# AOFlagger

*Release 3.2*

**Mar 20, 2023**

# Contents

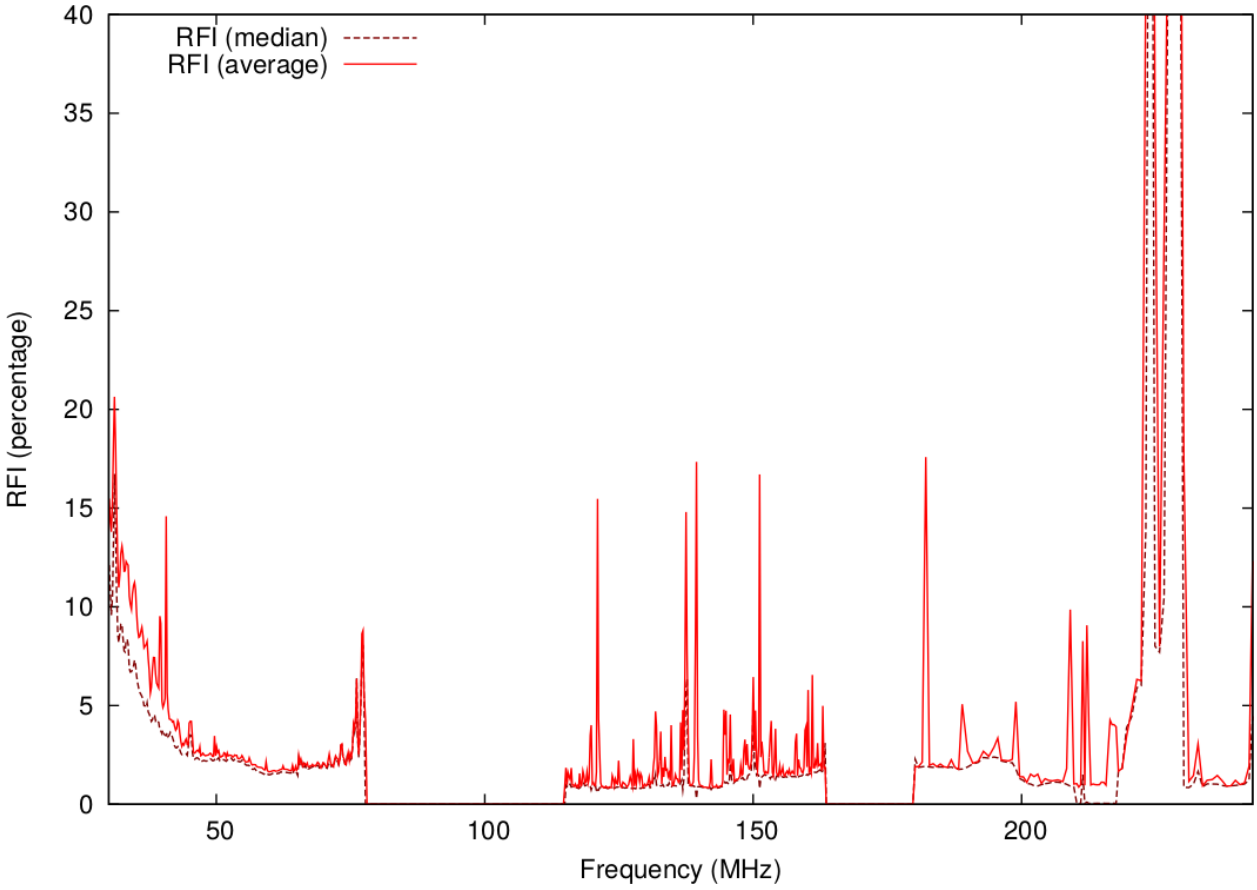
This is the manual for the AOFlagger software suite. The AOFlagger software can find and remove radio-frequency interference (RFI) in radio astronomical observations. The Lua language is used for writing flexible flagging strategies. The tools are applicable to a wide range of telescopes.

Contents:

# CHAPTER 1

# Introduction

AOFlagger makes it possible to detect radio-frequency interference in radio-observations. In this process, often called "flagging" the data, samples that are contaminated by interference are marked. As shown by the figure below, the frequencies covered by telescopes like LOFAR and the MWA are considerably affected by radio-frequency interference (RFI). Efficient RFI detection is essential to obtain high quality images.



The AOFlagger software is aimed at being a fast and accurate flagger. Implemented algorithms such as the SumThreshold method, background fitting techniques (smoothing, sliding window, median filters, high-pass filters) and morphological operators can be combined into strategy scripts. These are written in the Lua language. The default strategy performs well on a wide range of telescope observations. Further tweaking to accomodate for specific features of a telescope can sometimes improve the results.

The software consists of the flagger library (`libaoflagger`) that can be integrated into observatory pipelines by using the Application Programming Interface (API). It also provides several programs to execute the flagger on measurement sets, tweak strategies and visualize results. Tools are provided that can for example plot time-frequency diagrams and power spectra, both interactively or from scripts. The two main programs are *aoflagger* and *rfigui*. Furthermore, aoqplot can be used to visualize plots (*interactively* or *scripted*). The API is available as for programs written in C++ and Python.

AOFlagger is used by default for the LOFAR, MWA and Apertif radio telescope. The software can run in a fully automated way, but a graphical interface (rfigui) is provided to analyse results and tweak the strategy. The preferred input file format is the Casa Measurement Set (MS) format, but single dish SDFits files are also supported.

I believe that the AOFlagger is the best available flagger, both in terms of accuracy and speed. It has been succesfully used on several interferometric telescopes, including LOFAR, WSRT, Apertif, VLA, GMRT, ATCA and MWA, and the single-dish telescopes Parkes and Arecibo 305m.

## 1.1 History

The AOFlagger was originally written as part of my PhD thesis for the LOFAR Epoch of Reionization key science project, which needed a fast flagger with high accuracy. Since then it was made more generic and functionality was added to work on data from other observatories and at other frequency ranges.

CHAPTER 2

# Getting started

## 2.1 Installation

### 2.1.1 Installation on Redhat

This page describes installing AOFlagger 2.14 on RHEL 7.6 (Thanks to Leonardo Saavedra from NRAO)

aoflagger depends on many packages, most of then are provided by RHEL 7.6, but you will need to install the modern version of them. In particular: **fftw** and **boost**

- https://gitlab.com/aroffringa/aoflagger
- http://www.fftw.org
- https://github.com/casacore/casacore
- https://www.boost.org/

#### Install RPMs

You need install hdf from epel (or install from the sources as detailed below), so you need to set EPEL repo.

```
yum install hdf hdf-devel
yum install gtkmm30 gtkmm30-devel
```

#### Set environment

```
mkdir ~/Downloads2/aoflagger
cd ~/Downloads2/aoflagger
wget -c http://www.fftw.org/fftw-3.3.8.tar.gz
wget -c https://github.com/casacore/casacore/archive/v3.1.1.tar.gz
wget -c https://sourceforge.net/projects/aoflagger/files/aoflagger-2.14.0/aoflagger-2.
↪14.0.tar.bz2
```

(continues on next page)

(continued from previous page)

```
wget -c https://dl.bintray.com/boostorg/release/1.68.0/source/boost_1_68_0.tar.gz
LOCAL5=/opt/local5
```

### Install FFTW 3.3.8

```
tar xzvf fftw-3.3.8.tar.gz
cd fftw-3.3.8/
./configure --prefix=${LOCAL5} --enable-threads --enable-openmp  --enable-shared
make -j `nproc`
make install
cd ..
```

### Install Casacore 3.3.1

### Set new environment

```
export LD_LIBRARY_PATH=${LOCAL5}/lib:$LD_LIBRARY_PATH
export PATH=$PATH:${LOCAL5}/bin
```

```
tar xzvf v3.1.1.tar.gz
cd casacore-3.1.1/
mkdir build
cd build
cmake ../ -DCMAKE_PREFIX_PATH=${LOCAL5}
make -j `nproc`
make install
vim cmake_install.cmake <-- modified CMAKE_INSTALL_PREFIX to point to ${LOCAL5}
make install
```

### Install boost 1.68

Previous to compile boost, you have to make sure that your python installation uses a 4-byte representation for Unicode characters. To check that, you can run the following script, if it reports 'UCS2 Build', you have to either install a new python or use other python version.

https://docs.python.org/2/faq/extending.html#can-i-create-an-object-class-with-some-methods-implemented-in-c-and-others-in-pytho

```
python
cat CheckUnide.py
import sys

if sys.maxunicode > 65535:
    print ('UCS4 Build')
else:
    print ('UCS2 Build')
```

```
python CheckUnide.py
UCS4 Build
```

```
tar xzvf boost_1_68_0.tar.gz
cd boost_1_68_0
./bootstrap.sh --with-python=python --prefix=${LOCAL5} --with-icu
./b2
./b2 link=shared install --prefix=${LOCAL5}
(cd ${LOCAL5}/lib ; ln -s libboost_python27.so  libboost_python.so)
cd ..
```

#### Install HDF5

This is optional, you can skip if you have hdf5 from Epel. Download hdf-1.10.5.tar from https://www.hdfgroup.org/downloads/hdf5/source-code/

```
./configure --enable-cxx --with-szlib --enable-shared --enable-hl --disable-silent-
↪rules --enable-fortran --enable-fortran2003 --prefix=${LOCAL5}
make -j `nproc`
make check
make install
make installcheck
```

#### Install AOFlagger 2.14

```
tar xjvf aoflagger-2.14.0.tar.bz2
cd aoflagger-2.14.0
mkdir build
cd build
cmake ../ -DCMAKE_PREFIX_PATH=${LOCAL5} -DCMAKE_INSTALL_PREFIX={LOCAL5}
make -j `nproc`
vim cmake_install.cmake  <-- modified (or make sure) CMAKE_INSTALL_PREFIX  point to $
↪{LOCAL5}
make install
```

#### Check aoflagger version

```
which python
/opt/local5/bin/python
aoflagger --version
AOFlagger 2.14.0 (2019-02-14)
```

The easiest way to install AOFlagger is from a binary package. Binary packages for Debian and Ubuntu are available. For other distributions, AOFlagger can be compiled from source.

### 2.1.2 Binary distributions

Some distributions provide binaries. These might not always be the latest version, but if you don't require specific features from later versions, it is highly recommended to install aoflagger from binaries. If you do need a newer version, or if your distribution doesn't provide a binary, then AOFlagger should be compiled from source.

#### Debian/Ubuntu package

Debian and Ubuntu have binaries available. To install AOFlagger on Debian/Ubuntu system, run:

```
sudo apt-get install aoflagger
```

#### Kern

Kern is a distribution that provides the most commonly used astronomical software. It also provides AOFlagger, but is currently out of date.

### 2.1.3 Compilation

The AOFlagger source-code repository can be found here: https://gitlab.com/aroffringa/aoflagger

If you are compiling AOFlagger for the first time, take this into account:

- It is easiest to install AOFlagger on a Linux machine. Ubuntu, Debian and derived distributions are most easy to install to. Red Hat and derived distributions (e.g. CentOS) are harder, in particular when installing the graphical interface. Installation on MacOS is possible too, but installing the dependencies in the right way can be a lot of work. A Docker or Singularity container might in that case be a solution.
- It is easier when you can get root access to install distribution packages. This is particularly the case if you want the graphical interface.
- If you run into problems compiling the package, please ask colleagues or a local adminstrator first.
- If you can't solve an issue, please mail me. Please don't mail me with problems to install or compile Casacore. I get a lot of mail about the packages I wrote, so be sure that you have tried a few options yourself first. That said, if you cannot solve the issue, I'm more than happy to help.

#### On Red Hat

The following document lists some instructions that can be helpful for installing AOFlagger on Red Hat: *Installation on Redhat*.

#### Prerequisites

The following libraries are required to build the AOFlagger:

- Casacore, for opening measurement sets. Version >=2.0 is required.
- Lua, for scripting.
- FFTW, used to perform Fourier transformations.
- Boost, used for date and time calculations and some other general functionalities.
- LAPACK, for linear algebra, such as singular value decomposition.
- CFITSIO, for reading and writing FITS files.
- Gtkmm (only for rfigui and other graphical programs, but you probably want that). Version 3.10 or later is required (since AOFlagger 2.7; AOFlagger version 2.5 required gtkmm >= 3.0; earlier versions required gtkmm 2.x).
- GNU science library

- libpng
- libxml, used for saving scripts / configurations.
- Python 3 libraries.

The Gtkmm libraries and headers are required for the gui but not for the console executable. If they are not present, cmake will warn about it, but the console programs can still be compiled.

To compile Gtkmm "by hand" is discouraged; it has many dependencies. It is better to use a package manager (or ask your system administrator). Here is an apt command for Debian / Ubuntu system that installs all dependencies:

```
apt-get install   \
  cmake \
  build-essential \
  pkg-config \
  casacore-data casacore-dev \
  libblas-dev liblapack-dev \
  python3 \
  libboost-date-time-dev \
  libboost-test-dev \
  libcfitsio-dev \
  libfftw3-dev \
  libgsl-dev \
  libgtkmm-3.0-dev \
  liblua5.3-dev \
  libpng-dev  \
  libpython3-dev \
  libxml2-dev \
```

**Note:** Certain versions of Ubuntu ship the Lapack packages as "liblapack3gf".

### Compilation commands

The following commands are a quick summary of compilation and installation, assuming your working directory is the AOFlagger root directory:

```
mkdir build
cd build/
cmake ../
make -j 8
sudo make install
```

This will normally work when you have installed all libraries in their default locations. If you have not done so, you might need to tell cmake where the libraries are. The easiest way to do this is by adding search paths to cmake, e.g.:

```
cmake ../ -DCMAKE_PREFIX_PATH=/opt/cep/casacore/
```

This will include `/opt/cep/casacore` to the search path. Multiple paths can be specified by separating them with a colon ':'. Alternatively, if for some reason you need to specify individual libraries, the syntax is:

```
cmake ../ -DCASA_INCLUDE_DIR=/home/anoko/casacore/include -DCASA_MS_LIB=.. [..etc..]
```

One can set the `-D...` defines to the appropriate paths when CMake can not automatically find them. The 'make install' statement will install the binaries *aoflagger*, *rfigui*, *aoqplot* and *aoquality*, and the library `libaoflagger`

together with its header `aoflagger.h`. If you do not have administrator or sudo rights, you can install the software to a different location by specifying a `CMAKE_INSTALL_PREFIX` when calling cmake, e.g.:

```
cmake ../ -DCMAKE_INSTALL_PREFIX=/home/esmeralda/
```

### Building the Python interface

The Python lib that provides the *Python interface* is build by default. The Python aoflagger lib should be installed in your Python path after `make install`. Unlike the C++ library, the Python library does not start with `lib`. E.g., on my machine, the two libraries are:

- `lib/libaoflagger.so.0` -> C++ library
- `lib/aoflagger.cpython-38-x86_64-linux-gnu.so` -> Python library

Bottomline, if a machine is set up normally, one should be able to run `import aoflagger` from Python after installing AOFlagger.

### Building a machine independent binary

To build AOFlagger so that it can run on any machine, add `-DPORTABLE=True` to the cmake command line:

```
cmake ../ -DPORTABLE=True
```

## 2.2 Overview of programs

AOFlagger provides several programs. The *rfigui program* makes it possible to perform a "baseline by baseline" analysis of measurement sets (and other supported formats). When analyzing data from a telescope for the first time, it is recommended to first analyze the data with the `rfigui`, and use it to test and tweak flagging strategies, or if necessary to use it to *design your own*.

The *aoflagger program* runs strategies in a non-interactive, automated fashion. It is a command line program that, unlike `rfigui`, does not require a graphical terminal. It applies a flagging strategy to a dataset, and is optimized to run as fast as possible. `aoflagger` is therefore typically used to flag datasets "in production", once a good flagging strategy has been found with the `rfigui`.

After flagging, *aoqplot* can be used to inspect the results.

If the strategy works well for a number of observations, it might be desirable to integrate the flagging into an observational pipeline. This could involve calling `aoflagger` during the processing, but sometimes it might be desirable to have a tighter coupling, e.g. to keep data in memory. For this, the *C++ interface* or *Python interface* can be used. Another reason to use the API is to run a flagging strategy on a dataset with a format that is not natively supported by `aoflagger`, e.g. a numpy array.

CHAPTER 3

# Usage

The following chapters cover how to use the various AOFlagger tools:

## 3.1 Using `rfigui`

The program's main window shows a time-frequency plot (dynamic spectrum) of a selected baseline and/or polarization, and from there it is possible to analyse the data in various ways.

A few of its key features:

- Analyse Casa Measurement Sets, UVFits files and other supported formats;
- Make time-frequency diagrams and other cuts through the data;
- Export plots to pdfs (interactively or scripted);
- Execute and tweak flagging strategies;
- Visualize the uv plane and/or image plane for selected data.

### 3.1.1 Main window

After starting `rfigui`, the main window appears.

## 3.2 Using `aoflagger`

The '`aoflagger`' executable can be used to flag a measurement set. It is a stand-alone command-line tool, i.e., without a graphical interface, and is designed to work fast and non-interactively.

To flag a measurement set:

```
aoflagger [options] <observation.ms>
```

The list of options can be retrieved by running `aoflagger` without parameters.

### 3.2.1 Reading modes

The AOFlagger has several reading modes for opening measurement sets:

- Direct mode, enabled with `-direct-read`. This reads data when it is necessary. This results in scanning through the measurement set a lot, which can be very slow.
- Reordering mode, enabled with `-reorder`. Reorders the measurement set to disk, which is reasonably fast, but requires disk space. This used to be called 'indirect' mode.
- Memory mode, enabled with `-memory-read`: fastest mode, but only possible for sets that fit in memory.
- Automatic mode, enabled with `-auto-read-mode`: selects either reordering or memory mode based on available memory. This is the default.

---

**Note:** Before AOFlagger version 3.3, the automatic reading mode selected the direct reading mode instead of the reordering mode when not enough memory is available.

---

### 3.2.2 The reordering mode

This approach requires the size of the measurement set in free disk space (this would be the uncompressed size in case the measurement set is compressed with Dysco). AOFlagger will use reordering by specifying the "-reorder" parameter. In this case, the measurement set will be rewritten to a temporary location. Here is an example how to run AOFlagger in this mode:

```
lce032:/data/offringa/temp$ aoflagger -reorder SB4.MS
```

Please note that the current working directory will be used as a temporary storage location! Thus by running `aoflagger` as in the above command, temporary files will be created in `/data/offringa/temp`, and these will take up a volume equal to the size of the measurement set. Using fast storage (e.g. SSD drives or RAID setups) as temporary location will speed up AOFlagger with reordering considerably.

If you specify `-v` on the command line, you will see when the file is reordered. Here is an example output of `aoflagger` during initializing, when using the reordering mode:

```
[..]
0% : +-+-+-For each baseline...
Estimate of memory each thread will use: 1 MB.
Will process 91 baselines.
0% : +-+-+-+-Initializing...
Initializing observation times...
Opening temporary files.
Pre-allocating 16 MB...
Pre-allocating 2 MB...
Reordering data set...
Done reordering data set of 18 MB in 0.917 s (19.9788 MB/s)
[..]
```

If AOFlagger were to crash, please be sure to remove the temporary files (and send me a bug report if it is a bug).

## 3.3 Using aoqplot

aoqplot is an interactive tool to browse through quality statistics, but also allows non-interactive access to plots that can e.g. be stored automatically to disk. These quality statistics are added to a measurement set by the *aoquality* tool or within certain pipelines (e.g. DPPP, cotter).

The normal way of running aoqplot is:

```
aoqplot <ms1> [<ms2>..]
```

This will read the statistics from the given measurement set. If multiple measurement sets are specified, the statistics will be combined.

The following statement saves all standard deviation plots without user interaction:

```
aoqplot observation.ms -save Obs StandardDeviation
```

The non-interactive options for running aoqplot are described on the page *scripted plotting*.

## 3.4 Using aoquality

The aoquality program is a command-line tool to query or modify the quality statistics in the table. It is more or less a command-line counterpart to *aoqplot*.

### 3.4.1 What quality statistics are

Quality statistics are optional tables in a measurement set that describe statistics of the visibility in the measurement set. These can be computed from the visibilities and are therefore redundant, but they are much smaller in size, and therefore alleviate the need to read an entire set just to get some first idea of the quality of the visibilities. They can be calculated during flagging (using the lua *collect_statistics()* function), which avoids reading the data again just for getting the statistics.

---

**Note:** If the visibilities are changed, the quality statistics become out of date. This can be desirable in some cases, e.g. to keep the statistics of the raw, high-resolution data before averaging.

---

### 3.4.2 (Re)collecting statistics

To collect statistics, the following command can be used:

```
aoquality collect [-d <column name>] observation.ms
```

This will read all the visibilities and add the quality statistics tables to the measurement set afterwards. If quality tables already exist, they are overwritten. If no column name is specified, the DATA column is used.

### 3.4.3 Extracting and combining statistics

Normally, statistics are written inside a measurement set. The aoquality tool allows the extraction of statistics and to write these into separate (small) files, so that it becomes possible to store statistics separately from the measurement set. The stored statistics have the form of an empty measurement set with quality tables and just enough metadata

to interpret the statistics. The `aoquality combine` option can be used to create such separate statistic sets. The recommended extension for extracted statistics is `.qs`. For example, to extract statistics from a measurement set and write them in a separate, use:

```
aoquality combine mystatistics.qs input.ms
```

This would create `mystatistics.qs` and fill it with the quality statistics from `input.ms`. It is also possible to specify multiple measurement sets. In this case, the statistics are combined and then written to the output file, for example:

```
aoquality combine mystatistics.qs \
  observation1.ms observation2.ms observation3.ms
```

This would combine the statistics from the three observations and write these to `mystatistics.qs`. For quality statistics operations, the `.qs` statistics can be used any time that a `.ms` can be used, so `aoquality query` commands can be applied to it, they can be further combined with `aoquality combine`, and *aoqplot* can be used to view the contents of the `.qs` statistics.

### 3.4.4 Querying statistics

There are several dimensions over which statistics, such as the standard deviation, are stored and can be queried: over time (integrated over all other dimensions), frequency or baseline. To query these, the `aoquality` tool accepts these commands:

- `aoquality query_b`: Query the statistics per baseline (integrated over all other dimensions)
- `aoquality query_f`: Query the statistics per channel (integrated over all other dimensions)
- `aoquality query_t`: Query the statistics over time (integrated over all other dimensions)
- `aoquality query_g`: Query a statistic integrated over all dimensions

Each of these commands uses the same syntax. We will use `query_f` as an example. To query the standard deviation per channel, one would issue this command:

```
aoquality query_f StandardDeviation observation.ms
```

As shown, after `query_f` the type of statistic is given, followed by the name of the measurement set. A few other common statistics are `Mean`, `Variance`, `Count` (=nr of visibilities), `RFIRatio` and `DStandardDeviation` (=frequency-differenced standard deviation).

Here is an example output:

This shows the frequency in the first column, the standard deviation of the real values of XX in the second column, followed by imaginary, then real of XY, etc.

### 3.4.5 Other options

The `aoquality` command has a few more options. These can be queried by running `aoquality` without parameters.

## 3.5 Quality statistics

To be written

## 3.6 Scripted plotting

Storing plots that describe the (RFI) quality of an observation can be useful for quality assessment. Both `rfigui` and `aoqplot` have options that make it possible to save certain statistics or overview images. The command-line options of these tools give access to these plotting capabilities from non-interactive pipelines or scripts. When parameters are given on the command line that save images, the `rfigui` and `aoqplot` tools will not open their default window and therefore do not require an X windows system to be running[1]. In other words, the tools will run as non-interactive command line programs.

Both `rfigui` and `aoqplot` provide parameter `--help`, which will show a list of supported options.

### 3.6.1 `rfigui`

The `rfigui` tool can be used to save time-frequency "heatmap" images, that show the flux of the correlations in a two-dimensional time and frequency display. The syntax is as follows:

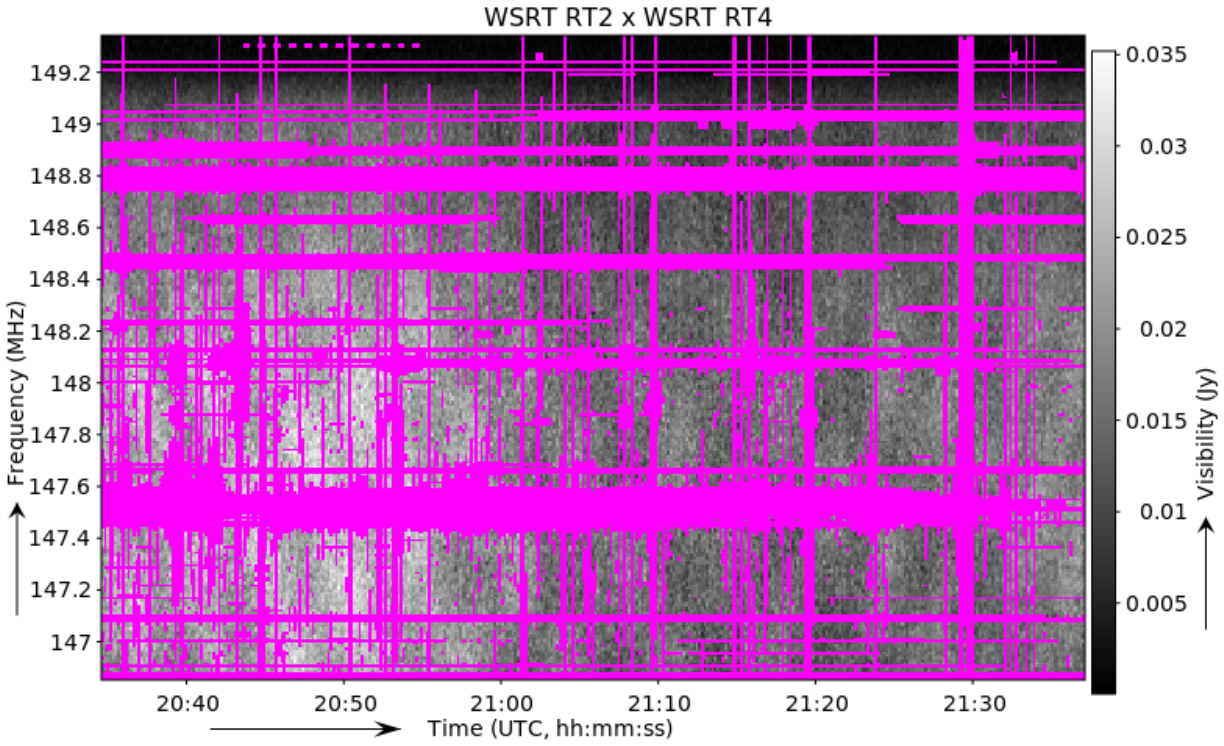
```
rfigui \
  -save-baseline <filename> <ant1> <ant2> <band> <seqindex>
```

Filename should end with an image extension. Supported formats are `.png`, `.pdf` and `.svg`. Note that `.pdf` and `.svg` are of higher quality than `.png` files. The created plots will by default be made from the "`DATA`" column. However, a different column can be selected with the option `-data-column <name>`. The following example will save the correlated data from antennas 1 and 3:

```
rfigui \
  -data-column CORRECTED_DATA \
  -save-baseline WSRT-RT2xRT4.pdf 1 3 0 0 3C196_spw5_sub1.MS/
```

Which creates the following pdf:

[1] Older versions of `rfigui` and `aoqplot` would require an X window system to be running even when running in command line mode. However, that was resolved in AOFlagger [version 2.10](changelog-2.10.0).




It is also possible to repeat the '`-save-baseline`' option to save multiple images at once. This saves quite a few computations over saving the baselines one by one with separate `rfigui` calls.

### 3.6.2 `aoqplot`

The `aoqplot` syntax to save plots is as follows:

```
aoqplot -save <filename prefix> <statistic name>
```

Note that unlike `rfigui`, the `aoqplot` tool saves several images at once. The plots created are i) a per-baseline matrix plot; ii) a per antenna plot; iii) a spectrum; iv) a time-plot; and v) a time-frequency heatmap. The statistic name parameter is a case-sensitive name of the statistic that will be plotted over antenna/time/frequency, etc. The common statistics are: `StandardDeviation`, `DStandardDeviation` (=stddev of difference between channels), `Variance`, `Mean`, `RFIPercentage`, `RFIRatio` and `Count` (=visibility count). A full list of allowed statistics can be retrieved by typing `aoquality liststats` on the command line. However, not all allowed statistics are stored by default in a measurement set. This is an example `aoqplot` run:

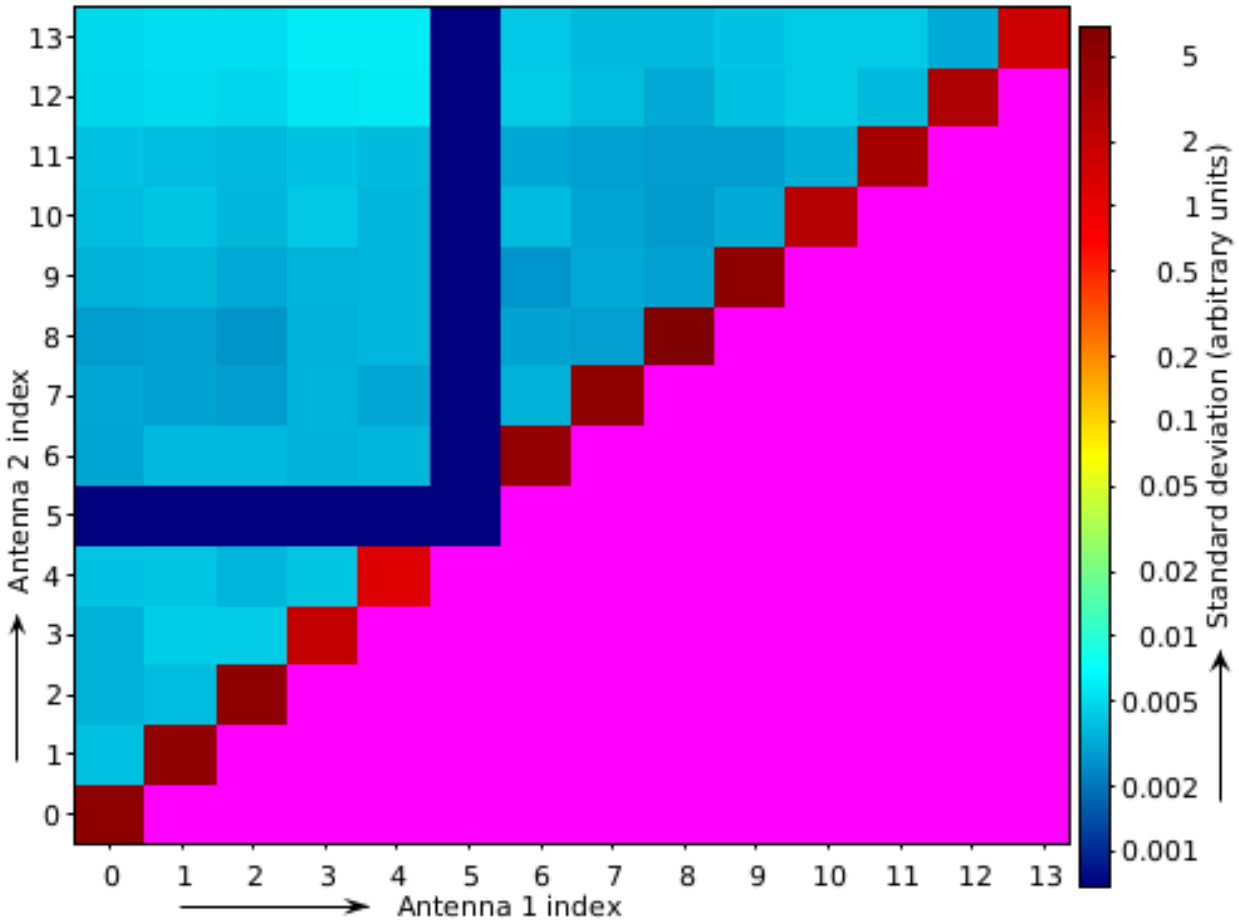
```
aoqplot -save aartfaac-stddev StandardDeviation aartfaac-testobs.ms/
 (1/1) Adding aartfaac-testobs.ms/ to statistics...
Lowering time resolution...
Lowering frequency resolution...
Integrating baseline statistics to one channel...
Regridding time statistics...
Copying statistics...
Integrating time statistics to one channel...
Opening statistics panel...
Saving aartfaac-stddev-antennas.pdf...
Saving aartfaac-stddev-baselines.pdf...
Saving aartfaac-stddev-baselinelengths.pdf...
Saving aartfaac-stddev-timefrequency.pdf...
```

(continues on next page)

(continued from previous page)

```
Saving aartfaac-stddev-time.pdf...
Saving aartfaac-stddev-frequency.pdf...
```

This is an example for the produced 'baselines' plot:




As can be seen from this image, antenna index 5 (WSRT RT5) is not working, and the autocorrelations show more power (as they should). Note that `aoqplot` uses the quality statistics tables inside the measurement set. These are normally produced by tools like `DPPP`, `cotter` or `aartfaac2ms`; however, if a measurement set is not produced by one of those tools, it is necessary to create these tables manually. This can be done with the `aoquality collect <obs.ms>` command.

# CHAPTER 4

# Lua strategies

AOFlagger flagging strategies are written in the Lua language. Lua is quite a simple language, and if you know Python, Lua will feel quite familiar. Most of its syntax should be evident when inspecting some of the example strategies. In Lua, comments start with two dashes "--", `if` and `for` blocks end with an `end` statement, and class methods can be called with either

```
object.method(object, parameter1, parameter2)
```

or

```
object:method(parameter1, parameter2)
```

## 4.1 Hello world

A flagging strategy script is a Lua script file that defines one or more execution functions, normally named *`execute()`*. It may additionally define an *`options()`* function to set options.

For a simple strategy, a flagging script can consist of a single `execute` function:

```
function execute (data)
  print("Hello world: " ..
        data:get_antenna1_name() ..
        " x " ..
        data:get_antenna2_name() )
end
```

The Lua `print` statement writes something to the console. Two dots are used in Lua to concatenate strings. Let's assume that the above script is saved as `helloworld.lua`. This strategy can now be run on a dataset as follows:

```
aoflagger -strategy helloworld.lua observation.ms
```

When running this strategy on a WSRT dataset, the output looks like this:

```
$ aoflagger -strategy helloworld.lua observation.ms
AOFlagger 3.0-alpha (2020-03-06) command line application
Author: André Offringa (offringa@gmail.com)

Starting strategy on 2020-Jun-21 22:26:56.154492
Hello world: RT0 x RT2
Hello world: RT0 x RT1
Hello world: RT0 x RT3
Hello world: RT0 x RT5
Hello world: RT0 x RT4
Hello world: RT0 x RT7
Hello world: RT0 x RT6
Hello world: RT0 x RT9
Hello world: RT0 x RT8
Hello world: RT0 x RTB
Hello world: RT0 x RTA
Hello world: RT0 x RTD
Hello world: RT0 x RTC
Hello world: RT1 x RT3
Hello world: RT1 x RT2
...
```

As can be seen from the output, the `execute()` function is called for every pair of antennas (antennas are named RT0/1/2/... for the WSRT). For every antenna pair, the dynamic spectrum data is loaded and passed to `execute()`. Additionally, the existing flags and some metadata are loaded. These data are all accessible through the `data` object (see class *Data*). Because this strategy does not modify the `data` object, the existing flags are written back to the data set, so nothing is changed.

When the data set is from an interferometric telescope, such as in this example, the default option is to only flag the cross-correlations. Because the function is called from multiple independent threads, the order in which the baselines are processed can differ.

### 4.1.1 Defining options

If a function named `options` exist, `aoflagger` will call this function before opening the dataset, and expects it to return a table with options. Here is the same hello world strategy that processes both auto-correlations and cross-correlations:

```
function execute (data)
  print("Hello world for " ..
        data:get_antenna1_name() ..
        " x " ..
        data:get_antenna2_name() )
end

function options ()
  main_options = { }
  main_options["baselines"] = "all"
  return { ["main"] = main_options }
end
```

If `aoflagger` is again used to run this strategy, the auto-correlations are indeed also shown. This is achieved by setting the `"baselines"` option to `"all"`. See the *list of options* for other options.

The `options()` function returns a table of tables. Elements of the first table can enumerate one or more runs, whereas the second table define the options. With this system, it is possible to define multiple execute functions.

These will process the data one after each other. Be aware that the dataset is first completely processed by the first function, then completely processed by the second function, etc.

**Note:** Multiple execute functions cause the data to be read multiple times.

Multiple passes can be useful to e.g. first flag all baselines separately, and then perform flagging based on the integrated statistics.

The following script sets up the options to perform two runs through the data:

```
function execute_per_baseline (data)
  -- Modify the 'data' object here
end

function execute_integrated (data)
  -- Modify the 'data' object here
end

function options ()
  opts1 = {
    ["baselines"] = "all"
    ["execute_function"] = "execute_per_baseline"
  }

  opts2 = {
    ["baselines"] = "cross"
    ["baseline_integration"] = "average"
    ["execute_function"] = "execute_integrated"
  }

  return {
    ["per baseline"] = opts1,
    ["baseline-integrated"] = opts2
  }
end
```

`rfigui`, `aoflagger` and the API make use of the same Lua structure, i.e., a script that runs in `rfigui` can also be run by `aoflagger` and by the *C++* and *Python* interfaces. However, not all options are relevant for the `rfigui` or interfaces.

## 4.2 Execute and options reference

The formal specification for the `execute()` and `options()` functions is as follows.

**`execute`**(*data*)

This is the main flagging function that is to be implemented by the flagging script. AOFlagger will look for the global `execute` function, and call this when data is available to be flagged. It is possible to give this function a different name, or have multiple functions by changing the defaults in *`options()`*.

On input, the `data` option contains the visibility data and the current flag mask. The object's interface is described by the *`Data`* class. The task of the `execute` function is to modify this data object.

The command line *aoflagger* program will go through a measurement set, call `execute` for each baseline and save the resulting flag mask. The visibilities themselves are not written back to the measurement set, so any changes to them (e.g. by *`low_pass_filter()`*) will not change the data on disk.

It is fine for the `execute` function to set global parameters. Note however that, when `execute` is called by `aoflagger`, different calls to `execute` might run within a different Lua context. Every thread is assigned its own Lua context.

**Parameters** **data** (*Data*) – Single-baseline data to be flagged

**options**()

This special function is called by `aoflagger` to obtain options that are relevant for the strategy (or strategies).

It should return a table for which each entry in the table maps a label to an option list. An option list is a table itself, mapping option names to their values. Any option that is not set, is left to its default.

Options on the command line override options that are set by the strategy for all runs defined. The `options` function is optional: if it is not defined, all options are left to their defaults. Available options can be found on the *list of options*.

**Returns** An option list

**Return type** Table

## 4.3 Example script

This section demonstrates a simple example script. Copying the script below into the `rfigui` Lua editor will allow you to interactively change this script.

```
function execute(data)

  data:clear_mask()

  for _,polarization in ipairs(data:get_polarizations()) do

    local pol_data = data:convert_to_polarization(polarization)
    pol_data = pol_data:convert_to_complex("amplitude")

    aoflagger.high_pass_filter(pol_data, 51, 51, 3, 3)
    aoflagger.visualize(pol_data, "Low-pass filtered", 0)

    aoflagger.sumthreshold(pol_data, 1, 1, true, true)
    aoflagger.scale_invariant_rank_operator(pol_data, 0.2, 0.2)

    pol_data = pol_data:convert_to_complex("complex")
    data:set_polarization_data(polarization, pol_data)

  end -- end of polarization iterations

end
```

We will discuss this script line by line. When AOFlagger calls the `execute()` function, the script starts by calling *Data.clear_mask()*, which unsets all flags:

```
function execute(data)

  data:clear_mask()
```

If the input data was already flagged, these flags are removed.

The next step is a `for` loop over the different polarizations in the set:

```
for _,polarization in ipairs(data:get_polarizations()) do
```

Almost any flagging script will need such a loop, because most operations can only work on a single polarization. The input data could consist of 1-4 linear polarizations (XX, XY, YX, YY), circular polarizations (LL, LR, RL, RR) or Stokes polarizations (I, Q, U, V). *Data.get_polarizations()* returns a table of strings, and `ipairs` converts this to an iterator. A loop over a table captures an (index, value) pair. In this case we only need the value (`polarization`), and ignore the index with the underscore.

To work on a single polarization only, *Data.convert_to_polarization()* is used to create a new data object that contains just that single polarization:

```
local pol_data = data:convert_to_polarization(polarization)
```

The new object is stored as `local` variable. In Lua, any variable that is not declared as `local`, is a global. This might cause the data to be stored longer than necessary, causing more memory usage, so local variables should be preferred.[1]

Most input data sets are complex. Thresholding is more effective on the amplitudes of the samples though. Function `Data.convert_to_complex` is used to calculate the amplitudes:

```
pol_data = pol_data:convert_to_complex("amplitude")
```

Another new object is created, but the previous `pol_data` object is overwritten.

The next step is high-pass filtering the data:

```
aoflagger.high_pass_filter(pol_data, 51, 51, 3, 3)
```

Function *aoflagger.high_pass_filter()* filters the visibility data in *pol_data*. A kernel of 51 x 51 samples is used (ntimes x nchannels), with a Gaussian width of 3 samples in both directions.

The filtered data is "visualized" with function *aoflagger.visualize()*:

```
aoflagger.visualize(pol_data, "Low-pass filtered", 0)
```

This statement makes it possible to display the result of filtering the data in `rfigui`. When the script is not running interactively from a gui, the call is ignored. Note that `visualize()` will be called for all polarizations. The gui will recombine visualizations from different polarizations, as long as they have the same name and sorting index.

The *aoflagger.sumthreshold()* searches the (filtered) data for consecutive high values in time or frequency:

```
aoflagger.sumthreshold(pol_data, 1, 1, true, true)
```

The resulting `pol_data` object will now contain a flag mask. This flag mask is morphologically extended in the time and frequency direction with the *aoflagger.scale_invariant_rank_operator()* function:

```
aoflagger.scale_invariant_rank_operator(pol_data, 0.2, 0.2)
```

Finally, the data are converted back to its original form, so that the polarizations can be combined. The first step is to convert the data back to complex values using *Data.convert_to_complex()*:

```
pol_data = pol_data:convert_to_complex("complex")
```

This "conversion" can be seen as an update of the metadata. The phases were lost while converting to amplitudes, so these are not restored, and just assumed to be constant. The `pol_data` now holds a complex data and our new flag mask. The last step is to update the input data with our new data using *Data.set_polarization_data()*:

[1] Data objects are emptied by AOFlagger at the end of the `execute()` function, so the user should normally not worry about memory usage.

```
data:set_polarization_data(polarization, pol_data)
```

By iterating over all polarizations in the input set, the polarizations in the input data object are replaced one by one.

## 4.4 Strategy option list

The following table lists all the options that can be set when implementing the *options()* function.

| option | type | description |
|---|---|---|
| bands | table | List of integer (zero-indexed) band ids to process. |
| baselines | string | "auto", "cross" or "all" for selecting auto/cross-correlations or both. |
| baseline-integration | string | Average baselines together to a single dynamic spectrum, with a specified method. Allowed values are: "count", "average", "average-abs", "squared" or "stddev". |
| chunk-size | integer | When not zero, aoflagger will process the data in chunks with the given maximum chunk size. |
| column-name | string | What data column to use, e.g. "DATA", "CORRECTED_DATA". etc. |
| combine-spws | boolean | Whether to concatenate all spectral windows together while flagging. |
| execute-file | string | Name of file to load for this run, which should provide the execute-function. By default, it is assumed to be the currently loaded file (which also provides the *options()* call). |
| execute-function | string | Name of function to run (note this is a string, not a function). Default: "execute". |
| fields | table | List of integer (zero-indexed) field ids to process. |
| files | table | List of strings that are the names of the files to process. |
| min-aoflagger-version | string | Minimum AOFlagger version required, of the form "major.minor". Defaults to "3.0". |
| quiet | boolean | Inhibits all output except errors. |
| read-mode | string | "direct", "indirect", "memory" or "auto". |
| read-uvws | boolean | Whether to read the UVWs. This is not done by default. |
| script-version | string | Version of this strategy. Can have the form "major.minor[.subminor] [extra description]", for example "2.4 beta" or "3.1.4 modified by André". |
| start-timestep /end-timestep | integer | Timestep (zero-indexed) from/at which to start/end processing. |
| threads | integer | Number of threads to use. The default is to use one thread per core. |
| verbose | boolean | Sets verbose logging. |

## 4.5 Functions

This section lists all the Lua functions provided by AOFlagger.

### 4.5.1 Summary

The `aoflagger` module provides the following functions:

- **Reporting and user interface:**
  - *print_polarization_statistics()*
  - *require_max_version()*
  - *require_min_version()*
  - *save_heat_map()*
  - *set_progress()*
  - *set_progress_text()*
  - *visualize()*
- **Mathematical operations**
  - *norm()*
  - *sqrt()*
- **Data scaling:**
  - *apply_bandpass()*
  - *normalize_bandpass()*
  - *normalize_subbands()*
- **Filtering and resolution:**
  - *downsample()*
  - *high_pass_filter()*
  - *low_pass_filter()*
  - *upsample_image()*
  - *upsample_mask()*
- **Thresholding:**
  - *sumthreshold()*
  - *sumthreshold_masked()*
  - *threshold_channel_rms()*
  - *threshold_timestep_rms()*
- **Morphological:**
  - *scale_invariant_rank_operator()*
  - *scale_invariant_rank_operator_masked()*
- **Ranges**
  - *copy_to_channel()*
  - *copy_to_frequency()*
  - *trim_channels()*
  - *trim_frequencies()*

- **Other:**
  - *collect_statistics()*

### 4.5.2 Detailed descriptions

aoflagger.**apply_bandpass**(*data*, *filename*)

Apply a bandpass file to the data. The data is changed in place. Each line in the file contains <antenna name> <X/Y polarization> <channel index> <gain>, separated by spaces, for example:

RT2 X 0 0.7022
RT2 X 1 0.7371
RT2 X 2 0.8092
...

**Parameters**

- **data** (*Data*) – Data to which the bandpass is applied.
- **filename** (*string*) – Path to bandpass textfile.

aoflagger.**collect_statistics**(*after_data*, *before_data*)

Calculate statistics, such as visibility standard deviation and flag percentages. When running the strategy on a measurement set, the statistics are stored inside the measurement set after finishing all baselines. These can be inspected by the `aoqplot` tool.

The function takes the data after and before flagging. Any data that are flagged in `before_data` will not contribute to the statistics. This avoids counting e.g. correlator faults or shadowing as interference.

**Parameters**

- **after_data** (*Data*) – Flagged data.
- **before_data** (*Data*) – Unflagged data.

aoflagger.**copy_to_channel**(*destination_data*, *source_data*, *channel*)

Copy the data (visibilities & flags) from the source data to the destination data with a specified channel offset. This function can be used together with *trim_channels()* to flag a subset of the data and copy the result back to the full data.

In that scenario, the start channel for *trim_channels()* equals the channel parameter in this call.

When the source does not fit into the destination at the specified offset, only the part that fits is copied.

Available since *v3.1*.

**Parameters**

- **destination_data** (*Data*) – Destination data
- **source_data** (*Data*) – Data to be copied. These data are unchanged.
- **channel** – A channel index that specifies the offset to which the data is copied in the destination data.

aoflagger.**copy_to_frequency**(*destination_data*, *source_data*, *frequency*)

This function is similar to *copy_to_channel()*, but instead of specifying the target channel offset as an index, it is specified as a frequency. This can be used as counterpart to *trim_frequencies()*.

See *copy_to_channel()* for further info.

Available since *v3.1*.

**Parameters**

- **destination_data** (*Data*) – Destination data
- **source_data** (*Data*) – Data to be copied. These data are unchanged.
- **channel** – A frequency in MHz that specifies the offset to which the data is copied in the destination data.

aoflagger.**downsample**(*data*, *xfactor*, *yfactor*, *masked*)

Decrease the resolution of the data using simple linear binning. This can be effective to increase the speed of data smoothing, for example when using *high_pass_filter()*. At the function end of *execute()*, the data should have the original size. Therefore, a call to downsample should normally be followed by a call to *upsample_image()* or *upsample_mask()* to restore the visibilities and flags, respectively, to their original resolution.

When the input data is not exactly divisable by the downsampling factors, fewer samples will be averaged into the last bins.

**Parameters**

- **data** (*Data*) – Input data (not modified).
- **xfactor** (*integer*) – Downsampling factor in time direction.
- **yfactor** (*integer*) – Downsampling factor in frequency direction.
- **masked** (*boolean*) – `true` means take flags into account during averaging

**Returns** Downsampled version of input data.

**Return type** *Data*

aoflagger.**high_pass_filter**(*data*, *xsize*, *ysize*, *xsigma*, *ysigma*)

Apply a Gaussian high-pass filter to the data. This removes the diffuse 'background' in the data. With appropriate settings, it can filter the signal of interest (slow sinusoidal signals), making the interference easier to detect.

The function convolves the data with a 2D "1 minus Gaussian" kernel. The kernel is clipped at the edges. The sigma parameters define the strength (band-limit) of the filter: lower values remove more of the diffuse structure.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **xsize** (*integer*) – Kernel size in time direction
- **ysize** (*integer*) – Kernel size in frequency direction
- **xsigma** (*number*) – Gaussian width in time direction.
- **ysigma** (*number*) – Gaussian width in frequency direction.

aoflagger.**low_pass_filter**(*data*, *xsize*, *ysize*, *xsigma*, *ysigma*)

Apply a Gaussian low-pass filter to the data. It convolves the data with a Gaussian. See *high_pass_filter()* for further details.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **xsize** (*integer*) – Kernel size in time direction
- **ysize** (*integer*) – Kernel size in frequency direction

- **xsigma** (*number*) – Gaussian width in time direction.
- **ysigma** (*number*) – Gaussian width in frequency direction.

aoflagger.**norm**(*input_data*)
Creates a new *Data* object for which the data values have been replaced by their norm. If the data is complex, a complex norm is performed. In all other cases, a real-valued norm is performed (i.e., the data is squared).

**Parameters** **input_data** (*Data*) – Data object (not modified)

**Returns** Norm of input_data, element-wise applied.

**Return type** *Data*

aoflagger.**normalize_bandpass**(*data*)
Normalizes the RMS over frequency. If multiple polarizations are present in the data, the RMS over the combination of all polarizations is calculated and normalized.

Available since *v3.1*.

**Parameters** **data** (*Data*) – The data (modified in place).

aoflagger.**normalize_subbands**(*data*, *nr_subbands*)
Remove jumps between subbands. A subband is in this context a number of adjacent channels, equally spaced over the bandwidth. This function therefore assumes that all subbands have an equal number of channels.

Each subband is scaled such that the standard deviation of the visibilities in a subband is unity. To avoid influence from interference, a stable method is used to estimate the standard deviation (Winsorized standard deviation).

A typical use-case for this function is the MWA phase 1 and 2. The 30 MHz bandwidth of the MWA is split in 24 'course channels', each consisting of 128 channels. Each course channel has an independent gain, and needs normalization before it can be compared with adjacent course channels.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **nr_subbands** (*integer*) – Number of subbands.

aoflagger.**print_polarization_statistics**(*data*)
Print RFI percentages per polarization to the command line.

**Parameters** **data** (*Data*) – Input data.

aoflagger.**require_max_version**(*version*)
Checks if the aoflagger version is lower or equal to the provided version. If the condition is not met, an error is thrown. This function can be used when it is known a strategy is making use of Lua functionality that was changed in newer aoflagger versions.

The version string can be of the form "major", "major.min" or "major.minor.subminor". The version is only checked up to the level that is specified: requiring at most version "`3.2`" will allow version "`3.2.1`", but not version "`3.3.0`" or "`4.0`". To disallow version "`3.2.1`", a maximum version of "`3.2.0`" should be specified.

Available since *v3.1*.

See also *require_min_version()*.

**Parameters** **version** (*string*) – Latest version that is allowed, e.g. `"3.0.4"`.

aoflagger.**require_min_version**(*version*)
Checks if the aoflagger version is equal to or newer than the provided version string. If the condition is not met, an error is thrown. This is a useful way of notifying users that their version of aoflagger is too old. A version of aoflagger should (only) be considered too old when the strategy requires a function, method or other functionality that is not available in versions before the specified version.

The version string can be of the form "major", "major.min" or "major.minor.subminor". The version is only checked up to the level that is specified: requiring version "`3.2`" will allow versions such as "`3.2-alpha`" and "`3.2.1`".

Available since *v3.1*.

**Parameters** **version** (*string*) – Minimum version that is allowed, e.g. `"3.0.4"`.

aoflagger.**save_heat_map**(*filename*, *data*)

Save the data as a "heat map" image. The type is determined from the extension. Supported extensions are `.svg`, `.png` and `.pdf`.

**Parameters**

- **filename** (*string*) – Path to image to be written.
- **data** (*Data*) – Input data.

aoflagger.**scale_invariant_rank_operator**(*data*, *xlevel*, *ylevel*)

Extend flags in time and frequency direction in a scale-invariant manner. This fills holes in the flag mask and makes flag sequences longer. Details are described in Offringa et al. 2012.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **xlevel** (*number*) – aggressiveness in time-direction
- **ylevel** (*number*) – aggressiveness in frequency-direction

aoflagger.**scale_invariant_rank_operator_masked**(*data*, *mask_data*, *xlevel*, *ylevel*, *penalty*)

Perform the same operation as *scale_invariant_rank_operator()*, but with an input mask that identifies invalid data. Invalid data is treated differently, and the penalty parameter selects how it is treated. With a penalty of 0, it is as if invalid samples are removed before applying the operator. With a penalty of 1, invalid samples are counted in the same way as unflagged samples (i.e., they penalize their extension). A typical penalty value is 0.1. For backwards compatibility, penalty may be left out, in which case a value of 0.1 is used.

Available since *v3.1*. The penalty parameter is available since *v3.2*.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **mask_data** (*Data*) – The data that is used as mask.
- **xlevel** (*number*) – aggressiveness in time-direction
- **ylevel** (*number*) – aggressiveness in frequency-direction
- **penalty** (*number*) – penalty given to the extension through invalid regions.

aoflagger.**set_progress**(*progress*, *max_progress*)

Notify user of the progress of this call. The gui uses this information to show a progress bar to the user. Example: when the *execute()* function iterates over the polarizations, progress can be reported by calling `aoflagger.set_progress(curpol, npol)` inside the loop.

**Parameters**

- **progress** (*integer*) – current progress
- **max_progress** (*integer*) – value of progress when complete

aoflagger.**set_progress_text**(*task_description*)

Notify user of the current task being done. The description can be anything, and can literally be presented to the user.

**Parameters** **task_description** (*string*) – Description string.

aoflagger.**sqrt**(*input_data*)

Creates a new *Data* object where all data values are replaced by their square root. If the data is complex, a complex square root is performed. In all other cases, a real-valued square root is performed.

**Parameters** **input_data** (*Data*) – Data object (not modified)

**Returns** Square root of input_data, element-wise applied.

**Return type** *Data*

aoflagger.**sumthreshold**(*data*, *x_threshold_factor*, *y_threshold_factor*, *x_direction*, *y_direction*)

Run the SumThreshold algorithm on the data. This algorithm detects sharp, line-shaped features in the time-frequency domain that are typical for RFI. See Offringa et al. (2010) for details about the algorithm.

The thresholds are relative to a (stable) estimate of the noise in the visibilities. They define the base sensitivity of the algorithm. Lower values will detect more features. A reasonable value for the thresholds is 1.

The x_direction/y_direction parameters turn detection in their particular directions on and off. If a direction is turned off, the threshold factor for that direction is ignored. Note that detection in *x*-direction (which is the time-direction) means detection of contiguous high-power samples in time, such as transmitters that occupy the same channel continuously. The *y*-direction detection is sensitive to transient, broadband RFI.

**Parameters**

- **data** (*Data*) – The data (modified in place)
- **x_threshold_factor** (*number*) – Threshold factor in time direction
- **y_threshold_factor** (*number*) – Threshold factor in frequency direction
- **x_direction** (*boolean*) – Enable flagging in time direction
- **y_direction** (*boolean*) – Enable flagging in frequency direction

aoflagger.**sumthreshold_masked**(*data*, *mask_data*, *x_threshold_factor*, *y_threshold_factor*, *x_direction*, *y_direction*)

Same as *sumthreshold()*, but with a mask. Visibilities that are flagged in the mask are considered to be visibilities that have not been sampled and are removed from the SumThreshold operation. A typical case for this is to make sure that correlator faults, shadowing and band-edges are correctly treated.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **mask_data** (*Data*) – The data that is used as mask
- **x_threshold_factor** (*number*) – Threshold factor in time direction
- **y_threshold_factor** (*number*) – Threshold factor in frequency direction
- **x_direction** (*boolean*) – Enable flagging in time direction
- **y_direction** (*boolean*) – Enable flagging in frequency direction

aoflagger.**threshold_channel_rms**(*data*, *threshold*, *flag_low_outliers*)

Calculate the root-mean-square (RMS) for each channel and flags channels that have an outlier RMS. The threshold is a "sigma level". Typical values for the threshold are therefore around 3.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **threshold** (*number*) – Sigma-level of threshold.
- **flag_low_outliers** (*boolean*) – Flag channels with low RMS.

aoflagger.**threshold_timestep_rms**(*data*, *threshold*)
Like *threshold_channel_rms()*, but thresholds *timesteps* with outlier RMS. Both timesteps with high and low RMS values are flagged.

**Parameters**

- **data** (*Data*) – The data (modified in place).
- **threshold** (*number*) – Sigma-level of threshold.

aoflagger.**trim_channels**(*data*, *start_channel*, *end_channel*)
Create a new data object from a subset of the input data. This can be used to flag a subset of the data, together with *copy_to_channel()* to copy the result back. All channels for which start_channel <= channel index < end_channel are copied into the result. All timesteps are copied.

Available since *v3.1*.

**Parameters**

- **data** (*Data*) – Input data (unchanged).
- **start_channel** (*integer*) – Index of first channel
- **end_channel** (*integer*) – Index of end of the channel range. The end range is excluding.

**Returns** A new data object, trimmed as specified.

**Return type** *Data*

aoflagger.**trim_frequencies**(*data*, *start_frequency*, *end_frequency*)
This function is equal to *trim_channels()*, except that the channel range is specified with frequency values. All channels for which start_frequency <= channel frequency < end_frequency are copied into the result. All timesteps are copied.

*copy_to_frequency()* can be used to copy the result back after processing.

Available since *v3.1*.

**Parameters**

- **data** (*Data*) – Input data (unchanged).
- **start_frequency** (*number*) – Start frequency in MHz of the selected range.
- **end_frequency** (*number*) – End frequency in MHz of the channel range.

**Returns** A new data object, trimmed as specified.

**Return type** *Data*

aoflagger.**upsample_image**(*input_data*, *destination_data*, *xfactor*, *yfactor*)
Increase the resolution of the data. This function is to restore the resolution of the data after having called *downsample()*. input_data is normally the data that was returned by *downsample()*, and destination_data is the input object that was specified as parameter. The upsampling is done by nearest neighbour interpolation.

The x and y factors should be the equal to the values specified in the call to *downsample*. The size of the destination_data is not changed: the input data is stretched by the given factors, and trimmed to the destination size in case the image dimensions were not exactly divisable by the factors.

The function only upsamples the visibilities, not the flags. To upsample the flags, see *upsample_mask()*.

**Parameters**

- **input_data** (*Data*) – Input low-resolution data (not modified).

- **destination_data** (*Data*) – Where the result will be stored.
- **xfactor** (*integer*) – Upsampling factor in time direction.
- **yfactor** (*integer*) – Upsampling factor in frequency direction.

aoflagger.**upsample_mask**(*input_data*, *destination_data*, *xfactor*, *yfactor*)
Increase the resolution of the mask. It is identical to *upsample_image()*, but works with the mask (flags) instead of the image (visibilities).

**Parameters**

- **input_data** (*Data*) – Input low-resolution data (not modified).
- **destination_data** (*Data*) – Where the result will be stored.
- **xfactor** (*integer*) – Upsampling factor in time direction.
- **yfactor** (*integer*) – Upsampling factor in frequency direction.

aoflagger.**visualize**(*data*, *label*, *sorting_index*)
Save a visualization of the data for inspection in `rfigui`. When this strategy runs outside of the `rfigui`, the call is ignored. Can be used to e.g. inspect partial results.

**Parameters**

- **data** (*Data*) – Input data (not modified).
- **label** (*string*) – A short description that is displayed to the user.
- **sorting_index** – Where to place this visualization in the list of visualization

## 4.6 Data class

**class Data**
The Data class contains the visibility data, flag masks and meta data.

It may hold information for multiple polarizations, channels and timesteps, such that it contains the data for a particular observing field, spectral window (band) and antenna-pair (or single dish). The data object thus holds a dynamic spectrum for each polarization.

The visibility data can consist of complex values, or single-float values representing the real, imaginary, amplitude or phase of the visibilities (see *get_complex_state()*).

The Data class is a Lua `userdata` object. Assigning one Data variable to another will therefore cause both variables to point to the same object, and changing one will also change the other. To create an independent object, *copy()* can be used.

Internally, AOFlagger uses a copy-on-write-like mechanism for the visibility sets, flag masks and meta data. It is not required for users of the class to know about these implementation details, but the relevant thing to know is that copy or assignment operations are generally fast and don't increase memory usage.

### 4.6.1 Method summary

- **Copy & Modification**
  - *clear_mask()*
  - *copy()*
  - *flag_nans()*

  - *flag_zeros()*
  - *invert_mask()*
  - *join_mask()*
  - *set_mask()*
  - *set_mask_for_channel_range()*
  - *set_masked_visibilities()*
  - *set_polarization_data()*
  - *set_visibilities()*
  - *__div()*
  - *__sub()*
- **Conversion**
  - *convert_to_complex()*
  - *convert_to_polarization()*
- **Meta-data**
  - *get_antenna1_index()*
  - *get_antenna1_name()*
  - *get_antenna2_index()*
  - *get_antenna2_name()*
  - *get_baseline_angle()*
  - *get_baseline_distance()*
  - *get_baseline_vector()*
  - *get_complex_state()*
  - *get_frequencies()*
  - *get_polarizations()*
  - *get_times()*
  - *has_metadata()*
  - *is_auto_correlation()*
  - *is_complex()*

### 4.6.2 Detailed descriptions

Data.**clear_mask**(*data*)
Clear the flag mask. Unflags all visibilities (sets all flags to false).

**Parameters** **data** (*Data*) – Data for which the mask is cleared.

Data.**convert_to_complex**(*data*, *new_state*)
Make a new *Data* object with a different complex-value state. Complex input data (data with *get_complex_state()* == "complex") can be converted to real, imaginary, amplitude or phase values. Amplitude data (*get_complex_state()* == "amplitude") can also be converted back to complex. In that case the phases become zero. Other conversions are not implemented and will cause an error.

The complex state of an *Data* object is stored internally and can be acquired by calling *get_complex_state()*.

**Parameters**
- **data** (*Data*) – Input data (unchanged).
- **new_state** (*string*) – `"complex"`, `"real"`, `"imaginary"`, `"amplitude"` or `"phase"`.

**Returns** New object with *get_complex_state()* == new_state

**Return type** *Data*

`Data.`**convert_to_polarization**(*data*, *new_polarization*)

Make a new *Data* object by converting the polarization. If the input data does not hold the polarimetric data to convert to the requested polarization, an error is thrown. For example, converting to `"i"` from data for which *get_polarizations()* == `{"xx", "yy"}` is possible, but converting to `"q"` from data with *get_polarizations()* == `{"ll"}` is not.

This method can also be used to extract a single polarization from the set of available polarizations, e.g.

```
xxdata = data:convert_to_polarization("xx")
```

for data with *get_polarizations()* == `{"xx", "xy", "yx", "yy"}`.

**Parameters**
- **data** (*Data*) – input data (unchanged).
- **new_polarization** (*string*) – `"i"`, `"q"`, `"u"`, `"v"`, `"xx"`, `"xy"`, `"yx"`, `"yy"`, `"rr"`, `"rl"`, `"lr"` or `"ll"`.

`Data.`**copy**(*data*)

Make a value copy of the data.

**Parameters** **data** (*Data*) – Source data.

**Returns** Value copy of input data.

**Return type** *Data*

`Data.`**flag_nans**(*data*)

Flag visibilities that are 'not a number' (nan) or hold overflow. Each polarization is independently searched for nans, and its mask is updated for that polarization (this is different from *Data.flag_zeros()*).

**Since** AOFlagger 3.1.

`Data.`**flag_zeros**(*data*)

Flag visibilities that are exactly zero. This corrects for correlators that output zeros during faults, such as network problems.

It flags samples when the sum of visibilities over polarizations is zero. When it is necessary to flag the polarizations independently, the statement should be placed inside a loop, e.g.:

```
for _,polarization in ipairs(data.get_polarizations()) do
  pol_data = data:convert_to_polarization(polarization)
  flag_zeros(pol_data)
  data:set_polarization_data(polarization, pol_data)
end
```

**Parameters** **data** (*Data*) – Data (modified inplace).

`Data.`**get_antenna1_index**(*data*)

Get first antenna index of the two correlated antennas. Throws an error of the antenna metadata is not available (see *has_metadata()*).

**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Index of first antenna
**Return type** integer

`Data.`**`get_antenna1_name`**`(`*data*`)`
Get name of first antenna of the two correlated antennas. Throws an error of the antenna metadata is not available (see *has_metadata()*).
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Name of first antenna
**Return type** string

`Data.`**`get_antenna2_index`**`(`*data*`)`
Get second antenna index of the two correlated antennas. Throws an error of the antenna metadata is not available (see *has_metadata()*).
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Index of second antenna
**Return type** integer

`Data.`**`get_antenna2_name`**`(`*data*`)`
Get name of second antenna of the two correlated antennas. Throws an error of the antenna metadata is not available (see *has_metadata()*).
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Name of second antenna
**Return type** string

`Data.`**`get_baseline_angle`**`(`*data*`)`
Get angle of this baseline. This is that angle between the line from antenna2 to antenna1 and North. Throws an error of the antenna metadata is not available (see *has_metadata()*).
**Parameters** **data** (*Data*) – Input data (unchanged)
**Returns** Baseline angle in radians
**Return type** number

`Data.`**`get_baseline_distance`**`(`*data*`)`
Get distance of the antenna1-antenna2 baseline in meters. Throws an error of the antenna metadata is not available (see *has_metadata()*).
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Baseline distance in meters
**Return type** number

`Data.`**`get_baseline_vector`**`(`*data*`)`
Get a table with items `x`, `y` and `z` that form the three- dimensional vector between antennas 1 and 2. Throws an error of the antenna metadata is not available (see *has_metadata()*).
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Baseline vector in meters
**Return type** table

`Data.`**`get_complex_state`**`(`*data*`)`
Get the state that the visibilities represent. This can be `"phase"`, `"amplitude"`, `"real"`, `"imaginary"` or `"complex"`. When the data is complex, each visibility consists of two number. Conversions can be performed with *convert_to_complex()*.
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** Complex state of data.
**Return type** string

`Data.`**`get_frequencies`**`(`*data*`)`

Get the frequencies of the channels. Throws an error of the spectral window metadata is not available (see *has_metadata()*).

**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** List that maps channel nr to frequency in Hz.
**Return type** table

Data.**get_polarizations**(*data*)
Get the list of polarizations provided by the data. See *convert_to_polarization()* for the list of possible polarization names.

**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** List that maps polarization nr to a string.
**Return type** table

Data.**get_times**(*data*)
Get the time of each timestep in these data. Throws an error of the time metadata is not available (see *has_metadata()*).

**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** List that maps timestep nr to MJD time in s.
**Return type** table

Data.**has_metadata**(*data*)
Returns whether metadata is completely present. Not all data formats (or simulations) provide all metadata items and some of the other methods (e.g. *get_times()*) may throw an error because of this. If this function returns `true`, all these functions will succeed.

**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** `true` in case all metadata is available, `false` otherwise.
**Return type** boolean

Data.**invert_mask**(*data*)
Changes masked values to be unmasked and vice versa.

**Parameters** **data** (*Data*) – Destination data (modified in place)

Data.**is_auto_correlation**(*data*)

Determine whether this baseline is an auto-correlation. This is the case if *get_antenna1_index()* == *get_antenna2_index()*. Unlike the `get_antenna*` functions, this method won't throw an error when no meta-data is available. `false` is returned in that case.

**param data** Input data (unchanged).

**type data** *Data*

**return** `true` when this is an auto-correlation.

**type** boolean

**is_complex**(*data*)
**Parameters** **data** (*Data*) – Input data (unchanged).
**Returns** `true` when *get_complex_state()* == `"complex"`
**Type** boolean

Data.**join_mask**(*first_data*, *second_data*)
Join two masks together. A flag will be set when it is set in either or both of the input data sets.

**Parameters**
- **first_data** (*Data*) – First mask and destination of operation.
- **second_data** (*Data*) – Second mask (unchanged).

Data.**set_mask**(*destination_data*, *mask_data*)
Assign the mask of one *Data* object to another.

**Parameters**
- **first_data** (*Data*) – Destination data (changed inplace).
- **second_data** (*Data*) – Source data (unchanged).

Data.**set_mask_for_channel_range**(*destination_data*, *mask_data*, *freq_start*, *freq_end*)
Partially assign the mask of one *Data* object to another. The flag mask of channels within the given frequency range are copied from `mask_data` to `destination_data`.

This can for example be useful when a certain channel range should not be flagged, by partially copying the initial flags to the mask produced by the RFI detection.

**Parameters**
- **first_data** (*Data*) – Destination data (changed inplace).
- **second_data** (*Data*) – Source data (unchanged).
- **freq_start** (*number*) – Frequency range start in MHz
- **freq_end** (*number*) – Frequency range end in MHz

Data.**set_polarization_data**(*destination_data*, *polarization*, *source_data*)
Replace one polarization of a *Data* object with some other data. The `source_data` should have only one polarization. The typical use-case for this method is to loop over polarizations and modify them one by one, e.g.:

```
for _,polarization in ipairs(data:get_polarizations()) do
  pol_data = input:convert_to_polarization(polarization)
  -- Change pol_data here...
  data:set_polarization_data(polarization, pol_data)
end
```

Note that the two data sets should have the same complex state (see *get_complex_state()*). This method copies both the mask and the visibilities. The meta-data is unchanged.

**Parameters**
- **destination_data** (*Data*) – Destination data (changed inplace).
- **polarization** (*string*) – Name of polarization to change
- **source_data** (*Data*) – Source data (unchanged).

Data.**set_visibilities**(*destination_data*, *visibility_data*)
Assign the visibility data from one *Data* object to another. The flagmask and meta-data are unchanged. The two sets should have the same number of polarizations and the same complex state.

**Parameters**
- **destination_data** (*Data*) – Destination data (changed inplace).
- **visibility_data** (*Data*) – Source data (unmodified).

Data.**set_masked_visibilities**(*destination_data*, *visibility_data*)
Conditionally assign the visibility data from one *Data* object to another. Only visibilities that are set to *true* in the destination mask are altered. The flagmask and meta-data are unchanged. The two sets should have the same number of polarizations and the same complex state.

**Parameters**
- **destination_data** (*Data*) – Destination data (changed inplace).
- **visibility_data** (*Data*) – Source data (unmodified).

Data.**__gc**(*data*)
Internal method for garbage collection function of the Data class. This will free the allocated data when those data are no longer used by other Data objects. Note that AOFlagger will

immediately clear data when the *execute()* function is finished, even when the class hasn't been garbage collected yet.

TODO add a method `set_persistent` to disable this.

**Parameters** **destination_data** (*Data*) – Garbage collected data.

`Data.__div`(*lhs_data*, *rhs_data*)

Special Lua method for the division operator. Performs element-wise division of *Data* objects. For example, with `data` and `rms` both of type `Data`, this would assign `corrected` to the element-wise divided data:

```
corrected = data / rms
```

The parameters `lhs_data` and `rhs_data` are required to have the same number of polarizations and are required to have the same complex state. The output data will have the same meta-data and masks as `lhs_data`.

**Parameters**

- **lhs_data** (*Data*) – Left-hand side data (not modified)
- **rhs_data** (*Data*) – Right-hand side data (not modified)

**Returns** Left / right

**Return type** *Data*

`Data.__sub`(*lhs_data*, *rhs_data*)

Lua-special method for element-wise subtraction of *Data* objects, e.g.:

```
data = data - filtered_data
```

For information about the requirements and produced meta data, see *__div()*.

**Parameters**

- **lhs_data** (*Data*) – Left-hand side data (not modified)
- **rhs_data** (*Data*) – Right-hand side data (not modified)

**Returns** Left - right

**Return type** *Data*

## 4.7 Upgrading .rfis files

If you have existing `.rfis` files but would like to use AOFlagger 3.0, you will have to upgrade those old `.rfis` files to Lua files. While Lua strategies can do everything that the old `.rfis` files could do, there is no automated method to convert them, because not every function in the old `.rfis` maps one to one to a Lua function. This unfortunately means you might have to do some manual work to make a Lua file. The good news is that Lua strategies are much more powerful.

The easiest way to create a Lua file is to start from an existing strategy, such as the default Lua strategy. If you started the old `.rfis` strategy from a default strategy as well, and you remember the changes necessary for the `.rfis` strategy, it should in most cases be simple to modify the default Lua strategy. If the modifications are unknown, I recommend to use a tool such as `diff` or `meld` to visualize the changes. Because the `.rfis` files are XML files, this should in most cases produce a human-parsable list of differences.

As a simple example, let's say that a `diff -Naur default.rfis modified.rfis`, i.e. the difference between aoflagger's default strategy and your own strategy, shows this:

```
--- default.rfis   2021-09-17 10:12:44.116199932 +0200
+++ modified.rfis   2021-09-17 10:11:10.094925901 +0200
@@ -31,8 +31,8 @@
 <sensitivity-start>4</sensitivity-start>
```

(continues on next page)

(continued from previous page)

```
 <children>
   <action type="SumThresholdAction">
-    <time-direction-sensitivity>1</time-direction-sensitivity>
-    <frequency-direction-sensitivity>1</frequency-direction-sensitivity>
+    <time-direction-sensitivity>1.5</time-direction-sensitivity>
+    <frequency-direction-sensitivity>1.5</frequency-direction-sensitivity>
    <time-direction-flagging>1</time-direction-flagging>
    <frequency-direction-flagging>1</frequency-direction-flagging>
    <exclude-original-flags>0</exclude-original-flags>
@@ -79,8 +79,8 @@
    </children>
  </action>
  <action type="SumThresholdAction">
-    <time-direction-sensitivity>1</time-direction-sensitivity>
-    <frequency-direction-sensitivity>1</frequency-direction-sensitivity>
+    <time-direction-sensitivity>1.5</time-direction-sensitivity>
+    <frequency-direction-sensitivity>1.5</frequency-direction-sensitivity>
    <time-direction-flagging>1</time-direction-flagging>
    <frequency-direction-flagging>1</frequency-direction-flagging>
    <exclude-original-flags>0</exclude-original-flags>
```

This shows that the sensitivity (both time and frequency direction) of the SumThreshold actions were changed from 1 to 1.5. In the old `.rfis` strateges, the sensitivity values are specified every time an action is executed. Because the SumThresholdAction occurs two times, there are two different places in the `.rfis` file. This is not the case for Lua files: in Lua, one can define variables, and the default strategy makes use of this. It lists the most commonly used parameters near the start of the Lua file, and changing the sensitivity of the SumThreshold step can be done simply by changing the base_threshold variable from its default:

```
local base_threshold = 1.0  -- lower means more sensitive detection
```

to:

```
local base_threshold = 1.5  -- lower means more sensitive detection
```

CHAPTER 5

# Python interface

The external Python API mirrors the external C++ API, and only differs in that it follows the common Python naming conventions. For a reference of the functions and a general overview of what can be done with the Python interface, see the *C++ interface*. A reference of the Python API can also be obtained by running `help(aoflagger)` in (i)python:

```
In [1]: import aoflagger

In [2]: help(aoflagger)
Help on module aoflagger:

NAME
    aoflagger - AOFlagger module for detection of radio-frequency interference

CLASSES
    pybind11_builtins.pybind11_object(builtins.object)
        AOFlagger
        FlagMask
        ImageSet
        QualityStatistics
        Strategy
        TelescopeId

    class AOFlagger(pybind11_builtins.pybind11_object)
     |  Main class that gives access to the aoflagger functions.
[..]
```

The rest of this chapter discusses a few practical topics related to the Python interface.

## 5.1 Installation

The AOFlagger Python module is compiled into an object library. It is compiled along when you run `make` as described on the *Installation* chapter. Currently, make compiles the following library on my computer:

```
build/python/aoflagger.cpython-39-x86_64-linux-gnu.so
```

The normal/proper way of installing this library into its correct location, is by running `make install`. On my computer, this copies that file to `/install-prefix/lib`. For Python to find this library, the path needs to be in your Python search path, which is normally set with the `PYTHONPATH` environment variable, e.g.:

```
export PYTHONPATH=/install-prefix/lib:${PYTHONPATH}
```

There's no need to run a `setup.py` for AOFlagger. Also, AOFlagger can't be installed via pip. When installing AOFlagger via a Debian/Ubuntu package, the library should be installed and found without any manual user tweaking. Be aware that the Python interface and binding tool was improved in version 3.0, and it is therefore recommended that the latest release of aoflagger (>= 3.0) is used.

## 5.2 Using the Python interface

The aoflagger module can be included in Python using a standard `import` command:

```
import aoflagger
```

A few examples are given in the `data` directory. The following is an example to calculate the false-positives ratio of the default strategy:

```
import aoflagger
import numpy

nch = 256
ntimes = 1000
count = 50        # number of trials in the false-positives test

flagger = aoflagger.AOFlagger()
path = flagger.find_strategy_file(aoflagger.TelescopeId.Generic)
strategy = flagger.load_strategy_file(path)
data = flagger.make_image_set(ntimes, nch, 8)

ratiosum = 0.0
ratiosumsq = 0.0
for repeat in range(count):
    for imgindex in range(8):
        # Initialize data with random numbers
        values = numpy.random.normal(0, 1, [nch, ntimes])
        data.set_image_buffer(imgindex, values)

    flags = strategy.run(data)
    flagvalues = flags.get_buffer()
    ratio = float(sum(sum(flagvalues))) / (nch*ntimes)
    ratiosum += ratio
    ratiosumsq += ratio*ratio

print("Percentage flags (false-positive rate) on Gaussian data: " +
    str(ratiosum * 100.0 / count) + "% +/- " +
    str(numpy.sqrt(
        (ratiosumsq/count - ratiosum*ratiosum / (count*count) )
        ) * 100.0) )
```

This takes about 10 seconds to run on my computer.

CHAPTER 6

# C++ interface

AOFlagger provides external C++ and Python Application Programming Interfaces (API) to call the aoflagger from those languages.

**Note:** These interfaces are not used to design a custom flagging strategy. Custom flagging strategies make use of an internal Lua interface, which are described in the *chapter on designing strategies*. The external interfaces described here make it possible to "push data" through AOFlagger from other software.

These external interfaces allow, for example, the integration of the AOFlagger inside a pipeline and observatory. To use the interface, the C++ header file "`aoflagger.h`" is installed as part of the package, and can be `#included` in a program's source code. Additionally, the program needs to be linked with libaoflagger. The best way to link a C++ program to aoflagger, is by using cmake's `find_package` utility, which can also handle versioning, etc. For example

```
find_package(AOFlagger 3 REQUIRED)
include_directories(${AOFLAGGER_INCLUDE_DIR})
target_link_libraries(yoursoftware ${AOFLAGGER_LIB})
```

## 6.1 Reference

The documentation for the external C++ API is automatically extracted from the code using Doxygen. The C++ API reference that summarizes all functions, classes, etc. can be found here:

- http://www.andreoffringa.org/aoflagger/doxygen/

The Python interface mirrors the C++ API, and these pages can therefore also be used as reference for the Python interface.

CHAPTER 7

# Further information

The core AOFlagger algorithms are described in the following two papers:

- A morphological algorithm for improving radio-frequency interference detection, Offringa, Van de Gronde and Roerdink (2012), A&A, 539, 10, A95.
- Post-correlation radio frequency interference classification methods, Offringa et al. (2010), MNRAS 405, 155-167.

Flagging results for LOFAR and MWA using the AOFlagger are described in the following papers, respectively:

- The LOFAR radio environment, Offringa et al. (2013), A&A, 549, A11.
- The low-frequency environment of the MWA: RFI analysis and mitigation, Offringa et al. (2015), PASA, 32, e008

# CHAPTER 8

# Change logs

## 8.1 Version 3 releases

AOFlagger version 3 is the current major release number. The first release of version 3 was released on 2022-05-19. Stable releases:

### 8.1.1 v3.03

| 2023-03-20 | André Offringa <of...nga@gmail.com> |
|---|---|

Version 3.3 is mostly a bugfix release that fixes various little issues. The full list of changes is given below.

#### New features

#### Generic improvements

- Add features to implement local Lua RMS thresholding (tested in this paper):
  - `Data.__div()` (division operator in Lua)
  - `aoflagger.norm()`
  - `aoflagger.sqrt()`
- Rfigui now has an option to save data to a numpy array.
- Support LOFAR beam-formed h5 files.
- Use the *reordering reading mode* by default when the data does not fit in memory.

### Bug fixes & refactoring

- Fix for crash when using `aoqplot -save` due to uninitialized pango.
- Fix crashes with empty quality statistics tables found by Tammo Jan.
- Fix compability issues with newest pylibs.
- Fix various plotting imperfections.
- Fix an exception when a recent file does not exist.
- Fix a cast error in the Python FlagMask.SetBuffer() method that caused it to misbehave.
- Improved a few warnings.
- Replace deprecated `math.pow` by `^` in Lua scripts.
- Fixes for Mac compilation.
- Format Python files using black.
- Format Lua scripts with StyLua.
- Add documentation for `aoquality combine` operation.
- Memory estimate for the reader was improved.

## 8.1.2 v3.02

| 2022-05-19 | André Offringa <of...nga@gmail.com> |
|---|---|

Version 3.2 of aoflagger allows concattenating measurement sets in frequency and process these in parallel. One of the use-cases for which this is useful, is the Prefactor/LINC pipeline for processing LOFAR data. Apart from that, a few smaller Lua options were added, and various bugs were squashed and performance improvements were made.

The full list of changes is given below.

### New features

### Generic improvements

- Faster vertical masked sumthreshold algorithm.
- New Lua functions:
  - *aoflagger.upsample_mask()*
  - *Data.invert_mask()*
  - *Data.set_masked_visibilities()*
- Allow specifying penalty in *aoflagger.scale_invariant_rank_operator_masked()*.
- Add support for Sdhdf files. This adds the requirement to have the hdf5 library installed.
- Add the Apertif 2021-03-09 strategy and set it as default for Apertif.
- Allow downsampling in `aoquality query_f`.
- Support for Function Multi Versioning, which makes the packaged versions run faster.
- Spectrally concatenate different measurement sets and perform IO in parallel.

#### New rfigui features

- New 'open file' window that allows opening multiple files and setting options.
- Ask confirmation when closing rfigui without saving changes.
- Add time/frequency options to rfigui.
- Add a new simulate-data window to rfigui with more simulations options.
- Add 'select zoom' option to rfigui.

#### Bug fixes & refactoring

- Data with NaNs would cause some of the sumthreshold algorithms to behave incorrectly.
- Various fixes for newer compilers (gcc 11/12) or different environments.
- Fix return value of *`Data.get_complex_state()`*, which always returned `complex`.
- Report clear error when the Lua version is older than 5.3.
- Fix default strategy when `exclude_original_flags = false`.
- Several plot-drawing improvements.
- Fix loading of ATCA strategy in rfigui.
- Let ATCA strategy work when not all polarizations are available.
- Make Apertif strategy also work when no bandpass is available.
- Remove libxml2 dependency.
- Add CMake option for skipping GTKMM builds.
- Add HINTS to find FFTW3 in cmake.
- Make sure all python examples run.
- Improve performance of calculating median window (time threshold operation).
- Require C++17, use std instead of Boost where possible.

### 8.1.3 v3.01

| 2021-02-24 | André Offringa <of...nga@gmail.com> |
|---|---|

This version provides new functionality which allows some further flagging flexibility. In particular, it is possible to flag parts of the frequency range with different strategies (for Apertif), changes were made to support querying metadata from Lua when using DP3 (for NenuFAR) and support for the LBA flagging strategy was added back. The gui also got several improvements, and quite a lot of bugs were squashed.

The full list of changes is given below.

### New features

- New open dialog that will eventually provide more options for opening multiple measurement sets at once. Fixes #26.
- New Lua functions to support flagging data partially:
  - *aoflagger.copy_to_channel()*
  - *aoflagger.copy_to_frequency()*
  - *aoflagger.trim_channels()*
  - *aoflagger.trim_frequencies()*

  As an example: this is used by Apertif to use a different strategy for the protected 1400-1420 MHz band, avoiding accidental flagging of HI.
- New Lua functions to specify version requirements:
  - *aoflagger.require_max_version()*
  - *aoflagger.require_min_version()*
- New Lua function *aoflagger.normalize_bandpass()* to normalize the bandpass (used by LBA strategy).
- The main `rfigui` window can now show a power-over-time plot above the dynamic spectrum heatmap and align the axes of the two.
- Several strategies were improved / added, including a LOFAR LBA wideband strategy, an improved Apertif strategy, an ATCA strategy and a NenuFAR strategy (the latter was made by F. Mertens).
- Make it possible to propagate metadata from the API to Lua scripts. Amongst other things, this makes it possible to use measurement set metadata from within Lua scripts when aoflagger is called from DP3. This is used for NenuFAR.

### Bug fixes & refactoring

- pybind11 is now put into the source as submodule, which means it is no longer a dependency.
- Fix the 'select active data' option so that it honours zoom settings.
- Fix "Export baseline" dialog so that it shows *.rfibl* files.
- Fix a crash when using *aoquality query_fr/query_g* on collected quality statistic sets.
- Fix hang when reading (sd)fits files.
- Add CPack options to CMakeLists.txt to enable creating Debian packages.
- Apply automated formatting of source code.
- Require at least Lua 5.2.
- Fixes for compilation on Macs (with help from JM van der Hulst).
- Fix build when SSE is not available.
- Fix issue causing crash when flagging multiple bands (Identified by I. van Bemmel, collect_statistics could crash when an MS has multiple bands).
- Don't output a confusing message when running on empty data.

### 8.1.4 v3.00

| 2020-07-21 | André Offringa <of...nga@gmail.com> |
|---|---|

In version 3.00, AOFlagger has fully switched to using *Lua-scripted strategies*. See *Upgrading from rfis to Lua scripts* for information on how to upgrade. For the Lua changes, the `rfigui` underwent a transformation, and now provides a simple Lua editor and runner. Furthermore, the AOFlagger repository has been migrated to Gitlab: https://gitlab.com/aroffringa/aoflagger is its new location. GitLab is faster, provides CI, an easier and more advanced ticketing system, merge requests, etc. Additionally, the old Wiki was transformed into Sphinx documentation that is automatically build and readable online at https://aoflagger.readthedocs.io/.

The full list of changes is given below.

#### New features

##### `rfigui`

- Options for loading default strategies.
- Add menu item for opening recent files.
- Some restructuring of toolbar buttons and menus.
- Combine open file/open directory options into one open option.
- Save baseline flags from the gui.
- Application icons are now found even when installed in a user prefix.
- Show a progress bar for long-running tasks.
- Directly load and save baseline-integrated data/flags (coherently/incoherently averaged, time/frequency differenced).

##### `aoflagger`

- Run `.lua` files instead of `.rfis` files.

#### Generic

- Add Lua files for all supported telescopes.
- Use pybind11 instead of boost-python.
- New C++/Python *external interfaces* that supports loading and running Lua scripts.
- Provide CMake config files for easier configuration for software that depends on AOFlagger.

#### Bug fixes & refactoring

- Simplify directory structure
- Add a few new unit tests and an automated CI system
- Fix polarization button logic

- Make use of 'aocommon' repository
- *v3.3* (2023-03-20)
- *v3.2* (2022-05-19)
- *v3.1* (2021-02-24)
- *v3.0* (2020-07-19)

## 8.2 Version 2 releases

### 8.2.1 v2.15

| 2020-02-20 | André Offringa <of...nga@gmail.com> |
|---|---|

The biggest change in AOFlagger 2.15 is the use of Lua as a scripting language for defining flagging strategies. After having tried to use Python for this, it unfortunately turned out that (without complicating things severely) Python is neither fast enough nor flexible enough to allow multi-threaded processing of the data. Lua, aimed at being a fast, embedded language, turned out to be a very good match for making aoflagger strategies. Lua strategies can be tested with the `aolua` command line tool. Another important thing about this release, is that it will be the last release of the major 2.x branch, whereas the 3.0 release will drop support for several things and move to Github.

As a result of using Lua, AOFlagger now depends on the lua 5.3 libraries. These are provided by the Ubuntu/Debian `liblua5.3-dev` package. Apart from the switch to Lua, as usual with new releases, AOFlagger 2.15 fixes various small issues and enhances the gui. It also includes a few new features that are aimed at flagging Apertif data more accurately.

#### Full list of changes

#### New features

- `aolua` command line tool can be used to run `.lua` scripted strategies. This allows for much more flexibility within the flagging scripts and, compared to the old 'xml' block-wise strategies, also makes it easier to understand what happens. This release implements all functionality, but the Lua tools are not completely stable or documented. This can be seen as a prerelease that allows testing with Lua strategies while all 'old' functionality still works, whereas 3.0 will fully switch to using Lua.
- The gui can also run lua scripts. This is done by adding a file called `strategy.lua` to your working directory. This will become easier / more flexible in AOFlagger 3.0.
- The *C++ interface* now has functionality to specify "input" flags that indicate data that is bad (e.g. because of a correlator malfunction, missing subband-edges, etc.), which are used to improve flagging accuracy by keeping the flagged visibilities out of collected statistics.
- The C++ API ("pipeline interface") is now completely wrapped with *a Python interface*. There are now two interfaces for Python: the "pipeline" interface that can be used to pass data that is flagged by a predefined strategy, and an "algorithmic" interface that can be used to fully define a strategy from within Python. The latter is too slow for the processing of large datasets, and is to be replaced by the Lua functionality. For the first part, the "pipeline interface", performance is not so relevant, and Python works well for this.
- New action "restore channel range" can be used to make sure that legimite lines (e.g. HI) are not flagged.
- Single baseline `.rfibl` files can now be processed by the `aoflagger` program.

- Add colormap 'cool' (similar to ds9) and 'rainbow'
- New parameters `-interval` and `-max-interval-size` to split the data and thereby lower the memory usage (suggested by A. Drabent; [[#84](#84)]).
- Some improvements to the *C++ interface*.
- Some small GUI improvements, related to metadata, plot axes, etc.
- Rewrote code for displaying menu
- Option `-hide-flags` added to rfigui (requested by R. Schulz)
- Include new, faster version of SumThreshold algorithm, written by Nicholas Dumas. This algorithm is used for Length>=64 when AVX is available.

#### Bug fixes

- Fixed a crash when opening corrupted input files
- Output error and warning messages when something goes wrong in a parallel thread
- Don't require write access to a measurement set when not actually requiring it
- Don't crash when a Python exception occurs in a Python strategy
- Better support for running several aoflagger instances in parallel
- Several small compilation issues and refactoring

### 8.2.2 v2.14

| 2019-02-14 | André Offringa <of...nga@gmail.com> |
|---|---|

This release contains some important bug fixes. Due to stricter requirements in gcc, 2.13 did no longer compile with new gcc versions. This is fixed in 2.14. There are also some new features that enable a good flagging strategy on concatenated LOFAR subbands.

Full list of changes:

#### New features

- Save and load single baseline into a binary file.
- Correct for smooth bandpasses.
- Add minimum version for boost: Require Boost 1.55.
- Return non-zero value when exceptions occurs (reported by W. Williams).

#### Bug fixes

- Fix VisualizeAction memory leak bug, causing memory problems (reported by A. Drabent).
- Fixing compilation issues with newer gcc versions (#81).
- Correct y-axis in logarithmic plots.
- Correct averaging in time/frequency with change resolution action and masks.

- Save the 'average with mask' property of changeresolutionaction.
- Fixing spelling error in debug output (pointed out by T.J. Dijkema).
- Remove unnecessary cmake search for signals resulting in compilation errors with Boost>=1.68 (reported by M. Brentjens).

### 8.2.3 v2.13

| 2018-11-07 | André Offringa <of...nga@gmail.com> |
|---|---|

This quick release, only two months after the previous release, is to make a feature available that was found to be required for Apertif: to temporary apply a passband by means of a text file. It also fixes some minor bugs.

Full list of changes:

#### New features

- Apply passband action that can be used to read in and apply a text file with gain values, implemented *-bandpass* parameter for aoflagger
- New colourmap: cubehelix and "colourful" cubehelix

#### Bug fixes

- AOQPlot required X/graphics when in non-interactive mode
- Slight improvement to API interface (Make QualityStatistics nullable)
- Baseline plot was saved without colour (reported by B. McKinley)
- Fixing potential use of uninitialized value in aoqplot

### 8.2.4 v2.12

| 2018-09-03 | André Offringa <of...nga@gmail.com> |
|---|---|

This version contains quite some changes and a new features. The gui has several usability additions, such as better zooming, visualization of intermediate products and improvements to plots. There are now options for taking into account flags that are already set in the observation before flagging, e.g. when they are set by the correlator. Several changes were added for APERTIF testing, such as coadding of different beams to improve RFI detection.

Full list of changes:

#### New features

- Zooming can now be performed by drawing a rectangle on the image widget or scrolling the mouse wheel (#50).
- Implemented panning by holding right mouse button.
- Enabled proper double buffering of the dynamic spectra, which improves snappiness.
- Improved system to show residual / background plots in `rfigui`: strategies can add visualizations to the gui, making it possible to inspect the smoothed and residual plots for different iterations.

- Plots in `rfigui` and `aoqplot` now contain a legend (#62), and have various other esthetic improvements.
- Improved "edit action" frames for change resolution and high pass filter actions.
- The binary file `rficonsole` that had been renamed to `aoflagger` has now been removed completely.
- In `rfigui`, measurement sets can now be "coadded" in amplitude by specifying multiple mses on the command line (requested by the APERTIF team).
- **Strategies can now take into account existing flags in the measurement set, e.g. those set by the correlator. In particular:**
  - Finished implementation of SumThreshold algorithm with missings, including new option in SumThreshold action to turn it on.
  - Implemented SIR operator that can apply on masks in which values are missing, such as when the original flags of an obs are set by the correlator.
- Added a 'clear original flags' button.
- Major cleanup and improvement of the strategy wizard, with preselected telescope and an extra page separating observational and technical settings.
- Added option 'interval' to `aoquality collect`.

#### Bug fixes

- Assigning from or to an empty FlagMask would lead to segfault.
- The "Change resolution" action with a resizing factor of 1 now works.
- Fixed problem with conversion of some circularly polarized data.
- Renamed 'statistical flagging' to more appropriate term 'morphological flagging'.
- Fixing error on Mac, `exp10` use without including math (reported by T. van der Hulst - Version 2.12 compiles succesfully on the Mac).

Version 2.12.1 fixes an issue with compilation on 32-bit and non-x86 systems, but is otherwise the same as 2.12.0.

### 8.2.5 v2.11

| 2018-06-12 | André Offringa <of...nga@gmail.com> |
|---|---|

Version 2.11 is mostly just incremental improvement, with no major new features but some important fixes. Worth mentioning is that the AOFlagger interface is integrated into the aartfaac preprocessing pipeline.

Full list of changes:

#### New features

- AARTFAAC telescope added to standard list of recognized telescopes, and integration with aartfaac2ms.
- SumThreshold's time and frequency sensitivity can now be separately specified (#20).
- Added an AVX2 SumThreshold algorithm. It is not much faster, but there seem to be cases where it is faster than the old SSE algorithm.
- The `badstations` tool has been optimized for AARTFAAC, where it seems very useful.

- Some minor plotting fixes / improvements.
- **Better support for various versions of the sdfits files that various telescopes use, in particular better support for GBT due**
  - TDIM specification is no longer required;
  - Several columns are now optional;
  - IFNUM is allowed as keyword instead of IF.

#### Bug fixes

- Compilation error, function `pow10` was removed from glibc 2.27 and `exp10` is used instead (#73, #74, #75).
- Possible segfault when running strategy caused by sumthreshold action with incorrect ptr use in c++11 changes.
- Possible bad_alloc exception due to bug in FrequencySelectionAction (#72), reported by P. Williams).
- The `rfigui` program no longer crashes when zooming in too far (#66).
- Updated dependencies/cmake config to reflect updates in boost python and glibmm (#71).
- PQ/QP spelling error.
- CMake prints "GTKMM found", regardless of whether GTKMM was found (#76).

### 8.2.6 v2.10

| 2018-02-06 | André Offringa <of. . . nga@gmail.com> |
|---|---|

Version 2.10 brings a few small but useful new features, and many small bugfixes. A lot of code was cleaned up and improved to use C++11 features.

Full list of changes:

#### New features

- Initial work on Python-scripted strategies; to be fully implemented in aoflagger 3.0.
- SPWs can be concattenated (#65).
- Added option in StatisticalFlagging to flag entire timesteps, channels or baselines when too little visibilities are left.
- A graphical termal ('X') is no longer necessary to run *rfigui* / *aoqplot* with drawing parameters, which allows saving plots in scripts.
- The aoqplot status window now also displays antenna index when hovering a baseline.
- Added interpolate flags menu item and extended algorithm to run multiple times to fill in spots that are still missing.

### Bug fixes

- Stokes I-only images are not shown because they cannot be selected (#63).
- Check polarizations when changing settings (#69).
- It could happen that exceptions were not shown properly (bug in baseexception).
- Unselected original flags would still show them in plots and 'keep..data' actions.
- Fixed x/y flip when opening a parmdb.
- Compilation improvements.
- Fixing frequency and time scatter plots in rfigui and plots in aoqplot for non-xx/xy/yx/yy polarized sets (#70) e.g. JVLA; reported by M. Verheijen).
- Don't plot polarizations when they are not available in aoqplot, as that can cause segfault
- **Esthetics of plots are improved.**
  - Use a clipping region to prevent points falling outside the canvas.
  - Unset value could cause the z-axis to disappear.
  - Make logarithmic Y ticks be better behaved.
  - Y axis in logarithmic plots could have the wrong range.
- **Major code cleanup:**
  - Many code has been cleaned up to use C++11 features.
  - Dependence on Boost has been decreased.
  - Directory structure was improved and old code was removed.
- Fixing multithreading of for-each-baseline action in ArtifactSet

## 8.2.7 v2.09

| 2016-12-20 | André Offringa <of...nga@gmail.com> |
|---|---|

This version adds support for circular polarizations in various locations, making AOFlagger visualize and interpret JVLA data correctly. Several other polarization issues were fixed as well (mostly reported by B. Adebahr while working on Apertif data), making it easier to browse polarized data with the rfigui.

### New features

- Toolbar buttons that select the displayed polarization (#61).
- Add proper support for circular polarizations (#17).
- In the plot options window, one can now select a logarithmic X axis.
- New plot: a frequency scatter plot, similar to time scatter plot.
- Close execute strategy frame when finished by default, unless asked to remain.
- Toolbar buttons of aoqplot now have tooltips.
- When the Viridis colour scheme is selected, masks will be white and black.

### Changes to default strategy

There are some slight changes in the workings of the actions, which changes the default strategy slightly:

- The "for-each-polarization" action now propagates the background/residual images, and actions after the f.o.p. action will now be executed on the residual image.
- This "for-each-complex-component" action now always propagates the data, despite whether "restore from amplitude" is checked. If that option is checked, it now means proper flux units are restored in amplitude mode, by dividing amp by sqrt(2).
- Threshold of final time selection action is now 4 instead of 3.5 sigma. This is because it is now executed on the foreground-removed data because of the change to the for-each-polarization action, and tends to be slightly more sensitive.
- Added a new buffering scheme to reordering which seems to speed up the reordering significantly.

### Bug fixes

- Corrected reading of Polarization table, affecting some measurement sets with two polarizations (#60, reported by B. Adebahr).
- Displaying a single polarization would still show the flags from all polarizations (#59, reported by B. Adebahr).
- Manually specifying -field, -bands or -j would not work when specifying a custom strategy (reported by B. Adebahr).
- The time-frequency display of aoqplot would show every set with only one channel.
- Use SONAME and SOVERSION for libaoflagger.so (#56, patch by O. Streicher).
- The 'clear alt flags' button did not longer work.
- Fixing unset state of original/alternative flag buttons.
- Major refactoring of TimeFrequencyData structure.
- A crash could occur in the plot frame when pressing clear plots button.
- Better increments for ranges in various action editting frames.
- Correcting help for *quality collect -d*
- **Several compilation problems were fixed:**
  - Compilation error on new platforms due to wrong Glib::RefPtr comparison (#57, patch by O. Streicher).
  - Compilation errors on Macs (reported by T. van der Hulst).
  - Compilation issue on gcc 5.3.1
  - Provide conditions on the use of SSE intrinsics, such that AOFlagger can be compiled on other platforms.

## 8.2.8 v2.08

| 2015-06-21 | André Offringa <of...nga@gmail.com> |
|---|---|

This version has no major new features, but some small enhancements and quite a few fixes for small bugs.

### New features

- Functionality in the interface to set a custom error handler and progress reporter (this also fixes exceptions ocurring whilst running within NDPPP).
- New aoqplot interface
- Adding a 'keep window open' checkbutton to the Goto window, requested by F. de Gasperin.
- Make the default size of the goto window somewhat larger.
- Adding option '-save-baseline' to rfigui for saving baseline plots in a non-interactive way
- Adding option '-data-column' to rfigui for selecting data column when saving baselines.
- Added an 'export data' tool to image widget.
- Use object libraries to speed up compilation and avoid compiling files more than once.
- Viridis colour scale.
- Making it possible to give column name in `aoquality collect`.
- Implementing CPU affinity patch by J. D. Mol, allowing e.g. SLURM to be used and have aoflagger use the right number of threads.

### Bug fixes

- Issue with `aoquality collect`, causing to not properly collect statistics.
- Turning on C++11 compilation, because this is now necessary to compile with the latest GTKMM.
- Segmentation fault in interface that seemed to occur because of bug exposed by newer compiler optimisations.
- When loading a strategy, -j did not work.
- Change assignment to real and imag part, for compatibility with libc++ and c++11 (patch by T. J. Dijkema)
- Avoiding compatilibity issues with different versions of GTKMM.
- Making sure that domain name in hostname doesn't make aoqplot halt.
- Improving warning when baseline not found in rfigui.
- Gracefully handle exceptions before gui has opened.
- Skip NaNs in the frequency plot.
- Fixing crash with error 'requested baseline is not available in measurement set', reported by D. van der Vlugt.
- When reporting progress, flush the correct stdout/stderr stream.
- Issue in CMakeLists.txt: GSL is not included when it is used.
- Fixed compilation warning (std::auto_ptr deprecated).
- Textual: Removing all references to my old e-mail address.
- Textual: Correcting wrong text above for-each-polarisation frame.
- Refactoring: Refactored source code with new 'structures' directory, clean up, removed timestepaccessor.

### 8.2.9 v2.07

| 2015-06-12 | André Offringa <of...nga@gmail.com> |
|---|---|

#### Summary

No really shocking changes, but lots of small improvements. It is now possible to save quality statistic plots from the command line and to re-flag by opening quality statistics. There are also many small gui tweaks and a lot of small bugfixes. Release 2.7.0 supports both Casacore's 1.x and 2.x versions – 2.7.1 will only support 2.0. The release of 2.7.1 was mainly to work around a bug in GTKMM in Ubuntu's latest distribution, causing the rfigui and aoqplot windows not to open.

#### Changes from 2.7.0 to 2.7.1

2.7.1 was released on 2015-09-02.

- Code was converted to Casacore 2
- Bugfix: Fixed two compiler warnings
- Bugfix: Applied patch by M. Sokolowski to correct time range of BIGHORN fits files
- Bugfix: crash when opening main windows in certain GTKMM installations (notably the one currently in Ubuntu)

#### Changes from 2.6 to 2.7

- Feature: Save the 'aoqplot' plots from the command line with option -save (#51).
- Feature: Quality sets can be opened in the rfigui & on the command line with aoflagger.
- Feature: Toolbar buttons for zooming in/out (related to #50).
- Feature: Icons for 'Show original/smoothed/residual visibilities' toolbar buttons.
- Feature: Tooltips for all toolbar buttons, which are displayed by holding the mouse above a button.
- Feature: When the AOFlagger icons are available, the text is no longer shown on buttons in the toolbar to save space.
- Feature: Added -version and -help parameters to aoqplot.
- Feature: Run 'aoquality liststats' to get list of stat names, and some more info in aoquality's help.
- Feature: Show progress on command line when opening multiple QS files.
- Feature: Change resampling method in plot properties window; default filter now nearest neighbour (related to #31).
- Feature: Collect time-frequency information in aoquality collect.
- Feature: New colour map, 'fire', going via black to red to yellow to white.
- Feature: Specify data column during collecting.
- Bugfix: Solving compilation issue when building with shared libraries (-DBUILD_SHARED_LIBS=ON) instead of a static library, to make it possible to link AOFlagger with new NDPPP.
- Bugfix: Adding newer FindCasacore.cmake from LOFAR repository, which supports casacore 2.

- Bugfix: Crash when pressing background or difference button when no image is loaded.
- Bugfix: Fixed all deprecated usage of GTKMM. Everything can now be compiled with '-DGTKMM_DISABLE_DEPRECATED' set (#52).
- Bugfix: Application icon not shown in About window.
- Bugfix: Patch by M. Brentjes to prevent using Casapy's boost, which causes compilation errors.
- Bugfix: Compiling without GTKMM results in undefined reference due to missing vdsfile.cpp on some platforms (reported by G. van Diepen).
- Bugfix: Making it possible to run strategies from the API that include plot actions, which includes the default LOFAR strategy.
- Bugfix: Solved several Mac compilation issues (reported by A. Biggs).
- Bugfix: Make use of cmake package finders for casacore and cfitsio (patch by T.J. Dijkema)
- Bugfix: Sometimes plots were not updated when changing their parameters.
- Cleanup: Using find_package(Threads) instead of find_library in cmake (suggested by M. Brentjens).
- . . . and some more clean ups.

### 8.2.10 v2.06

| 2014-06-26 | André Offringa <of. . . nga@gmail.com> |
|---|---|

#### Summary

This release fixes a lot of small bugs, in particular in the GUI. Support for Filterbank sets was added, and exporting strategies from the GUI to the command line has been made easier. Several other small improvements were made too, of which my personal favourite is that it is now possible to zoom in on the time-frequency plot by pressing ctrl plus (+).

#### Change list

- Implemented the 32-bit SIGPROC Filterbank format for flagging pulsar data.
- Implemented zoom in, zoom out and zoom to fit commands in View menu, with accelerator keys ctrl =/-/0. Moving mouse over area of interest and pressing ctrl '=' zooms in on that area. I use this all the time now :-). (#40).
- Automatically add for-each-baseline/write actions when not present in loaded strategy. This makes it possible to create a strategy in the gui, and when it works on one baseline, save it without changes and run it with 'aoflagger' on the entire set (#47).
- Added default strategy optimization option for high-time-resolution observations.
- aoflagger now accepts -bands and -fields followed by comma-separated list of ids to select specific bands/fields. Requested by O. Smirnov (#29).
- Added GUI editor for the FrequencySelectionAction, patch by P. Williams.
- Renamed menu 'Go' to 'Browse'.
- Added buttons to browse to shortest and longest baseline and to reload current baseline.
- Removed 'large step' browse buttons

- Disable browse buttons when only one baseline available (#34)
- Improved time axis when ticks have a sub-second distance.
- Improved *C++ interface* added methods to collect histograms and to get AOFlagger's version.
- Making it possible to set the axis descriptions and horizontal range on a 2d plot.
- Output sample count in summary page of aoqplot.
- Bugfix: Unable to compile on Ubuntu 14.04 with gcc 4.8 due to incorrect pthreads linking, patch by G. Molenaar.
- Bugfix: Fixing bug in opening of files of length 6 chars, reported by P. Williams.
- Bugfix: Edit strategy window crashes when the strategy is changed while an action is selected.
- Bugfix: Crash when moving mouse over a zoomed image.
- Bugfix: Solved a problem when rewriting data with indirect baseline reader.
- Bugfix: Edit strategy window is too small and hides action properties (#48).
- Bugfix: Axes invalid for very large sets (#49).
- Bugfix: Compiler issues with linking lapack/cblas on some platforms.
- Bugfix: Problem in writing the flags of Bighorns fits file: zeros were replaced with nans.
- Bugfix: In ChangeResolutionAction, restoring the revised image when decreasing in frequency resolution did not work.
- Bugfix: Issues with calculation of free mem on 32-bit machines & improved mem reporting.
- Bugfix: Updated old e-mail address in flagger strategy files, write aoflagger version to the files.
- Bugfix: "make check" fails for 32 bit build, reported by G. Molenaar.
- Bugfix: Boost.Signals deprecated: switch to Boost.Signals2 (#43).
- Bugfix: Fixed documentation of HighPassFilterAction, patch by P. Williams.
- Bugfix: Crash when pressing "clear plots" button in plot window.
- Bugfix: Warnings issued by Intel compiler (#42)
- Bugfix: Compilation errors when using newest Casacore
- Bugfix: Fixed some compilations error on the Mac (but it still doesn't compile flawless). Thanks to E. Lenc.
- Bugfix: Issue with NaNs in 2d plot.
- Code cleaning: Removed deprecated code: Collect statistics action and various old observation formats that are no longer in use.

### 8.2.11 v2.05

| 2013-08-24 | André Offringa <of...nga@gmail.com> |
|---|---|

The GUI has been converted to GTK 3. Consequently, you will need the GTKMM 3 library to compile the graphical programs. This library version is at present not available in some (common) distributions, so if you need the graphical programs, make sure your distribution ships GTKMM 3. Also nice to report is that the AOFlagger has been succesfully applied on Arecibo 305m and BIGHORNS data, and default strategies were added for the two. Also, various bugs and performance issues were squashed which should make it worthwhile to update.

### List of changes

- Switched to GTK/GTKMM 3 (#26).
- A default strategy for Arecibo 305m telescope was added.
- A default strategy for "BIGHORNS" telescope was added, and support for its file format was added with help from M. Solovsky (#36).
- Multiple sets can now be opened in aoqplot without using a cluster file: aoqplot `*.ms` will work now (#39).
- Added an execute strategy button to the toolbar (#35).
- Version info can be retrieved with "aoflagger –version" and the about box in the rfigui (#28).
- Name of open sdfits files is now displayed in GUI (#30), requested by J. Delhaize.
- Antenna map feature was removed because of bugginess and little use.
- API change: Image buffers can now be resized without reallocation, which was necessary to avoid memory fragmentation in the Cotter MWA preprocessing pipeline.
- API performance fix: Collecting statistics was quite slow in the API. Speed of this task is now 7x improved (#16).
- Bugfix: Antenna plot in aoqplot did not properly integrate statistics over all antennae (#37).
- Bugfix: Plots in rfigui sometimes use unselected flags (#38).
- Bugfix: Arecibo single-dish files caused GUI to crash (#33), reported by L. Hoppmann.
- Bugfix: Reading with direct reader on large sets is too slow (#27), tested by F. de Gasperin.
- Bugfix: Extensions '.uvfits', '.uvf' and '.fits' are now recognized as uvfits files.
- Bugfix: The command line 'aoflagger' application did not properly write the flags for MWA datasets.
- Bugfix: Crash when setting change resolution factor to 0 (#24), reported by F. de Gasperin.
- Bugfix: Crash when pressing Vert EVD in multi-polarization datasets (#24).
- Bugfix: Possibly race condition in producer-consumer construct used at various locations.
- Bugfix: Compilation error with gcc >= 4.8.1 (#32), reported by M. Brentjens.
- Bugfix: Fixed several other compilation issues on more exotic systems.
- Some cleaning of some code (#15).

## 8.2.12 v2.04

| 2013-04-06 | André Offringa <of...nga@gmail.com> |
|---|---|

AOFlagger changes for version 2.4.0

This version took longer than usual, but this is reflected by many large changes, most prominently: JVLA compatibility due to multi-SPW and multi-field support, improved Parkes compatibility due to SDFITS fixes, automatic loading of a strategy that matches the telescope and observation properties and switching between plots in the RFIGui.

### Changes

- Added a simple Parkes strategy (#19) – not extensively optimized, but seems to do a reasonable job (. . . much better than simple thresholding)
- Measurement sets with multiple spectral windows are now fully supported (#11). All [reading modes](reading mode) will correctly read and write such measurement sets.
- Measurement sets with multiple fields are now fully supported (#18). This and the previous feature has been requested many times, in particular to be able to easily flag JVLA data without 'split()'-ting off the bands or sources. As with the previous feature, all reading modes will now correctly read and write multi-field measurement sets. Each sequence of correlations on a single source will be flagged individually. Thanks to R. van Weeren for providing test sets.
- The RFIGui and aoflagger, when not specified otherwise, will now load an optimized strategy when an observation is opened. (#10) Telescopes for which this is currently supported are LOFAR, MWA, Parkes, JVLA (beta) and WSRT. In other cases, a "generic" strategy is loaded, which is the LOFAR strategy (which is actually not very generic – I hope to improve that in the next version).
- An initial JVLA strategy was added, that I've optimized for a ~1.1 GHz observation that I've received from R. van Weeren. Together with the other 3 previous features, this allows to simply execute "aoflagger my_jvla_set.ms" on a standard JVLA multi-source/spw file and get good results (if your sets look like the set I got) without any further tweaking. If you have, send me "typical" observations of other frequencies so I can optimize this further.
- The default WSRT strategy has been tweaked.
- The strategy wizard has been improved: a "fast" option was added and the sensitivity can be changed. (#12)
- The plotting window in the RFIGui now shows a list of all previous plots, which can be quickly reselected and its style can be changed (#9)
- The main window in the rfigui now shows the selected correlation name (e.g. RT0 x RT1). Can also be turned off and/or changed. (#8)
- The aoflagger console application will now automatically select the optimal number of threads when not overriden on cmd line (#6)
- The [indirect reader](reading mode) has been made a lot more efficient. It will "preallocate" the files it will write (if [your filesystem supports it](http://linux.die.net/man/1/fallocate)) and will no longer create a file for every correlation, but rather uses only one for data and one for flags. This overcomes problems with "too many open files" on some environments.
- When properly installed, the RFIGui and aoqplot applications will have a application/taskbar icon and show up in the menu of common desktop environments. (#14)
- Toolbar has been cleaned up, the more commonly used buttons are now there & some icons were added. (#13)
- Several menu items now got accelerator key, e.g. "F9" is "run strategy".
- Bugfix: G van Diepen send me a patch to fix variable length arrays and class/struct mixing, which caused problems on the Mac.
- Bugfix: Fixed a bug in the SDFITS reader/writer. With some help from J. Delhaize, this version is able to open and flag Parkes SDFITS file both with the RFIGui and the aoflagger console application. (#21)
- Bugfix: MWA strategy could silently fail
- Bugfix: aoqplot will ask for filename if none given (#22)
- Bugfix: Deadlock when showing error dialog box during execution of strategy
- Bugfix: fixing rare bug in RFIGui when a set contains invalid times

- Refactoring: The main RFIGUI class is huge, major efforts to clean it up and split some things in new classes in a model-view-control way.

### 8.2.13 v2.03

| 2012-12-16 | André Offringa <of...nga@gmail.com> |
|---|---|

Summary: A wizzard for creating strategies & solved compilation issues under Ubuntu

#### List of changes

- Fixing Ubuntu compilation error reported by A. Chippendale, D. Jacobs, R. Wayth and O. Smirnov. Ubuntu seems to use a different linker (~configuration?) than other Linux distributions, which causes the order of libraries that are linked to matter. The flagger should now also compile fine on Ubuntu 11-13.
- The AOFlagger now knows how to create custom strategies with certain modifications, that can be specified in a small wizzard window in the GUI. Some modifications include making the strategy more or less sensitive, making it insensitive for transient effects, making it more robust and being more aware of off-axis sources. The default strategy inside the GUI is slightly changed, which I think alows better experimenting for generic cases. (#4)
- Test suite is now again working (seems to have been broken in the transfer to new repository structure)
- It is now possible to open a png file inside the rfigui.
- Slight performance improvements to quality stat collector.
- Bug fix: VDS files (remote obs specifiers) could not be opened after the new repository structure. (#7)
- Bug fix: Added support for custom CFITSIO directory.
- Bug fix: Some supported GTKMM versions gave compilation errors (reported by M. Loose).

### 8.2.14 v2.02

| 2012-11-10 | André Offringa <of...nga@gmail.com> |
|---|---|

Summary: The AOFlagger was moved to a new repository, an API + docs were added, performance was (again :-)) improved.

#### List of changes

- Major repository restructering and moved to SourceForge! Many thanks to the LOFAR developers for hosting my project for so long in the LOFAR daily build, but it was time to separate the AOFlagger sources from LOFAR. This means the source structure is now much simpler, the project compiles faster and it is easier for me to package. Moreover, anyone can now enjoy the latest fixes by using the public SourceForge git repository. Sources are still published under the GNU GPL version 3 license.
- Official site is now http://aoflagger.sourceforge.net/ (NB: moved again to https://gitlab.com/aroffringa/aoflagger for version 3)
- The "rficonsole" executable has been renamed to "aoflagger". For the time being, a placeholder will warn you about the new name.

- The software is no longer depending on the log4cpp library (this used to be required for the LOFAR infrastructure).
- Faster high-pass filtering algorithm using SSE instructions, which has replaced the sliding window algorithm in the default algorithm (3 x faster filtering, 10% benefit in full strategy).
- New reading mode that reads a full set in memory. It is 25% faster compared to indirect when every thing fits in memory (on a 1gb set: Direct: 2m53, Indirect: 1m55, Memory: 1m27). This is now the default reading mode when enough memory is available.
- New public API. This API exposes a few simple & documented interface classes in a single header, that can be used to integrate the flagger and/or quality collector in an observatory's pipeline. It is used by the Cotter MWA preprocessing pipeline and will hopefully be used by LOFAR's preprocessing pipeline (NDPPP) some time in the future. Latest API docs can be found here: *C++ interface*
- New feature: action that can preliminary calibrate the passband (used in default MWA strategy)
- Bug fix: conversion from XX,YY to Stokes U did not work when no XY,YX polarization were available, as well as some other uncommon conversions.
- Bug fix: allow inverted vertical axes in plots, as well as some other plotting fixes.
- Bug fix: sdfits reader did not read polarizations properly.
- Bug fix: uvfits reader could not open MWA fits files properly.
- Bug fix: various uncommon crashes in rfigui.
- Bug fix: on opening measurement sets, option window now always opens on top.
- Bug fix: cmake will now fail when gtkmm version < 2.18.
- Bug fix: fixing problem reported by K. Buckley, which caused compilation errors with older (2.10) gtkmms.
- Bug fix: clarified some error messages in rfigui.

### 8.2.15 v2.01

| 2012-08-12 | André Offringa <of...nga@gmail.com> |
|---|---|

- This release adds support for .sdfits files that are used in Parkes data reduction and has various little enhancements. Strategy file format has not changed.
- New feature: Parkes' sdfits files can now be opened (with help from J. Delhaize) and the AOFlagger seems to be able to flag such observations well
- New feature: Save flags action can store flags to sdfits files
- New feature: New spectrum plot options in Rfigui (plot mean or sum of time steps)
- New feature: binary 'badstations', will use quality statistics to determine bad stations. Meant to be a fast tool for LOFAR data.
- New feature: allow reading non-standard columns in the gui (requested by R. van Weeren)
- New feature: new action 'Normalize variance', observations can be normalized by using the quality statistics.
- Bug fix: Sinc convolution in time direction can now be accomplished with FFT
- Bug fix: Added many improvements to the aoqplot quality plotting tool, i.e., better error msgs, more plots, bandpass/time correction, less crashes.
- Bug fix: No longer silently ignoring write errors during reordering – before this, when writing failed, flag results were wrong without warning.

- Enhancement: more options for aoquality tool.
- Bug fix: The uvfits reader did not work any more on WSRT files because of architectural changes in the AOFlagger. Fixed.
- Bug fix: AOFlagger can now flag data without time stamps (wich allows me to flag data from my personal hobby telescope).
- Bug fix: Open button now has key accelerator.
- Bug fix: AOFlagger did not compile on gcc 4.7 (reported by J. Swinbank).
- Bug fix: Toolbar in the Rfigui will now always show both text and icons
- Bug fix: Interface changed to allow NDPPP to read the raw data, NDPPP's problems fixed that concerned quality collecting.
- Performance enhancement: Improved speed of horizontal sumthreshold algorithm with SSE instructions, leading to about 10% improved speed on overal strategy.
- Bug fix: fixed bug in previously mentioned new SumThreshold algorithm causing crashes (reported by J. Swinbank), also adding test case to validate result.
- Performance enhancement: Improved performance of reading meta data of measurement sets.
- Experimental additions: the frequency filters are now optimized and can be applied on huge (LOFAR) observations.

### 8.2.16 v2.00.1

| 2012-03-16 | André Offringa <of...nga@gmail.com> |
|---|---|

Changes:

- This release is a quick fix for a bug that caused a strategy read failure on machines with non-standard locales
- Bug fix: fixing bug causing strategy reader to malfunction with different locales (reported by Arpad Miskolczi)
- Bug fix: show an error box when a strategy fails to load instead of crashing
- Bug fix: complex plane plot was still using gnuplot – now uses the internal plotter.
- Bug fix: fixed a bug that sometimes caused a crash after having resized the time-frequency diagram.
- Bug fix: building without gtkmm was not possible, patched by Marcel Loose.

### 8.2.17 v2.00

| 2012-03-08 | André Offringa <of...nga@gmail.com> |
|---|---|

Changes:

- Main points of this release: no longer depence on pdfviewer & gnuplot for drawing plots and new tools for quality analysis. Strategy file format did not change, thus .rfis files from 1.5.0 can be opened without a problem in 2.0.0.
- New tool: aoqplot, for very quick but superficial analysis of observations (see LOFAR Cookbook for info).
- New tool: aoquality, for collecting statistics (also see Cookbook).

- New feature: An internal plot renderer was added, which is used for all plots in the plot menu. The plots are therefore much faster and better integrated within rfigui, and no longer require gnuplot and a pdf viewer.
- New feature: ticks and text for logarithmic colour scales are visualized in a more esthetically pleasing way.
- New feature: Text along x,y,z axes of time-frequency plot can be manually set.
- New feature: log-log distribution plot in rfigui, with various analysis possible (slope calculation, rayleigh fitter).
- New feature: rfigui will now immediately ask which baseline to load, instead of loading the first baseline (requested by Raymond Oonk).
- New feature: you can now specify a MS on the commandline with the rfigui: `rfigui <ms>`
- Bug fix: Baseline name disappeared in bottom status bar when moving mouse over time-frequency plot. Baseline name is now reported when mouse is moved outside time-frequency plot (requested by Raymond Oonk).
- Bug fix: very large sets were not displayed correctly in the rfigui due to cairo limitations.
- Bug fix: clicking on statistics button sometimes crashed the gui.
- Bug fix: Times along x-axis in rfigui were not correct when splitting the data (reported by Raymond Oonk).
- Performance of dilation algorithm improved (is not used in default strategy).
- Slight performance improvement of SSE SumThreshold method.
- New library interface: the statistics collector is used by NDPPP to accumulate statistics during averaging (NDPPP and AOFlagger remain independ though).

AOFlagger version 2 was in development from 2011-10-20 to 2020-02-20. Stable releases:

- *2.15* (2020-02-20)
- *2.14* (2019-02-14)
- *2.13* (2018-11-07)
- *2.12* (2018-09-03)
- *2.11* (2018-06-12)
- *2.10* (2018-02-06)
- *2.9* (2016-12-20)
- *2.8* (2016-06-21)
- *2.7* (2015-06-16)
- *2.6* (2014-06-26)
- *2.5* (2013-08-24)
- *2.4* (2013-04-06)
- *2.3* (2012-12-16)
- *2.2* (2012-11-10)
- *2.1* (2012-08-12)
- *2.0.1* (2012-03-16)
- *2.0* (2012-03-08)

## 8.3 Version 1 releases

### 8.3.1 v1.5

| 2011-10-20 | André Offringa <of...nga@gmail.com> |
|---|---|

Changes:

- The main points of this release: Better performance, improved visualization and raw file support.
- The computational performance has been significantly improved (2x faster) using various optimizations, including rewriting algorithms to use the SSE instruction set.
- New feature: The time-frequency plots are customizable in various ways (View->Plot properties), and have been polished somewhat.
- New feature: Export visualizations as vector graphics (SVG, PDF) or bitmap (PNG).
- New feature: FITS export of image plane.
- New feature: Logarithmic colour scale (both in TF diagram and Image plane window).
- New feature: Tool tips in some windows.
- New feature: Raw files produced for the Transient Key Science project can now be read and written by the tools (format description from P. Zarka).
- The performance of the indirect reader has been improved (flags are also reordered during write).
- Added a different interpolation algorithm
- Menus in the gui have been restructured.
- New testset simulating sinusoidal RFI.
- Bugfix: Cairo behaved differently on some platforms, and did not show everything.
- Change resolution action can now optionally take flags into account.
- Fixed some issues with UV projection and simulation runs.
- RFI console now returns a status upon finishing, useful for automated pipelines (suggested by C. Coughlan).
- Heavy refactoring of visualization code.
- Better test coverage.

### 8.3.2 v1.4

| 2011-07-22 | André Offringa <of...nga@gmail.com> |
|---|---|

Changes:

- New action: absolute threshold. Useful for experimenting (not to be used otherwise).
- Speed increase of about 10% due to using a novel linear performing algorithm for the dilation. Algorithm provided by J. van de Gronde (article coming up).
- Several enhancements to fringe filters (see Offringa et al, 2011, in prep.).
- Enhanced performance of statistic collecting strategies.

- Added / enhanced possibility to experiment with spatial filters, thanks to a lot of input from U.-L. Pen.
- Esthetic changes to the gui by regrouping the menus.
- msinfo reports somewhat more useful info now.
- colormapper can now average fits images together and has some other useful features.
- Bugfix: Converging of sets with lots of RFI did not work correctly. Reported by R. van Weeren and D. Rafferty. Won't work for old strategies(!).
- Bugfix: Winsorized variance calculation is now more accurate with large amounts of RFI.
- Bugfix: Since the new actions screen was getting too large, the edit strategy now shows a menu of possible actions. Reported by A. Miskolczi.
- Bugfix: Removed unused parameters of the time selection action. Reported by P. Serra.
- Bugfix: Parameter -c removed from rfistrategy, as it did not work. Reported by F. De Gasperin.
- Several new unit tests to validate flagger steps.
- Some small fixes everywhere around.

### 8.3.3 v1.3

| 2011-04-26 | André Offringa <of...nga@gmail.com> |
|---|---|

- Several esthetic changes to the gui, including rounded values on the axes, a z-axis scale and fixed z-scale
- Feature to create a graphical map of the antennas in a set
- Various improvements to collection of statistics, such as time-frequency noise plots
- Dynamic noise spectra generated by the CollectNoiseAction can be opened and re-flagged in RFI gui.
- Added several unit tests to test correctness of the more complicated algorithms
- New action that acts like it resamples the data in a specific way, for testing it side effects.
- Fringe fitting can be performed on any source now
- Bugfix: Fix of 1d convolution out-of-bounds error, reported by R. van Nieuwpoort
- Bugfix: fitting of curve when no data is present
- Bugfix: Fix of somewhat confusing side effect of the dilation operation when the Equal polarisation setting was removed. Reported by A. Biggs.
- Bugfix: Fix for bug reported by P. Serra, flagging on Stokes Q in a 2 polarization (XX,YY) set did not work
- Many other small bugfixes

### 8.3.4 v1.2

| 2011-02-28 | André Offringa <of...nga@gmail.com> |
|---|---|

- Dependency on GSL removed
- Added RMS plot option
- Added option to rficonsole to specify data column used for flagging

- Added opening of solution tables
- Added some checks and error messages when making common strategy errors
- RFI console will now output the total percentages per polarization
- Added support for skipping mses that have already been flagged
- The edit strategy window works more intuitively now
- Goto window will select current baseline when opening
- RFI Console now writes an entry in the HISTORY table of the Measurement Set
- Logging has been formatted and start time was added.
- Added "Set and show image plane" button in plot menu
- Added feature to continue with already resorted MS in indirect baseline reader
- The AdapterAction has been replaced by the ForEachComplexComponentAction
- Strategy XML file is now formatted to make it human readable
- Allow indirect reading of an MS
- The indirect reading mode can be used in the GUI
- Allow reading of raw RCP files
- Changing the default threshold for baseline selection suggestor to 8 sigma
- Fixed several GUI issues that made the GUI crash when having multiple windows open
- Fixed error when opening reference tables, e.g. concattenated tables.
- Fixed two race conditions found by Helgrind
- Fixed bug in reported coordinates when zooming
- Fixed NaN issues while imaging certain sets
- Fixed bug that prevented column selection to actually have effect
- Allow lower memory machines
- Default strategy is about 20% faster and equally accurate
- Changed directory structure of source code
- Fixed various rare segmentation faults
- A lot of bug fixes and feature enhancements related to filtering
- A lot of doc fixes
- Fixed some bugs that caused NDPPP not to work
- Some Mac fixes by Ger van Diepen

AOFlagger version 1 was in development from 2010-10-22 to 2011-10-20. Stable releases:

- *1.5* (2011-10-20)
- *1.4* (2011-07-22)
- *1.3* (2011-04-26)
- *1.2* (2011-02-28)
- 1.1 (2010-10-22)

Other sources of information:

- AOFlagger on Gitlab
- Debian AOFlagger tracker
- The LOFAR cookbook for LOFAR-specific info

CHAPTER 9

# Navigate

- genindex
- search

# Lua Module Index

## a

# Index

## Symbols



## A



## C



## D



## E



## F



## G



## H



## I



## J



## L



## N



## O



## P



## R



## S

## T



## U



## V