-measured errors on source positions when performing source association. These variables refer to an absolute angular error along an east-west and north-south axis respectively. (NB Although these values are *stored* during the source-extraction process, they affect the source-association process.)

`association` Section

Parameters used in source-association. See *Source association stage* for details. NB the `ew_sys_err`, `ns_sys_err` parameters detailed above also affect source-association.

deruiter_radius Float. Maximum DeRuiter radius for two sources to be considered candidates for association.

beamwidths_limit Float. Maximum separation for two sources to be considered candidates for association, as a multiple of the restoring-beam semimajor-axis length. Default is 1.0, which was the fixed default prior to TraP release 2.1. It may be necessary to use a larger number if your data has large systematic position errors, i.e. if the sources 'jitter' between images, but note that using a large value can cause slowdown of database operations.

`transient_search` Section

Parameters used in transient-detection. See also the *Variability and new-source detection stage*.

new_source_sigma_margin Float. A newly detected source is considered transient if it is significantly above the best (lowest) previous detection limit for that point on-sky. 'Significantly above' is defined by a 'margin of error,' intended to screen out steady sources that just happen to be fluctuating around the detection threshold due to measurement noise. This value sets that margin as a multiple of the RMS of the previous-best image.

`celeryconfig.py` - Task distribution via Celery

Warning: TRAP runs in parallel on a single multi-core machine by default now, using the standard multiprocessing functionality. As such, you should only try to use celery if you want to distribute a single job over multiple machines.

Celery provides a mechanism for distributing tasks over a cluster of compute machines by means of an "asynchronous task queue". This means that users submit jobs to a centralised queueing system (a "broker"), and then one or more worker processes collect and process each job from the queue sequentially, returning the results to the original submitter.

Celery is a flexible but complex system, and the details of its configuration fall outside the scope of this document. The user is, instead, referred to the Celery documentation. Here, we provide only some brief explanation.

If you would like to take advantage of the task distribution system, you will need to set up a broker and one or more workers which will process tasks from it. There are a number of different brokers available, each with their own pros and cons: RabbitMQ is a fine default choice.

Workers can be started by using the `celery worker` option to the *trap-manage.py* script. Indeed, `trap-manage.py` provides a convenient way of interfacing with a variety of Celery subcommands: try `trap-manage.py celery -h` for information.

When you start a worker, you will need to configure it to connect to an appropriate broker. If you are using the `trap-manage.py` script, you can configure the worker through the file *celeryconfig.py* in your *project folder*: set the BROKER_URL variable appropriately. Note that if you are running the broker and a worker on the same host with a standard configuration, the default value should be fine.

Note that a single broker and set of workers can be used by multiple different pipeline users. If running on a shared system, it is likely sensible to regard the broker and workers as a "system service" that all users can access, rather than having each user try to run their own Celery system.

Note also that a worker loads all the necessary code to perform its tasks into memory when it is initalized. If the code on disk changes after this point (for example, if a bug is fixed in the TraP installation), the worker *will continuing executing the old version of the code* until it is stopped and restarted. If, for example, you are using a "daily build" of the TraP code, you will need to restart your workers after each build to ensure they stay up-to-date.

Finally, always bear in mind that it is possible to disable the whole task distribution system and run the pipeline in a single process. This is simpler to set up, and likely simpler to debug in the event of problems. But keep in mind that a running broker is still required. To enable this mode, simple edit your celeryconfig.py file and ensure it contains the (uncommented) line:

```
CELERY_ALWAYS_EAGER = CELERY_EAGER_PROPAGATES_EXCEPTIONS = True
```

Run Celery workers

If you want to parallelize TraP operations using celery, you need to run a separate Celery worker. This worker will receive jobs from a broker, so it is assumed you installed and started a broker in the installation step. Start a Celery worker by running:

```
% trap-manage.py celery worker
```

If you want to increase the log level add --loglevel=info or maybe even debug to the command. If you dont want to use a Celery worker (run the pipeline is serial mode) uncomment this line in the celeryconfig.py file in your pipline directory:

```
#CELERY_ALWAYS_EAGER = CELERY_EAGER_PROPAGATES_EXCEPTIONS = True
```

Note that a running broker is still required.

Celery Configuration File

The *management script* may be used to start a *Celery* worker. The worker is configured using the file celeryconfig.py in the *project directory*. The default contents of this file are:

```
# TraP Celery Configuration

# This file uses the standard Celery configuration system.
# Please refer to the URL below for full documentation:
# http://docs.celeryproject.org/en/latest/configuration.html

# Uncomment the below for local use; that is, bypassing the task distribution
# system and running all tasks in serial in a single process. No broker or
# workers are required.
#CELERY_ALWAYS_EAGER = CELERY_EAGER_PROPAGATES_EXCEPTIONS = True

# Prevents issues with using a separate threading.Thread in addition to Celery.
CELERYD_FORCE_EXECV = True

# Otherwise, configure the broker to which workers should connect and to which
# they will return results. This must be started independently of the
# pipeline.
```

```
BROKER_URL = CELERY_RESULT_BACKEND = 'amqp://guest@localhost//'

# This is used when you run a worker.
CELERY_IMPORTS = ("tkp.distribute.celery.tasks", )
```

Note that this file is Python code, and will be parsed as such. In fact, it is a fully-fledged Celery configuration file, and the reader is referred to the main Celery documentation for a complete reference. Here, we highlight just the important parameters defined in the defualt configuration.

Note the line:

```
#CELERY_ALWAYS_EAGER = CELERY_EAGER_PROPAGATES_EXCEPTIONS = True
```

By uncommenting this line (removing the initial #), the pipeline is forced to run in serial mode. That is, tasks are executed sequentially by a single Python process. No broker and no workers are required. This will likely have a significant impact on performance, but makes the system simpler and easier to debug in the event of problems.

The line:

```
BROKER_URL = CELERY_RESULT_BACKEND = 'amqp://guest@localhost//'
```

specifies the URL of the Celery broker, which is also the location to which workers will return their results. Various different types of broker are available (see our *introduction to Celery* for suggestions), and they must be configured and started independently of the pipeline: the appropriate URL to use will therefore depend on the configuration chosen for your local system.

The other parameters in the file – `CELERY_IMPORTS` and `CELERYD_HIJACK_ROOT_LOGGER` – should be left set to their default values.

1.3.3 Pipeline Design

This section presents an overview of the fundamental algorithms used by, and data flow through, the TraP. It is designed such that everyday users have a full understanding of how their data is being processed. Note that the top-level logic is defined in `tkp.main`, further implementation details for specific sub-sections may be found in the *Developer's Reference Guide*.

As images flow through the TraP, they are processed by a series of distinct pipeline components, or "stages". Each stage consists of Python logic, often interfacing with the pipeline database.

A complete description of the logical design of the TraP is beyond the scope of this document. Instead, the reader is referred to an upcoming publication by *Swinbank et al*. Here, we sketch only an outline of the various pipeline stages.

Pipeline topology and code re-use

An early design goal of the TraP was that the various stages should be easily re-usable in different pipeline topologies. That is, rather than simply relying on "the" TraP, users should be able to mix-and-match pipeline components to pursue their own individual science goals. This mode of operation is not well supported by the current TraP, but some effort is made to ensure that stages can operate as independent entities

Image ordering and reproducibility

The material below describes each of the stages an image goes through as it is processed through the pipeline. It is important to realise, though, that the order in which images are processed is important due to the way in which lightcurves are generated within the database: see the material on *Source association stage* for details. Reproducibility of pipeline results is of paramount importance: the TraP guarantees that results will be reproducible provided that images are always processed *in order of time*. That is, an image from time t_n must always be processed before an image from time t_{n+1}. In order to satisfy this condition, the TraP will internally re-order images provided to it in the *images_to_process.py file* so that they are in time order. *If multiple TraP runs are to be combined in a single dataset, the user must ensure that the runs are in an appropriate sequence.*

Configuration and startup

The pipeline configuration and job management system is described under *Pipeline Configuration*.

Pipeline stages

Persistence stage

(See also the *relevant configuration parameters*.)

A record of all images to be processed is made in the database. Within the database, images are sorted into *datasets*, which group related images together for processing: searches for transients are performed between images in a single databset, for instance. All images being processed are added to the same dataset.

Optionally, a copy of the image pixel data may be stored to a *MongoDB* instance at the same time. This is configured in the *image_cache section* of the pipeline config.

Note that only images which meet the *data accessor* requirements are stored in the database. Any other data provided to the pipeline will be processed: an error will be logged, and that data will not be included in further processing.

Quality check stage

(See also the *relevant configuration parameters*.)

Images are checked to ensure they meet a minimum "quality" standard such that useful scientific information can be extracted from them. If an image fails to meet quality standards, it is rejected and not included in further processing. However, it is still recorded in the database for book-keeping purposes.

The quality check code is structured such that different sets of tests can be applied to images from different telescopes (see source of the `tkp.steps.quality.reject_check()` function for implementation details). Currently, only a selection of tests designed to process LOFAR images are available. Three separate tests are performed:

Image RMS The central subsection of the image is iteratively sigma-clipped until it reaches a user-defined convergence. The RMS of the clipped value is compared to the theoretically expected image noise based on the LOFAR configuration in use. The image is rejected if the noise is signifcantly greated than expected.

Beam shape The restoring beam is represented as an ellipse, parameterized by lengths of its major and minor axes and a position angle. These parameters are checked for sanity. Four separate checks are applied:

- None of the beam parameters should be infinite.
- Both beam axes should be at least two pixels long (the beam is not undersampled).
- Neither beam axis should be longer than a user-defined threshold (the beam is not oversampled).
- The ratio of the major to the minor axis should be lower than a user defined threshold (the beam is not excessively elliptical).

Nearby bright sources There should be no bright radio sources within a user-defined radius of the image centre. The sources checked for are:

- Cassiopeia A
- Cygnus A
- Tauraus A
- Hercules A
- Virgo A
- The Sun
- Jupiter

"Blind" source extraction stage

(See also the *relevant configuration parameters*.)

A source finding and measurement step is run on the image. For more details on the procedure employed, the reader is referred to *Spreeuw (2010)*. Here, we present only a brief introduction.

The algorithm employed is:

1. The image is divided into a square grid of user defined size.
2. Within each cell, the data is sigma-clipped to remove the effect of bright sources.
3. The median pixel value within each clipped cell is calculated, and interpolated across the image to form a background map.
4. The RMS pixel value within each clipped cell is calculated, and interpolated across the image to form a noise map.
5. The background map is subtracted from the data.
6. Pixels in the image data which are more than a "detection threshold" times the value of the noise map at the pixel position are identified as sources.
7. Pixels in the image data which are adjacent to source pixels and which more than an "analysis threshold" times the value of the noise map at the pixel position are appended to the source pixels.
8. The source pixel groups are (optionally) "deblended": multi-component sources are split into their constituent parts by a multi-thresholding technique. The method is based upon that described by *Bertin & Arnouts (1996)*; see *Spreeuw (2010)* for a discussion of the differences.
9. An estimate of the source parameters are made based on the source pixels. The barycentre is taken as the position of the source, and the moments about that centre are used to estimate axis lengths and position angles.
10. A least squares fit of an elliptical Gaussian to the pixel values is performed, starting with the estimated source parameters. If the fit converges, the fitted values are returned as the source measurement; if not, we return the earlier estimate together with an appropriate flag.

For each source, the following measurements are stored:

- Position (RA and declination, including uncertainties);
- An estimate of the absolute on sky angular error on the position (NB this is *not* equivalent to the errors on RA and/or declination);
- The peak flux value;
- The integrated flux value;

- The lengths of the major and minor axes;
- The position angle, measured counterclockwise from the Y axis;
- The significance of the detection (that is, the ratio of the peak flux value to the RMS map at that point).

After the blind extraction has been performed, the list of source measurements is stored in the database.

Source association stage

(See also the *relevant configuration parameters*.)

After all *blind* source measurements have been inserted into the database, they are "associated" with existing sources to form lightcurves. A word on the database nomenclature may be helpful: we store source measurements (the result of a fit to a particular collection of image pixels) in a table called *extractedsource*, and multiple extracted sources are collected together to form a lightcurve by means of the *runningcatalog* table (see the *database schema* documentation for details). This terminology "leaks" into TraP interfaces, and one will often see references to (for example) a "runningcatalog source".

The extracted sources are therefore associated with runningcatalog sources. The association procedure is complex, taking account of multitudinous different ways in which sources may be related. The detailed documentation on *Source Association Logic* covers all the possible association topologies and describes the code pathways in detail; see also *Scheers (2011)*.

Forced source-fitting stage

(See also the *relevant configuration parameters*.)

When the *source association* stage is complete, the pipeline proceeds to handle forced-fits. These may be required either for measuring 'null detections' or sources added to the monitoringlist.

Null detection handling A "null detection" is the term used to describe a source which was expected to be measured in a particular image (because it has been observed in previous images covering the same field of view) but was in fact not detected by the *blind extraction* step.

After the blindly-extracted source measurements have been stored to the database and have been associated with the known sources in the running catalogue, the null detection stage starts. We retrieve from the database a list of sources that serve as the null detections. Sources on this list come from the *runningcatalog* and

- Do not have a counterpart in the extractedsources of the current image after source association has run.
- Have been seen (in any band) at a timestamp earlier than that of the current image.

We determine null detections only as those sources which have been seen at earlier times which don't appear in the current image. Sources which have not been seen earlier, and which appear in different bands at the current timestep, are *not* null detections, but are considered as "new" sources.

For all sources identified as null detections, we measure fluxes by performing a forced elliptical Gaussian fit to the expected source position on the image. The procedure followed is similar to that used for *blind extraction*, but rather than allowing the pixel position of the barycentre to vary freely, it is held to the known source position. No deblending is performed.

The results of these "forced" source measurements are marked as such and appended to the database.

After being added to the database, the forced fits are matched back to their running catalog counterparts in order to append it as a datapoint in the light curve. This matching is does not include the De Ruiter radius, since the source position came from the running catalog. It is sufficient to use the weighted positional error as a cone search, since the positions are identical. Therefore the forced fit position is not included as an extra datapoint in the position of the running catalog. The fluxes, however, are included into the statistical properties and the values are updated.

It is worth emphasizing that the above procedure guarantees that every known source will have either a blind detection or a forced-fit measurement in every image from the moment it was detected for the first time.

Null-detection depends upon the same *job configuration file* parameters as the *"blind" source-extraction stage*.

Monitoringlist If monitoringlist positions have been *specified when running a job*, then these are always force-fitted whenever an image's source-extraction region contains the designated co-ordinates. These forced fits are used to build up a special *runningcatalog* entry which is excluded from association with regular extractions.

Variability and new-source detection stage

(See also the *relevant configuration parameters*.)

Variability index calculation After all images for a given timestep have been processed and the resulting source measurements have been assigned to *runningcatalog* entries (effectively lightcurves), variability indices are calculated for the most recent timestep, and stored as part of the association recorded in the *assocxtrsource* table.

Note that a single runningcatalog source may contain entries from multiple independent frequency bands. The variability indices are calculated independently for each frequency band, hence the ν suffix in the calculations below denotes an index over the different bands.

For a comprehensive discussion of the transient and variability detection algorithms currently being employed, see *Scheers (2011)* chapter 3. Here, we provide a brief outline.

We define two metrics for identifying variability in a lightcurve. The flux coefficient of variation, which we denote V_ν, is defined as

$$V_\nu \equiv \frac{s_\nu}{\overline{I_\nu}} = \frac{1}{\overline{I_\nu}}\sqrt{\frac{N}{N-1}(\overline{I_\nu^2} - \overline{I_\nu}^2)}$$

where $\overline{I_\nu}$ is the mean flux of all measurements in the lightcurve at frequency ν, s_ν is the standard deviation of those flux measurements and N is the number of measurements.

The second metric is η_ν, which is defined based on reduced χ^2 statistics as

$$\eta_\nu \equiv \chi^2_{N-1} = \frac{1}{N-1}\sum_{i=1}^{N}\frac{(I_{\nu,i} - \overline{I_\nu}^*)^2}{\sigma^2_{I_{\nu,i}}}$$

Where $\overline{I_\nu}^*$ is the average of the flux measurements weighed by their uncertainties. η_ν is the χ^2 probability distribution. The probability that the source is "flat" (i.e. has no significant variability) is then the integral of the distribution from the measured value of η_ν to ∞; the probability that it *isn't* flat is thus 1 minus this quantity.

See also the *appendices on the database schema* for details of how these are iteratively updated.

'New source' detection We also attempt to identify any newly extracted sources which we suspect are intrinsically variable in nature (i.e. they are getting brighter, as opposed to our observations getting deeper or even simply looking at a previously unobserved patch of sky). The algorithm for evaluating new sources is encoded by `tkp.db.associations._determine_newsource_previous_limits()`. Sources we deem to be intrinsically 'new' are then recorded in the *newsource* table.

1.4 Developer's Reference Guide

The developer's reference guide presents the TraP from a code-oriented perspective: it aims to enable developers to quickly get up-to-speed with the structure of the project so they are able to start contributing. It also describes the development process followed and the requirements regarding code reviews, testing and documentation.

It is assumed that the reader is already familiar with the *User's Reference Guide*.

1.4.1 Development Procedure

Here we describe the development process used when working on the TraP. All developers are encouraged to familiarize themselves with this material before making any changes to the code.

Accessing the Code

All code relating to the core TraP functionality is hosted in the transientskp/tkp `git` repository on GitHub.

The repository is currently only available to authorized project members. If you require access, contact `git@transientskp.org` for help. Please *do not* redistribute the code without authorization.

You will need to be familiar with basic `git` operation before you can work with the codebase. There are many excellent tutorials available: start at the Git front page.

Planning

We aim to make releases of the TraP at the cadence of a few per year. Broadly, the plan is to alternate technically focused releases, which clean up the codebase and make behind-the-scenes improvements, with science based releases, which provide new functionality to end users. Technical releases have odd numbers; science releases even. We will provide bugfixes, but no new development, for the most recent release; support for earlier releases is on a "best efforts" basis only.

At the start of a release cycle, the developers and commissioners will meet to discuss the issues which will be addressed in the upcoming release. Having agreed upon a set of goals, a roadmap for the release will be created on the *Issue Tracker*.

It is generally acceptable to submit minor changes and tweaks, as well as bugfixes, to the codebase at any time. However, if you are planning a major change which will have significant repercussions for other developers, or which causes end-user visible changes, it should be included in the goals for a particular release and there should be a broad consensus about the plan for implementation before you begin coding.

Issue Tracker

We keep track of bug reports and feature requests using the Github repository issue tracker.

We use the issue tracker extensively for project planning and managing releases. It is not a *hard* requirement that every commit to the repository refers to a particular issue, but you are strongly encouraged to record your activities on the tracker and to refer to it in commit messages as appropriate.

Coding Standards

We do not rigorously enforce a set of coding style rules, with the sole exception that all indents in Python code should consist of 4 spaces. However, all code is expected to be considerately written, appropriately (but not excessively) commented, and as easy to read as possible. Please read PEP 8 carefully and bear it in mind as you work.

You may wish to run `pylint` or similar tools on the codebase. Occasionally, such tools can provide useful hints about how to make your code clearer: by all means act upon these. However, much of the output of such tools is subjective: be sure you understand and agree with their recommendations, and be very reluctant to apply them to pre-existing code without fully considering the implications.

Testing & Continuous Integration

The `tests` directory contains a (reasonably) comprehensive TraP test suite. Tests are written following the Python unittest conventions. Although they can be run using just the standard Python library tools, you might find that nose provides a more convenient interface.

Requirements

Whenever any changes are made to the code, all pre-extant tests *must* continue to pass. If the change involves changing the result of one or more tests, any such changes must be clearly explained and check during *Code Review*.

New features and bug fixes must be accompanied by appropriate tests. These should both demonstrate the correct operation of the code with good input data as well as demonstrating how it responds to bad inputs. Again, the code reviewer should check that the tests provided are clear and comprehensive.

Test data

Many of the tests require data files which are distributed separately from the TraP. Having cloned the TraP repository (into the directory `tkp` in this example), fetch the appropriate test data as follows:

```
$ cd tkp
$ git submodule init
$ git submodule update
```

In future, running `git submodule update` again will fetch the latest version of the test data if required.

By default, the test suite will look for data in the appropriate subdirectory of your checked out Trap source. However, if the data has been installed elsewhere, or if you are running the test suite against an installed copy of the TraP, you can specify the data location by exporting the `TKP_TESTPATH` environment variable.

If the data is not available, the relevant tests will be skipped. The partial test suite should still complete successfully; note that your copy of the TraP will not be fully tested.

Database

Many of the tests require interaction with the *pipeline database*. A convient way to configure the database is by using environment variables. For example:

```
$ export TKP_DBENGINE=xxx
$ export TKP_DBNAME=xxx
$ export TKP_DBHOST=xxx
$ export TKP_DBPORT=xxx
$ export TKP_DBUSER=xxx
$ export TKP_DBPASSWORD=xxx
$ trap-manage.py initdb
```

If you do not have or need a database, you can disable all the tests which require it by exporting the variable `TKP_DISABLEDB`. The partial test suite should still complete successfully, but your copy of the TraP will not be fully tested.

Running the tests

Within the tests directory, use the runtests.py script to start the test suite using nose:

```
$ cd tests
$ python runtests.py -v
```

Command line arguments (such as -v, above) are passed onwards to nose; you can use them, for example, to select a particular subset of the suite to run.

Often it is convenient to run the TraP against a work-in-progress version of the TraP while continuing to use other libraries and tools installed on the system. This just requires setting the PYTHONPATH environment variable to the root of the development tree:

```
$ cd tkp
$ export PYTHONPATH=$(pwd):${PYTHONPATH}
$ cd tests
$ python runtests.py -v
```

Alternatively, you may choose to work with a virtualenv and install the TraP in 'development mode':

```
$ pip install --editable .
```

Continuous integration

There is a Jenkins instance at https://jenkins.transientskp.org/ which builds and tests all code committed to the master branch as well as any incoming pull requests. This constitutes the reference system on which tests are required to pass: a failure here should result in a change being rejected, regardless of its successful operation on some other system.

Documentation

Requirements

All parts of the TraP should be documented *both* at a level that is suitable for end users using the pipeline and for developers who wish to understand, fix and extend the codebase. Broadly, that distinction reflects the structure of this manual. Note that we expect that developers will benefit from both automatically generated API documentation making use of appropriate docstrings in the code, and higher-level descriptions of system architecture and functionality.

All new features or changes *must* be accompanied by appropriate documentation. Reviewers are required to check that pull requests are well documented before merging. See the material on *Code Review* for details.

Technical details

Documentation is written using Sphinx. The documentation for the HEAD of the master branch of the transientskp/tkp repository is, together with the documentation for all released versions, is automatically build every night and put online at http://docs.transientskp.org/.

Docstrings should make use of the "Napoleon" syntax. For example:

```
Args:
    path (str): The path of the file to wrap
    field_storage (FileStorage): The :class:`FileStorage` instance to wrap
    temporary (bool): Whether or not to delete the file when the File
       instance is destructed
```

```
Returns:
    BufferedFileStorage: A buffered writable file descriptor
```

Code Review

Rather than pushing changes directly to `transientskp/tkp`, developers are asked to submit their changes for review before they are merged into the project. Ideally, this applies to all changes; however, it is recognized that in certain cases – e.g. recovering from a previous mistake, or making trivial formatting changes – pushing directly may be appropriate.

Code reviews are carried out using GitHub's Pull Request functionality.

Submitting code for review

Using the GitHub web interface, "fork" a copy of the repository to your own account (note that even if you do not have a paid GitHub account, forks of private repositories remain private, so you are not exposing the code to the outside world).

Clone a copy of your forked repository to your local system:

```
$ git clone git@github.com:<username>/tkp.git
```

Create a branch which you will use for working on your changes:

```
$ git checkout -b my_new_branch
```

Work on that branch, editing, adding, removing, etc as required. When you are finished, push your changes pack to GitHub:

```
$ git push orign my_new_branch
```

Return to the GitHub web interface, and issue a pull request to merge your `<username>:my_new_branch` into `transientskp:master`.

Reviewing code

Reviewing code is just as valuable an activity as creating it: *all* developers are expected to handle a share of code reviews. The procedure is simple: visit the GitHub web interface, and choose a pull request to review. Look through it carefully, ensuring that it adheres to the guidelines below. If you are happy with it, and it can be automatically merged, simply hit the big green "Merge Pull Request". If an automatic merge isn't possible, you will have to check out the code onto your system and merge it manually: this takes a little time, but GitHub document the process.

If you *aren't* happy with the code as submitted, you can use the GitHub web interface to add both general comments covering the whole PR and to comment on specific lines explaining what the problem is. You can even issue your own PR suggesting new commits that the submitter could merge with their own work. Please be as clear as possible and make constructive suggestions as to how the submitter can make improvements: remember, the aim is to get high quality code merged into the repository in a timely fashion, not to argue over obscure minutiae!

Requirements for Pull Requests

When submitting or reviewing a pull request, please bear the following guidelines in mind:

- PRs should be as concise and self-contained as possible. Sometimes, major functionality changes will require large amounts of code to be changed, but this should be the exception rather than the rule. Be considerate to the reviewer and keep changes minimal!
- It is not required that reviewers check every line for correctness, but they should read through the code and check that it is clearly structured and reasonably transparent in operation.
- Effectively all requests should be accompanied by appropriate additions to the test suite. If it is not possible to provide tests, the submitter should explain why, and the reviewer must check and agree with this justification.
- Any changes to user-visible functionality must be accompanied by appropriate updates to the Users' Guide.
- Any changes to APIs or the structure of code must be accompanied by appropriate updates to the Developers' Reference.
- Both submitter and reviewer should check that the PR does not introduce any regressions into the unit test results (ie, no tests which previously passed should fail after merging.)
- Make sure that the version control history is readable. This means both using descripive commit messages (future developers will not thank you for recording that you did "stuff"), and appropriate use of `git rebase` to eliminate dead-end and work-in-progress commits before submitting the code for review.

Release procedure

Code and repository management

We use what is sometimes known as semantic release versioning. We use branches to track major releases, and we tag all point releases, which makes it easy to track down the relevant git history.

Versions are numbered in the format MAJOR.MINOR.PATCH, where major releases break backwards compatibility, minor releases add functionality without breaking backwards compatibility, and patch releases are backwards-compatible bug fixes.

To make a release you should first create a new branch (if appropriate: see below), then set the version number in the code, then tag the new release to note the version number.

Major releases are sequentially numbered (`1`, `2`, `N`). They happen on a branch named `releaseN`. Create the branch as follows:

```
$ git checkout -b <releaseN>
```

Minor releases happen on existing release branches. They are named `N.M`, where `N` is the major release version and `M` the minor version. The first commit on every release branch corresponds to `N.0.0`. Check out the relevant branch:

```
$ git checkout <releaseN>
```

Next, edit the code to set the version number in `tkp/__init__.py`.

Commit your changes. This commit is the basis of the release:

```
$ git commit -am "Release N.M.O"
```

Tag the release. This is important, as we use the tags to indicate which versions should be built and added to the documentation site:

```
$ git tag -a "rN.M.O"
```

Push everything, including the tag, to GitHub:

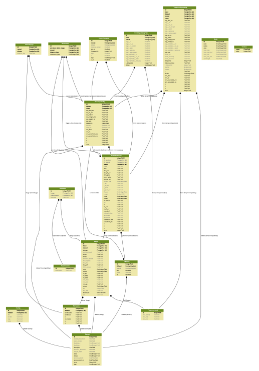
```
$ git push --tags origin releaseN
```

Information about the systems available to users and developers working on LOFAR data using the TraP are available on the LOFAR wiki.

1.4.2 The Pipeline Database

This section presents material related to the structure and maintenance of the database used by the TraP.

Schema reference



Note that this diagram is periodically updated and may not reliably reflect the current status of the development version.

Table Listing

assocxtrsource This table stores the association between an extracted source and its runningcatalog counterpart source, where the relation might be of type 1-1, 1-n or n-1.

runcat A reference to the `runcatid` in `runningcatalog`. It may be regarded as the "base" identitfier of a lightcurve, whereas the lightcurve consist of multiple frequency bands and Stokes parameters.

xtrsrc This is the ID of the extracted source that could be associated to runningcatalog source. Together, the `runcat_id` and the `xtrsrc` form a unique pair.

type Type of association, and its description. n-m, where n is the number of `runningcatalog` sources, and m the number of extracted sources. (The assignment of numbers is to association types is arbitrary.)

`type = 2` Base point of a 1-n (one-to-many) association (relates to `type = 6`)

`type = 3` 1-1 association.

`type = 4` 0-1 (zero-to-one) association, i.e. a new source.

`type = 6` Associations of 1-n (one-to-many) type. These are updates of pre-existing associations, due to the 1-n case (relates to `type = 2` association).

`type = 7` Forced-fit to position of the null detection of a known source.

`type = 8` Initial forced fit to a monitoring list position.

`type = 9` Subsequent forced fit to a monitoring list position (relates to `type = 8`.

Note that many-to-1 relations reduce to 1-to-1 and 1-to-many associations. Therefore, there is no type specifying such a case.

distance_arcsec The distance in arcsec between the associated sources, calculated by the database using the dot product Cartesian coordinates

r The dimensionless distance (De Ruiter radius) between the associated sources. It is determined as the positional differences weighted by the errors, calculated by the association procedure inside the database (*Scheers, 2011*, chapter 3).

loglr The logarithm of the likelihood ratio of the associated sources, defaults to `NULL` if not calculated (*Scheers, 2011*, chapter 3).

v_int The flux coefficient of variation , V_ν, based on the integrated flux values up to this datapoint.

eta_int The 'reduced chi-squared' variability index, η_ν, based on the integrated flux values up to this datapoint.

f_datapoints The number of flux datapoints (including the extractedsource referenced in this entry) used to calculate the variability indices.

assocskyrgn (See also *skyregion table*.)

This table records which *runningcatalog* sources we expect to see in any given skyregion. This serves two purposes: it allows us to determine when we *do not* see previously detected sources, presumably because they have dropped in flux (see `tkp.db.nulldetections.get_nulldetections()`). It also allows us to determine whether a new runningcatalog entry (i.e. a newly detected source without associated historical detections) is being detected for the first time because it is actually a new transient, or if it is simply the first time that region of sky has been surveyed (see `tkp.db.associations._determine_newsource_previous_limits()`).

This table is updated under 2 circumstances:

- A new skyregion is processed, and associations must be made with pre-existing runcat entries (see SQL function `updateSkyRgnMembers`).
- A new runningcatalog source is added, and must be associated with pre-existing skyregions (see `tkp.db.associations._insert_new_runcat_skyrgn_assocs()`).

runcat References the associated `runningcatalog` ID.

skyrgn References the associated `skyregion` ID.

distance_deg Records the angular separation between the `runningcatalog` source and the `skyregion` centre, at time of first association.

config A simple table of `(dataset, section, key, value, type)` tuples, providing a means of recording the contents of the *job_params.cfg - Job Parameters Configuration* file used to initiate a particular TraP processing run. This provides provenance and reproducibility for any given dataset.

dataset The dataset table provides an id for grouping together results, usually from a single run of the TraP. As such, it represents a collection of images processed with a particular configuration (see also the *config* table).

Note that it is possible to specify a dataset_id in order to process more recent images as part of a pre-existing dataset, in which case the previously stored configuration is loaded from the *config* table, to ensure consistency across the dataset.

id Every dataset gets a unique ID. The ID is generated by the database.

rerun The value indicates how many times a dataset with a given description was processed by the pipeline. Note that every dataset still has a unique id, even when it was reprocessed. At insertion time this is incremented by 1 when the description of the dataset is already present in the table; otherwise, it defaults to 0.

type Not being used.

process_start_ts The timestamp of the start of processing the dataset, generated by the database.

process_end_ts The timestamp of the completion of processing the dataset, generated by the database. `NULL` if processing is ongoing.

detection_threshold The detection threshold that was used by source finder to extract sources. Value read from either the source finder parset file or the tkp.cfg file. See the *PySE documentation* for more information.

analysis_threshold The analysis threshold that was used by source finder to extract sources. Value read from either the source finder parset file or the tkp.cfg file. See the *PySE documentation* for more information.

assoc_radius The association radius that is being used for associating sources. Value read from either the source finder parset file or the tkp.cfg file.

backsize_x Background grid segment size in x. Value read from either the source finder parset file or the tkp.cfg file. See the *PySE documentation* for more information.

backsize_y Background grid segment size in y. Value read from either the source finder parset file or the tkp.cfg file. See the *PySE documentation* for more information.

margin_width Margin applied to each edge of image (in pixels). Value read from either the source finder parset file or the tkp.cfg file. See the *PySE documentation* for more information.

description A description of the dataset, with a maximum of 100 characters.

node(s) Determine the current and number of nodes in case of a sharded database set-up.

extractedsource This table contains all the extracted sources (measurements) of an image. Maybe source is not the right description, because measurements may be made that were erronous and do not represent a source.

Most values come from the sourcefinder procedures, and some are auxiliary deduced values generated by the database.

This table is empty *before* an observation. *During* an observation new sources are inserted into this table. *After* an observation this table is dumped and transported to the catalog database.

All detections (measurements) found by sourcefinder are appended to this table. At insertion time some additional auxiliary parameters are calculated by the database as well. At anytime, no entries will be deleted or updated. The TraP may add forced-fit entries to this table as well. Then `extract_type` is set to 1.

id Every inserted source/measurement gets a unique id, generated by the database.

image The reference ID to the image from which this sources was extracted.

zone The zone ID in which the source declination resides, calculated by the database. The sphere is devided into zones of equal width: currently fixed to 1 degree, and the zone is effectively the truncated declination. (decl=31.3 => zone=31, decl=31.9 => zone=31). This column is primaly for speeding up source look-up queries.

ra Right ascension of the measurement (J2000 degrees). Calculated by the sourcefinder procedures.

decl Declination of the measurement (J2000 degrees). Calculated by the sourcefinder procedures.

ra_err The 1-sigma error on `ra` (degrees), i.e. the square root of the quadratic sum of the fitted error (`ra_fit_err`) and the systematic error (`ew_sys_err`) after the latter has been corrected for ra inflation depending on declination. It is calculated by the database at insertion time. Note that this error is declination dependent and the source lies in the range [ra - ra_err, ra + ra_err].

decl_err The 1-sigma error on declination (degrees), i.e. the square root of the quadratic sum of the fitted error (`decl_fit_err`) and the systematic error (`ns_sys_err`), calculated by the database at insertion time. Note that the source lies in the range [`decl - decl_err`, `decl + decl_err`]

uncertainty_ew The 1-sigma on-sky error on `ra` (in the east-west direction) (degrees), ie. the square root of the quadratic sum of the error radius (`error_radius`) and the systematic error (`ew_sys_err`). It is calculated by the database at insertion time. Note that this is a positional uncertainty and is declination independent. This error is being used in the De Ruiter calculations.

uncertainty_ns Analogous to `uncertainty_ew`.

ra_fit_err The 1-sigma error on `ra` (degrees) from the source gaussian fitting, calculated by the sourcefinder procedures. It is important to note that a source's fitted ra error increases towards the poles, and is thus declination dependent (see also `error_radius`).

decl_fit_err The 1-sigma error from the source fitting for declination (degrees), calculated by the sourcefinder procedures (see also `error_radius`).

ew_sys_err The systematic error on RA (arcsec). (As an on-sky angular uncertainty, independent of declination.) It is a telescope dependent error and is provided by the user in the pipeline configuation.

ns_sys_err Analogous to `ew_sys_err`.

error_radius Estimate of the absolute angular error on a source's central position (arcsec). It is a pessimistic estimate, because it takes the sum of the error along the X and Y axes.

x, y, z Cartesian coordinate representation of RA and declination.

racosdecl The product of RA and cosine of the declination. Helpful in source look-up association queries where we use the De Ruiter radius as an association parameter.

margin Used for association procedures to take into account sources that lie close to ra=0 & ra=360 meridian. *NOTE:* Not currently used.

- `True`: source is close to ra=0 meridian
- `False`: source is far away enough from the ra=0 meridian

det_sigma

The significance level of the detection: $20 \times f_{\mathrm{peak}}/\mathrm{det_sigma}$ provides the detection RMS. See *Spreeuw (2010)*.

semimajor Semi-major axis that was used for gauss fitting (arcsec), calculated by the sourcefinder procedures.

semiminor Semi-minor axis that was used for gauss fitting (arcsec), calculated by the sourcefinder procedures.

pa Position Angle that was used for gauss fitting (from north through local east, in degrees), calculated by the sourcefinder procedures.

f_peak peak flux (Jy), calculated by the sourcefinder procedures.

f_peak_err 1-sigma error (Jy) of `f_peak`, calculated by the sourcefinder procedures.

f_int integrated flux (Jy), calculated by the sourcefinder procedures.

f_int_err 1-sigma error (Jy) of `f_int`, calculated by the sourcefinder procedures.

chisq, reduced_chisq Goodness of fit metrics for fitted Gaussian profiles.

Note: These provide useful information for e.g. machine-classification and filtering of transient candidates, but strictly speaking are not statistically valid (we might rename them in a future release to avoid confusion). See *`tkp.sourcefinder.fitting.goodness_of_fit()`* for details.

extract_type Reports how the source was extracted by sourcefinder (*Spreeuw (2010)*), Currently implemented values are:

- `0`: blind fit
- `1`: forced fit to pixel
- `2`: manually monitored position

fit_type Reports what fitting type was used by sourcefinder (*Spreeuw (2010)*). Currently implemented values are:

- `0`: moments-based analysis
- `1`: Gaussian fitting

ff_runcat Null, except when the extractedsource is a forced fit requested due to a null-detection. In that case, it is used to link null-detection extractions to their appropriate runningcatalog entry via the `assocxtrsource` table. It will initially point to the runningcatalog id which was null-detected, but may change back to Null later on (after the initial association is recorded in assocxtrsource) if the runningcatalog entry forks due to a one-to-many association.

ff_monitor Null, except when the extractedsource is a forced fit requested for a position in the `monitor` table. In that case, it identifies the relevant `monitor` entry, and is used in the association process.

node(s) Determine the current and number of nodes in case of a sharded database set-up.

frequencyband This table contains the frequency bands that are being used inside the database. Here we adopt the set of pre-defined Standard LOFAR Frequency Bands and their bandwidths as defined for MSSS. Included are frequency bands outside the LOFAR bands, in order to match the external catalogue frequency bands. When an image is taken at an unknown band, it is added to this table by the SQL function `getBand()`. To make it possible to easily compare images with slightly different effective frequencies, new bands are constructed by rounding the effective frequency to the nearest MHz, and assuming a band width of 1 MHz.

id Every frequency band has its unique ID, generated by the database.

freq_central The central frequency (Hz) of the defined frequency band. (Note that this is not the effective frequency, which is stored as a property in the image table.)

freq_low The low end of the frequency band (Hz).

freq_high The high end of the frequency band (Hz).

image This table contains the images that are being or were processed in the TraP. Note that the format of the image is not stored as an image property. An image might be a composite of multiple images, but it is not yet defined how the individual values for effective frequency, integration times, etc are propagated to the columns of the `image` table. The CASA image description for LOFAR describes the structure of a LOFAR CASA Image, from which most of the data of the `image` table originates.

An image is characterised by

- observation timestamp (`taustart_ts`);
- integration time (`tau`);
- frequency band (`band`);
- Stokes parameter (`stokes`).

A group of images that belong together (defined by user, but not specified any further) are in the same data set (i.e. they have the same reference to dataset).

id Every image is assigned a unique ID by the database.

dataset The dataset to which the image belongs.

tau The integration time of the image. This is a quick reference number related to tau_time. Currently this is not used.

band The frequency band at which the observation was carried out. Its value refers to the ID in frequencyband, where the frequency bands are predefined. The image's effective frequency falls within this band. If an image has observation frequency that is not in this table, a new entry will be created based an the effective

stokes The Stokes parameter of the observation. 1 = I, 2 = Q, 3 = U and 4 = V. The Stokes parameter originates or is read from the CASA Main table in the coords subsection from the `stokesX` record. The char value is converted by the database to one of the four (tiny) integers.

tau_time The integration time (in seconds) of the image. The value originates or is read from the CASA `LOFAR_OBSERVATION` table by differencing the `OBSERVATION_END` and `OBSERVATION_START` data fields.

freq_eff The effective frequency (or synonymously rest frequency) (in Hz) at which the observation was carried out. The value originates or is read from the CASA Main table in the coords subsection from the `spectralX` record and the `crval` field. Note that in the case of FITS files the header keywords representing the effective frequency are not uniquely defined and may differ per FITS file.

freq_bw The frequency bandwidth (Hz) of the observation. Value originates or is read from the CASA Main table in the coords subsection from the `spectralX` record and the `cdelt` field. N This is a required value and when it is not available an error is thrown.

taustart_ts The timestamp of the start of the observation, originating or read from the CASA LOFAR_OBSERVATION table from the `OBSERVATION_START` data field.

skyrgn The sky region to which the image belongs.

rb_smaj The semi-major axis of the restoring beam, in degrees. Full major axis value originates or is read from the CASA Main table in the imageinfor subsection from the `restoringbeam` record and is converted at db insertion time.

rb_smin

The semi-minor axis of the restoring beam, in degrees. Full minor axis value originates or is read from the CASA Main table in the imageinfor subsection from the `restoringbeam` record and is converted at db insertion time.

rb_pa The position angle of the restoring beam (from north to east to the major axis), in degrees. Value originates or is read from the CASA Main table in the imageinfor subsection from the `restoringbeam` record.

deltax, deltay Pixel sizes along the X & Y axes in degrees.

fwhm_arcsec The full width half maximum of the primary beam, in arcsec. Value not yet stored in table.

fov_degrees The field of view of the image, in square degrees. Not yet stored in table.

rms_qc RMS for quality-control. This is the sigma-clipped RMS value from the central region of the image, calculated in the persistence step.

rms_min, rms_max The minimum and maximum values of the estimated-RMS-map within the source-extraction region. Used when determining if a newly-detected source is a probable transient, or just due to deeper imaging.

detection_thresh, analysis_thresh The detection and analysis thresholds (as a multiple of the local RMS value) used in the source extraction process for this image.

url The url of the physical location of the image at the time of processing. NOTE that this needs to be updated when the image is moved.

node(s) Determine the current and number of nodes in case of a sharded database set-up.

monitor This table stores the user-requested monitoring positions for a dataset.

id Every position in the monitor table gets a unique id.

dataset The relevant dataset ID - monitoring positions are dataset-specific.

ra, decl The position coordinates (J2000 degrees).

runcat Initially `NULL`. When a forced-fit is first made to a monitoring position, this column is updated to point to the relevant entry in the runningcatalog.

name A short descriptive name, e.g. GRB140101A or SNe150101, for more user-friendly display of results. This functionality is not currently implemented, but the presence of this column allows it to be trivially implemented in future without requiring a database migration.

newsource For discovering transient or variable sources, our primary tools are variability statistics. However, a bright single-epoch source cannot sensibly be assigned variability statistics until at least a second measurement (possibly non-detection) has been made.

This table tracks new sources, in the hopes that new sources considered sufficiently bright enough to be interesting may be flagged up immediately.

See *`tkp.db.associations._determine_newsource_previous_limits()`* for details on how these values are assigned.

id Unique identifier, set by the database.

runcat Reference to the associated `runningcatalog` entry.

trigger_xtrsrc Reference to the extracted source that caused insertion of this newsource.

newsource_type Refers to how certain we are that the newly discovered source is really "physically new", i.e. transient. Since we do not store fine-grained noise-maps in the database, we must be fairly conservative in our labelling here.

Type 0 sources may be a steady source located in a high-RMS region, newly detected due to noise fluctuations, or may be a real transient in a low-RMS region.

Type 1 sources are bright enough that we can be fairly certain they are really new - they are significantly brighter than the `rms_max` in the previous image with best detection limits.

previous_limits_image The ID of the previous image with the best upper limits on previous detections of this source. Can be used to calculate the significance level of the new-source detection.

node This table keeps track of zones (declinations) of the stored sources on the nodes in a sharded database configuration. Every node in such a set-up will have this table, but with different content.

node The ID of the node

zone The zone that is available on the node

zone_min The minimum zone of the zones

zone_max The maximum zone of the zones

zone_min_incl Boolean determining whether the minimum zone is included.

zone_max_incl Boolean determining whether the maximum zone is included.

zoneheight The zone height of a zone, in degrees

nodes The total number of nodes in the sharded database configuration.

Note: The following sections on the `runningcatalog`, `runningcatalog_flux` and `temprunningcatalog_flux` are annotated using the style of mathematical notations developed in the *Appendix*.

rejection This table contains all rejected images and a reference to the reason.

id The database ID of the rejection.

image A foreign key relationship to the image ID of the rejected image.

rejectreason A foreign key relationship to the ID of the rejection reason.

comment A textfield with more details about the reason for rejection. For example in the case of a rejection because of RMS value to high, this field will contain the theoretical noise value and the calculated RMS value of the image.

rejectreason This table contains all the possible reasons for rejecting an image.

id The database ID of the rejection reason.

description An description of the rejection.

runningcatalog (See *Appendices* for explanation of mathematical notation.)

While a single entry in `extractedsource` corresponds to an individual source measurement, a single entry in `runningcatalog` corresponds to a unique astronomical source detected in a specific dataset (series of images). The position of this unique source is a weighted mean of all its individual source measurements. The relation between a `runningcatalog` source and all its measurements in `extractedsource` is maintained in `assocxtrsource`.

The association procedure matches extracted sources with counterpart candidates in the runningcatalog table. Depending on their association parameters (distance and De Ruiter radius) of the `runningcatalog` source and `extractedsource` source, the source pair ids are added to `assocxtrsource`. The source properties, position, fluxes and their errors in the `runningcatalog` and `runningcatalog_flux` tables are then updated to include the counterpart values from the extracted source as a new datapoint.

If no counterpart could be found for an extracted sources, it is appended to `runningcatalog` as a "new" source (datapoint=1).

id Every source in the running catalog gets a unique ID.

xtrsrc The ID of the extractedsource for which this runningcatalog source was detected for the first time.

dataset The dataset to which the runningcatalog source belongs to.

datapoints $= N_\alpha$ **or equivalently** N_δ The number of datapoints (or number of times this source was detected) that is included in the calculation of the *position* averages. It is assumed that a source's position stays relatively constant across bands and therefore all bands are included in averaging the position.

zone The zone ID in which the source declination resides. The sphere is divided into zones of equal width: here fixed to 1 degree, and the zone is effectively the truncated declination. (decl=31.3 => zone=31, decl=31.9 => zone=31)

wm_ra $= \xi_\alpha$ The weighted mean of RA of the source [in J2000 degrees].

wm_decl $= \xi_\delta$ The weighted mean of Declination of the source [in J2000 degrees].

wm_uncertainty_ew The positional on-sky uncertainty in the east-west direction of the weighted mean RA (degrees).

wm_uncertainty_ns The positional on-sky uncertainty in the north-south direction of the weighted mean Dec (degrees).

avg_ra_err The average of the `ra_err` of the source (degrees).

avg_decl_err The average of the `decl_err` of the source (degrees).

avg_wra $= \overline{w_\alpha \alpha}$ The average of (the square of `ra/uncertainty_ew`). Used for calculating the weighted mean of the RA.

avg_wdecl $= \overline{w_\delta \delta}$ Analogous to `avg_wra`.

avg_weight_ra $= \overline{w_\alpha}$ The average of the reciprocal of the square of `uncertainty_eq`. Used for calculating the weighted mean of the RA.

avg_weight_decl $= \overline{w_\delta}$ Analogous to `avg_weight_ra`.

x, y, z The Cartesian coordinate representation of `wm_ra` and `wm_decl`.

inactive Boolean to set an entry to inactive. This is done during the *source association* procedure, where e.g. the many-to-many cases are handled and an existing entry is replaced by two or more entries.

mon_src Boolean to indicate whether an entry is from the user-specified monitoring list. Default value is false.

runningcatalog_flux The runningcatalog_flux table contains the averaged flux measurements of a runningcatalog source, per band and stokes parameter. The combination runcat, band and stokes is the primary key.

The flux squared and weights are used for calculations of the variability indices, V_ν and η_ν.

runcat The `id` of the `runningcatalog` entry to which this band/stokes/flux belongs.

band Reference to the frequency band of this flux.

stokes Stokes parameter: 1 = I, 2 = Q, 3 = U, 4 = V.

f_datapoints $= N_I$ The number of *flux* datapoints for which the flux averages were calculated.

avg_f_peak $= \overline{I}$ Average of peak flux.

avg_f_peak_sq $= \overline{I^2}$ Average of (peak flux squared).

avg_f_peak_weight $= \overline{w_I}$ Average of one over peak flux errors squared.

avg_weighted_f_peak $= \overline{w_I I}$ Average of ratio of (peak flux) and (peak flux errors squared).

avg_weighted_f_peak_sq $= \overline{w_I I^2}$ Average of (weighted peak flux squared).

avg_f_int, avg_f_int_sq, avg_f_int_weight, avg_weighted_f_int, avg_weighted_f_int_sq Analogous to those above, except for the *integrated* flux.

skyregion Entries in this table represent regions of sky which have been, or will shortly be, processed via the usual extract-sources-and-associate procedures. By listing regions of sky in a dedicated table, we de-duplicate information that would otherwise be repeated for many images.

When an image is first inserted into the database, the SQL function `getSkyRgn` is called. This first checks for the pre-existence of a matching skyregion entry. If none exists, then a new entry is created and the SQL function `updateSkyRgnMembers` is called to update the *assocskyrgn* table as necessary.

See also *assocskyrgn*.

dataset Reference to the `dataset id`, for the dataset to which the skyregion belongs. This field is needed in order to restrict association to the current dataset.

centre_ra and centre_decl The central coordinates (J2000) (or pointing centre) of the region, in degrees. RA and Dec values are read from `DataAccessor` metadata.

xtr_radius The radius of the circular mask used for source extraction, in degrees. This is calculated from the `extraction_radius_pix` parameter and the image metadata during the *persistence pipeline stage*.

x, y and z The Cartesian coordinates of `centre_ra` and `centre_decl`.

temprunningcatalog (See also *source association detailed logic*.)

Most of the entries in the `temprunningcatalog` are identical to those of the same name in *runningcatalog* and *runningcatalog_flux*, except updated to include the information from a new `extractedsource`. Those without direct counterparts in those tables are listed below.

runcat Reference to the `runningcatalog id`. `runcat` and `xtrsrc` together form a unique key.

xtrsrc Reference to the `extractedsource id`. `runcat` and `xtrsrc` together form a unique combination.

distance_arcsec The distance in arcsec on the sky of the `runcat` - `xtrsrc` association, as calculated by the database.

r The De Ruiter radius of the `runcat` - `xtrsrc` association, calculated by the database.

inactive During evaluation of the association pairs, some pairs might be set to inactive (`TRUE`), defaults to `FALSE`.

beam_semimaj, beam_semimin, beam_pa Not currently used.

version This table contains the current schema version of the database. Every schema upgrade will increment the value by 1.

name The name of the version.

value The version number, which increments after every database change.

Appendices

On iteratively updated weighted means We now take a diversion to note the mechanics of storing and updating weighted means - this happens a lot in the database.

We define the average (specifically, the *arithmetic mean*) of x as

$$\overline{x}_N = \frac{1}{N}\sum_{i=1}^{N} x_i$$

where x_i is the i th measurement of x.

We may update this in an iterative fashion. If we add the next datapoint, x_{N+1}, to it, we can build the new average as:

$$\overline{x}_{N+1} = \frac{N\overline{x}_N + x_{N+1}}{N+1}. \tag{1.1}$$

We now treat weighted means.

We first define the weight of the i th measurement of x,

$$w_{x_i} = 1/{e_{x_i}}^2$$

where e_{x_i} is the one-sigma error in the i th measurement of x.

We can now define a weighted mean of N measurements of x; ξ_{x_N} as:

$$\xi_{x_N} = \frac{\sum_{i=1}^{N} w_{x_i} x_i}{\sum_{i=1}^{N} w_{x_i}}.$$

To update this weighted average, we first define the sum of the weights as

$$W_{x_N} = \sum_{i=1}^{N} w_{x_i}$$

we may then calculate the weighted average after N+1 measurements as:

$$\xi_{x_{N+1}} = \frac{W_{x_N}\xi_{x_N} + w_{x_{N+1}}x_{N+1}}{W_{x_N} + w_{x_{N+1}}} \tag{1.2}$$

Note, if we define the mean or 'bar' operator such that:

$$\overline{y}_N = \frac{\sum_{i=1}^{N} y_i}{N}$$

for any variable y, then

$$\overline{w}_{x_N} = \frac{\sum_{i=1}^{N} w_{x_i}}{N} = \frac{W_{x_N}}{N}$$

and we may use the formula:

$$\xi_{x_{N+1}} = \frac{N\overline{w}_{x_N}\xi_{x_N} + w_{x_{N+1}}x_{N+1}}{N\overline{w}_{x_N} + w_{x_{N+1}}} \tag{1.3}$$

(Note how this simplifies if $w_i = 1 \quad \forall i$)

Warning: For tracking Ra and Dec (α and δ) weighted means, we substitute

$$N\overline{w_{\alpha_N}}\xi_{\alpha_N} = N\overline{(w_\alpha \alpha)_N}$$

to yield another manipulation of the update formula:

$$\xi_{\alpha_{N+1}} = \frac{N\overline{(w_\alpha \alpha)_N} + w_{\alpha_{N+1}}\alpha_{N+1}}{N\overline{w}_{\alpha_N} + w_{\alpha_{N+1}}} \tag{1.4}$$

Note that this requires that we also keep track of the extra aggregate value: $\overline{(w_\alpha \alpha)_N}$, which is probably unnecessary given that we are not performing reduced-χ^2 stats on the position.

In general, we perform similar tricks with aggregate values (i.e. storing the 'barred' values of variables) throughout the database code. This has pros and cons - it makes the equations below a little prettier (and possibly simpler to compute), but requires many multiplications and divisions by the factor N (hence, also possibly harder to compute - this may be worth careful consideration during the next big code review).

On 'aggregated' variability indexes We now explain how running averages are used to compute the 'variability indices' we use in identifying sources which may be intrinsically transient or variable. Adapted from *Scheers (2011)*.

The first variability indicator, the proportional flux variability of a source, is expressed as the ratio of the sample standard deviation, and mean, of the flux I; that is to say:

$$V = \frac{s}{\overline{I}}$$

where s is the unbiased sample standard deviation:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}\left(I_i - \overline{I}\right)^2}$$

Note: In general, we may consider calculating all these values per frequency-band and subscript them by band central frequency ν, but we neglect such details here for simplicity.

Written in its well known 'aggregate' form, it is now easy to handle bulk data, and is defined as

$$V = \frac{1}{\overline{I}}\sqrt{\frac{N}{N-1}\left(\overline{I^2} - \overline{I}^2\right)}$$

The second indicator, the significance of the flux variability, is based on reduced χ^2 statistics. We derive the aggregate form here.

We begin with the familiar reduced-χ^2 formula, except with the regular arithmetic mean $\overline{I}$ replaced by the weighted mean ξ_{I_N},

$$\xi_{I_N} = \frac{\sum_{i=1}^{N} w_i I_i}{\sum_{i=1}^{N} w_i} = \frac{\overline{w_i I_i}}{\overline{w_i}},$$

resulting in:

$$\eta = \frac{1}{N-1}\sum_{i=1}^{N}\frac{\left(I_i - \xi_{I_N}\right)^2}{e_i^2}$$

where e_i is the estimated uncertainty, or standard deviation, in I_i. We may rewrite this using $\frac{1}{e_i^2} = w_i$:

$$\eta = \frac{N}{N-1}\left(\frac{1}{N}\sum_{i=1}^{N} w_i \left(I_i - \xi_{I_N}\right)^2\right)$$

Expanding inside the brackets gives:

$$\begin{aligned} &\frac{1}{N}\sum_{i=1}^{N} w_i \left(I_i^2 - 2\xi_{I_N} I_i + \xi_{I_N}^2\right) \\ &= \frac{1}{N}\sum_{i=1}^{N} w_i I_i^2 - 2\xi_{I_N}\frac{1}{N}\sum_{i=1}^{N} w_i I_i + \xi_{I_N}^2 \frac{1}{N}\sum_{i=1}^{N} w_i \\ &= \overline{w_i I_i^2} - 2\xi_{I_N}\overline{w_i I_i} + \xi_{I_N}^2\overline{w_i} \qquad . \end{aligned}$$

Expanding for ξ_{I_N} results in the final aggregate form of the reduced-χ^2:

$$\eta = \frac{N}{N-1}\left(\overline{wI^2} - \frac{\overline{wI}^2}{\overline{w}}\right)$$

Source Association Logic

Source association—the process by which individual measurements recorded from an image corresponding to a given time, frequency and Stokes parameter are combined to form a lightcurve representing an astronomical source—is fundamental to the goals of the Transients KSP. However, the process is complex and may be counterintuitive. A thorough understanding is essential both to enable end-users to interpret pipeline results and to inform pipeline design decisions.

Here, we summarise the various possible results of source association, highlighting potential issues. For a full discussion of the algorithms involved, the user is referred to *Scheers (2011)*.

Database Structure & Association Procedure

The structure of the database is discussed in detail *elsewhere*: here, only a brief overview of the relevant tables is presented.

Each measurement (that is, a set of coordinates, a shape, and a flux) taken is inserted into the `extractedsource` table. Many such measurements may be taken from a single image, either due to "blind" source finding (that is, automatically attempting to locate islands of significant bright pixels), or by a user-requested fit to a specific position.

The association procedure knits together ("associates") the measurements in `extractedsource` which are believed to originate from a single astronomical source. Each such source is given an entry in the `runningcatalog` table which ties together all of the measurements by means of the `assocxtrsource` table. Thus, an entry in `runningcatalog` can be thought of as a reference to the lightcurve of a particular source.

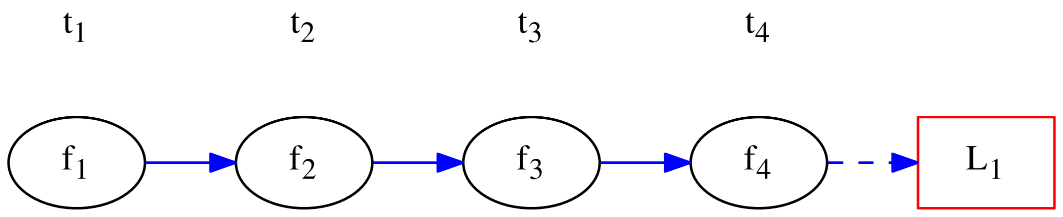
Each lightcurve may be composed of measurements in one or more frequency bands (as defined in the `frequencyband` table). Within each band flux measurements are collated. These include the average flux of the source in that band, as well as assorted measures of variability. Each row in the `runningcatalog_flux` table contains flux statistics of this sort for a given band of a given flux. Thus, each row in `runningcatalog` may be associated with both multiple rows in `extractedsource` and in `runningcatalog_flux`. Bear in mind, however, that each lightcurve has a *single* average position associated with it, stored in the `runningcatalog` table.

When a new source measurement is made, the association procedure compares it against the records in `runningcatalog`. Note that the comparison is based upon position *only*: an association is made if the new measurement is a good fit with a position in `runningcatalog`, regardless of the time or frequency associated with it. After the association has been made, the position recorded for the source in `runningcatalog` is updated to take account of the new measurement.

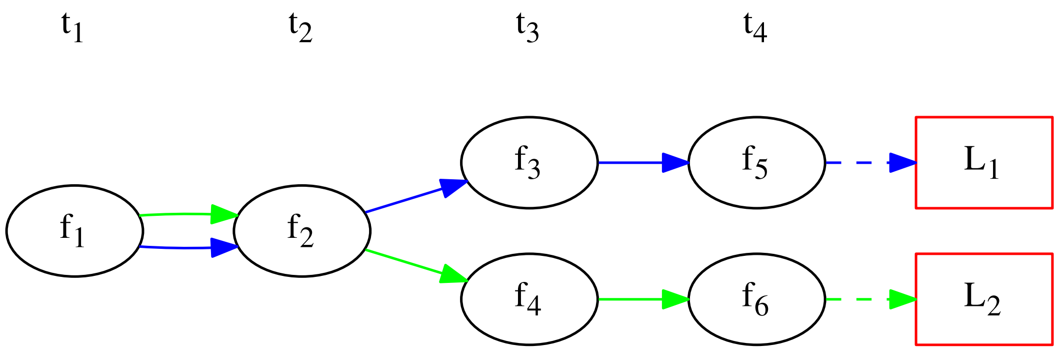
It is important to note that source association must as far as possible be *commutative*. That is, given a set of measurements, the final contents of the database should not be dependent upon the order of their insertion. This is not possible in the general case—it would involve a quadratic number of source comparisons—but source association procedures should be designed with this goal in mind. In particular, we require that source association be commutative if all measurements made at time t_n are inserted and associated before any measurements made at time t_{n+1}.

Case Studies Here we will discuss the various outcomes which are possible from the source association process under different conditions. In the following, individual timesteps are indicated by the notation t_i and individual flux measurements (that is, at a particular time/band/Stokes) by f_j. Lightcurves (entries in `runningcatalog`) are indicated by L_k; the flux measurements which constitute a particular lightcurve are linked to the L_k symbol by means of a coloured line.

Single Frequency Band We start by considering observations with only a single frequency band.

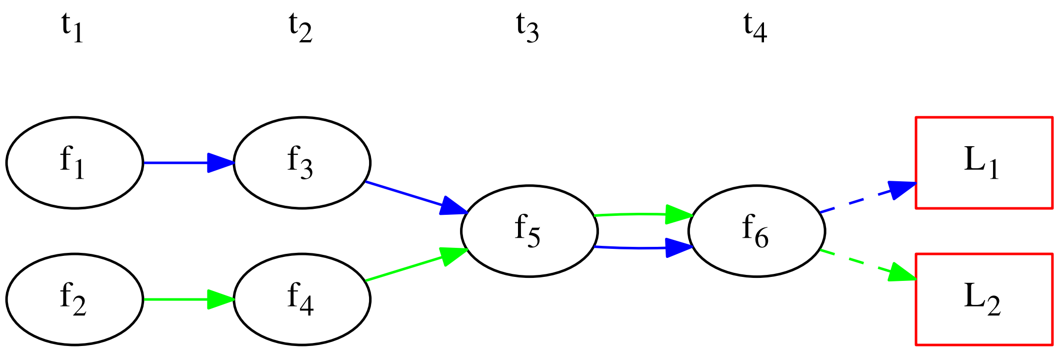


One-to-One Association In this simplest case all the flux measurements are unambiguously associated in order. A single lightcurve is generated. The calculated average flux of the lightcurve L_1 is $\overline{f_{1\cdots4}}$.



One-to-Many Association Here, both f_3 and f_4 can be associated with the lightcurve containing f_2 and f_1: a "one-to-many" association. Since f_3 and f_4 are distinct, though, they result in *two* entries in the `runningcatalog` table, or, equivalently, two lightcurves: L_1 with average flux $\overline{f_{1,2,3,5}}$ and L_2 with average flux $\overline{f_{1,2,4,6}}$.

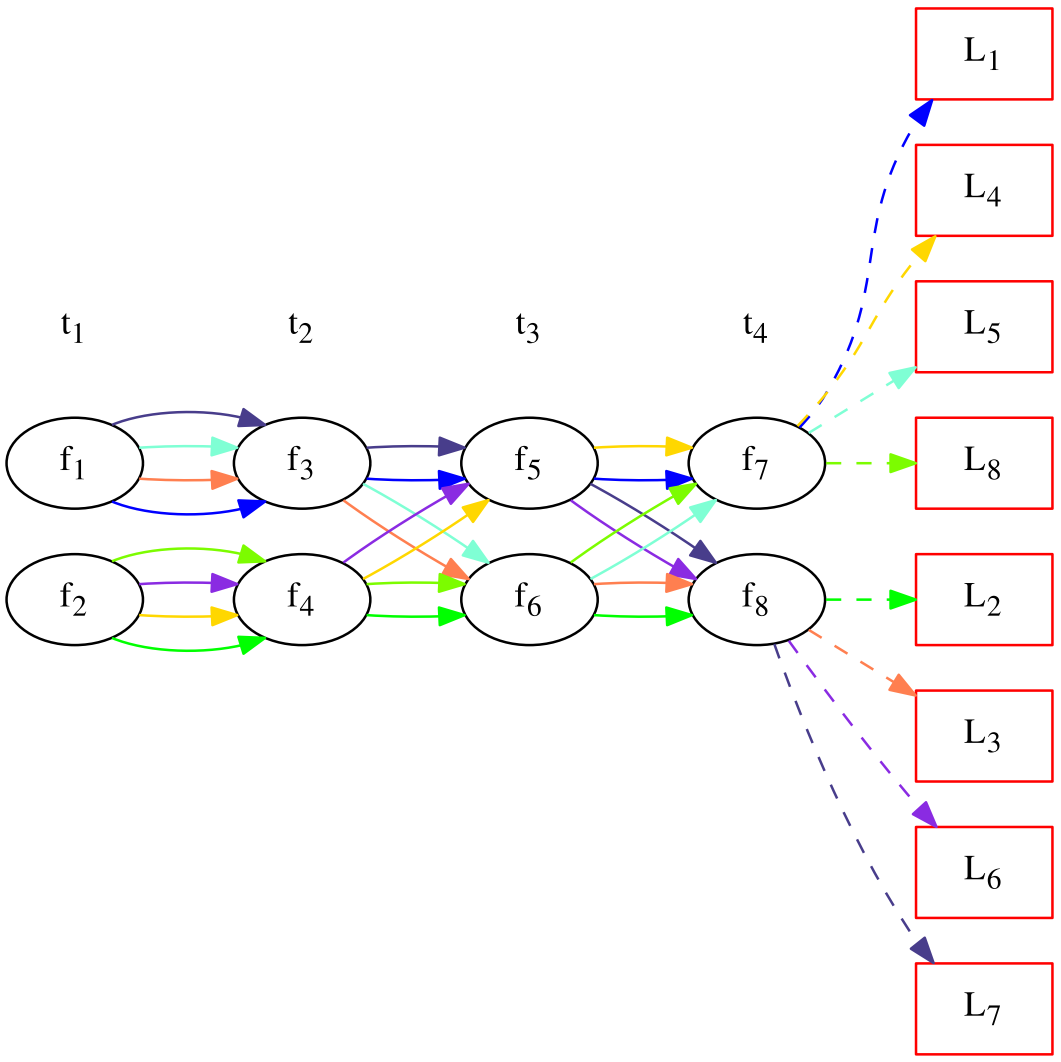
Note that f_1 and f_2: are now being counted *twice*. Even if f_3 and f_4 each contribute only half the total flux of f_2, the total brightness reached by summing all the lightcurve fluxes *increases* when this occurs. Equivalently, increasing the spatial resolution of the telescope causes the sky to get brighter!



Many-to-One Association This situation is similar to that seen above, but in reverse. Initially, two lightcurves are seen L_1 consisting of f_1 and f_3 and L_2 consisting of f_2 and f_4. However, at timestep t_3 a new measurement is made, f_5, which is associated with both L_1 and L_2. This, and the subsequent measurement f_6, are then appended to both lightcurves, resulting in L_1 having average flux $\overline{f_{1,3,5,6}}$ and L_2 having average flux $\overline{f_{2,4,5,6}}$. Again, note that f_5 and f_6 are counted twice.

Many-to-Many Association

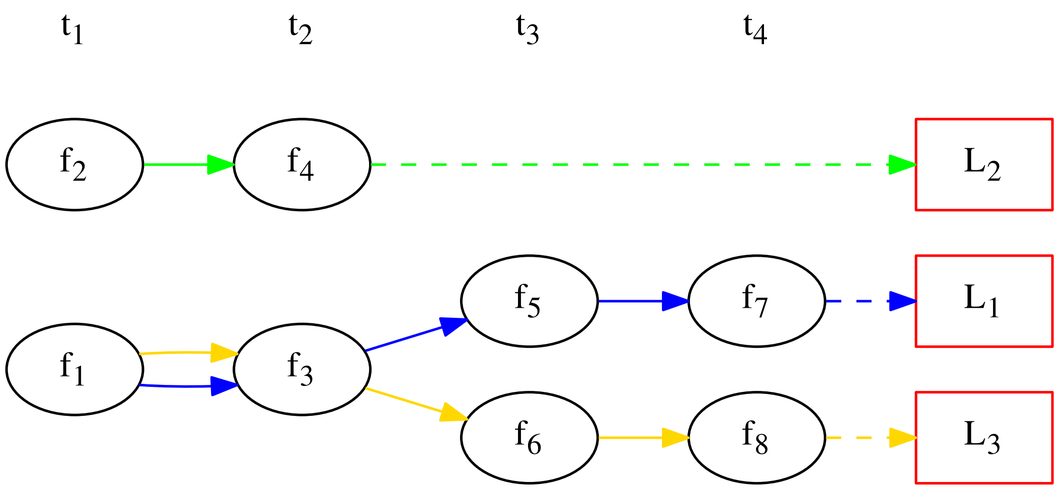
Note: First we illustrate "true" many-to-many association. However, for reasons that will become obvious, this is never actually performed: instead, we reduce it to a simpler, one-to-one or one-to-many association.



As shown above, many-to-many association grows quadratically in complexity, as every possible combination of sources involved in the association results in a new lightcurve. Further, assuming that neither the sky nor the telescope configuration change significantly from observation to observation, it's likely that subsequent measurements will also result in many-to-many associations, doubling the number of lightcurves at every timestep.

It should be obvious that the scenario described is untenable. Instead, all many-to-many associations are automatically reduced by only taking the source pairs with the smallest De Ruiter radii such that they become either one-to-one or one-to-many associations.

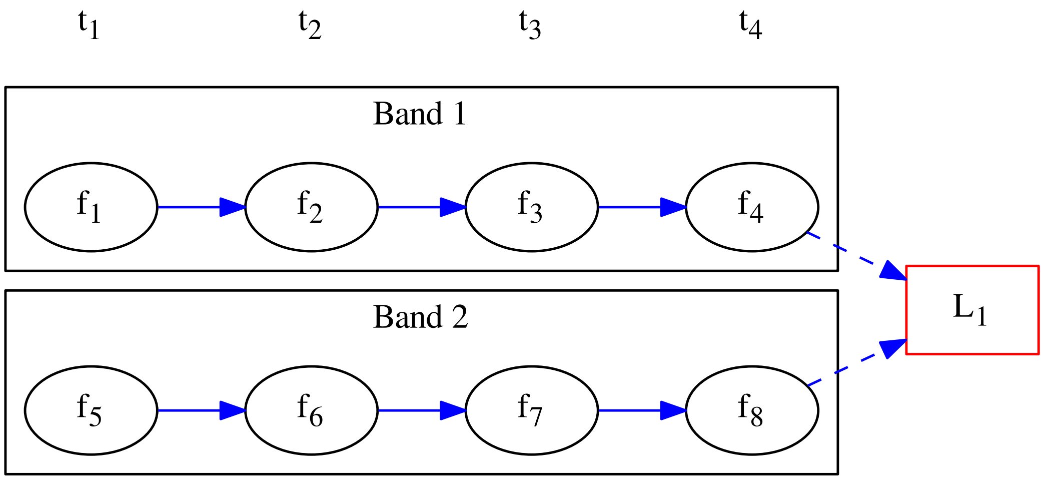
For example, using this criterion, both f_5 and f_6 might be associated with a lightcurve consisting of f_1 and f_3 in the above. The following situation results:



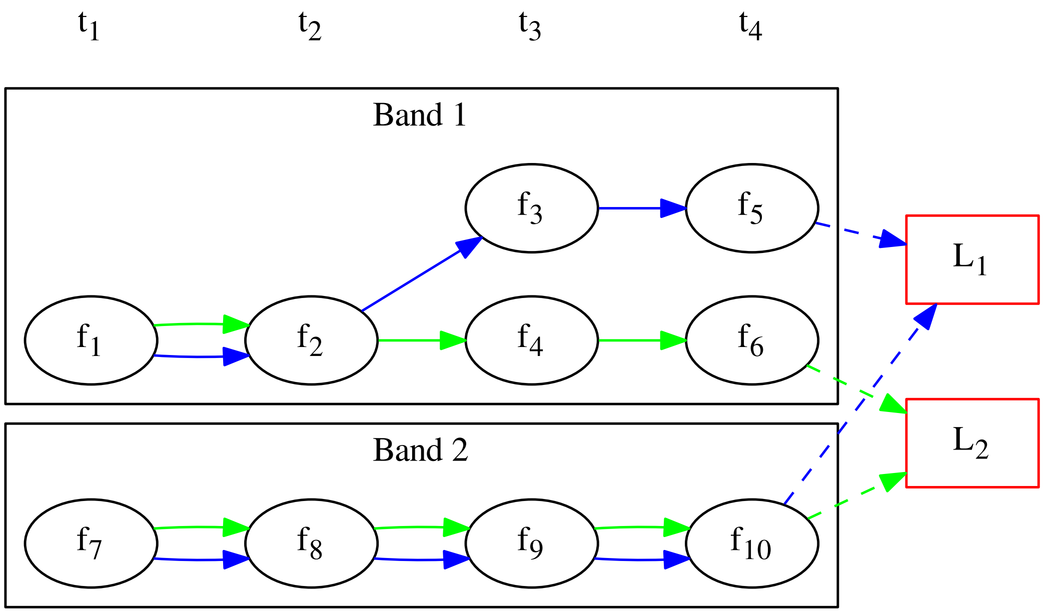
Note that L_2 contains no measurements for timesteps later than t_2: the many-to-many association is removed, but at the cost of truncating this lightcurve.

Multiple Frequency Bands We now introduce the added complexity of multiple bands: the same part of the sky being observed at the same time, but at different frequencies. Here, we use just two bands for illustration, but in practice several could be involved.

When considering multiple frequency bands, the same association procedure, based only on position, as described above, is employed. However, extra care must be taken to ensure that the commutative nature of association is preserved.



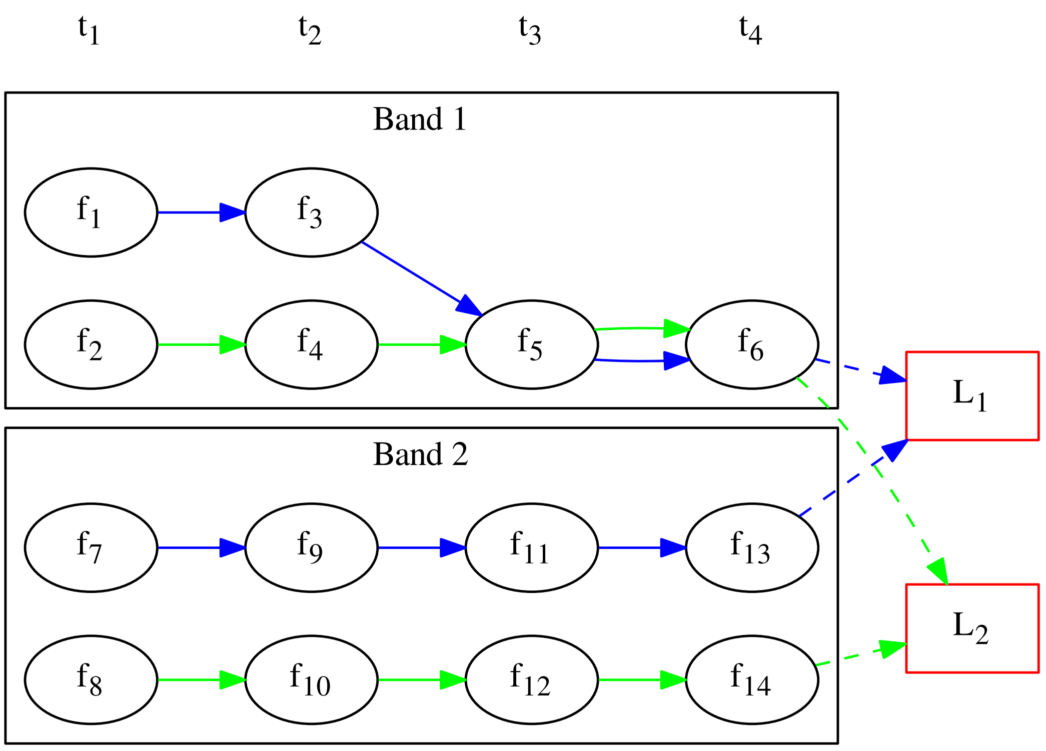
Multi-Band One-to-One Association In the simplest case, a one-to-one association is made between each measurement and an entry in the `runningcatalog` table. A single lightcurve results, which we label L_1, but for which two average fluxes are calculated: $\overline{f_{1 \cdots 4}}$ in band 1 and $\overline{f_{5 \cdots 8}}$ in band 2.



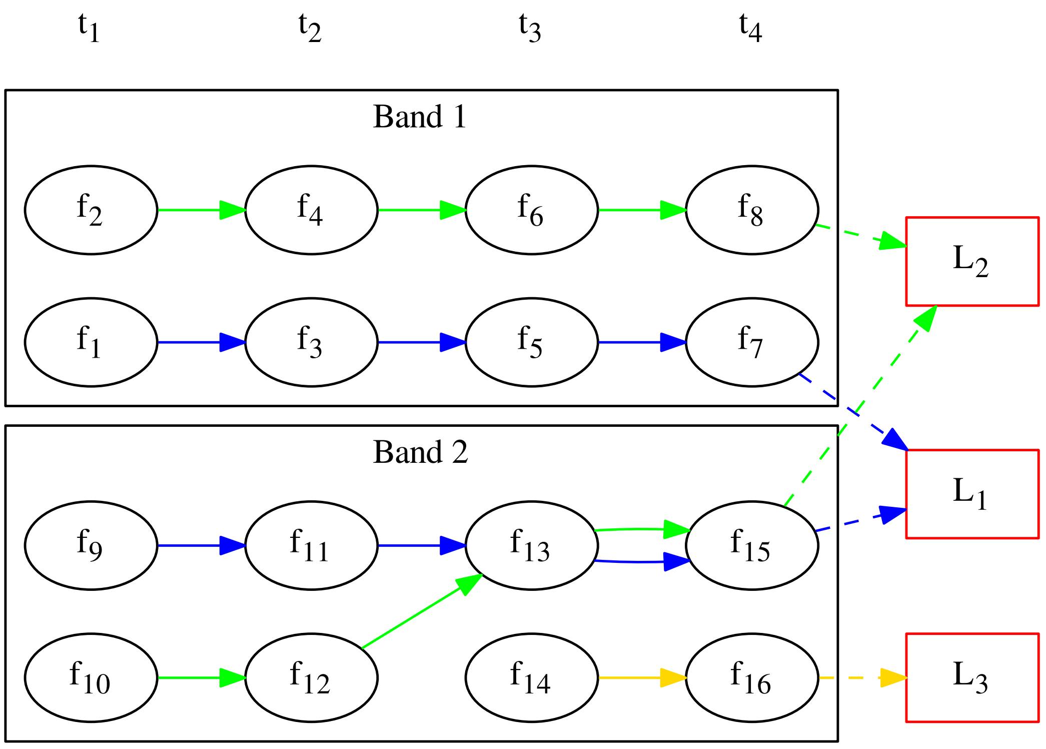
Multi-Band One-to-Many Association Initially, we proceed as above. However, at t_3, a one-to-many association takes place in Band 1. That band therefore bifurcates, and we are left with two lightcurves: L_1 and L_2.

No such bifurcation is seen in Band 2. The single measurement f_9 may be associated with one or both of L_1 and L_2, depending on their relative positions. In the former case, one of the lightcurves is truncated in Band 2. In the latter, a chain of one-to-many associations takes place with measurements in this band, as both f_9 and f_{10} are associated with both lightcurves.

In the situation shown, the resulting average fluxes for L_1 are $\overline{f_{1,2,3,5}}$ in Band 1 and $\overline{f_{7\cdots10}}$ in Band 2, while those for L_2 are $\overline{f_{1,2,4,6}}$ and $\overline{f_{7\cdots10}}$ respectively. Note that the entire flux in Band 2, as well as f_1 and f_2, is now counted twice.



Multi-Band Many-to-One Association At first, L_1 and L_2 are completely independent. However, at t_3, f_5 undergoes a many-to-one association with both of them. The same applies to f_6. In Band 2, the lightcurves remain independent. L_1 therefore has average fluxes $\overline{f_{1,3,5,6}}$ in Band 1 and $\overline{f_{7,9,11,13}}$ in Band 2, and L_2 has average fluxes $\overline{f_{2,4,5,6}}$ in Band 1 and $\overline{f_{8,10,12,14}}$ in Band 2.



Multi-Band Many-to-One Association (2) In this case, we initially have two separate lightcurves. However, at t_3, f_{13} is associated with both lightcurves in Band 2, while f_{14} is associated with neither. Three lightcurves result, as shown.

It is worth considering the ordering of database insertion at this point. In particular, consider that either one of f_6 and f_{14} may be inserted before the other. After each insertion, the average position of the `runningcatalog` entry is recalculated, and this may affect future associations.

For example, assume that f_6 is inserted before f_{14}. In this case, the average position of $f_{2,4,6,10,12}$ is not associated with f_{14}. However, if f_{14} were to be inserted first, it would be compared for association with the average position of $f_{2,4,10,12}$. This may well produce a different result!

It is desirable for the database contents to be independent of the order of insertion, since this is an arbitrary choice which should not effect scientifically significant outcomes. However, in the current implementation we simply settle for a *deterministic and repeatable* arbitrary choice, by ingesting images in order of band frequency.

Discussion

It is immediately obvious from the examples given above that, in all but the simplest cases, there is potential for confusion here. In particular, note that simply summing the average fluxes of all the lightcurves in the `runningcatalog_flux` table in a given band is not an appropriate way to estimate the total brightness of the sky: this may count individual flux measurements multiple times.

Further, the way the source association is handled may result in false detections of transients. In the case of a one-to-many association, for example, a single bright source can be associated with two sources each of a fraction of the brightness. This results in two lightcurves, both containing a (very transient like!) sudden step in flux. A similar outcome can, of course, also result from a many-to-one association.

There are two potential areas of improvement which should be investigated.

Flux division

In a one-to-many or many-to-one association, rather than simply allocating the full flux of the "one" measurement to each of the "many" lightcurves, it could be split such that each was only allotted a portion of the total. In this way, the total brightness of the sky could be maintained.

The most appropriate division is not obvious. A simple model could allocate each of n lightcurves a fraction $1/n$ of the total flux of the single measurement. A more elaborate procedure would weight the allocation by the flux in each of the n lightcurves, such that brighter sources are allocated a larger fraction of the flux.

Whatever flux allocation procedure is adopted, however, involves making assumptions about what fraction should be allocated to each source. Further, it may also increase the computational complexity in the database, as lightcurve statistics are no longer simply calculated over source measurements, but must also take account of fractional allocations.

Smarter association

The current association procedure is purely based on the positions of the sources and their uncertainties. By incorporating more information about the sources, ambiguities in association could often be avoided.

For example, consider the case of a many-to-many association involving an extended source and a point source. It is likely perfectly reasonable to assume that the measurement of the extended source at time t_2 should only be associated with the extended source at time t_1, and similarly for the point source: in this way, the many-to-many association can be easily reduced to a much simpler case.

Again, though, a number of assumptions go into any procedure like this. In particular, given that our ultimate aim is to detect transient and variable sources, we should be wary of any procedure that implicitly assumes the sky is unchanging. Further, again the issue of database complexity should be considered: incorporating more logic of this sort is expensive, in terms of both compute and developer time.

Recommendations

Although it is clear that improvements can and will need to be made to the procedures adopted, it is not immediately obvious how best to proceed. Therefore, it is suggested that refinements be deferred until more practical experience has been obtained.

To that end, we suggest the following:

1. Commissioners and scientists working with the lightcurve database, as well as developers of tools designed to detect transients based upon it, must familiarize themselves with the issues described above.
2. The TKP Lightcurve Archive should be explicit about which measurements have gone into a displayed lightcurve or other measurement. The figures which accompany this document are easy to programmatically generate using GraphViz, and show clearly the heritage of a given lightcurve; we suggest, therefore, that they or a derivative of them should be shown on the website.
3. As more source measurements are collected, statistics can be collected to demonstrate to what extent the problems anticipated are observed in real-world use. For example, in the ideal case, the total number of measurements included in all the lightcurves would be equal to the number of measurements made on images; in practice,

however, the former will be bigger, since measurements may be counted twice. Observing the "overcounting fraction" as the database grows will help understand the nature and severity of the problem.

Detailed logic flow

Herein we give an algorithmic description of how the source association routines work.

Warning: The following detail is really aimed at developers or particularly interested users only, and can certainly be skipped on first reading.

We assume that source extraction has been run on input images, and new measurements have been inserted into the `extractedsource` table.

Clean any previously created temporary listings. To ensure a clean start, we first run `_empty_temprunningcatalog`, which does what it says on the tin.

Generate a list of candidate runningcatalog-extractedsource associations Performed by: *`tkp.db.associations._insert_temprunningcatalog()`*

(See also: *temprunningcatalog* table.)

This function generates a temporary table listing possible associations with previously catalogued sources.

For a given `image_id`:

- Select all the relevant extractedsource entries, and
- For each extractedsource, create a bunch of table entries detailing candidate associations with runningcatalog entries which are:
  - In the same declination zone as the extractedsource
  - Have a weighted mean position for which the RA and DEC are within a box of half-width `radius` degrees from the extractedsource. (This places a hard limit on the maximum association radius).
  - Have a weighted mean position within a user-specified De Ruiter radius of the extractedsource.
- Each of these rows representing a candidate association is populated with all the values which would represent an update to the corresponding runningcatalog and runningcatalog_flux entries, if the association is later determined to be definitive.

Trim the 'many-to-many' links to prevent exponentional database growth Performed by: *`tkp.db.associations._flag_many_to_many_tempruncat()`*

Especially if we employ a large De Ruiter radius limit, we may generate a large number of candidate associations which result in a complex web of possible lightcurves. We reduce this to a more manageable situation by trimming some of the 'weaker' candidate associations.

First, inspect the temprunningcatalog table:

- Select entries for which the extractedsource is listed more than once.
- Of these entries, select those for which the runcat id is listed more than once in temprunningcatalog.
- Use this selection to determine the runningcatalog id of minimum De Ruiter radius, for each extracted source which is part of a many-to-many set.

- Then, using this per-extractedsource minimum DR radius, reapply the above filters to select multiply-associated entries, and select all entries for which the runcat id has a larger than minimum DR radius to the extractedsource.
- Return the runcat-extractedsource identifying pair values for all non-optimal entries in many-to-many sets.

Finally, use these identifiers to set all these entries as `inactive = TRUE`.

Or, in pseudo-mathematical terms, tempruncat describes the edges of a graph, linking nodes (sources) from two spaces (previous runcat entries, newly extracted entries). (There are no intra-space links). *`_flag_many_to_many_tempruncat()`* trims this graph using the De Ruiter radius as a ranking metric, to ensure that any connected sub-graph has multiple nodes in *at most* one of the two spaces.

Deal with the 'one-to-many' runcat-to-extractedsource link sub-graphs When we observe two new sources in the region of a previous known source, it is unclear if this is due to increased resolution, or a new source. To resolve this, we hedge our bets and replace the old single runcat entry with two new entries - these are identical up to the current 'fork'. This is done in *`tkp.db.associations._insert_1_to_many_runcat()`*, and *`tkp.db.associations._flag_1_to_many_inactive_runcat()`* then flags the old entries as ready for deletion.

Having inserted these new runningcatalog entries, we must copy over all the relevant information to new entries in the associated tables, then delete the outdated rows; see

- *`tkp.db.associations._insert_1_to_many_runcat_flux()`*
- *`tkp.db.associations._delete_1_to_many_inactive_runcat_flux()`*
- *`tkp.db.associations._insert_1_to_many_basepoint_assocxtrsource()`*
- *`tkp.db.associations._insert_1_to_many_replacement_assocxtrsource()`*
- *`tkp.db.associations._delete_1_to_many_inactive_assocxtrsource()`*
- *`tkp.db.associations._insert_1_to_many_assocskyrgn()`*
- *`tkp.db.associations._delete_1_to_many_inactive_assocskyrgn()`*
- *`tkp.db.associations._insert_1_to_many_newsource()`*
- *`tkp.db.associations._delete_1_to_many_inactive_newsource()`*

Finally, *`tkp.db.associations._flag_1_to_many_inactive_tempruncat()`* flags the one-to-many associations in `temprunningcatalog` as inactive, so we can easily distinguish remaining one-to-one associations.

Process all remaining associations Performed by:

- *`tkp.db.associations._insert_1_to_1_assoc()`*
- *`tkp.db.associations._update_1_to_1_runcat()`*
- *`tkp.db.associations._update_1_to_1_runcat_flux()`*
- *`tkp.db.associations._insert_1_to_1_runcat_flux()`*

We now process all the remaining active associations listed in temprunningcatalog.

`_insert_1_to_1_assoc()` Inserts all the remaining active links listed in tempruncat, into assocxtrsource. These links all refer to a still-valid runningcatalog entry from a previous source association run. (This actually includes those candidate links in 'many-to-one' sets, e.g. sources merged due to a lower-resolution image - hence we set `type = 3`).

`_update_1_to_1_runcat()` then performs the corresponding update on the runningcatalog table, copying across the values calculated during the generation of temprunningcatalog.

_update_1_to_1_runcat_flux() grabs all the columns relevant to the runnincatalog_flux entries, from the still active entries in temprunningcatalog, and updates the `runningcatalog_flux` table accordingly.

Process remaining extractedsources (those without associations) Performed by:

- *tkp.db.associations._insert_new_runcat()*
- *tkp.db.associations._insert_new_runcat_flux()*
- *tkp.db.associations._insert_new_runcat_skyrgn_assocs()*
- *tkp.db.associations._insert_new_assocxtrsource()*

We still need to insert the 'new' sources, i.e. those extractions without an identified association.

_insert_new_runcat() is run first, since the database constraints are already satisfied (pre-existent xtrsrc and dataset-id). First, we pre-select those extractedsources which were discovered in the current image. Then we filter to just those which do not have any associations, by selecting those extractedsources listed in the image but not in the temprunningcatalog (A left outer join on xtrsrc where temprunningcatalog.xtrsrc is NULL).

We initialise the averages (position, flux, etc) by pulling in the relevant values from extractedsource, and the dataset id from the image table.

_insert_new_runcat_flux() performs a similar trick to select the 'new-source' extractsources, then cross-matches against the xtrsrc id to select the new runcat entries. With these in hand it's easy to insert new runcat_flux entries, pulling in the relevant id from runningcatalog, band and stokes from image table, and flux values from extractedsource.

_insert_new_runcat_skyrgn_assocs() performs a positional check against all known skyregions to see which regions this source lies within, and inserts links in the `assocskyrgn` table accordingly.

_insert_new_assocxtrsource() Performs the same routine of grab 'new-source' entries, match new runcat entries, as *_insert_new_runcat_flux()* - it's then trival to insert the relevant entries in assocxtrsource. These are then marked as a `type = 4` association.

Determine if a new source is a likely transient Performed by *tkp.db.associations._determine_newsource_previou*

Cleanup Performed by:

- *tkp.db.associations._empty_temprunningcatalog()*
- *tkp.db.associations._delete_inactive_runcat()*

Now that all the new extractions have been dealt with, we take care of some loose ends. We delete all rows from the `temprunningcatalog` table, and finally delete those runningcatalog entries which we have now superceded, via a simple `inactive = TRUE` filter.

"How-To" notes on common tasks

Recover from disappearing clients

Current (as of April 2014) versions of the MonetDB server do not check for clients timing out. That is, if a remote client connects to the server and opens a transaction, then dies without shutting down cleanly for some reason (power failure, network glitch, ...), the transaction will remain open indefinitely. MonetDB logs all incremental updates during the transaction. Eventually, this will both cripple performance and take a huge amount of space.

Recover by stopping and restarting the database:

```
$ monetdb stop ${database}
stoping database '${database}'... done
```

Start it again and the logs will be replayed and then removed:

```
$ monetdb start ${database}
starting database '${database}'... done
```

Note that this procedure may take a long time (several hours) to replay a large volume of logs.

Future versions of MonetDB (unreleased as of April 2014) will include a fix for this issue by enabling `SO_KEEPALIVE` on the TCP socket.

Create a schema diagram

As shown in the *schema documentation*.

documentation/devref/database/schema in the TKP tree contains a mini django projects that can be used to update the schema image. See the README file in the directory for instructions how to use and set up the django project.

1.4.3 TKP Package API Reference

Subpackages

`tkp.accessors` – Image container classes

Introduction The "accessors" system attempts to abstract away the physical image storage format from the library logic. The higher-level data access routines should not care whether the data is stored in a CASA table, or a FITS file, or in some other format: they should be coded against a uniform interface provided by the appropriate accessor.

An accessor should subclass *`tkp.accessors.dataaccessor.DataAccessor`*, which is an abstract base class. Accessors *must* conform to the DataAccessor interface by defining the relevant attributes (see the attributes listed under *`tkp.accessors.dataaccessor.DataAccessor`* for full details). An accessor which provides all of these attributes is guaranteed to be usable with all core TraP functionality.

In some cases, most notably *quality control*, specialized (eg, per-telescope) metadata may also be required. A further abstract base class should be constructed to define the interface required. For example, *`tkp.accessors.lofaraccessor.LofarAccessor`* defines the following metadata which must be provided for LOFAR quality control:

`antenna_set` Antenna set in use during observation. String; `LBA_INNER`, `LBA_OUTER`, `LBA_SPARSE`, `LBA` or `HBA`

`ncore` Number of core stations in use during observations constituting this image. Integer.

`nremote` Number of remote stations in use during observations constituting this image. Integer.

`nintl` Number of international stations in use during observations constituting this image. Integer.

`subbandwidth` Width of a subband, in Hz.

`subbands` Number of subbands.

API Documentation

`tkp.accessors` API reference

`tkp.accessors` – Base data accessor utilities Data accessors.

These can be used to populate ImageData objects based on some data source (FITS file, array in memory... etc).

`tkp.accessors.`**`open`**(*path*, **args*, ***kwargs*)
Returns an accessor object (if available) for the file or directory 'path'.

We try all the possible accessors in order from most specific to least specific. That is, if possible, we prefer an accessor providing LofarAccessor to one providing DataAccessor, but we accept the latter if that's the only possible match.

Will raise an exception if something went wrong or no matching accessor class is found.

`tkp.accessors.`**`sourcefinder_image_from_accessor`**(*image*, ***args*)
Create a source finder ImageData object from an image 'accessor'

Parameters **`image`** (-) – FITS/AIPS/HDF5 image available through an accessor.

Returns a source finder image.

Return type (*`tkp.sourcefinder.image.ImageData`*)

`tkp.accessors.`**`writefits`**(*data*, *filename*, *header={}*)
Dump a NumPy array to a FITS file.

Key/value pairs for the FITS header can be supplied in the optional header argument as a dictionary.

`tkp.accessors.detection` **– File type detection**
class `tkp.accessors.detection.`**`FitsTest`**(*accessor*, *test*)

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

`__repr__`()
Return a nicely formatted representation string

`accessor`
Alias for field number 0

`test`
Alias for field number 1

`tkp.accessors.detection.`**`casa_detect`**(*filename*)
Detect which telescope produced CASA data, return corresponding accessor.

Checks for known CASA table types where we expect additional metadata. If the telescope is unknown we return nothing.

`tkp.accessors.detection.`**`detect`**(*filename*)
returns the accessor class that should be used to process filename

`tkp.accessors.detection.`**`fits_detect`**(*filename*)
Detect which telescope produced FITS data, return corresponding accessor.

Checks for known FITS image types where we expect additional metadata. If the telescope is unknown we default to a regular FitsImage.

`tkp.accessors.detection.`**`iscasa`**(*filename*)
returns True if filename is a lofar casa directory

`tkp.accessors.detection.`**`isfits`**(*filename*)
returns True if filename is a fits file

`tkp.accessors.detection.`**`islofarhdf5`**(*filename*)
returns True if filename is a hdf5 container

Data Accessor Variants

Basic data accessors

class `tkp.accessors.dataaccessor.`**`DataAccessor`**
Base class for accessors used with `tkp.sourcefinder.image.ImageData`.

Data accessors provide a uniform way for the ImageData class (ie, generic image representation) to access the various ways in which images may be stored (FITS files, arrays in memory, potentially HDF5, etc).

This class cannot be instantiated directly, but should be subclassed and the abstract properties provided. Note that all abstract properties are required to provide a valid accessor.

Additional properties may also be provided by subclasses. However, TraP components are required to degrade gracefully in the absence of this optional properties.

The required attributes are as follows:

`beam`
tuple – Restoring beam. Tuple of three floats: semi-major axis (in pixels), semi-minor axis (pixels) and position angle (radians).

`centre_ra`
float – Right ascension at the central pixel of the image. Units of J2000 decimal degrees.

`centre_decl`
float – Declination at the central pixel of the image. Units of J2000 decimal degrees.

`data`
numpy.ndarray – Two dimensional numpy.ndarray of floating point pixel values. (TODO: Definitive statement on orientation/transposing.)

`freq_bw`
float – The frequency bandwidth of this image in Hz.

`freq_eff`
float – Effective frequency of the image in Hz. That is, the mean frequency of all the visibility data which comprises this image.

`pixelsize`
tuple – (x, y) tuple representing the size of a pixel along each axis in units of degrees.

`tau_time`
float – Total time on sky in seconds.

`taustart_ts`
float – Timestamp of the first integration which constitutes part of this image. MJD in seconds.

`url`
string – A (string) URL representing the location of the image at time of processing.

`wcs`
`tkp.utility.coordinates.WCS` – An instance of `tkp.utility.coordinates.WCS`, describing the mapping from data pixels to sky-coordinates.

The class also provides some common functionality: static methods used for parsing datafiles, and an 'extract_metadata' function which provides key info in a simple dict format.

static `degrees2pixels`(*bmaj*, *bmin*, *bpa*, *deltax*, *deltay*)
Convert beam in degrees to beam in pixels and radians. For example Fits beam parameters are in degrees.

Parameters

- **bmaj** (-) – Beam major axis in degrees
- **bmin** (-) – Beam minor axis in degrees
- **bpa** (-) – Beam position angle in degrees
- **deltax** (-) – Pixel size along the x axis in degrees
- **deltay** (-) – Pixel size along the y axis in degrees

Returns Beam semi-major axis in pixels - semimin: Beam semi-minor axis in pixels - theta: Beam position angle in radians

Return type

- semimaj

`extract_metadata`()
Massage the class attributes into a flat dictionary with database-friendly values.

While rather tedious, this is easy to serialize and store separately to the actual image data.

May be extended by subclasses to return additional data.

`parse_pixelsize`()

Returns pixel size along the x axis in degrees - deltay: pixel size along the x axis in degrees

Return type

- deltax

The following accessors are derived from the basic `DataAccessor` class:

class `tkp.accessors.fitsimage.FitsImage`
Generic FITS data access.

class `tkp.accessors.casaimage.CasaImage`
Generic CASA image access.

class `tkp.accessors.kat7casaimage.Kat7CasaImage`
KAT-7 specific CASA image access.

LOFAR-specific data accessors

class `tkp.accessors.lofaraccessor.LofarAccessor`

The following accessors are derived from the generic `LofarAccessor`:

class `tkp.accessors.lofarfitsimage.LofarFitsImage`
LOFAR FITS access.

class `tkp.accessors.lofarcasaimage.LofarCasaImage`
LOFAR CASA image access.

`tkp.config` – Pipeline Configuration

Outline The tkp.config folder contains the job configuration file templates, and code for handling config files.

API

`tkp.config.`**`get_database_config`**(*config_passed=None*, *apply=False*)
Determine database config and (optionally) use to set up the Database.

Determines a database configuration using the settings defined in a dict (if supplied) and possibly overridden by environment variables. The config resulting from the combination of defaults, supplied dict, and environment variables is returned as a dict. If apply==True, the Database singleton is configured using these resulting settings.

The following environment variables are recognized, and take priority:

- TKP_DBENGINE
- TKP_DBNAME
- TKP_DBUSER
- TKP_DBPASSWORD
- TKP_DBHOST
- TKP_DBPORT

Parameters

- **`config_passed`** – Dict of db settings. Relevant keys: (engine, database, user, password, host, port, passphrase)
- **`apply`** – apply settings (configure db connection) or not

Returns Dict containing the resulting combined settings (resulting from defaults, `config_passed` and possibly environment variables.)

`tkp.config.`**`initialize_pipeline_config`**(*pipe_cfg_file*, *job_name*)
Initializes the default variables and loads the ConfigParser file.

Sets defaults for start_time, job_name and cwd; these can then be used via variable substitution in other config values.

`tkp.config.parse` **– Magic ConfigParser to dict functionality** Utilities for loading parameters from config files, with automatic type conversion.

`tkp.config.parse.`**`loads_timestamp_w_microseconds`**(*dt_str*)
Loads and returns timestamp with microsecond precission

`tkp.config.parse.`**`parse_to_dict`**(*config*)
Loads the ConfigParser object as a nested dictionary.

Automatically converts strings representing ints and floats to their respective types, through the magic of ast.literal_eval. This functionality is extensible via the loads_methods list. Each loads (load string) method is tested in turn to see if it throws an exception. If all throw, we assume the value is meant to be string.

Any values that you don't want to be converted should simply be surrounded with quote marks in the parset - then ast.literal_eval knows to load it as a string.

Parameters **`config`** – A ConfigParser object.

Returns Nested dict {sections -> keys -> values } representing parsed params.

`tkp.db` – Database routines

Modules from the `tkp.db` package.

Introduction The database subpackage consists of several modules dedicated to their tasks. The main modules are:

- database
- general

Database The database module provides a single class, `tkp.db.database.Database`, which takes care of the connection to the database. It provides a *connection* and a *cursor* object to the database, as well as a few shortcut utility functions for executing SQL queries.

The most typical use for the `tkp.db.database.Database` class is something as follows, assuming the default login settings (taken from the tkp configuration file) are appropriate:

```
from tkp.db import execute as execute

cursor = execute(<sql query>, <args>)
results = cursor.fetchall()
```

For more details, see the *database* section.

General: dataset, image, extracted sources The dataset module provides a miniature object relation mapper (ORM) interface to the database. This interface is not fully complete, but it does allow one to treat several database tables and their data as Python classes and instances. The mapped tables and their classes are:

- datasets: DataSet
- images: Image
- extractedsource: ExtractedSource

Each of these classes inherits from the DBOject class. This class provides a few general methods to interface with the underlying database. A typical usage example could look like this:

```
db_image = DBImage(id=image_id, database=database)
db_image.insert_extracted_sources(results,'blind')
```

where *database* is the database opened above, and image_id points to an existing row in the images table. *results* is obtained from the source finder, and are stored in the database per image into extractedsource. In case of a new image, one would leave out the *id* keyword in the first line, and instead supply a *data* keyword argument that is a dictionary with the necessary information (again, see the *database documentation*).

Currently, the usage described above is sometimes used in the unittest code. In the pipeline code we try to restrict the usage of the orm, since it can get quite complex. Inserting the extracted sources for a particular image looks like this:

```
from tkp.db import general as dbgen
dbgen.insert_extracted_sources(image.id, results.sources, 'blind')
```

where image.id is the id of the image to which the sources belong to.

Other The remaining modules contain the actual SQL queries used for the various database routines, such as source insertion, source association, keeping track of the monitoring sources and null detections, etc. All functions and queries within each function call follow a fixed pattern: functions inherit the connection object, and each query is executed within a try-except block in case the database raises an error. There is some overhead in the sense that a cursor is created for each function call, but other than that, the routines provide the fastest and most direct interaction with the database.

Several functions are called from methods of the classes in the dataset module, to provide a hopefully clearer interface, but can of course be called directly as well. Finally, a number of functions are private to the module, and their name is therefore preceded with an underscore. These functions tend to be called from another funtion within the same module, but may be called from another module as well.

`tkp.db` – TKP database root package

`tkp.db.`**`commit`**`()`

A generic wrapper to commit a query transaction

It saves the changes involved by a transaction

`tkp.db.`**`connect`**`()`

A generic wrapper to connect to the configured database

`tkp.db.`**`connection`**`()`

A generic wrapper to create a connection to the database if it does not exist

`tkp.db.`**`execute`**`(`*query*, *parameters={}*, *commit=False*`)`

A generic wrapper for doing any query to the database

Parameters

- **`query`** – the query string
- **`parameters`** – The query parameters. These will be converted and escaped.
- **`commit`** – should a commit be performed afterwards, boolean

Returns a database cursor object

`tkp.db.`**`rollback`**`()`

A generic wrapper to rollback a query transaction

Undo changes involved by a transaction that have not been saved

`tkp.db.associations` – source association

A collection of back end subroutines (mostly SQL queries), In this module we deal with source association.

`tkp.db.associations.`**`_check_meridian_wrap`**`(`*image_id*`)`

Checks whether an image is close to the meridian ra = 0 or ra = 360

When so, the association query needs to be rewritten to take into account sources across the 0/360 meridian.

The query returns:

q_across: true, if the extraction region of the image crosses the ra=0/360 border

ra_min: the min value of the ra-between for the normal case, when the image is outside the ra=0/360 meridian, otherwise NULL

ra_max: the max value of the ra-between for the normal case, when the image is outside the ra=0/360 meridian, otherwise NULL

ra_min1/max1 and ra_min2/max2 are the values which may be used for the case of a cross-meridian image. F.ex. using a search radius of 5 degrees, and when a source is at 359.99 the ra-betweens 1 and 2 are : ... AND (ra BETWEEN ra_min1 AND ra_max1 OR ra BETWEEN ra_min2 AND ra_max2) ... ... AND (ra BETWEEN 354.99 AND 360 OR ra BETWEEN 0 AND 4.99) ...

ra_min1: the min value of the high-end ra-between, if the extraction region of the image crosses the ra=0/360 border, otherwise NULL

ra_max1: the min value of the high-end ra-between, if the extraction region of the image crosses the ra=0/360 border, otherwise NULL ra_min2, ra_max2: As ra_min1/max1, but for the low-end ra values.

These values are not being used in the cross-meridian association query, but are merely reported to notice the search area. The cross-meridian association query uses the cartesian dot product, to get the search area.

`tkp.db.associations.`**`_delete_1_to_many_inactive_assocskyrgn`**`()`

Delete the assocskyrgn links of the old runcat

Since we replaced this runcat.id with multiple new ones, we now delete the old links.

`tkp.db.associations.`**`_delete_1_to_many_inactive_assocxtrsource`**`()`

Delete the association pairs of the old runcat from assocxtrsource

NOTE: It might sound confusing, but those are not qualified as inactive in tempruncat (read below). Since we replaced this runcat.id with multiple new one, we first flag it as inactive, after which we delete it from the runningcatalog

The subselect selects those valid "old" runcat ids (i.e., the ones that were not set to inactive for the many-to-many associations).

NOTE: We do not have to flag these rows as inactive, no furthr processing depends on these in the assoc run

`tkp.db.associations.`**`_delete_1_to_many_inactive_newsource`**`()`

Delete the newsource sources of the old runcat

Since we replaced this runcat.id with multiple new ones, we now delete the old one.

`tkp.db.associations.`**`_delete_1_to_many_inactive_runcat_flux`**`()`

Flag the old runcat ids in the runningcatalog to inactive

Since we replaced this runcat.id with multiple new one, we first flag it as inactive, after which we delete it from the runningcatalog

`tkp.db.associations.`**`_delete_bad_blind_extractions`**`(`*image_id*`)`

Remove blind extractions centred outside designated extract region.

These occur sometimes due to highly elliptical fits on noisy data, creating a best fit centred outside the original pixel region. The source-extraction code has been modified to (probably) prevent this, but we check for them anyway.

NB. We currently only delete blind extractions. We expect that occasionally forced fits to sources just inside the extraction radius might converge just outside, but these should be restricted to a very small additional margin. By not deleting these edge cases, the data allows us to construct proper lightcurves, and (I think) does not contribute to their weighted mean positions (so sources cannot 'migrate' across the border). TODO(TS): Check this.

Only extractions from the specified image are checked for deletion.

Returns Number of extractedsource rows deleted.

`tkp.db.associations.`**`_delete_inactive_runcat`**`()`

Delete the one-to-many associations from temprunningcatalog, and delete the inactive rows from runningcatalog.

After the one-to-many associations have been processed, they can be deleted from the temporary table and the runningcatalog.

`tkp.db.associations.`**`_determine_newsource_previous_limits`**`(`*image_id*, *new_source_sigma_margin*`)`

Determines which new-runcat sources are also probably transient.

Looks up previous images relevant to this source-position, using the following criteria - images must:

- overlap the new-source position, according to the skyregion information;

- be in the same dataset;
- be in the same frequency band;
- have an earlier timestamp than the current image;
- have not been rejected.

For those images we calculate the per-previous-image detection-thresholds, which are defined as follows.

A new source is 'possibly transient' (type 0) if it passes the following tests:

- Was not detected in a skyregion being surveyed for the first time.
- Has a flux-value such that:

  flux > MIN_OVER_I [(rms_min_I*(det_I + new_source_sigma_margin)]

  (where I indexes the images) i.e. if it was a steady-source, it should have been already detected if it was in the *low-RMS* area of the previous image with best detection threshold, even allowing for noise fluctuations.

Furthermore, a new source is 'likely transient' (type 1) if it is additionally bright enough that, if it were a steady source, it should have been detected even if it was in the *high-RMS* area of the aforementioned 'low rms_min' image, i.e.

flux > (rms_max_I*(det_I + new_source_sigma_margin))

Note that, once we have located the image with best 'low rms threshold', we then use that image to *also* generate the 'high rms threshold'. Strictly speaking, this is non-optimal - we should run a fresh search against all images to find the best 'high rms threshold'. However, I'm working on the assumption that most of the time the image with best low-threshold will also have best high-threshold, and even when that is not the case we won't lose too much accuracy. The benefits of this assumption are simplicity, and possibly faster performance, but this might need to be re-examined in future, especially if we start ingesting images of wildly differing sizes and noise non-uniformity characteristics (e.g. single pointings vs mosaics) etc.

We use peak flux (f_peak) as the flux value here, since that is likely to be the deciding factor in whether a source gets blindly extracted or not. (NB This is a hunch, rigorous investigation welcome.)

`tkp.db.associations.`**`_empty_temprunningcatalog`**`()`
Initialize the temporary storage table

Initialize the temporary table temprunningcatalog which contains the current observed sources.

`tkp.db.associations.`**`_flag_1_to_many_inactive_runcat`**`()`
Flag the old runcat ids in the runningcatalog to inactive

We do not delete them yet, because we still need to clear up all the superseded entries in assocskyrgn, etc.

`tkp.db.associations.`**`_flag_1_to_many_inactive_tempruncat`**`()`
Flag the one-to-many associations from temprunningcatalog.

(Since we are done processing them, now.)

We do not delete them yet- if we did, we would not be able to cross-match extractedsources to determine which sources did not have a match in temprunningcatalog ('new' sources).

`tkp.db.associations.`**`_flag_many_to_many_tempruncat`**`()`
Select the many-to-many association pairs in temprunningcatalog.

By flagging the many-to-many associations, we reduce the processing to one-to-many and many-to-one (identical to one-to-one) relationships

tkp.db.associations.**_insert_1_to_1_assoc**()
Insert remaining associations from temprunningcatalog into assocxtrsource.

We also calculate the variability indices at the timestamp of the the current image.

tkp.db.associations.**_insert_1_to_1_runcat_flux**()
Insert the fluxes in runningcatalog_flux of a new band for an existing runcat source.

If the runcat, band, stokes entry does not exist (yet) in runcat_flux, we need to insert the new values from tempruncat. This might be the case if a source has been observed at other frequencies, but not in the current band, so there does not exist an entry for this band.

tkp.db.associations.**_insert_1_to_many_assocskyrgn**()
Copy skyregion associations from old runcat entries for new one-to-many runningcatalog entries.

tkp.db.associations.**_insert_1_to_many_basepoint_assocxtrsource**()
Insert 'base points' for one-to-many associations

Before continuing, we have to insert the 'base points' of the associations, i.e. the links between the new runningcatalog entries and their associated (new) extractedsources.

We also calculate the variability indices at the timestamp of the the current image.

tkp.db.associations.**_insert_1_to_many_newsource**()
Update the runcat id for the one-to-many associations, and delete the newsource entries of the old runcat id (the new ones have been added earlier).

In this case, new entries in the runningcatalog and runningcatalog_flux were already added (for every extractedsource one), which will replace the existing ones in the runningcatalog. Therefore, we have to update the references to these new ids as well.

tkp.db.associations.**_insert_1_to_many_replacement_assocxtrsource**()
Insert links into the association table between the new runcat entries and the old extractedsources. (New to New ('basepoint') links have been added earlier).

In this case, new entries in the runningcatalog and runningcatalog_flux were already added (for every extractedsource one), which will replace the existing ones in the runningcatalog. Therefore, we have to update the references to these new ids as well. So, we will append to assocxtrsource and delete the entries from runningcatalog_flux.

NOTE: 1. We do not update the distance_arcsec and r values of the pairs.

TODO: 1. Why not?

tkp.db.associations.**_insert_1_to_many_runcat**()
Insert the extracted sources that belong to one-to-many associations in the runningcatalog.

Since for the one-to-many associations (i.e. one runcat source associated with multiple extracted sources) we cannot a priori decide which counterpart pair is the correct one, or whether all are correct (in the case of a higher-resolution image), all extracted sources are added as a new source to the runningcatalog, and they will replace the (old; lower resolution) runcat source of the association.

As a consequence of this, the resolution of the runningcatalog is increasing over time.

tkp.db.associations.**_insert_1_to_many_runcat_flux**()
Insert the fluxes of the extracted sources that belong to a one-to-many association in the runningcatalog.

Analogous to the runningcatalog, extracted source properties are added to the runningcatalog_flux table.

tkp.db.associations.**_insert_new_assocxtrsource**(*image_id*)
Insert new associations for previously unknown sources.

tkp.db.associations.**_insert_new_runcat**(*image_id*)
Insert previously unknown sources into the `runningcatalog` table.

Extractedsources for which no counterpart was found in the runningcatalog (i.e. no pair exists in tempruncat), will be added as a new source to the assocxtrsource, runningcatalog and runningcatalog_flux tables.

`tkp.db.associations.`**`_insert_new_runcat_flux`**`(`*image_id*`)`
Insert previously unknown sources into the `runningcatalog_flux` table.

(i.e. those without *any* previous runcat-counterpart)

`tkp.db.associations.`**`_insert_new_runcat_skyrgn_assocs`**`(`*image_id*`)`
Process newly created entries from the runningcatalog, determine which skyregions they lie within.

Upon creation of a new runningcatalog entry, we need to determine which previous fields of view (skyrgns) we expect to see it in. This knowledge helps us to make accurate guesses as whether a new source is really transient or simply being surveyed for the first time.

`tkp.db.associations.`**`_insert_temprunningcatalog`**`(`*image_id*, *deRuiter_r*, *beamwidths_limit*, *meridian_wrap*`)`
Select matched sources

Here we select the extractedsource that have a positional match with the sources in the running catalogue table (runningcatalog). Those sources which *do* have a potential match, will be inserted into the temporary running catalogue table (temprunningcatalog).

See also: http://docs.transientskp.org/tkp/database/schema.html#temprunningcatalog

Explanation of some column name prefixes/suffixes used in the SQL query:

- avg_X := average of X
- avg_X_sq := average of X^2
- avg_weight_X := average of weight of X, i.e. mean(1/error^2)
- **avg_weighted_X := average of weighted X,** i.e. mean(X/error^2)
- **avg_weighted_X_sq := average of weighted X^2,** i.e. mean(X^2/error^2)

This result set might contain multiple associations (1-n,n-1) for a single known source in runningcatalog.

The n-1 assocs will be treated similar as n 1-1 assocs.

NOTE: Beware of the extra condition on x0.image in the WHERE clause, preventing the query to grow exponentially in response time

`tkp.db.associations.`**`_update_1_to_1_runcat`**`()`
Update the running catalog with the values in temprunningcatalog

`tkp.db.associations.`**`_update_1_to_1_runcat_flux`**`()`
Updates the fluxes in runningcatalog_flux of an existing band for an existing runcat source.

If the runcat, band, stokes entry does exist in runcat_flux, it will be updated with the values from tempruncat.

`tkp.db.associations.`**`_update_ff_runcat_extractedsource`**`()`
We are about to delete the runcats that are inactivated, and therefore have to set the ff_runcat reference in extractedsource to NULL.

`tkp.db.associations.`**`associate_extracted_sources`**`(`*image_id*, *deRuiter_r*, *beamwidths_limit=1*, *new_source_sigma_margin=3*`)`
Associate extracted sources with sources detected in the running catalog.

See the “developer’s reference” section of the docs for a step-by-step breakdown of the logic encapsulated here.

The dimensionless distance between two sources is given by the “De Ruiter radius”, see Chapters 2 & 3 of Scheers’ thesis.

`tkp.db.configstore` – Key-value record of job configuration params store and retrieve pipeline settings to/from database

`tkp.db.configstore.`**`fetch_config`**(*dataset_id*)
Retrieve the stored config for given dataset id

Returns nested dict [section][key] -> [value]

`tkp.db.configstore.`**`store_config`**(*config*, *dataset_id*)
Store a config defined in d into the database.

Parameters **`config`** (*dict*) – nested dict containing config, [section][key] -> [value]

`tkp.db.consistency` – Database integrity checking check database for consistency

`tkp.db.consistency.`**`check`**()
Checks for any inconsistent values in tables.

Returns False if any inconsistency is found, otherwise True.

`tkp.db.consistency.`**`isconsistent`**(*query*)
Counting rows should return 0, otherwise database is in an inconsistent state.

If the database is consistent we return True, otherwise False.

`tkp.db.database` – Database connection interface

class `tkp.db.database.`**`DBExceptions`**(*engine*)
This provides an engine-agnostic wrapper around the exceptions that can the thrown by the database layer: we can refer to eg DBExcetions(engine).Error rather than <engine specific module>.Error.

We handle both the PEP-0249 exceptions as provided by the DB engine, and add our own as necessary.

class `tkp.db.database.`**`Database`**(***kwargs*)
An object representing a database connection.

`close`()
close the connection if open

`connect`()
connect to the configured database

`connection`
The database connection, will be created if it doesn't exists.

This is a property to be backwards compatible with the rest of TKP.

Returns a database connection

`vacuum`(*table*)
Force a vacuum on a table, which removes dead rows. (Postgres only)

Normally the auto vacuum process does this for you, but in some cases (for example when the table receives many insert and deletes) manual vacuuming is necessary for performance reasons.

Parameters **`table`** – name of the table in the database you want to vacuum

`tkp.db.database.`**`sanitize_db_inputs`**(*params*)
Replace values in params with alternatives suitable for database insertion.

That includes:

- Convert numpy.floating types into Python floats;
- Convert infs into the string "Infinity".

Parameters **`params`** (*dict/list/tuple*) – (Potentially) dirty database inputs

Returns **cleaned** – Sanitized database inputs

Return type dict/list/tuple

`tkp.db.dump` **– Dump database backup to file** Dump database schema and content

`tkp.db.dump.`**`dump_db`**(*engine*, *hostname*, *port*, *dbname*, *dbuser*, *dbpass*, *output*)
Dumps a database

Parameters

- **`engine`** – the name of the database system (either monetdb or postgresql)
- **`hostname`** – the hostname of the database
- **`port`** – the port of the database server
- **`dbname`** – the database name to be dumped
- **`dbuser`** – the user authorised to do the dump
- **`dbpass`** – the pw for the user
- **`output`** – the output file to which the dump is written

`tkp.db.dump.`**`dump_monetdb`**(*hostname*, *port*, *dbname*, *dbuser*, *dbpass*, *output_filename*)
Dumps a MonetDB database in specified output file

`tkp.db.dump.`**`dump_pg`**(*hostname*, *port*, *dbname*, *dbuser*, *dbpass*, *output_filename*)
Dumps a PostgreSQL database in specified output file

`tkp.db.general` **– general database functions** A collection of back end subroutines (mostly SQL queries).

In this module we collect together various routines that don't fit into a more specific collection.

`tkp.db.general.`**`insert_dataset`**(*description*)
Insert dataset with description as given by argument.

DB function insertDataset() sets the necessary default values.

`tkp.db.general.`**`insert_extracted_sources`**(*image_id*, *results*, *extract_type*, *ff_runcat_ids=None*, *ff_monitor_ids=None*)
Insert all detections from sourcefinder into the extractedsource table.

Besides the source properties from sourcefinder, we calculate additional attributes that are increase performance in other tasks.

The strict sequence from results (the sourcefinder detections) is given below. Note the units between sourcefinder and database. (0) ra [deg], (1) dec [deg], (2) ra_fit_err [deg], (3) decl_fit_err [deg], (4) peak_flux [Jy], (5) peak_flux_err [Jy], (6) int_flux [Jy], (7) int_flux_err [Jy], (8) significance detection level, (9) beam major width (arcsec), (10) - minor width (arcsec), (11) - parallactic angle [deg], (12) ew_sys_err [arcsec], (13) ns_sys_err [arcsec], (14) error_radius [arcsec] (15) gaussian fit (bool) (16), (17) chisq, reduced_chisq (float)

ra_fit_err and decl_fit_err are the 1-sigma errors from the gaussian fit, in degrees. Note that for a source located towards the poles the ra_fit_err increases with absolute declination. error_radius is a pessimistic on-sky error estimate in arcsec. ew_sys_err and ns_sys_err represent the telescope dependent systematic errors and are in arcsec. An on-sky error (declination independent, and used in de ruiter calculations) is then: uncertainty_ew^2 = ew_sys_err^2 + error_radius^2 uncertainty_ns^2 = ns_sys_err^2 + error_radius^2 The units of uncertainty_ew and uncertainty_ns are in degrees. The error on RA is given by ra_err. For a source with an RA of ra and an error of ra_err, its RA lies in the range [ra-ra_err, ra+ra_err]. ra_err^2 = ra_fit_err^2 + [alpha_inflate(ew_sys_err,decl)]^2 decl_err^2 = decl_fit_err^2 + ns_sys_err^2. The units of ra_err and decl_err are in degrees. Here alpha_inflate() is the RA inflation function, it converts an angular on-sky distance to a ra distance at given declination.

Input argument "extract" tells whether the source detections originate from: 'blind': blind source extraction 'ff_nd': from forced fits at null detection locations 'ff_ms': from forced fits at monitoringlist positions

Input argument ff_runcat is not empty in the case of forced fits from null detections. It contains the runningcatalog ids from which the source positions were derived for the forced fits. In that case the runcat ids will be inserted into the extractedsource table as well, to simplify further null-detection processing. For blind extractions this list is empty (None).

For all extracted sources additional parameters are calculated, and appended to the sourcefinder data. Appended and converted are:

- the image id to which the extracted sources belong to
- the zone in which an extracted source falls is calculated, based on its declination. We adopt a zoneheight of 1 degree, so the floor of the declination represents the zone.
- the positional errors are converted from degrees to arcsecs
- the Cartesian coordinates of the source position
- ra * cos(radians(decl)), this is very often being used in source-distance calculations

`tkp.db.general.`**`insert_image`**(*dataset*, *freq_eff*, *freq_bw*, *taustart_ts*, *tau_time*, *beam_smaj_pix*, *beam_smin_pix*, *beam_pa_rad*, *deltax*, *deltay*, *url*, *centre_ra*, *centre_decl*, *xtr_radius*, *rms_qc*, *rms_min*, *rms_max*, *detection_thresh*, *analysis_thresh*)

Insert an image for a given dataset.

Parameters

- **dataset** (*int*) – ID of parent dataset.
- **freq_eff** – See *Image table definitions*.
- **freq_bw** – See *Image table definitions*.
- **taustart_ts** – See *Image table definitions*.
- **taus_time** – See *Image table definitions*.
- **beam_smaj_pix** (*float*) – Restoring beam semimajor axis length in pixels. (Converted to degrees before storing to database).
- **beam_smin_pix** (*float*) – Restoring beam semiminor axis length in pixels. (Converted to degrees before storing to database).
- **beam_pa_rad** (*float*) – Restoring beam parallactic angle in radians. (Converted to degrees before storing to database).
- **deltax** (*float*) – Degrees per pixel increment in x-direction.
- **deltay** (*float*) – Degrees per pixel increment in y-direction.
- **centre_ra** (*float*) – Image central RA co-ord, in degrees.
- **centre_decl** (*float*) – Image central Declination co-ord, in degrees.
- **xtr_radius** (*float*) – Radius in degrees from field centre that will be used for source extraction.

`tkp.db.general.`**`insert_monitor_positions`**(*dataset_id*, *positions*)

Add entries to the `monitor` table.

Parameters

- **dataset_id** (*int*) – Positions will only be monitored when processing this dataset.
- **positions** (*list of tuples*) – List of (RA, decl) tuples in decimal degrees.

tkp.db.general.**lightcurve**(*xtrsrcid*)
Obtain a light curve for a specific extractedsource

Parameters **xtrsrcid** (*int*) – the source identifier that corresponds to a point on the light curve. Note that the point does not have to be the start (first) point of the light curve.

Returns a list of tuples, each containing: - observation start time as a datetime.datetime object - integration time (float) - integrated flux (float) - integrated flux error (float) - database ID of this particular source - frequency band ID - stokes

Return type list

tkp.db.general.**update_dataset_process_end_ts**(*dataset_id*)
Update dataset start-of-processing timestamp.

tkp.db.generic – generic data-query generation functions A collection of generic functions used to generate SQL queries and return data in an easy to use format such as dictionaries.

tkp.db.generic.**columns_from_table**(*table*, *keywords=None*, *alias=None*, *where=None*, *order=None*)
Obtain specific column (keywords) values from 'table', with kwargs limitations.

A very simple helper function, that builds an SQL query to obtain the specified columns from 'table', and then executes that query. Optionally, the WHERE clause can be specified using the where dictionary. It returns a list of a dict (with the originally supplied keywords as dictionary keys), which can be empty.

Example

```
>>> columns_from_table('image',             keywords=['taustart_ts', 'tau_time', 'freq_eff', 'fr
    [{'freq_eff': 133984375.0, 'taustart_ts': datetime.datetime(2010, 10, 9, 9, 4, 2), 'tau_time
```

This builds the SQL query: "SELECT taustart_ts, tau_time, freq_eff, freq_bw FROM image WHERE imageid=1"

This function is implemented mainly to abstract and hide the SQL functionality from the Python interface.

Parameters

- **conn** – database connection object
- **table** (*string*) – database table name

Kwargs:

keywords (list): column names to select from the table. None indicates all ('*')

where (dict): where clause for the query, specified as a set of 'key = value' comparisons. Comparisons are and-ed together. Obviously, only 'is equal' comparisons are possible.

alias (dict): Chosen aliases for the column names, used when constructing the returned list of dictionaries

order (string): ORDER BY key.

Returns list of dicts. Each dict contains the given keywords, or all if keywords=None. Each element of the list corresponds to a table row.

Return type list

`tkp.db.generic.`**`convert_db_rows_to_dicts`**(*results*, *cursor_description*, *alias_map=None*)
Takes a list of rows as returned by cursor.fetchall(), converts to a list of dictionaries.

`tkp.db.generic.`**`get_db_rows_as_dicts`**(*cursor*, *alias_map=None*)
Grab results of cursor.fetchall(), convert to a list of dictionaries.

`tkp.db.generic.`**`set_columns_for_table`**(*table*, *data=None*, *where=None*)
Set specific columns (keywords) for 'table', with 'where' limitations.

A simple helper function, that builds an SQL query to update the specified columns given by data for 'table', and then executes that query. Optionally, the WHERE clause can be specified using the 'where' dictionary.

The data argument is a dictionary with the names and corresponding values of the columns that need to be updated.

`tkp.db.orm` – Object Relational Model interface This module contains lightweight container objects that corresponds to a dataset, image or extracted source in the database; it is actually a mini Object Relation Mapper (ORM). The correspondence between the object and table row is matched through the private _id attributes.

Each dataset contains several database Images; each Image contains a number of ExtractedSources. The database Images correspond to the images table in the database, *not* to sourcefinder images or actual image data files on disk (this distinction is important; while there are certainly parts in common, several are not).

The current setup is done in large part to keep the database and sourcefinder (and other parts of the TKP package) separate; tightly integrated database tables/sourcefinder images/disk files make it more difficult to improve the code or distribute parts separately.

Usage In practice, a DataSet object is created, and separate Images are created referencing that DataSet() instance; ids are automatically assigned where necessary (i.e., on creation of a new entry (row) in the database).

Objects can also be created using an existing id; data is then taken from the corresponding table row in the database.

Creating new objects The following code is an usage example, but should not be used as a doc test (since the database value can differ, and thus the test would fail):

```
# database sets up and holds the connection to the actual database
>>> database = tkp.db.database.Database()

# Each object type takes a data dictionary on creation, which for newly objects
# has some required keys (& values). For a DataSet, this is only 'description';
# for an Image, the keys are 'freq_eff', 'freq_bw_', 'taustart_ts',
# 'tau_time' & 'url'
# The required values are stored in the the REQUIRED attribute
>>> dataset = DataSet(data={'description': 'a dataset'}, database=database)

# Here, dataset indirectly holds the database connection:
>>> dataset.database
DataBase(host=heastro1, name=trap, user=trap, ...)
>>> image1 = Image(data={'freq_eff': '80e6', 'freq_bw': 1e6,         'taustart_ts': datetime(2011, 5,
    # note the dataset kwarg, which holds the database connection
>>> image1.tau_time
1800.
>>> image1.taustart_ts
datetime.datetime(2011, 5, 1, 0, 0, 0)
>>> image2 = Image(data={'freq_eff': '80e6', 'freq_bw': 1e6,         'taustart_ts': datetime(2011, 5,
>>> image2.tau_time
1500
```

```
>>> image2.taustart_ts
datetime.datetime(2011, 5, 1, 0, 1, 0)
# Images created with a dataset object, are automatically added to that dataset:
>>> dataset.images
set([<tkp.database.dataset.Image object at 0x26fb6d0>, <tkp.database.dataset.Image object at 0x26fb7
```

Updating objects To update objects, use the update() method.

This method does two things, in the following order:

1. it updates from the database to the object: if there have been changes in the database, the object will reflect that after executing update()

2. then, it updates the object (and the database) with values supplied by the user. The latter values are optional; no supplied values simply means there aren't any updates.

```
>>> image2.update(tau_time=2500)     # updates the database as well
>>> image2.tau_time
2500
>>> database.cursor.execute("SELECT tau_time FROM images WHERE imageid=%s" %
>>> database.cursors.fetchone()[0]
2500
# Manually update the database
>>> database.cursor.execute("UPDATE images SET tau_time=2000.0 imageid=%s" %
>>> image2.tau_time    # not updated yet!
2500
>>> image2.update()
>>> image2.tau_time
2000
```

Assigning objects to a table row on creation It is also possible to create a DataSet, Image or ExtractedSource instance from the database, using the `id` in the initializer:

```
>>> dataset2 = DataSet(id=dataset.id, database=database)
>>> image3 = Image(imageid=image2.id, database=database)
>>> image3.tau_time
2000
```

If an `id` is supplied, `data` is ignored.

class `tkp.db.orm.`**`DBObject`**(*data=None*, *database=None*, *id=None*)

Generic mini-ORM object

Derived objects will need to implement __init__, which for practical reasons is split up in __init__ and _init_data: the latter is called at the end __init__, so a derived __init__ would have super(Derived, self).__init__() at the start and super(Derived, self)._init_data() at the end.

__init__ takes care of setting the id, the supplied *data* dictionary and the connection to the database.

_init_data sets the actual data either from the database (in case of a supplied id) or from the *data* dictionary.

Basic initialization.

Inherited classes need to implement any actual database action, by calling self._init_data() at the end of their __init__ method.

`__getattr__`(*name*)

Obtain the 'name' attribute, where 'name' is a database column name

id

Add or obtain an id to/from the table

The id is generated if self._id does not exist, effectively creating a new row in the database.

Several containers have their specific SQL function to create a new object, so this property will need to overridden.

update(***kwargs*)

Update attributes from database, and set database values to kwargs when provided

This method performs two functions, the first always and the second optionally after the first:

- it updates the attributes from the database. That is, it makes sure the Python instance is synchronized with the database.
- (optional): it sets the column values in the database to the values provided through kwargs, for the associated database row. Attributes for the instance are of course also set to these values. Any kwargs that do not correspond to a column name are simply ignored.

This function therefore first updates the instance from the database, and then optionally the database from the instance (with the provided keyword arguments).

class tkp.db.orm.**DataSet**(*data=None*, *database=None*, *id=None*)

Class corresponding to the dataset table in the database

If id is supplied, the data and image arguments are ignored.

frequency_bands()

Return a list of distinct bands present in the dataset.

id

Add or obtain an id to/from the table

This uses the SQL function insertDataset().

runcat_entries()

Returns

a list of dictionarys representing rows in runningcatalog, for all sources belonging to this dataset

Column 'id' is returned with the key 'runcat'

Currently only returns 3 columns: [{'runcat,'xtrsrc','datapoints'}]

Return type list

update_images()

Renew the set of images by getting the images for this dataset from the database. Implemented separately from update(), since normally this would be too much overhead

class tkp.db.orm.**ExtractedSource**(*data=None*, *image=None*, *database=None*, *id=None*)

Class corresponding to the extractedsource table in the database

If id is supplied, the data and image arguments are ignored.

lightcurve()

Obtain the complete light curve (within the current dataset) for this source.

Returns list of 5-tuples, each tuple being: - observation start time as a datetime.datetime object - integration time (float) - integrated flux (float) - integrated flux error (float) - database ID of this particular source

Return type list

class `tkp.db.orm.`**`Image`**(*data=None*, *dataset=None*, *database=None*, *id=None*)
Class corresponding to the images table in the database

If id is supplied, the data and image arguments are ignored.

`associate_extracted_sources`(*deRuiter_r*, *new_source_sigma_margin*)
Associate sources from the last images with previously extracted sources within the same dataset

Parameters **`deRuiter_r`** (*float*) – The De Ruiter radius for source association. The default value is set through the tkp.config module

`id`
Add or obtain an id to/from the table

This uses the SQL function insertImage()

`insert_extracted_sources`(*results*, *extract='blind'*)
Insert a list of sources

Parameters

- **`results`** (*list*) – list of utility.containers.ExtractionResult objects (as returned from sourcefinder.image.ImageData().extract()), or a list of data tuples with the source information as follows: (ra, dec, ra_fit_err, dec_fit_err, peak, peak_err, flux, flux_err, significance level, beam major width (as), beam minor width(as), beam parallactic angle ew_sys_err, ns_sys_err, error_radius).
- **`extract`** (*str*) – 'blind', 'ff_nd' or 'ff_ms' (see db.general.insert_extracted_sources)

`update_rejected`()
Update self.rejected with the rejected status. Will be false if not rejected, will be a list of reject descriptions if rejected

`update_sources`()
Renew the set of sources by getting the sources for this image from the database

This method is separately implemented, because it's not always necessary and potentially (for an image with dozens or more sources) time & memory consuming.

`tkp.db.monitoringlist` – handling of monitoring sources A collection of back end subroutines (mostly SQL queries).

This module contains the routines to deal with the monitoring sources, provided by the user via the command line.

`tkp.db.monitoringlist.`**`_insert_1_to_1_assoc`**()
The runcat-monitoring pairs are appended to the assocxtrsource (light-curve) table as a type = 9 datapoint.

`tkp.db.monitoringlist.`**`_insert_new_1_to_1_assoc`**(*image_id*)
The forced fits of the monitoring sources which are new are appended to the assocxtrsource (light-curve) table as a type = 8 datapoint.

`tkp.db.monitoringlist.`**`_insert_new_runcat`**(*image_id*)
Insert the fits of the monitoring sources as new sources into the runningcatalog

`tkp.db.monitoringlist.`**`_insert_new_runcat_flux`**(*image_id*)
Insert the fitted fluxes of the monitoring sources as new datapoints into the runningcatalog_flux.

Extractedsources for which not a counterpart was found in the runningcatalog, i.e., those that do not have an entry in the tempruncat table (t0) will be added as a new source in the runningcatalog_flux table.

`tkp.db.monitoringlist.`**`_insert_runcat_flux`**()
Monitoring sources that were not yet fitted in this frequency band before, will be appended to it. Those have their first f_datapoint.

`tkp.db.monitoringlist.`**`_insert_tempruncat`**`(`*image_id*`)`
Here the associations of forced fits of the monitoring sources and their runningcatalog counterparts are inserted into the temporary table.

We follow the implementation of the normal association procedure, except that we don't need to match with a De Ruiter radius, since the counterpart pairs are from the same runningcatalog source.

`tkp.db.monitoringlist.`**`_update_monitor_runcats`**`(`*image_id*`)`
Update `runcat` col of `monitor` table for newly extracted positions.

`tkp.db.monitoringlist.`**`associate_ms`**`(`*image_id*`)`
Associate the monitoring sources, i.e., their forced fits, of the current image with the ones in the running catalog. These associations are treated separately from the normal associations and there will only be 1-to-1 associations.

The runcat-monitoring source pairs will be inserted in a temporary table. Of these, the runcat and runcat_flux tables are updated with the new datapoints if the (monitoring) source already existed, otherwise they are inserted as a new source. The source pair is appended to the light-curve table (assocxtrsource), with a type = 8 (for the first occurence) or type = 9 (for existing runcat sources). After all this, the temporary table is emptied again.

`tkp.db.monitoringlist.`**`get_monitor_entries`**`(`*dataset_id*`)`
Returns the `monitor` entries relevant to this dataset.

Parameters **`dataset_id`** (*int*) – Parent dataset.

Returns list of tuples [(monitor_id, ra, decl)]

`tkp.db.nulldetections` **– handling of null detections** A collection of back end subroutines (mostly SQL queries).

This module contains the routines to deal with null detections.

`tkp.db.nulldetections.`**`_insert_1_to_1_assoc`**`()`
The null detection forced fits are appended to the assocxtrsource (light-curve) table as a type = 7 datapoint. Subtable t1 has to take care of the cases where values and differences might get too small to cause divisions by zero.

`tkp.db.nulldetections.`**`_insert_tempruncat`**`(`*image_id*`)`
Here the associations of forced fits and their runningcatalog counterparts are inserted into the temporary table.

We follow the analogies of the normal association procedure. The difference here is that we know what the runcat ids are for the extractedsource.extract_type = 1 (ff_nd) sources are, since these were inserted at the same time as well.

This is why subtable t0 looks simpler than in the normal case. We still have to do a left outer join with the runcat_flux table (rf), because fluxes might not be detected in other bands. Before being inserted the additional properties are calculated.

`tkp.db.nulldetections.`**`associate_nd`**`(`*image_id*`)`
Associate the null detections (ie forced fits) of the current image.

They will be inserted in a temporary table, which contains the associations of the forced fits with the running catalog sources. Also, the forced fits are appended to the assocxtrsource (light-curve) table. The runcat_flux table is updated with the new datapoints if it already existed, otherwise it is inserted as a new datapoint. (We leave the runcat table unchanged.) After all this, the temporary table is emptied again.

`tkp.db.nulldetections.`**`get_nulldetections`**`(`*image_id*`)`
Returns the runningcatalog sources which:

- Are associated with the skyregion of the current image.
- Do not have a counterpart in the extractedsources of the current image after source association has run.

- Have been seen (in any band) at a timestamp earlier than that of the current image.

NB This runs *after* the source association.

We determine null detections only as those sources which have been seen at earlier times which don't appear in the current image. Sources which have not been seen earlier, and which appear in different bands at the current timestep, are *not* null detections, but are considered as "new" sources.

Returns: list of tuples [(runcatid, ra, decl)]

`tkp.db.quality` – Routines handling the 'rejectreason' table. check image quality

class `tkp.db.quality.`**`RejectReason`**(*id*, *desc*)

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

`__repr__`()
Return a nicely formatted representation string

`desc`
Alias for field number 1

`id`
Alias for field number 0

`tkp.db.quality.`**`isrejected`**(*imageid*)
Find out if an image is rejected or not :param imageid: The image ID of the image to reject :returns: False if not rejected, a list of reason id's if rejected

`tkp.db.quality.`**`reject`**(*imageid*, *reason*, *comment*)
Add a reject intro to the db for a given image :param imageid: The image ID of the image to reject :param reason: why is the image rejected, a defined in 'reason' :param comment: an optional comment with details about the reason

`tkp.db.quality.`**`unreject`**(*imageid*)
Remove all rejection of a given imageid :param imageid: The image ID of the image to reject

`tkp.distribute` – routines for running on distributed nodes

tkp.distribute implement various computation distribution methods. All sub modules should communicate with the other parts of TKP through the *tkp.steps* submodule.

`tkp.telescope` – Telescope specific functionality

`tkp.telescope.lofar` Functions for calculating LOFAR hardware specific properties. This module contains numbers of phsysicial properties of the LOFAR array.

For performance reasons these distances are precomputed. One can recompute them using *`parse_antennafile()`*, *`shortest_distances()`* and a `AntennaArrays.conf` file from `LOFAR/MAC/Deployment/data/StaticMetaData/AntennaArrays` in the lofar system software source tree.

`tkp.telescope.lofar.antennaarrays.`**`parse_antennafile`**(*positions_file*)

Parses an antenna file from the LOFAR system software repository.

Parameters **`positions_file`** – a antenna file

Returns a dictionary with array as key and positions as values

`tkp.telescope.lofar.antennaarrays.`**`pretty_print`**(*file_*)

Pretty prints a parsed antenna file. Use this function to generate copy paste code to be used in the top of this file.

Parameters **`file`** – a file location

`tkp.telescope.lofar.antennaarrays.`**`shortest_distances`**(*coordinates*, *full_array*)

returns a list of distances for each antenna relative to its closest neighbour.

Parameters

- **`coordinates`** – a list of 3 value tuples that represent x,y and z coordinates of a subset of the array
- **`full_array`** – a list of x,y,z coordinates of a full array

Returns a list of floats of distances

Beam characterization calculations.

For more information and the math behind this code go to the LOFAR imaging capabilities page.

`tkp.telescope.lofar.beam.`**`fov`**(*fwhm*)

The Field of View (FoV) of a LOFAR station

Parameters **`fwhm`** – nominal Full Width Half Maximum, caulculated with *`fwhm()`*.

`tkp.telescope.lofar.beam.`**`fwhm`**(*lambda_*, *d*, *alpha1=1.3*)

The nominal Full Width Half Maximum (FWHM) of a LOFAR Station beam.

Parameters

- **`lambda`** – wavelength in meters
- **`d`** – station diameter.
- **`alpha1`** – depends on the tapering intrinsic to the layout of the station, and any additional tapering which may be used to form the station beam. No electronic tapering is presently applied to LOFAR station beamforming. For a uniformly illuminated circular aperture, alpha1 takes the value of 1.02, and the value increases with tapering (Napier 1999).

Returns the nominal Full Width Half Maximum (FWHM)

functions for calculating theoretical noise levels of LOFAR equipment.

For more information about the math used here read the sensitivity of the LOFAR array page.

To check the values calculated here one can use this LOFAR image noise calculator.

`tkp.telescope.lofar.noise.`**`Aeff_dipole`**(*freq_eff*, *distance=None*)

The effective area of each dipole in the array is determined by its distance to the nearest dipole (d) within the full array.

Parameters

- **`freq_eff`** – Frequency
- **`distance`** – Distance to nearest dipole, only required for LBA.

`tkp.telescope.lofar.noise.`**`noise_level`**(*freq_eff*, *bandwidth*, *tau_time*, *antenna_set*, *Ncore*, *Nremote*, *Nintl*)

Returns the theoretical noise level (in Jy) given the supplied array antenna_set.

Parameters

- **bandwidth** – in Hz
- **tau_time** – in seconds
- **inner** – in case of LBA, inner or outer
- **antenna_set** – LBA_INNER, LBA_OUTER, LBA_SPARSE, LBA or HBA

`tkp.telescope.lofar.noise.`**`system_sensitivity`**(*freq_eff*, *Aeff*)
Returns the SEFD of a system, given the freq_eff and effective collecting area. Returns SEFD in Jansky's.

`tkp.quality` – Data-quality control

Data-quality checking related code.

The quality checks are described in the LOFAR Transients Key Science Project Quality Control Document V1.1 and on the wiki.

`tkp.quality.restoringbeam`

`tkp.quality.restoringbeam.`**`beam_invalid`**(*semibmaj*, *semibmin*, *theta*, *oversampled_x=30*, *elliptical_x=2.0*)
Are the beam shape properties ok?

Parameters **semibmaj/semibmin** – size of the beam in pixels

Returns True/False

`tkp.quality.restoringbeam.`**`highly_elliptical`**(*semibmaj*, *semibmin*, *x=2.0*)
If the beam is highly elliptical it can cause source association problems within TraP. Again further testing is required to determine exactly where the cut needs to be.

Parameters **Semibmaj/semibmin** – describe the beam size in pixels

Returns True if the beam is highly elliptical, False otherwise

`tkp.quality.restoringbeam.`**`infinite`**(*smaj*, *smin*, *bpa*)
If the beam is not correctly fitted by AWimager, one or more parameters will be recorded as infinite.

Parameters

- **smaj** – Semi-major axis (arbitrary units)
- **smin** – Semi-minor axis
- **bpa** – Postion angle

`tkp.quality.restoringbeam.`**`not_full_fieldofview`**(*nx*, *ny*, *cellsize*, *fov*)
This has been raised as an interesting test, as if the full field of view (FOV) has not been imaged we may want to image the full dataset. The imaged FOV information can be estimated using the number of pixels and the size of the pixels.

Parameters

- **nx** – number of pixels in x direction
- **ny** – number of pixels in y direction

Returns True if the full FOV is imaged, False otherwise

`tkp.quality.restoringbeam.`**`oversampled`**(*semibmaj*, *semibmin*, *x=30*)
It has been identified that having too many pixels across the restoring beam can lead to bad images, however further testing is required to determine the exact number.

Parameters **`Semibmaj/semibmin`** – describe the beam size in pixels

Returns True if beam is oversampled, False otherwise

`tkp.quality.restoringbeam.`**`undersampled`**(*semibmaj*, *semibmin*)

We want more than 2 pixels across the beam major and minor axes.

Parameters **`Semibmaj/semibmin`** – describe the beam size in pixels

Returns True if beam is undersampled, False otherwise

`tkp.quality.brightsource`

`tkp.quality.brightsource.`**`check_for_valid_ephemeris`**(*measures*)

Checks whether the ephemeris data in use by `measures` is valid. `measures` should already have a valid reference frame.

`tkp.quality.brightsource.`**`is_bright_source_near`**(*accessor*, *distance=20*)

Checks if there is any of the bright radio sources defined in targets near the center of the image.

Parameters

- **`accessor`** – a TKP accessor
- **`distance`** – maximum allowed distance of a bright source (in degrees)

Returns False if not bright source is near, description of source if a bright source is near

`tkp.quality.rms`

`tkp.quality.rms.`**`rms_invalid`**(*rms*, *noise*, *low_bound=1*, *high_bound=50*)

Is the RMS value of an image outside the plausible range?

Parameters

- **`rms`** – RMS value of an image, can be computed with tkp.quality.statistics.rms
- **`noise`** – Theoretical noise level of instrument, can be calculated with tkp.lofar.noise.noise_level
- **`low_bound`** – multiplied with noise to define lower threshold
- **`high_bound`** – multiplied with noise to define upper threshold

Returns True/False

`tkp.quality.statistics` functions for calculating statistical properties of LOFAR images

`tkp.quality.statistics.`**`clip`**(*data*, *sigma=3*)

Remove all values above a threshold from the array. Uses iterative clipping at sigma value until nothing more is getting clipped. :param data: a numpy array

`tkp.quality.statistics.`**`rms`**(*data*)

Returns the RMS of the data about the median. :param data: a numpy array

`tkp.quality.statistics.`**`rms_with_clipped_subregion`**(*data*, *rms_est_sigma=3*, *rms_est_fraction=4*)

RMS for quality-control.

Root mean square value calculated from central region of an image. We sigma-clip the input-data in an attempt to exclude source-pixels and keep only background-pixels.

Parameters

- **`data`** – A numpy array

- **rms_est_sigma** – sigma value used for clipping
- **rms_est_fraction** – determines size of subsection, result will be 1/fth of the image size where f=rms_est_fraction

returns the rms value of a iterative sigma clipped subsection of an image

tkp.quality.statistics.**subregion**(*data*, *f=4*)

Returns the inner region of a image, according to f.

Resulting area is 4/(f*f) of the original. :param data: a numpy array

tkp.sourcefinder – source finding and fitting

tkp.sourcefinder.deconv Gaussian deconvolution.

tkp.sourcefinder.deconv.**deconv**(*fmaj*, *fmin*, *fpa*, *cmaj*, *cmin*, *cpa*)

Deconvolve a Gaussian "beam" from a Gaussian component.

When we fit an elliptical Gaussian to a point in our image, we are actually fitting to a convolution of the physical shape of the source with the beam pattern of our instrument. This results in the fmaj/fmin/fpa arguments to this function.

Since the shape of the (clean) beam (arguments cmaj/cmin/cpa) is known, we can deconvolve it from the fitted parameters to get the "real" underlying physical source shape, which is what this function returns.

Parameters

- **fmaj** (*float*) – Fitted major axis
- **fmin** (*float*) – Fitted minor axis
- **fpa** (*float*) – Fitted position angle of major axis
- **cmaj** (*float*) – Clean beam major axis
- **cmin** (*float*) – Clean beam minor axis
- **cpa** (*float*) – Clean beam position angle of major axis

Returns **rmaj** – Real major axis rmin (float): Real minor axis rpa (float): Real position angle of major axis ierr (int): Number of components which failed to deconvolve

Return type float

tkp.sourcefinder.image – Image class & routines This module provides simple access to an image, without database overhead. The Image class handles the actual data (a (2D) numpy array), the world coordinate system (a tkp.utility.coordinates.WCS instance) and the beam information. While these three objects are supplied upon instantiation of an Image, one can use a *tkp.accessors.dataaccessor.DataAccessor* object to automatically derive these from the image file itself (provided the header information in the file is correct). Some generic utility routines for number handling and calculating (specific) variances

class tkp.sourcefinder.image.**ImageData**(*data*, *beam*, *wcs*, *margin=0*, *radius=0*, *back_size_x=32*, *back_size_y=32*, *residuals=True*)

Encapsulates an image in terms of a numpy array + meta/headerdata.

This is your primary contact point for interaction with images: it icludes facilities for source extraction and measurement, etc.

Sets up an ImageData object.

Args:

- data (2D numpy.ndarray): actual image data
- wcs (utility.coordinates.wcs): world coordinate system specification
- beam (3-tuple): beam shape specification as (semimajor, semiminor, theta)

backmap
Background map

static box_slice_about_pixel (*x*, *y*, *box_radius*)
Returns a slice centred about (x,y), of width = 2*int(box_radius) + 1

clearcache ()
Zap any calculated data stored in this object.

Clear the background and rms maps, labels, clip, and any locally held data. All of these can be reconstructed from the data accessor.

Note that this *must* be run to pick up any new settings.

data
Masked image data

data_bgsubbed
Background subtracted masked image data

extract (*det*, *anl*, *noisemap=None*, *bgmap=None*, *labelled_data=None*, *labels=None*, *deblend_nthresh=0*, *force_beam=False*)
Kick off conventional (ie, RMS island finding) source extraction.

Kwargs:

det (float): detection threshold, as a multiple of the RMS noise. At least one pixel in a source must exceed this for it to be regarded as significant.

anl (float): analysis threshold, as a multiple of the RMS noise. All the pixels within the island that exceed this will be used when fitting the source.

noisemap (numpy.ndarray):

bgmap (numpy.ndarray):

deblend_nthresh (int): number of subthresholds to use for deblending. Set to 0 to disable.

force_beam (bool): force all extractions to have major/minor axes equal to the restoring beam

Returns `tkp.utility.containers.ExtractionResults`

fd_extract (*alpha*, *anl=None*, *noisemap=None*, *bgmap=None*, *deblend_nthresh=0*, *force_beam=False*)
False Detection Rate based source extraction. The FDR procedure guarantees that <FDR> < alpha.

See Hopkins et al., AJ, 123, 1086 (2002).

fit_fixed_positions (*positions*, *boxsize*, *threshold=None*, *fixed='position+shape'*, *ids=None*)
Convenience function to fit a list of sources at the given positions

This function wraps around fit_to_point().

Parameters

- **positions** (*list*) – list of (RA, Dec) tuples. Positions to be fit, in decimal degrees.
- **boxsize** – See `fit_to_point()`

- **`threshold`** – as above.
- **`fixed`** – as above.
- **`ids`** (*list*) – A list of identifiers. If not None, then must match the length and order of the `requested_fits`. Any successfully fit positions will be returned in a tuple along with the matching id. As these are simply passed back to calling code they can be a string, tuple or whatever.

In particular, boxsize is in pixel coordinates as in fit_to_point, not in sky coordinates.

Returns A list of successful fits. If `ids` is None, returns a single list of *`tkp.sourcefinder.extract.Detection`* s. Otherwise, returns a tuple of two matched lists: ([detections], [matching_ids]).

Return type list

`fit_to_point` (*x*, *y*, *boxsize*, *threshold*, *fixed*)
Fit an elliptical Gaussian to a specified point on the image.

The fit is carried on a square section of the image, of length *boxsize* & centred at pixel coordinates *x*, *y*. Any data below *threshold* * rmsmap is not used for fitting. If *fixed* is set to `position`, then the pixel coordinates are fixed in the fit.

Returns an instance of *`tkp.sourcefinder.extract.Detection`*.

`flux_at_pixel` (*x*, *y*, *numpix=1*)
Return the background-subtracted flux at a certain position in the map

`grids`
Gridded RMS and background data for interpolating

`label_islands` (*detectionthresholdmap*, *analysisthresholdmap*)
Return a lablled array of pixels for fitting.

Parameters

- **`detectionthresholdmap`** (*numpy.ndarray*) –
- **`analysisthresholdmap`** (*numpy.ndarray*) –

Returns

list of valid islands (list of int)

labelled islands (numpy.ndarray)

`pixmax`
Maximum pixel value (pre-background subtraction)

`pixmin`
Minimum pixel value (pre-background subtraction)

`reverse_se` (*det*)
Run source extraction on the negative of this image.

Obviously, there should be no sources in the negative image, so this tells you about the false positive rate.

We need to clear cached data – backgroung map, cached clips, etc – before & after doing this, as they'll interfere with the normal extraction process. If this is regularly used, we'll want to implement a separate cache.

`rmsmap`
RMS map

xdim
X pixel dimension of (unmasked) data

ydim
Y pixel dimension of (unmasked) data

tkp.sourcefinder.extract – Source extraction routines Source Extraction Helpers.

These are used in conjunction with image.ImageData.

class tkp.sourcefinder.extract.**Detection**(*paramset*, *imagedata*, *chunk=None*, *eps_ra=0*, *eps_dec=0*)
The result of a measurement at a given position in a given image.

distance_from(*x*, *y*)
Distance from center

serialize(*ew_sys_err*, *ns_sys_err*)
Return source properties suitable for database storage.

We manually add ew_sys_err, ns_sys_err

returns: a list of tuples containing all relevant fields

class tkp.sourcefinder.extract.**Island**(*data*, *rms*, *chunk*, *analysis_threshold*, *detection_map*, *beam*, *deblend_nthresh*, *deblend_mincont*, *structuring_element*, *rms_orig=None*, *flux_orig=None*, *subthrrange=None*)
The source extraction process forms islands, which it then fits. Each island needs to know its position in the image (ie, x, y pixel value at one corner), the threshold above which it is detected (analysis_threshold by default, but will increase if the island is the result of deblending), and a data array.

The island should provide a means of deblending: splitting itself apart and returning multiple sub-islands, if necessary.

deblend(*niter=0*)
Return a decomposed numpy array of all the subislands.

Iterate up through subthresholds, looking for our island splitting into two. If it does, start again, with two or more separate islands.

fit(*fixed=None*)
Fit the position

noise()
Noise at maximum position

sig()
Deviation

threshold()
Threshold

class tkp.sourcefinder.extract.**ParamSet**(*clean_bias=0.0*, *clean_bias_error=0.0*, *frac_flux_cal_error=0.0*, *alpha_maj1=2.5*, *alpha_min1=0.5*, *alpha_maj2=0.5*, *alpha_min2=2.5*, *alpha_maj3=1.5*, *alpha_min3=1.5*)
All the source fitting methods should go to produce a ParamSet, which gives all the information necessary to make a Detection.

calculate_errors(*noise*, *beam*, *threshold*)
Calculate positional errors

Uses _condon_formulae() if this object is based on a Gaussian fit, _error_bars_from_moments() if it's based on moments.

deconvolve_from_clean_beam(*beam*)
Deconvolve with the clean beam

keys()

`tkp.sourcefinder.extract.`**source_profile_and_errors**(*data*, *threshold*, *noise*, *beam*, *fixed=None*)
Return a number of measurable properties with errorbars

Given an island of pixels it will return a number of measurable properties including errorbars. It will also compute residuals from Gauss fitting and export these to a residual map.

In addition to handling the initial parameter estimation, and any fits which fail to converge, this function runs the goodness-of-fit calculations - see `tkp.sourcefinder.fitting.goodness_of_fit()` for details.

Parameters

- **data** (*numpy.ndarray*) – array of pixel values, can be a masked array, which is necessary for proper Gauss fitting, because the pixels below the threshold in the corners and along the edges should not be included in the fitting process
- **threshold** (*float*) – Threshold used for selecting pixels for the source (ie, building an island)
- **noise** (*float*) – Noise level in data
- **beam** (*tuple*) – beam parameters (semimaj,semimin,theta)

Kwargs:

fixed (dict): Parameters (and their values) to hold fixed while fitting. Passed on to fitting.fitgaussian().

Returns a populated ParamSet, and a residuals map. Note the residuals map is a regular ndarray, where masked (unfitted) regions have been filled with 0-values.

Return type tuple

tkp.sourcefinder.gaussian – defines a two dimensional Gaussian function Definition of a two dimensional elliptical Gaussian.

`tkp.sourcefinder.gaussian.`**gaussian**(*height*, *center_x*, *center_y*, *semimajor*, *semiminor*, *theta*)
Return a 2D Gaussian function with the given parameters.

Parameters

- **height** (*float*) – (z-)value of the 2D Gaussian
- **center_x** (*float*) – x center of the Gaussian
- **center_y** (*float*) – y center of the Gaussian
- **semimajor** (*float*) – major axis of the Gaussian
- **semiminor** (*float*) – minor axis of the Gaussian
- **theta** (*float*) – angle of the 2D Gaussian in radians, measured between the semi-major and y axes, in counterclockwise direction.

Returns 2D Gaussian (function of pixel coords `(x,y)`)

Return type lambda

`tkp.sourcefinder.fitting` – Source fitting routines Source fitting routines.

`tkp.sourcefinder.fitting.`**`fitgaussian`**(*pixels*, *params*, *fixed=None*, *maxfev=0*)
Calculate source positional values by fitting a 2D Gaussian

Parameters

- **`pixels`** (*numpy.ma.MaskedArray*) – Pixel values (with bad pixels masked)
- **`params`** (*dict*) – initial fit parameters (possibly estimated using the moments() function, above)

Kwargs:

fixed (dict): parameters & their values to be kept frozen (ie, not fitted)

maxfev (int): maximum number of calls to the error function

Returns peak, total, x barycenter, y barycenter, semimajor, semiminor, theta (radians)

Return type dict

Raises `exceptions.ValueError` – In case of a bad fit.

Perform a least squares fit to an elliptical Gaussian.

If a dict called fixed is passed in, then parameters specified within the dict with the same names as fit_params (below) will be "locked" in the fitting process.

`tkp.sourcefinder.fitting.`**`goodness_of_fit`**(*masked_residuals*, *noise*, *beam*)
Calculates the goodness-of-fit values, *chisq* and *reduced_chisq*.

Warning: We do not use the standard chi-squared formula for calculating these goodness-of-fit values, and should probably rename them in the next release. See below for details.

These goodness-of-fit values are related to, but not quite the same as reduced chi-squared. Strictly speaking the reduced chi-squared is statistically invalid for a Gaussian model from the outset (see arxiv:1012.3754). We attempt to provide a resolution-independent estimate of goodness-of-fit ('reduced chi-squared'), by using the same 'independent pixels' correction as employed when estimating RMS levels, to normalize the chi-squared value. However, as applied to the standard formula this will sometimes imply that we are fitting a fractional number of datapoints less than 1! As a result, it doesn't really make sense to try and apply the 'degrees-of-freedom' correction, as this would likely result in a negative `reduced_chisq` value. (And besides, the 'degrees of freedom' concept is invalid for non-linear models.) Finally, note that when called from *`source_profile_and_errors()`*, the noise-estimate at the peak-pixel is supplied, so will typically overestimate the noise and hence under-estimate the chi-squared values.

Parameters

- **`masked_residuals`** (*numpy.ma.MaskedArray*) – The pixel-residuals from the fit
- **`noise`** (*float*) – An estimate of the noise level. Could also be set to a masked numpy array matching the data, for per-pixel noise estimates.
- **`beam`** (*tuple*) – Beam parameters

Returns chisq, reduced_chisq

Return type tuple

`tkp.sourcefinder.fitting.`**`moments`**(*data*, *beam*, *threshold=0*)
Calculate source positional values using moments

Parameters

- **data** (*numpy.ndarray*) – Actual 2D image data
- **beam** (*3-tuple*) – beam (psf) information, with semi-major and semi-minor axes

Returns peak, total, x barycenter, y barycenter, semimajor axis, semiminor axis, theta

Return type dict

Raises `exceptions.ValueError` – in case of NaN in input.

Use the first moment of the distribution is the barycenter of an ellipse. The second moments are used to estimate the rotation angle and the length of the axes.

`tkp.sourcefinder.stats` – Statistics specific to source finding for radio images. Generic utility routines for number handling and calculating (specific) variances used by the TKP sourcefinder.

`tkp.sourcefinder.stats.`**`sigma_clip`**(*data*, *beam*, *sigma=<function unbiased_sigma>*, *max_iter=100*, *centref=<function median>*, *distf=<function var>*, *my_iterations=0*, *corr_clip=1.0*)

Iterative clipping

By default, this performs clipping of the standard deviation about the median of the data. But by tweaking centref/distf, it could be much more general.

max_iter sets the maximum number of iterations used.

my_iterations is a counter for recursive operation of the code; leave it alone unless you really want to pretend to jump into the middle of a loop.

sigma is subtle: if a callable is given, it is passed a copy of the data array and can calculate a clipping limit. See, for e.g., unbiased_sigma() defined above. However, if it isn't callable, sigma is assumed to just set a hard limit.

To do: Improve documentation -Returns??? -How does it make use of the beam? (It estimates the noise correlation)

`tkp.sourcefinder.stats.`**`unbiased_sigma`**(*N_indep*)

Calculate an unbiased sigma for using in sigma clipping.

The formula below for cliplim is pretty subtle. Kappa, sigma clipping should be such that the noise is not biased by it. Consequently, the clipping boundaries should be such that exactly half an independent pixel should exceed it if the map were source free. A rigid boundary of 3 sigma is appropriate only if the number of independent pixels is about 185 (the number of independent pixels equals the number of pixels divided by the beamsize in pixels).

The condition that kappa, sigma clipping may not bias the noise is translated in the formula below, using Gaussian statistics. A disadvantage of this is that more iterations of kappa, sigma clipping are needed, compared to 3 sigma clipping. However, the noise values derived are generally significantly different (lower) compared to 3 sigma clipping.

`tkp.sourcefinder.stats.`**`var_helper`**(*N*)

Correct for the fact the rms noise is computed from a clipped distribution.

That noise will always be lower than the noise from the complete distribution. The correction factor is a function of the computed rms noise only.

`tkp.sourcefinder.utils` – Utility routines This module contain utilities for the source finding routines

`tkp.sourcefinder.utils.`**`calculate_beamsize`**(*semimajor*, *semiminor*)

Calculate the beamsize based on the semi major and minor axes

`tkp.sourcefinder.utils.`**`calculate_correlation_lengths`**(*semimajor*, *semiminor*)
Calculate the Condon correlation length

In order to derive the error bars from Gauss fitting from the Condon (1997, PASP 109, 116C) formulae, one needs the so called correlation length. The Condon formulae assumes a circular area with diameter theta_N (in pixels) for the correlation. This was later generalized by Hopkins et al. (2003, AJ 125, 465) for correlation areas which are not axisymmetric.

Basically one has theta_N^2 = theta_B*theta_b.

Good estimates in general are:

- theta_B = 2.0 * semimajar
- theta_b = 2.0 * semiminor

`tkp.sourcefinder.utils.`**`circular_mask`**(*xdim*, *ydim*, *radius*)
Returns a numpy array of shape (xdim, ydim). All points with radius of the centre are set to 0; outside that region, they are set to 1.

`tkp.sourcefinder.utils.`**`flatten`**(*nested_list*)
Flatten a nested list

Nested lists are made in the deblending algorithm. They're awful. This is a piece of code I grabbed from http://www.daniweb.com/code/snippet216879.html.

The output from this method is a generator, so make sure to turn it into a list, like this:

```
flattened = list(flatten(nested)).
```

`tkp.sourcefinder.utils.`**`fudge_max_pix`**(*semimajor*, *semiminor*, *theta*)
Estimate peak flux correction at pixel of maximum flux

Previously, we adopted Rengelink's correction for the underestimate of the peak of the Gaussian by the maximum pixel method: fudge_max_pix = 1.06. See the WENSS paper (1997A&AS..124..259R) or his thesis. (The peak of the Gaussian is, of course, never at the exact center of the pixel, that's why the maximum pixel method will always underestimate it.)

But, instead of just taking 1.06 one can make an estimate of the overall correction by assuming that the true peak is at a random position on the peak pixel and averaging over all possible corrections. This overall correction makes use of the beamshape, so strictly speaking only accurate for unresolved sources.

`tkp.sourcefinder.utils.`**`generate_result_maps`**(*data*, *sourcelist*)
Return a source and residual image

Given a data array (image) and list of sources, return two images, one showing the sources themselves and the other the residual after the sources have been removed from the input data.

`tkp.sourcefinder.utils.`**`generate_subthresholds`**(*min_value*, *max_value*, *num_thresholds*)
Generate a series of `num_thresholds` logarithmically spaced values in the range (min_value, max_value) (both exclusive).

`tkp.sourcefinder.utils.`**`get_error_radius`**(*wcs*, *x_value*, *x_error*, *y_value*, *y_error*)
Estimate an absolute angular error on the position (x_value, y_value) with the given errors.

This is a pessimistic estimate, because we take sum of the error along the X and Y axes. Better might be to project them both back on to the major/minor axes of the elliptical fit, but this should do for now.

`tkp.sourcefinder.utils.`**`maximum_pixel_method_variance`**(*semimajor*, *semiminor*, *theta*)
Estimate variance for peak flux at pixel position of maximum

When we use the maximum pixel method, with a correction fudge_max_pix, there should be no bias, unless the peaks of the Gaussians are not randomly distributed, but relatively close to the centres of the pixels due to selection effects from detection thresholds.

Disregarding the latter effect and noise, we can compute the variance of the maximum pixel method by integrating (the true flux-the average true flux)^2 = (the true flux-fudge_max_pix)^2 over the pixel area and dividing by the pixel area (= 1). This is just equal to integral of the true flux^2 over the pixel area - fudge_max_pix^2.

`tkp.steps` – Define logic for each pipeline stage

`tkp.steps.forced_fitting`

`tkp.steps.forced_fitting.`**`perform_forced_fits`**(*fit_posns*, *fit_ids*, *image_path*, *extraction_params*)

Perform forced source measurements on an image based on a list of positions.

Parameters

- **`fit_posns`** (*list*) – List of (RA, Dec) tuples: Positions to be fit.
- **`fit_ids`** – List of identifiers for each requested fit position.
- **`image_path`** (*str*) – path to image for measurements.
- **`extraction_params`** (*dict*) – source extraction parameters, as a dictionary.

Returns A matched pair of lists (serialized_fits, ids), corresponding to successfully fitted positions. NB returned lists may be shorter than input lists if some fits are unsuccessful.

Return type tuple

`tkp.steps.misc` Various subroutines used in the main pipeline flow.

We keep them separately to make the pipeline logic easier to read at a glance.

`tkp.steps.misc.`**`check_job_configs_match`**(*job_config_1*, *job_config_2*)

Check if job configs match, except dataset_id which we expect to change.

`tkp.steps.misc.`**`group_per_timestep`**(*images*)

groups a list of TRAP images per time step.

Per time step the images are order per frequency and then per stokes. The eventual order is:

(t1, f1, s1), (t1, f1, s2), (t1, f2, s1), (t1, f2, s2), (t2, f1, s1), ...) where:

- t is time sorted by old to new
- f is frequency sorted from low to high
- s is stokes, sorted by ID as defined in the database schema

Parameters **`images`** (*list*) – Images to group.

Returns List of tuples. The list is sorted by timestamp. Each tuple has the timestamp as a first element, and a list of images sorted by frequency and then stokes as the second element.

Return type list

`tkp.steps.misc.`**`load_job_config`**(*pipe_config*)

Locates the job_params.cfg in ‘job_directory’ and loads via ConfigParser.

`tkp.steps.misc.`**`setup_log_file`**(*log_dir*, *debug=False*, *basename=’trap.log’*)

sets up a catch all logging handler which writes to *log_file*.

Parameters

- **`log_file`** – log file to write

- **debug** – do we want debug level logging?
- **basename** – basename of the log file

tkp.steps.persistence This *step* is used for the storing of images and metadata to the database and image cache (mongodb).

tkp.steps.persistence.**create_dataset**(*dataset_id*, *description*)
Creates a dataset if it doesn't exists Note: Should only be used in a master recipe :returns: the database ID of this dataset

tkp.steps.persistence.**extract_metadatas**(*images*, *rms_est_sigma*, *rms_est_fraction*)
Extracts metadata and rms_qc values from the list of images.

Parameters

- **images** – list of image urls
- **rms_est_sigma** – used for RMS calculation, see *tkp.quality.statistics*
- **rms_est_fraction** – used for RMS calculation, see *tkp.quality.statistics*

Returns a list of metadata's. The metadata will be False if extraction failed.

tkp.steps.persistence.**image_to_mongodb**(*filename*, *hostname*, *port*, *db*)
Copy a file into mongodb

tkp.steps.persistence.**node_steps**(*images*, *image_cache_config*, *rms_est_sigma*, *rms_est_fraction*)
this function executes all persistence steps that should be executed on a node. Note: Should only be used in a node recipe

tkp.steps.persistence.**store_images**(*images_metadata*, *extraction_radius_pix*, *dataset_id*)
Add images to database. Note that all images in one dataset should be inserted in one go, since the order is very important here. If you don't add them all in once, you should make sure they are added in the correct order e.g. sorted by observation time.

Note: Should only be used in a master recipe

Parameters

- **images_metadata** – list of dicts containing image metadata
- **extraction_radius_pix** – (float) Used to calculate the 'skyregion'
- **dataset_id** – dataset id to be used. don't use value from parset file since this can be -1 (TraP way of setting auto increment)

Returns the database ID of this dataset

tkp.steps.prettyprint Pretty print the classified transients

tkp.steps.quality All generic quality checking routines.

tkp.steps.quality.**reject_check**(*image_path*, *job_config*)

checks if an image passes the quality check. If not, a rejection tuple is returned.

NOTE: should only be used on a NODE

Parameters

- **id** – database ID of image. This is not used but kept as a reference for distributed computation!
- **image_path** – path to image
- **parset_file** – parset file location with quality check parameters

Returns (rejection ID, description) if rejected, else None

`tkp.steps.quality.`**`reject_image`**(*image_id*, *reason*, *comment*)
Adds a rejection for an image to the database

NOTE: should only be used on a MASTER node

tkp.steps.source_extraction

class `tkp.steps.source_extraction.`**`ExtractionResults`**(*sources*, *rms_min*, *rms_max*)

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

`__repr__`()
Return a nicely formatted representation string

rms_max
Alias for field number 2

rms_min
Alias for field number 1

sources
Alias for field number 0

`tkp.steps.source_extraction.`**`extract_sources`**(*image_path*, *extraction_params*)
Extract sources from an image.

Parameters

- **image_path** – path to file from which to extract sources.
- **extraction_params** – dictionary containing at least the detection and analysis threshold and the association radius, the last one a multiplication factor of the de Ruiter radius.

Returns list of ExtractionResults named tuples containing source measurements, min RMS value and max RMS value

`tkp.testutil` – helper functions for writing tests

tkp.testutil.db_queries A collection of back end db query subroutines used for unittesting

`tkp.testutil.db_queries.`**`convert_to_cartesian`**(*conn*, *ra*, *decl*)
Returns tuple (x,y,z)

`tkp.testutil.db_queries.`**`get_assoc_entries`**(*db*, *runcat_id*)
Return the full history of variability indices for a runcat entry, ordered by time.

tkp.testutil.db_subs

class tkp.testutil.db_subs.**ExtractedSourceTuple**(*ra*, *dec*, *ra_fit_err*, *dec_fit_err*, *peak*, *peak_err*, *flux*, *flux_err*, *sigma*, *beam_maj*, *beam_min*, *beam_angle*, *ew_sys_err*, *ns_sys_err*, *error_radius*, *fit_type*, *chisq*, *reduced_chisq*)

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

__repr__()
Return a nicely formatted representation string

beam_angle
Alias for field number 11

beam_maj
Alias for field number 9

beam_min
Alias for field number 10

chisq
Alias for field number 16

dec
Alias for field number 1

dec_fit_err
Alias for field number 3

error_radius
Alias for field number 14

ew_sys_err
Alias for field number 12

fit_type
Alias for field number 15

flux
Alias for field number 6

flux_err
Alias for field number 7

ns_sys_err
Alias for field number 13

peak
Alias for field number 4

peak_err
Alias for field number 5

ra
Alias for field number 0

ra_fit_err
Alias for field number 2

`reduced_chisq`
Alias for field number 17

`sigma`
Alias for field number 8

class `tkp.testutil.db_subs.`**`MockSource`**(*template_extractedsource*, *lightcurve*)
Defines a MockSource for generating mock source lists.

(These can be used to test the database routines.)

The lightcurve-dict entries define the times of non-zero flux (we do not support time-ranges here, discretely defined datapoints are sufficiently complex for the current unit-test suite). In this case, any undefined datetimes requested will produce a zero-flux measurement. A defaultdict may be supplied to simulate a steady-flux source.

Parameters

- **`template_extractedsource`** (ExtractedSourceTuple) – This defines everything **except** the flux and significance of the extraction (i.e. position, fit error, beam properties, etc.).
- **`lightcurve`** (*dict*) – A dict mapping datetime -> flux value [Jy]. Any undefined datetimes will produce a zero-flux measurement. A defaultdict with constant-valued default may be supplied to represent a steady source, e.g.

  >>>MockSource(base_source, defaultdict(lambda:steady_flux_val))

`simulate_extraction`(*db_image*, *extraction_type*, *rms_attribute='rms_min'*)
Simulate extraction process, returns extracted source or none.

Uses the database image properties (extraction region, rms values) to determine if this source would be extracted in the given image, and return an extraction or None accordingly.

Parameters

- **`db_image`** (*int*) – Database Image object.
- **`extraction_type`** – Valid values are 'blind', 'ff_nd'. If 'blind' then we only return an extracted source if the flux is above rms_value * detection_threshold.
- **`rms_attribute`** (*str*) – Valid values are 'rms_min', 'rms_max'. Determines which rms value we use when deciding if this source will be seen in a blind extraction.

Returns ExtractedSourceTuple or None.

`value_at_dtime`(*dtime*, *image_rms*)
Returns an *extractedsource* for a given datetime.

If lightcurve is defined but does not contain the requested datetime, then peak, flux, sigma are all set to zero.

`tkp.testutil.db_subs.`**`deRuiter_radius`**(*src1*, *src2*)
Calculates the De Ruiter radius for two sources

`tkp.testutil.db_subs.`**`delete_test_database`**(*database*)
Use with caution!

NB. Not the same as a freshly initialised database. All the sequence counters are offset.

`tkp.testutil.db_subs.`**`example_dbimage_data_dict`**(***kwargs*)
Defines the canonical default image-data for unit-testing the database.

By defining this in one place we make it simple to make changes. A subset of the default values may be overridden by passing the keys as keyword-args.

Note that while RA, Dec and extraction radius are arbitrary, they should (usually) be close enough and large enough to enclose the RA and Dec of any fake source extractions inserted, since the association routines reject sources outside of designated extraction regions.

`tkp.testutil.db_subs.`**`example_extractedsource_tuple`**(*ra=123.123*, *dec=10.5*, *ra_fit_err=0.0013888888888888889*, *dec_fit_err=0.0016666666666666668*, *peak=0.015*, *peak_err=0.0005*, *flux=0.015*, *flux_err=0.0005*, *sigma=15.0*, *beam_maj=100.0*, *beam_min=100.0*, *beam_angle=45.0*, *ew_sys_err=20.0*, *ns_sys_err=20.0*, *error_radius=10.0*, *fit_type=0*, *chisq=5.0*, *reduced_chisq=1.5*)

Generates an example 'fake extraction' for unit testing.

Note that while RA and Dec are arbitrary, they should (usually) be close to the RA and Dec of any fake images used, since the association routines reject sources outside of designated extraction regions.

`tkp.testutil.db_subs.`**`generate_timespaced_dbimages_data`**(*n_images*, *timedelta_between_images=datetime.timedelta(1)*, ***kwargs*)

Generate a list of image data dictionaries.

The image-data dicts are identical except for having the taustart_ts advanced by a fixed timedelta for each entry.

These can be used to create known entries in the image table, for unit-testing.

A subset of the image-data defaults may be overridden by passing the relevant dictionary values as keyword args.

`tkp.testutil.db_subs.`**`get_newsources_for_dataset`**(*dsid*)

Returns dicts representing all newsources for this dataset.

Parameters **`dsid`** – Dataset id

Returns (list of dicts) Each dict represents one newsource. The dict keys are all the columns in the newsources table, plus the 'taustart_ts' from the image table, which represents the trigger time.

Return type list

`tkp.testutil.db_subs.`**`get_sources_filtered_by_final_variability`**(*dataset_id*, *eta_min*, *v_min*)

Search the database to find high-variability lightcurves.

Uses the variability associated with the last datapoint in a lightcurve as the key criteria.

Parameters

- **`dataset_id`** (*int*) – Dataset to search
- **`eta_min`** (*float*) – Minimum value of eta-index to return.
- **`v_min`** (*float*) – Minimum value of V-index to return.

Returns (list of dicts) Each dict represents a runningcatalog_flux entry matching the filter criteria.

Return type list

tkp.testutil.db_subs.**insert_image_and_simulated_sources**(*dataset*, *image_params*, *mock_sources*, *new_source_sigma_margin*, *deruiter_radius=3.7*)

Simulates the standard database image-and-source insertion logic using mock sources.

Parameters

- **dataset** – The dataset object
- **image_params** (*dict*) – Contains the image properties.
- **mock_sources** (*list of MockSource*) – The mock sources to simulate.
- **new_source_sigma_margin** (*float*) – Parameter passed to source-association routines.
- **deruiter_radius** (*float*) – Parameter passed to source-association routines.

Returns 3-tuple (image, list of blind extractions, list of forced fits).

tkp.testutil.db_subs.**lightcurve_metrics**(*src_list*)

Calculates various metrics for a lightcurve made up of source extractions

These are normally calculated internally in the database - this function serves as a sanity check, and is used for unit-testing purposes.

Returns a list of dictionaries, the nth dict representing the value of the metrics after processing the first n extractions in the lightcurve. The dict keys mirror the column names in the database, to make cross-checking of results trivial.

Final note: this function is very inefficient, recalculating over the first n extractions for each step. We could make it iterative, updating the weighted averages as we do in the database. However, this way provides a stronger cross-check that our iterative SQL approaches are correct - less chance of making the same mistakes in two languages!

tkp.testutil.decorators

tkp.testutil.decorators.**high_ram_requirements**()

Used to disable tests that break Travis due to out-of-memory issues.

tkp.testutil.decorators.**requires_test_db_managed**()

This decorator is used to disable tests that do potentially low level database management operations like destroy and create. You can enable these tests by setting the TKP_TESTDBMANAGEMENT environment variable.

tkp.testutil.mock Mock data objects for use in testing.

tkp.utility – miscellaneous utility routines

Submodules:

tkp.utility.containers – Generic container classes Container classes for the TKP pipeline.

These provide convenient means of marshalling the various types of data – lightcurves, detections, sources, etc – that the pipeline must handle.

class tkp.utility.containers.**ExtractionResults**

Container for the results of running source extraction on an ImageData object

class `tkp.utility.containers.`**`ObjectContainer`**

A container class for objects.

What sort of objects? Well, anything that has a position and we want to keep lists of, really. So detections (ie, an individual source measurement on an image), sources (ie all the detections of a given object in a given image stack) and lightcurves (ie, all the sources associated with a given object through time).

You probably don't want to use this on it's own: see ExtractionResults, TargetList or source for more useful derived classes.

`__iadd__`(*y*)
Not implemented.

`__imul__`(*y*)
Not implemented.

`__mul__`(*y*)
Not implemented.

`__rmul__`(*y*)
Not implemented.

`__setslice__`(*section*, *items*)
Not implemented.

`tkp.utility.coordinates` **– Coordinate routines** General purpose astronomical coordinate handling routines.

class `tkp.utility.coordinates.`**`CoordSystem`**

A container for constant strings representing different coordinate systems.

`FK4` **= 'B1950 (FK4)'**

`FK5` **= 'J2000 (FK5)'**

class `tkp.utility.coordinates.`**`WCS`**

Wrapper around pywcs.WCS.

This is primarily to preserve API compatibility with the earlier, home-brewed python-wcslib wrapper. It includes:

- A fix for the reference pixel lying at the zenith;
- Raises ValueError if coordinates are invalid.

`ORIGIN` **= 1**

`WCS_ATTRS` **= ('crpix', 'cdelt', 'crval', 'ctype', 'cunit', 'crota')**

`p2s`(*pixpos*)
Pixel to Spatial coordinate conversion.

Parameters **`pixpos`** (*list*) – [x, y] pixel position

Returns **ra** – Right ascension corresponding to position [x, y] dec (float): Declination corresponding to position [x, y]

Return type float

`s2p`(*spatialpos*)
Spatial to Pixel coordinate conversion.

Parameters **`pixpos`** (*list*) – [ra, dec] spatial position

Returns x – X pixel value corresponding to position [ra, dec] y (float): Y pixel value corresponding to position [ra, dec]

Return type float

`tkp.utility.coordinates.`**`alpha`**`(`*l*, *m*, *alpha0*, *delta0*`)`

Convert a coordinate in l,m into an coordinate in RA

Keyword arguments: l,m – direction cosines, given by (offset in cells) x cellsi (radians) alpha_0, delta_0 – centre of the field

Return value: alpha – RA in decimal degrees

`tkp.utility.coordinates.`**`alpha_inflate`**`(`*theta*, *decl*`)`

Compute the ra expansion for a given theta at a given declination

Keyword arguments: theta, decl are both in decimal degrees.

Return value: alpha – RA inflation in decimal degrees

For a derivation, see MSR TR 2006 52, Section 2.1 http://research.microsoft.com/apps/pubs/default.aspx?id=64524

`tkp.utility.coordinates.`**`alphasep`**`(`*ra1*, *ra2*, *dec1*, *dec2*`)`

Find the angular separation of two sources in RA, in arcseconds

Keyword arguments: ra1,dec1 - RA and Dec of the first source, in decimal degrees ra2,dec2 - RA and Dec of the second source, in decimal degrees

Return value: angsep - Angular separation, in arcseconds

`tkp.utility.coordinates.`**`altaz`**`(`*mjds*, *ra*, *dec*, *lat=52.9088*`)`

Calculates the azimuth and elevation of source from time and position on sky. Takes MJD in seconds and ra, dec in degrees. Returns (alt, az) in degrees.

`tkp.utility.coordinates.`**`angsep`**`(`*ra1*, *dec1*, *ra2*, *dec2*`)`

Find the angular separation of two sources, in arcseconds, using the proper spherical trig formula

Keyword arguments: ra1,dec1 - RA and Dec of the first source, in decimal degrees ra2,dec2 - RA and Dec of the second source, in decimal degrees

Return value: angsep - Angular separation, in arcseconds

`tkp.utility.coordinates.`**`convert_coordsystem`**`(`*ra*, *dec*, *insys*, *outsys*`)`

Convert RA & dec (given in decimal degrees) between equinoxes.

`tkp.utility.coordinates.`**`coordsystem`**`(`*name*`)`

Given a string, return a constant from class CoordSystem.

`tkp.utility.coordinates.`**`dectodms`**`(`*decdegs*`)`

Convert Declination in decimal degrees format to hours, minutes, seconds format.

Keyword arguments: decdegs – Dec. in degrees format

Return value: dec – list of 3 values, [degrees,minutes,seconds]

`tkp.utility.coordinates.`**`delta`**`(`*l*, *m*, *delta0*`)`

Convert a coordinate in l, m into an coordinate in Dec

Keyword arguments: l, m – direction cosines, given by (offset in cells) x cellsi (radians) alpha_0, delta_0 – centre of the field

Return value: delta – Dec in decimal degrees

`tkp.utility.coordinates.`**`deltasep`**`(`*dec1*, *dec2*`)`

Find the angular separation of two sources in Dec, in arcseconds

Keyword arguments: dec1 - Dec of the first source, in decimal degrees dec2 - Dec of the second source, in decimal degrees

Return value: angsep - Angular separation, in arcseconds

`tkp.utility.coordinates.`**`dmstodec`**`(`*decd*, *decm*, *decs*`)`

Convert Dec in degrees, minutes, seconds format to decimal degrees format.

Keyword arguments: decd, decm, decs – list of Dec values (d,m,s)

Return value: decdegs – Dec in decimal degrees

`tkp.utility.coordinates.`**`eq_to_cart`**`(`*ra*, *dec*`)`

Find the cartesian co-ordinates on the unit sphere given the eq. co-ords.

ra, dec should be in degrees.

`tkp.utility.coordinates.`**`eq_to_gal`**`(`*ra*, *dec*`)`

Find the Galactic co-ordinates of a source given the equatorial co-ordinates

Keyword arguments: (alpha,delta) – RA, Dec in decimal degrees

Return value: (l,b) – Galactic longitude and latitude, in decimal degrees

`tkp.utility.coordinates.`**`gal_to_eq`**`(`*lon_l*, *lat_b*`)`

Find the Galactic co-ordinates of a source given the equatorial co-ordinates

Keyword arguments: (l, b) – Galactic longitude and latitude, in decimal degrees

Return value: (alpha, delta) – RA, Dec in decimal degrees

`tkp.utility.coordinates.`**`hmstora`**`(`*rah*, *ram*, *ras*`)`

Convert RA in hours, minutes, seconds format to decimal degrees format.

Keyword arguments: rah,ram,ras – RA values (h,m,s)

Return value: radegs – RA in decimal degrees

`tkp.utility.coordinates.`**`jd2lst`**`(`*jd*, *position=None*`)`

Converts a Julian Date into Local Apparent Sidereal Time in seconds at a given position. If position is None, we default to the reference position of CS002.

Parameters

- **`jd`** (*float*) – Julian Date
- **`position`** (*casacore measure*) – Position for LST calcs.

`tkp.utility.coordinates.`**`julian2unix`**`(`*timestamp*`)`

Convert a modifed julian timestamp (number of seconds since 17 November 1858) to Unix timestamp (number of seconds since 1 January 1970).

Parameters **`timestamp`** (*numbers.Number*) – Number of seconds since the Unix epoch.

Returns Number of seconds since the modified Julian epoch.

Return type numbers.Number

`tkp.utility.coordinates.`**`julian_date`**`(`*time=None*, *modified=False*`)`

Calculate the Julian date at a given timestamp.

Parameters

- **`time`** (*datetime.datetime*) – Timestamp to calculate JD for.

- **`modified`** (*bool*) – If True, return the Modified Julian Date: the number of days (including fractions) which have elapsed between the start of 17 November 1858 AD and the specified time.

Returns Julian date value.

Return type float

`tkp.utility.coordinates.`**`l`**(*ra*, *dec*, *cra*, *incr*)
Convert a coordinate in RA,Dec into a direction cosine l

Keyword arguments: ra,dec – Source location cra – RA centre of the field incr – number of degrees per pixel (negative in the case of RA)

Return value: l – Direction cosine

`tkp.utility.coordinates.`**`lm_to_radec`**(*ra0*, *dec0*, *l*, *m*)
Find the l direction cosine in a radio image, given an RA and Dec and the field centre

`tkp.utility.coordinates.`**`m`**(*ra*, *dec*, *cra*, *cdec*, *incr*)
Convert a coordinate in RA,Dec into a direction cosine m

Keyword arguments: ra,dec – Source location cra,cdec – centre of the field incr – number of degrees per pixel

Return value: m – direction cosine

`tkp.utility.coordinates.`**`mjd2datetime`**(*mjd*)
Convert a Modified Julian Date to datetime via 'unix time' representation.

NB 'unix time' is defined by the casacore/casacore package.

`tkp.utility.coordinates.`**`mjd2lst`**(*mjd*, *position=None*)
Converts a Modified Julian Date into Local Apparent Sidereal Time in seconds at a given position. If position is None, we default to the reference position of CS002.

mjd – Modified Julian Date (float, in days) position – Position (casacore measure)

`tkp.utility.coordinates.`**`mjds2lst`**(*mjds*, *position=None*)
As mjd2lst(), but takes an argument in seconds rather than days.

Parameters

- **`mjds`** (*float*) – Modified Julian Date (in seconds)
- **`position`** (*casacore measure*) – Position for LST calcs

`tkp.utility.coordinates.`**`propagate_sign`**(*val1*, *val2*, *val3*)
casacore (reasonably enough) demands that a minus sign (if required) comes at the start of the quantity. Thus "-0D30M" rather than "0D-30M". Python regards "-0" as equal to "0"; we need to split off a separate sign field.

If more than one of our inputs is negative, it's not clear what the user meant: we raise.

Parameters **`val1`** (*float*) – (,val2,val3) input values (hour/min/sec or deg/min/sec)

Returns "+" or "-" string denoting sign, val1, val2, val3 (numeric) denoting absolute values of inputs.

Return type tuple

`tkp.utility.coordinates.`**`radec_to_lmn`**(*ra0*, *dec0*, *ra*, *dec*)

`tkp.utility.coordinates.`**`ratohms`**(*radegs*)
Convert RA in decimal degrees format to hours, minutes, seconds format.

Keyword arguments: radegs – RA in degrees format

Return value: ra – tuple of 3 values, [hours,minutes,seconds]

`tkp.utility.coordinates.`**`sec2days`**(*seconds*)
Seconds to number of days

`tkp.utility.coordinates.`**`sec2deg`**(*seconds*)
Seconds of time to degrees of arc

`tkp.utility.coordinates.`**`sec2hms`**(*seconds*)
Seconds to hours, minutes, seconds

`tkp.utility.coordinates.`**`unix2julian`**(*timestamp*)
Convert a Unix timestamp (number of seconds since 1 January 1970) to a modified Julian timestamp (number of seconds since 17 November 1858).

Parameters **`timestamp`** (*numbers.Number*) – Number of seconds since the modified Julian epoch.

Returns Number of seconds since the Unix epoch.

Return type numbers.Number

`tkp.utility.uncertain` – Uncertainty class

class `tkp.utility.uncertain.`**`Uncertain`**(*value=0.0*, *error=0.0*, **a*, ***t*)
Represents a numeric value with a known small uncertainty (error, standard deviation...).

Numeric operators are overloaded to work with other Uncertain or numeric objects. The uncertainty (error) must be small. Otherwise the linearization employed here becomes wrong.

`tkp.utility.sigmaclip` – Generic sigma clipping routine

Generic kappa-sigma clipping routine.

Note: this does *not* replace the specialized sigma_clip function in utilities.py

`tkp.utility.sigmaclip.`**`calcmean`**(*data*, *errors=None*)
Calculate the mean and the standard deviation of the mean

`tkp.utility.sigmaclip.`**`calcsigma`**(*data*, *errors=None*, *mean=None*, *axis=None*, *errors_as_weight=False*)
Calculate the sample standard deviation

Parameters **`data`** (*numpy.ndarray*) – Data to be averaged. No conversion from eg a list to a numpy.array is done.

Kwargs:

errors (numpy.ndarray, None): Eerrors for the data. Errors needs to be the same shape as data (this is different than for numpy.average). If you want to use weights instead of errors as input, set errors_as_weight=True. If not given, all errors (and thus weights) are assumed to be equal to 1.

mean (float): Provide mean if you don't want the mean to be calculated for you. Pay careful attention to the shape if you provide 'axis'.

axis (int): Specify axis along which the mean and sigma are calculated. If not provided, calculations are done over the whole array

errors_as_weight (bool): Set to True if errors are weights.

Returns (2-tuple of floats) mean and sigma

`tkp.utility.sigmaclip.`**`clip`**(*data*, *mean*, *sigma*, *siglow*, *sighigh*, *indices=None*)
Perform kappa-sigma clipping of data around mean

Parameters

- **`data`** (*numpy.ndarray*) – N-dimensional array of values
- **`mean`** (*float*) – value around which to clip (does not have to be the mean)
- **`sigma`** (*float*) – sigma-value for clipping
- **`siglow`** (*float*) – lower kappa clipping values
- **`sighigh`** (*float*) – higher kappa clipping values

Kwargs:

indices (numpy.ndarray): data selection by indices

Returns (numpy.ndarray) indices of non-clipped data

`tkp.utility.sigmaclip.`**`sigmaclip`**(*data*, *errors=None*, *niter=0*, *siglow=3.0*, *sighigh=3.0*, *use_median=False*)

Remove outliers from data which lie more than siglow/sighigh sample standard deviations from mean.

Parameters **`data`** (*numpy.ndarray*) – Numpy array containing data values.

Kwargs:

errors (numpy.ndarray, None): Errors associated with the data values. If None, unweighted mean and standard deviation are used in calculations.

niter (int): Number of iterations to calculate mean & standard deviation, and reject outliers, If niter is negative, iterations will continue until no more clipping occurs or until abs('niter') is reached, whichever is reached first.

siglow (float): Kappa multiplier for standard deviation. Std * siglow defines the value below which data are rejected.

sighigh (float): Kappa multiplier for standard deviation. Std * sighigh defines the value above which data are rejected.

use_median (bool): Use median of data instead of mean.

Returns (2-tuple) Boolean numpy array of indices indicating which elements are clipped (False), with the same shape as the input; number of iterations

Return type tuple

`tkp.utility.fits` – Some FITS routines

`tkp.utility.fits.`**`combine`**(*fitsfiles*, *outputfile*, *method='average'*)

Combine a set of FITS files, taking care of header keywords

Parameters

- **`fitsfiles`** (*list*) – FITS filenames to combine
- **`outputfile`** (*str*) – output FITS filename
- **`method`** (*str*) – average or sum the images

Returns None

`tkp.utility.fits.`**`convert`**(*casa_image*, *ms*, *fits_filename=None*)

Convert a CASA image to FITS, taking care of header keywords

Parameters

- **`casa_image`** (*casacore.images.image*) – CASA image

- **ms** (*casacore.tables.table*) – CASA measurement set
- **fits_filename** (*str*) – FITS output filename

Returns None

tkp.utility.fits.**fix_reference_dec**(*imagename*)
If the FITS file specified has a reference dec of 90 (or pi/2), make it infinitesimally less. This works around problems with ill-defined coordinate systems at the north celestial pole.

tkp.utility.memoize – Memoization decorator

class tkp.utility.memoize.**Memoize**(*funct*)
Decorator to cache the results of methods.

Examples in e.g. image.py:

```
@Memoize
def _grids(self):
    return self.__grids()
grids = property(fget=_grids, fdel=_grids.delete)
```

delete(*instance*)
Forget a memoized value

Root level *tkp.utility* functions that don't justify a submodule:

class tkp.utility.**adict**
Accessing dict keys like an attribute.

tkp.utility.**substitute_inf**(*value*, *sub='Infinity'*)
If value is not infinite, return value. Otherwise, return sub.

tkp.utility.**substitute_nan**(*value*, *sub=0.0*)
If value is not NaN, return value. Otherwise, return sub.

Top-level modules

tkp.main – Top-level pipeline logic flow

tkp.main Main pipeline logic is defined here.

The science logic is a bit entwined with celery-specific functionality. This is somewhat unavoidable, since how a task is parallelised (or not) has implications for the resulting logic.

In general, we try to keep functions elsewhere so this file is succinct. The exceptions are a couple of celery-specific subroutines.

tkp.main.**run**(*job_name*, *supplied_mon_coords=[]*)

1.5 Standalone Tools

1.5.1 PySE

Preamble

This document briefly describes the means by which the Transients Project source extraction & measurement code (henceforth `pyse.py`) may be used to obtain a list of sources found in a collection of images stored as FITS files. It does not attempt to act as a complete reference to the TKP codebase.

Introduction

Pyse provides the following capabilities:

- Identification of sources in astronomical images:
  - By a simple thresholding technique (ie, locating contiguous islands of pixels above some multiple of the noise in the image), or
  - By making use of a False Detection Rate (FDR) algorithm (Hopkins et al., AJ, 123, 1086, 2002).
- Deblending merged sources.
- Quick estimation of source properties based on the calculation of moments.
- Fitting of identified sources with elliptical Gaussians for accurate measurement of source properties.
- All measurements made are accompanied by a comprehensive error analysis.

For details of all algorithms implemented, the reader is referred to the PhD thesis by Spreeuw (University of Amsterdam, 2010).

It is worth emphasizing that there are a number of differences compared to projects such as, for example, BDSM. In particular, the `pyse.py` code is made available in the form or Python modules, primarily designed for integration into a pipeline or other script, rather than for use as an interactive analysis environment. Further, it is reasonable to assume that astronomical transients are unresolved, so the code does not attempt to decompose complex, extended sources into a multiple component model.

Command Line Usage

A script is available to make it possible to test the basic functionality of the `pyse.py` code. It does not make all the features listed above available.

Assuming `pyse.py` exists on your `$PATH`, it is involed by simply providing a list of filenames:

```
$ pyse.py file1.fits ... fileN.fits
```

For each file specified, a list of sources identified is printed to the screen.

By default, source extraction is carried out by thresholding: that is, identifying islands of pixels which exceed a particular multiple of the RMS noise.

A list of available command line option may be obtained with the `-h`/`--help` option:

Source extraction for radio-synthesis images

```
usage: pyse.py [-h] [--detection DETECTION] [--analysis ANALYSIS] [--fdr]
               [--alpha ALPHA] [--deblend-thresholds DEBLEND_THRESHOLDS]
               [--grid GRID] [--margin MARGIN] [--radius RADIUS] [--bmaj BMAJ]
               [--bmin BMIN] [--bpa BPA] [--force-beam]
               [--detection-image DETECTION_IMAGE] [--fixed-posns FIXED_POSNS]
               [--fixed-posns-file FIXED_POSNS_FILE] [--ffbox FFBOX]
               [--skymodel] [--csv] [--regions] [--rmsmap] [--sigmap]
               [--residuals] [--islands]
               files [files ...]
```

Positional arguments:

files	Image files for processing

Options:

--detection=10	Detection threshold
--analysis=3	Analysis threshold
--fdr=False	Use False Detection Rate algorithm
--alpha=0.01	FDR Alpha
--deblend-thresholds=0	Number of deblending subthresholds; 0 to disable
--grid=64	Background grid segment size
--margin=0	Margin applied to each edge of image (in pixels)
--radius=0	Radius of usable portion of image (in pixels)
--bmaj	Set beam: Major axis of beam (deg)
--bmin	Set beam: Minor axis of beam (deg)
--bpa	Set beam: Beam position angle (deg)
--force-beam=False	Force fit axis lengths to beam size
--detection-image	Find islands on different image
--fixed-posns	List of position coordinates to force-fit (decimal degrees, JSON, e.g [[123.4,56.7],[359.9,89.9]]) (Will not perform blind extraction in this mode)
--fixed-posns-file	Path to file containing a list of positions to force-fit (Will not perform blind extraction in this mode)
--ffbox=3.0	Forced fitting positional box size as a multiple of beam width.
--skymodel=False	Generate sky model
--csv=False	Generate csv text file for use in programs such as TopCat
--regions=False	Generate DS9 region file(s)
--rmsmap=False	Generate RMS map
--sigmap=False	Generate significance map
--residuals=False	Generate residual maps
--islands=False	Generate island maps

The `--detection` argument specifies the multiple of the RMS noise which is required for detection; ie, setting `--detection=5` is equivalent to requiring pixels used for detecting sources to be at 5 sigma.

The `--analysis` argument specifies the significance level used when performing fitting. It should be lower than `--detection`, such that once islands have been identified a larger number of pixels is included for the fitting process.

However, if the `--fdr` option is given, a False Detection Rate algorithm is used instead. In this case, an additional `--alpha` argument may be given to specify the α parameter (as defined by Hopkins).

Note that if `--fdr` is specified, any values given for `--detection` and `--analysis` are *not used*. Conversely, if `--fdr` is not specified, any value given for `--alpha` is *not used*.

If the `--regions` option is specified, a DS9-compatible region file is output showing the shapes & positions of the sources. It is named according to the input filename with the extension changed to `.reg`.

If the `--residuals` option is specified, a FITS file is produced showing the residuals left after the fitted sources have been subtraced from the input image. It is named according to the input filename with `.residuals` inserted before the extension.

If the `--islands` option is specified, a FITS file is produced showing the Gaussians which have been fitted in the data. It is named according to the input filename with `.islands` inserted before the extension. The sum of this file with that produced by `--residuals` above should total the input image.

If the `--skymodel` option is given, a skymodel file suitable for use with BBS will be generated. It is named according to the input filename with the extension changed to `.skymodel`.

If the `--csv` option is given, a comma-separated list of sources will be written to file. It is named according to the input filename with the extension changed to `.csv`.

If the `--rmsmap` option is given, a FITS file is produced showing the noise map which has been generated during the source-finding process. It is named according to the input filename with `.rms` inserted before the extension.

If the `--sigmap` option is given, a FITS file is produced showing the significance of each pixel: that is, the background-subtracted image pixel value divided by the RMS noise at that pixel. It is named according to the input filename with `.sigmap` inserted before the extension.

If the `--deblend` option is specified, `pyse.py` will attempt to separate composite sources into multiple components and fit each one independently. The number of subthresholds used in this process can be specified using the `--deblend-thresholds` argument. Refer to Spreeuw's thesis for a detailed description of the algorithm used.

`--bmaj`, `--bmin` and `--bpa` specify the shape of the restoring beam. They are equivalent to the `BMAJ`, `BMIN` and `BPA` FITS headers. Normally, the code will read the beam shape from the image metadata; however, if it is not available, it must be manually specified using these arguments or the process will abort.

When generating background and RMS maps of the image prior to source detection, it is segmented into a grid. The size of the grid can be specified using the `--grid` option. The optimal value is a compromise: it should be significantly larger than the most extended sources in the image, but small enough to account for small-scale variation across the image.

Sometimes, it is useful to exclude the edge regions of an image from processing. The `--margin` takes an argument given in pixels and masks off all portions of the image within the given distance of the edge before processing. The `--radius` argument is similar, but rather masks off all parts of the image more than the given distance from the centre. This options are cumulative.

If the `--force-beam` option is given, PySE will insist that all sources have axis lengths and position angles equal to the restoring beam parameters. This is (might be...) a good assumption if you are observing only point sources.

If the `--detection-image` option is specified, PySE will identify sources and the positions of pixels which comprise them on the deteciton image, but then use the corresponding pixels on the target images to perform measurements. Of course, the detection image and the target image(s) must have the same pixel dimensions. Note that only a single detection image may be specified, and the same pixels are then used on all target images. Note further that this `--detection-image` option is incompatible with `--fdr`.

It is possible to configure PySE to perform a fit to user-specified positions in the image _rather_ than "blindly" locating sources and attempting to fit them. (Note that it is not possible to do both at once: that requires invoking PySE twice.) This mode may be invoked either by using either of the `--fixed-posns` or `--fixed-posns-file` options. The former directly reads a list of positions from the command line; the latter accepts a filename, and reads the positions to fit from that. In both cases, the positions themselves are provided in JSON format, and should consist of a _list_ of RA, declination _pairs_ given in decimal degrees.

When fitting to a fixed position, a square "box" of pixels is chosen around the requested position, and the optimization procedure allows the source position to vary within that box. The size of the box may be changed with the `--ffbox` option. Note that this parameter is given in units of the major axis of the beam.

All of these arguments are optional (with the caveat that the beam shape must be provided if not included with the image).

Output Definition

The Gaussian fitted to sources is defined as:

$$peak * \exp(\ln(2.0) * ((x\cos(\theta) + y\sin(\theta))/semiminor)^2 + ((y\cos(\theta) - x\sin(\theta))/semimajor)^2)$$

In other words:

- x and y are the Cartesian coordinates of the centre of the Gaussian;
- $peak$ is the value at the centre of the Gaussian;
- $theta$ is the position angle of the major axis measured counterclockwise from the y axis;
- $semimajor$ and $semiminor$ are the half-widths at half-maximum of the Gaussian along its major and minor axes, respectively.

1.5.2 Image Metadata Injection

Preamble

In order to images through the TraP, they are required to provide a comprehensive set of metadata, including details such as observation time, frequency and the shape of the restoring beam.

Unfortunately, not all images are produced with all the required metadata embedded. The metadata injection tool, `tkp-inject.py`, makes it possible to annotate images with a user-supplied set of metadata. This can be used to either replace incorrect metadata provided with an image, or to provide it from scratch.

Configuration

`tkp-inject.py` is configured by means of a `ConfigParser` format file named `inject.cfg` in the users job directory. See the documentation on *pipeline configuration* for details.

The default `inject.cfg` file contains the following settings:

```
[inject]
taustart_ts = "2007-07-20T14:18:09.909001" ; start time
freq_eff = 128613281.25                   ; frequency in Hz
freq_bw = 1940917.96875                   ; bandwidth in Hz
tau_time = 58141509.156864166             ; integration time
antenna_set = "HBA_DUAL"                    ; which antenna set is used
subbands = 10
ncore = 41                                ; number of core stations
nremote = 0                                 ; number of remote stations
nintl = 0                                 ; number of international stations
subbandwidth = 128613281.25
bmaj = 1.9211971282958984
bmin = 1.7578132629394532
bpa = 1.503223674140207
itrf_position_x = 3.8269e+06
itrf_position_y = 460979
itrf_position_z = 5.06466e+06
```

Any or all of these may be changed by the user to reflect their requirements.

1.6 Bibliography

Bertin, E. and Arnouts, S. *SExtractor: Software for source extraction*, A&AS 117, 393–404, 1996. Scheers, L.H.A. *Transient and variable radio sources in the LOFAR sky: An architecture for a detection framework*. PhD thesis, University of Amsterdam, 2011.

- Scheers (2011) in University of Amsterdam Repository

Spreeuw, J.N. *Search and detection of low frequency radio transients*. PhD thesis, University of Amsteram, 2010.

- Spreeuw (2010) in University of Amsterdam Repository

Swinbank, J.D. et al. *The LOFAR Transients Pipeline*. In prep.

- Github Repository (members only)

1.7 Colophon

This document was generated using Sphinx and a theme based on the Python documentation. The latest version is available from the LOFAR Transients KSP documentation repository.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Symbols