DPPP

DPPP (the Default Preprocessing Pipeline, previously NDPPP for New Preprocessing Pipeline) is the LOFAR data pipelined processing program. It can be used to do all kind of operations on the data in a pipelined way, so the data are read and written only once.

DPPP started as a new and faster version of IDPPP. The original differences can be seen here.

DPPP preprocesses the data of a LOFAR observation by executing steps like flagging or averaging. Such steps can be used for the raw data as well as the calibrated data by defining the data column to use. One or more of the following steps can be defined as a pipeline. DPPP has an implicit input and output step. It is also possible to have intermediate output steps.

DPPP comes with quite some predefined steps, but it is possible to plugin arbitrary steps, either implemented in C++ or Python.

The following steps are possible:

- **Flagging** and **Filtering**
  - **AOFlagger** for automatic flagging in time/freq windows using Andre Offringa's advanced aoflagger.
  - **Preflagger** to flag given baselines, time slots, etc.
  - **UVWFlagger** to flag based on UVW coordinates, possibly in the direction of another source.
  - **MADFlagger** for automatic flagging in time/freq windows based on median filtering.
  - **Filter** to filter on baseline and/or channel (only the given baselines/channels are kept). The reader step has an implicit filter.
- **Averaging**
  - **Averager** to average data in time and/or freq.
- **Phase Shifting**
  - **PhaseShift** to shift data to another phase center.
- **Demixing** to remove strong sources (A-team) from the data.
  - **Demixer** to demix in the old way.
  - **SmartDemixer** to demix in a new, smarter way.
- **Station summation**
  - **StationAdder** to add stations (usually the superterp stations) forming new station(s) and baselines.
- **Counter** to count the number of flags per baseline, frequency, and correlation. A flagging step also counts how many visibilities it flagged. Counts can be saved to a table to be plotted later using function `plotflags` in python module `lofar.dppp`.
- **Data calibration** and **Data scaling**
  - **ApplyCal** to apply an existing calibration to a MeasurementSet.
  - **GainCal** to calibrate gains using StefCal.
  - **DDECal** to calibrate direction dependent gains.
  - **Predict** to predict the visibilities of a given sky model.
  - **H5ParmPredict** to predict visibilities corrupted by an instrument model (in H5Parm)
  - **ApplyBeam** to apply the LOFAR beam model, or the inverse of it.
  - **ScaleData** to scale the data with a polynomial in frequency (based on SEFD of LOFAR stations).
  - **Upsample** to upsample visibilities in time
  - **Out** to add intermediate output steps

- **start**Interpolate]] **for improving the accuracy of data averaging. * User defined steps provide a plugin mechanism for arbitrary steps implemented in C++. * Python defined steps provide a plugin mechanism for arbitrary steps implemented in Python. The input is one or more (regularly shaped) MeasurementSets (MSs). The data in the given column are piped through the steps defined in the parset file and finally written (if needed). It makes it possible to, say, flag at the full resolution, average, flag on a lower resolution, average further, and finally write the data.**
  Regularly shaped means that all time slots in the MS must contain the same baselines and channels. DPPP can handle only one spectral window. If the MS has multiple spectral windows, one has to be selected. If multiple MSs are given as input, their data are combined in frequency. It means that the time, phase direction, etc. of the different MSs have to be the same. Note that other steps (like averaging) can still be used.
  When combining MSs (thus combining subbands), it is possible that one or more of them do not exist. Flagged data will be inserted for them. The missing frequency info is deduced from the other subbands. Note that in order to insert missing subbands in the data, the names of the missing MSs have to be given at the right place in the list of MS names. Otherwise DPPP does not know that subbands are missing. The output can be a new MeasurementSet, but it is also possible to update the flags if the input is a single MS. If averaging or phase-shifting to another phase center is done, the only option is to create a new MeasurementSet. At the end the run time is shown. Note that on a multi-core machine the user time can exceed the elapsed time (user time is counted per core). By default the percentage of time each step took is also shown. The AOFlagger, MADFlagger, and Demixer, by far the most expensive parts of DPPP, can run multi-threaded if DPPP is built with OpenMP. It is possible to define the number of threads to use by the global key `numthreads`. Is that is not set, it uses the environment variable `OMP_NUM_THREADS`. If also that variable is undefined, an DPPP run uses as many threads as there are CPU cores. Thus if multiple DPPP runs are started on a machine, the default total number of threads will exceed the number of CPU cores. === MeasurementSet Access === * The 'msin' step defines which MS and which DATA column to use. It is possible to specify multiple MSs using a glob-pattern or a vector of MS names. * If multiple MSs are given, they will be concatenated in frequency. It means that all MSs must have the same times, baselines, etc. Flagged data can be inserted for MSs that are specified, but do not exist. * It is possible to select baselines and/or a band (spectral window) and/or skip leading or trailing channels. This is the same for each input MS. * Optionally proper weights can be calculated using the auto-correlation data. * It sets flags for invalid data (NaN or infinite). * Dummy, fully flagged data with correct UVW coordinates will be inserted for missing time slots in the MS. This can only be done if a single input MS is used. * Missing time slots at the beginning or end of the MS can be detected by giving the correct start and end time. This is particularly useful for the imaging pipeline where BBS requires that the MSs of all subbands of an observation have the same time slots. When updating an MS, those inserted slots are temporary and not put back into the MS. * The 'msout' step step defines the output. If a band is selected, the output MS (including its SPECTRAL_WINDOW subtable) contains that band only (its id is 0).
  The input MS is updated if no output name is given or if the output name is equal to the input name or equal to a dot. The calculation of the weights is done as follows. <code> Weight[ANT1_POL1, ANT2_POL2] = N / (autocorr[ANT1_POL1] * autocorr[ANT2_POL2]) N = EXPOSURE * CHAN_WIDTH * WGHT </code> where WGHT

is the weight put in by RTCP (number of samples used / total number of samples). This note discusses weighting in some more detail. === Flagging === It is important to realize that a MeasurementSet contains columns FLAG and FLAG_ROW to indicate if data are flagged. If FLAG_ROW is set, all data in that row are flagged. DPPP will set FLAG_ROW if all FLAG are set (and vice-versa).
When clearing the flags manually, it is important to realize that both columns have to be cleared. For example: <code> taql 'update my.ms set FLAG=F, FLAG_ROW=F' </code> DPPP flagging behaviour is as follows. * If one correlation is flagged, all correlations will be flagged (e.g. XX,YX,YY are flagged if XY is flagged). * The `msin` step flags data containing NaNs or infinite numbers or if FLAG_ROW is set. * An `AOFlagger` step can be used to flag using Andre Offringa's rficonsole code. Because DPPP always reads entire time slots, the flagging can be done on limited time windows only (depending on the available memory). An overlap can be defined to reduce boundary effects.
By default QUALITY subtables will be created containing statistical flagging quality information. They can be inspected using tools like `aoqplot`.
The default strategy works well for HBA data, but not for LBA data. The strategy `LBAdefault` should be used for it. * A `Preflagger` step can be used to flag (or unflag) on time, baseline, elevation, azimuth, simple uv-distance, channel, frequency, amplitude, phase, real, and imaginary. Multiple values (or ranges) can be given for one or more of those keywords. A keyword matches if the data matches one of the values. The results of all given keywords are AND-ed. For example, only data matching given channels and baselines are flagged.
Keywords can be grouped in a set making it a single (super) keyword. Such sets can be OR-ed or AND-ed. It makes it possible to flag, for example, channel 1-4 for baseline A and channel 34-36 for baseline B. Here it is explained in a bit more detail. * A `UVWFlagger` step can be used to flag on UVW coordinates in meters and/or wavelengths. It is possible to base the UVW coordinates on a given phase center. If no phase center is given, the UVW coordinates in the input MS are used. * A `MADFlagger` step can be used to flag on the amplitudes of the data. It flags based on the median of the absolute difference of the amplitudes and the median of the amplitudes. It uses a running median with a box of the given size (number of channels and time slots). It is a rather expensive flagging method with usually good results.
The flagging parameters can be given as an expression to make them dependent on baseline length.
It is possible to specify which correlations to use in the MADFlagger. Flagging on XX only, can save a factor 4 in performance.
Furthermore it is possible to only flag the auto-correlations and apply the results to the cross-correlations with a baseline length within optionally given limits. === Averaging === * Unflagged visibility data are averaged in frequency and/or time taking the weights into account. New weights are calculated as the sum of the old weights.
Some older LOFAR MSs have weight 0 for unflagged data points. These weights are set to 1. * The UVW coordinates are also averaged (not recalculated). * It fills the new column LOFAR_FULL_RES_FLAG with the flags at the original resolution for the channels selected from the input MS. It can be used by BBS to deal with bandwidth and time smearing. * Averaging in frequency requires that the average factor fits integrally. E.g. one cannot average every 5 channels when having 256 channels. * When averaging in time, dummy time slots will be inserted for the ones missing at the end. In that way the output MeasurementSet is still regular in time. * An

averaged point can be flagged if too few unflagged input points were available === Demixing === Demixing (or Smart Demixing explained below) is a faster and more flexible way of the old demixing python script to demix and subtract strong sources (A-team). Jones matrices can be estimated for the direction of the subtract-sources, model-sources, and the optional target-source. * It is possible to have different averaging for the demix and subtract step. * Selected (e.g. shorter) baselines can be demixed (others will be averaged only). By default only the cross-correlations are used. * Four different direction types can be given: * The subtract-sources are subtracted from the data. They must have a source model. * The model-sources can be given to take the contribution of other strong sources into account when solving for the gains. They must have a source model as well. The target source should NOT be part of this list. * The other-sources directions are taken into account when demixing. They are projected away when solving for the gains. * If the target source is given, it must have a source model and no other-sources can be given. If no target source is given, the target direction can be projected away like the extra-sources. Weak target sources should not be projected away. * A source model mentioned above is the patch name in the SourceDB (e.g. CasA). At the moment only point and Gaussian sources are supported. The direction used for demixing is the centroid of the sources that belong to the patch. The direction for an extra source (for which no model is used) can be given as a parameter if that is needed. * It is important to note that the target source model must NOT be given using the subtract-sources or model-sources. If it has to be used, give it using the `targetsource` parameter. * The Jones matrices will be estimated jointly for all directions, so better results are expected if the sources are close to the target. However, joint estimation of the Jones matrices for all directions is slower than estimating the Jones matrices for each direction separately. In the near future an option will be added to estimate the Jones matrices for each direction separately like the old demixing script is doing. === Smart Demixing === Smart Demixing does demixing as above, but in a smarter way using a scheme developed by Reinout van Weeren. For each time chunk (say 2 minutes) it is decided how to demix. It needs three source models, which are made from a text file using makesourcedb. Note that for performance it is best to run makesourcedb with parameter `outtype=blob`. * A detailed model of the A-team sources used in the solve and subtract steps. * A coarse model of the A-team sources used in the estimate step. If not given, the detailed model will be used. * A model of the target field. Usually the user can create it from the GSM using gsm.py. Smart demixing works as follows: * If an A-team source is at about the same position as a source in the target model, the source is removed from the A-team list and its detailed model replaces the source in the target model used in the solve step (not for the estimate step). * Using the coarse A-team model, the visibilities are estimated per baseline for each A-team source. By default the beam model is applied to get the apparent visibilities. The sources and baselines are selected for which the maximum amplitude exceeds a given threshold. A source/station will be solved for if the station appears in at least N of the selected baselines for that source. A detailed source model is used in that step to get as accurate gains as possible. * The visibilities of the target are estimated in a similar way using the target model. The target is included in the solve if its maximum amplitude exceeds a threshold or if the amplitude ratio Target/Ateam exceeds a threshold. The target is also included if it is close to an A-team source and the ratio exceeds another (smaller) threshold. Otherwise, the target is ignored (if close) or deprojected. A detailed decision tree that the smart demixing algorithm follows is available here. When solving for the

complex gains of the selected A-team sources, the detailed A-team model is used to get the correct gains. Note that by default the sources/stations not solved for are still used in the solve step. There Jones matrices will have a small gain value on the diagonal and zeroes for the off-diagonal values. At the end a log is produced showing how the demixing behaved. It shows: * percentage of converged solves and the average number of iterations used for them. * percentage of times the target was included, deprojected, and ignored. * percentage of times a source/station was solved for (thus matched the threshold/ratio criteria). * average and standard deviation of percentage amplitude subtracted per source per baseline === Phase shifting === * Data can be shifted to another phase center. * A shift step can shift back to the original phase center (by giving an empty center). If that is done by the last shift step, no new MS needs to be created. === Upsample === * Upsampling data can be useful for at least one use case. Consider data that has been integrated for two seconds, by a correlator (the AARTFAAC correlator) that sometimes misses one second of data. The times of the visibilities will then look like [0, 2, 4, 7, 9, 12], each having integration time 2 seconds. DPPP will automatically fill missing time slots, which will lead to times [0, 2, 4, 6, 7, 9, 11, 12]. This is still a nonuniform time coverage, which is not desirable. Calling the upsample step with `timestep=2` on this data will create times [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13] (it will remove the inserted dummy time slots that overlap, i.e. at 7 and 12). This data is then useful for further processing, e.g. averaging to 10 seconds. === Station summation === * One or more new stations can be defined from a list of existing stations. An existing station can occur in only one new station. * The data of baselines containing only one of the stations are added to form a new baseline. * Optionally the auto-correlations can be added to form a new auto-correlation 'baseline'. * The data can be added with or without weight. * Optionally averaging instead of summing can be done. === Data scaling === * The data can be scaled with a polynomial in frequency to correct for the SEFD of the LOFAR stations. * The default coefficients have been determined empirically. It is possible to specify them per station. * It can take the number of used dipoles/tiles into account when scaling (e.g. for remote/international or for failing ones). === Filtering === * Similar to the `msin` step a filter makes it possible to keep only the given channels and/or baselines. * By default, a station is always kept in the ANTENNA table, even if all its baselines are removed. This can be changed with the key `remove`. === Flag statistics and plotting === Several steps show statistics about flagged data points. * A MADFlagger and AOFlagger step show the percentage of visibilities flagged by that flagging step. It shows: * The percentages per baseline and per station. * The percentages per channel. * The number of flagged points per correlation, i.e. which correlation triggered the flagging. This may help in determining which correlations to use in the MADFlagger. * A UVWFlagger and PreFlagger step show the percentage of visibilities flagged by that flagging step. It shows percentages per baseline and per channel. * The `msin` step shows the number of visibilities flagged because they contain a NaN or infinite value. It is shown which correlation triggered the flagging, so usually only the first correlation is really counted. * A Counter step can be used to count and show the number of flagged visibilities. Such a step can be inserted at any point to show the cumulative number of flagged visibilities. For example, it can be defined as the first and last step to know how many visibilities have been flagged in total by the various steps. * Each step giving flagging percentages can save the percentages per frequency and per station to a table. The extension `.flagfreq` is used for the table containing the flags per frequency; the extension `.flagstat` for the flags per station. The full basename of the table is the main part of the MS followed by

_<stepname> followed by the extension. The path for these tables can be specified in the parset file. * The plotflags function in the Python module lofar.dppp can be used to plot those tables. It can plot multiple subbands by giving it a list of table names. The flags per station will be averaged for those subbands. === Intermediate output step === The step out can write data to disk at an intermediate stage. It takes the same arguments as the 'msout' step. As an example, the following reduction will flag, save flagged data at high resolution, then average and save the result in another measurement set. On the averaged data, it will also apply a calibration table and save that in the CORRECTED_DATA column. <code> msin=L123.MS steps=[aoflag,out1,average,out2,applycal] # Write out flagged data at full resolution out1.type=out out1.name=L123-flagged.MS average.timestep=4 # Write out averaged data out2.type=out out2.name=L123-averaged.MS out2.datacolumn=DATA applycal.parmdb=instrument.parmdb # Write the corrected data to CORRECTED_DATA msout=L123-averaged.MS msout.datacolumn=CORRECTED_DATA </code> === User defined step === Besides the predefined DPPP steps like AOFlagger, etc., it is possible to use any user-defined DPPP step implemented in C++ or Python. If implemented in C++ such a step has to reside in a shared library, that will dynamically be loaded by DPPP. The name of such a shared library has to be the step type name. DPPP will try to load the library libdppp_xxx.so (or .dylib on OS-X) for a step type xxx. To make this a bit more flexible it is possible to define multiple steps in a single shared library. In such a case the step type name has to consist of 2 parts separated by a dot. The first part is the library name, the second part the step type in that library. For example: <code> steps=[averager, mystep1, mystep2] mystep1.type = mystep.stepa mystep2.type = mystep.stepb </code> defines two user steps. Both step implementations reside in library libmystep.so.

A description and example of a dynamically loaded step can be found in the LOFAR source code repository in LOFAR/CEP/DPPP/TestDyDPPP. === Python defined step === The mechanism described above is used to make it possible to implement a user step in Python. The step type has to be pythoDPPP and the name of the Python module and class containing the code have to be given. DPPP will load the library libdppp_pythonDPPP.so, which will start an embedded Python shell, load the module, and instantiate an object of the class.

A detailed description is available. ===== ParSet File ===== Similar to most LOFAR programs, the parameters for the DPPP program are given in a so-called parset file. Note that it is possible to add parameters or overwrite parameters, defined in the parset file, using command line arguments. For example: <code> DPPP DPPP.pset parm1=value1 parm2=value2 … </code> The steps to perform have to be defined in the parset file. They are executed in the given order, where the data are piped from one step to the other until all data are processed. Each step has a name to be used thereafter as a prefix in the keyword names specifying the type and parameters of the step. The most basic parset is as follows. It copies the DATA column of the MS and flags NaN and infinite data. <code> msin = ~/SB0.MS msout = SB0_DPPP.MS steps=[] </code> The following example is more elaborate. It flags (using a median flagger), averages all channels, flags the result of the average, and finally averages in time.

Note that 'msin' and 'msout' can be seen as an implicit first and last step. <code> msin = ~/SB0.MS msin.startchan = 8 msin.nchan = 240 msin.datacolumn = DATA # is the default msout = "SB0_DPPP.MS" # if empty, the input MS is updated and # no averaging steps can be done msout.datacolumn = DATA # is the default steps =

[flag1,count,avg1,flag2,avg2,count] flag1.type=madflagger flag1.threshold=1 flag1.freqwindow=31 flag1.timewindow=5 flag1.correlations=[0,3] # only flag on XX and YY flag1.count.save = true # save flag percentages flag1.count.path = $HOME # to a table in $HOME avg1.type = average avg1.freqstep = 240 avg1.timestep = 1 # is the default flag2.type=madflagger flag2.threshold=2 flag2.timewindow=51 avg2.type = average avg2.timestep = 5 </code> Plotting the flag percentages, saved by the first MADFlagger step, could be done in python like: <code> import lofar.dppp as ld ld.plotflags ('$HOME/SB0_flag1.flagfreq') # step name was flag1 </code> ===== Description of all parameters ===== The parameters in the parset are divided into several groups like input (msin), output (msout), madflagger, average, preflagger, and uvwflagger. Because multiple flagging and averaging steps can be specified, their parameters have to be prefixed with the step name as shown in the example above. ^ Parameter ^ type ^ default ^ description ^ ||||| ^General |||| | steps | string vector | | Names of the steps to perform. Each step has to be defined using the step name as a prefix.
The step `type` parameter defines the type of step (averager, madflagger, preflagger, uvwflagger, counter). The step type defaults to the name of the step, which is especially handy for count steps.
`msin` and `msout` are implicit steps which should not be given here.
An empty vector `[]` means that the input MS is copied to the output MS while flagging NaN and infinite numbers.
Note that a step name can be used more than once meaning that the same step will be executed multiple times (e.g., multiple times count). | | numthreads | int | ${OMP_NUM_THREADS} | Maximum number of threads to use. | | showprogress | bool | true | Show a progress bar? | | showcounts | bool | true | Show flagging statistics? | | showtimings | bool | true | At the end the percentage of elapsed time each step took can be shown; the overall time is always shown. | | checkparset | integer | 0 | What to do if parameters in the ParSet file are not used.
-1 means ignore.
0 means give a warning showing those parameters. In this way misspelled parameters can be detected.
1 means give an error and stop.
For backward compatibility False (0) and True (1) can also be given. | | uselogger | bool | false | If false, all DPPP messages are written on stdout. If true, the logging framework is used. | ===Counter=== | <step>.type | string | | Case-insensitive step type; must be 'counter' (or 'count').
Note that the type defaults to the step name, so if step name `count` is used, nothing more needs to be specified. | | <step>.showfullyflagged | bool | false | If true, all fully flagged baselines are shown in the baseline selection format using their antenna indices (not names). For example: 0&1; 3&7 | | <step>.save | bool | false | If true, the flag percentages per frequency are saved to a table with extension `.flagfreq` and percentages per station to a table with extension `.flagstat`. The basename of the table is the MS name (without extension) followed by the stepname and extension. | | <step>.path | string | “” | The directory where to create the flag percentages table. If empty, the path of the input MS is used. | | <step>.warnperc | double | 0 | If > 0, print an extra message for each baseline or channel with a percentage flagged higher than this value. Such a message line can be easily grep-ed. | | <step>.flagdata | bool | false | If COUNT is the only step in an DPPP run, the data won't be read, so unflagged invalid data (NaN. infinite) won't be noticed and counted as flagged. Setting this flag forces DPPP to read and check the data. | ==== Input ==== | msin msin.name | string | | Name of the input MeasurementSets. If a single name is given,

it can be a glob-pattern (like L23456_SAP000_SB*) meaning that all MSs matching the pattern will be used. A glob-pattern can contain *, ?, [], and {} pattern characters (as used in bash).
If multiple MSs are to be used, their data are concatenated in frequency, thus multiple subbands are combined to a single band. In principle all MSs should exist, but if 'missingdata=true' and 'orderms=false' flagged zero data will be inserted for missing MS(s) and their frequency info will be deduced from the other MSs. | | msin.sort | bool | false | Does the MS need to be sorted in TIME order? | | msin.orderms | bool | true | Do the MSs need to be ordered on frequency? If true, all MSs must exist, otherwise they cannot be ordered. If false, the MSs must be given in order of frequency. | | msin.missingdata | bool | false | true = it is allowed that a data column in an MS does not exist. In that case its data will be 0 and flagged. It can be useful if the CORRECTED_DATA of subbands are combined, but a BBS run for one of them failed.
If 'orderms=false', it also makes it possible that a MS is specified but does not exist. In such a case flagged data will be used instead. The missing frequency info will be deduced from the other MSs where all MSs have to have the same number of channels and must be defined in order of frequency. | | msin.baseline | string | | Baselines to be selected (default is all baselines). See Description of baseline selection parameters. Only the CASA baseline selection syntax as described in this note can be used. | | msin.band | integer | -1 | Band (spectral window) to select (<0 is no selection). This is mainly useful for WSRT data. | | msin.startchan | integer | 0 | First channel to use from the input MS (channel numbers start counting at 0). Note that skipped channels will not be written into the output MS. It can be an expression with `nchan` (nr of input channels) as parameter. E.g.
`nchan/32`
will be fine for LOFAR observations with 64 and 256 channels. | | msin.nchan | integer | 0 | Number of channels to use from the input MS (0 means till the end). It can be an expression with `nchan` (nr of input channels) as parameter. E.g.
`15*nchan/16` **| | msin.starttime | string | first time in MS | Center of first time slot to use; if < first time in MS, dummy time slots are inserted. A date/time must be specified in the casacore MVTime format, e.g. 19Feb2010/14:01:23.817 | | msin.endtime | string | last time in MS | Center of last time slot to use; if > last time in MS, dummy time slots are inserted. | | msin.ntimes | integer | 0 | Number of time slots to use (0 means till the end). | | msin.useflag | bool | true | Use the current flags in the MS? If false, all flags in the MS are ignore and the data (except NaN and infinite values) are assumed to be good and will be used in later steps. | | msin.datacolumn | string | DATA | Data column to use, i.e. the name of the column in which the visibilities are written. | | msin.weightcolumn | string | WEIGHT_SPECTRUM or WEIGHT | Weight column to use. Defaults to WEIGHT_SPECTRUM if this exists, otherwise the WEIGHT column is used. | | msin.modelcolumn | string | MODEL_DATA | Model data column. Currently only used in gaincal | | msin.autoweight | bool | false | Calculate weights using the auto-correlation data? It is meant for setting the proper weights for a raw LOFAR MeasurementSet. | | msin.forceautoweight | bool | false | In principle the calculation of the weights should only be done for the raw LOFAR data. It appeared that sometimes the `autoweight` switch was accidently set in a DPPP run on already dppp-ed data. To make it harder to make such mistakes, the `forceautoweight` flag has to be set as well for MSs containing dppp-ed data. | ==== Output ==== | msout**
msout.name | string | | Name of new output MeasurementSet; if empty, the input MS

is updated. The other msout parameters are not applicable (apart from `countflag`). Normally an update is only done if a step is given that can change the data (e.g. PreFlagger). However, a name '.' or a name equal to the name of the input MS means that the input MS will always be updated, even if no step is given. This is useful if only flagging of NaN-s in the MS needs to be done.
Note that when doing averaging, the input MS cannot be updated. | | msout.overwrite | bool | false | When creating a new MS, overwrite if already existing? | | msout.datacolumn | string | DATA | The column in which to write the data. When creating a new MeasurementSet, only column DATA can be used. When updating the input MeasurementSet, any column can be used. If not existing, it will be created first. | | msout.weightcolumn | string | WEIGHT_SPECTRUM | The column in which to write the weights. When creating a new MeasurementSet, only WEIGHT_SPECTRUM can be used. When updating the input Measurementset, any column can be used. If not existing, it will be created first. | | msout.writefullresflag | bool | true | Write the full resolution flags? | | msout.tilesize | integer | 1024 | For expert user: tile size (in Kbytes) for the data columns in the output MS. | | msout.tilenchan | integer | 8 | For expert user: maximum number of channels per tile in output MS. | | msout.clusterdesc | string | “” | If not empty, create the VDS file using this ClusterDesc file. | | msout.vdsdir | string | “” | Directory where to put the VDS file; if empty, the MS directory is used. | | msout.storagemanager msout.storagemanager.name| string | “” | What storage manager to use. When empty (default), the data will be stored uncompressed. When set to “dysco”, the data will be compressed. Settings below will set the compression settings; see the Dysco wiki and the paper for more info. The default settings are reasonably conservative and safe. | | msout.storagemanager.databitrate | integer | 10 | Number of bits per float used for columns containing visibilities. Can be set to zero to compress weights only. | | msout.storagemanager.weightbitrate | integer | 12 | Number of bits per float used for WEIGHT_SPECTRUM column. | | msout.storagemanager.distribution | string | “TruncatedGaussian” | Assumed distribution for compression; “Uniform”, “TruncatedGaussian”, “Gaussian” or “StudentsT”.| | msout.storagemanager.disttruncation | double | 2.5 | Truncation level for compression with the Truncated Gaussian distribution.| | msout.storagemanager.normalization | string | “AF” | Compression normalization method: AF, RF or Row.| ==== Filter ==== | <step>.type | string | | Case-insensitive step type; must be 'filter'| | <step>.startchan | integer | 0 | First channel to use from the input MS (channel numbers start counting at 0). Note that skipped channels will not be written into the output MS. It can be an expression with `nchan` (nr of input channels) as parameter. E.g.
`nchan/32`
will be fine for LOFAR observations with 64 and 256 channels. | | <step>.nchan | integer | 0 | Number of channels to use from the input MS (0 means till the end). It can be an expression with `nchan` (nr of input channels) as parameter. E.g. `15*nchan/16` | | <step>.baseline | string | “” | Baselines to keep. See Description of baseline selection parameters. | | <step>.blrange | double vector | “” | Baselines to keep. See Description of baseline selection parameters. | | <step>.corrtype | string | “” | Correlation type to match? Must be auto, cross, or an empty string. | | <step>.remove | bool | false | If true, the stations not used in any baseline will be removed from the ANTENNA subtable and the antenna ids in the main table will be renumbered accordingly. To have a consistent output MeasurementSet, other subtables (FEED, POINTING, SYSCAL, LOFAR_ANTENNA_FIELD, LOFAR_ELEMENT_FAILURE, and QUALITY_BASELINE_STATISTIC) will also be updated.

**Note that stations filtered previously (e.g. using msselect) will also be removed, even if no baseline selection is done in the filter step. | ==== Upsample ==== | <step>.type | string | | Case-insensitive step type; must be 'upsample'| | <step>.timestep | integer | | Number of times into which each timestep will be expanded | ==== AOFlagger ==== | <step>.type | string | | Case-insensitive step type; must be 'aoflagger' (or 'aoflag'). | | <step>.count.save | bool | false | If true, the flag percentages per frequency are saved to a table with extension `.flagfreq` and percentages per station to a table with extension `.flagstat`. The basename of the table is the MS name (without extension) followed by the stepname and extension. | | <step>.count.path | string | "" | The directory where to create the flag percentages table. If empty, the path of the input MS is used. | | <step>.strategy | string | "" | The name of the strategy file to use. If no name is given, the default strategy is used which is fine for HBA. For LBA data the strategy `LBAdefault` should be used.
A strategy file is looked up as given. If not found, it is looked up in $LOFARROOT/share/rfistrategies that contains the standard strategies. | | <step>.memoryperc | integer | 0 | If >0, percentage of the machine's memory to use. If `memorymax` nor `memoryperc` is given, all memory will be used (minus 2 GB (at most 50%) for other purposes). Accepts only integer values (LOFAR v2.16). Limiting the available memory too much affects flagging accuracy; in general try to use at least 10 GB of memory. | | <step>.memorymax | double | 0 | Maximum amount of memory (in GB) to use. ⇐0 means no maximum. As stated above, this affects flagging accuracy.| | <step>.timewindow | integer | 0 | Number of time slots to be flagged jointly. The larger the time window, the better the flagging performs. 0 means that it will be deduced from the memory to use. Note that the time window can be extended with an overlap on the left and right side to minimize possible boundary effects.| | <step>.overlapperc | double | 0 or 1 | If >0, percentage of time window to be added to the left and right side for overlap purposes (to minimize boundary effects). If `overlapmax` is not given, it defaults to 1%. | | <step>.overlapmax | integer | 0 | Maximum overlap value (0 is no maximum). | | <step>.autocorr | bool | true | Flag autocorrelations? | | <step>.pulsar | bool | false | Use flagging strategy optimized for pulsar observations? | | <step>.pedantic | bool | false | Be more pedantic when flagging? | | <step>.keepstatistics | bool | true | Write the quality statistics? | ==== MADFlagger ==== | <step>.type | string | | Case-insensitive step type; must be 'madflagger' (or 'madflag'). | | <step>.count.save | bool | false | If true, the flag percentages per frequency are saved to a table with extension `.flagfreq` and percentages per station to a table with extension `.flagstat`. The basename of the table is the MS name (without extension) followed by the stepname and extension. | | <step>.count.path | string | "" | The directory where to create the flag percentages table. If empty, the path of the input MS is used. | | <step>.threshold | float | 1 | The flagging threshold that can be baseline dependent.
It can be any (TaQL-like) expression that evaluates to a float. In the expression the variable 'bl' can be used which is the baseline length (in meters). In this way the value can be made baseline dependent. For example:
`iif(bl<100, 0.5, iif(bl<500, 0.75, iif(bl<1000, 0.9, 1)))`
defines the threshold between the baseline lengths 100, 500, and 1000 meter. | | <step>.timewindow | integer | 1 | Number of times in the median box. If not odd, 1 is subtracted. It is silently reduced if exceeding the actual number of time slots.
In a way similar to 'threshold' it can be made baseline length dependent. | | <step>.freqwindow | integer | 1 | Number of channels in the median box. If not odd, 1 is subtracted. It is silently reduced if exceeding the actual number of channels.**

In a way similar to 'threshold' it can be made baseline length dependent. | | <step>.correlations | integer vector | [] | The correlations to use in the flagger; an empty vector means all. They are handled in the order given; if the flagging criterium holds for one correlation, the other correlations are not tested anymore. So if one knows that most RFI is found in YY, then in XX and finally some in XY and YX, the vector should be `[3,0,1,2]` because it makes the program run faster. Note that the statistics printed at the end show how many flagged data points have been found per correlation. | | <step>.applyautocorr | bool | False | True means that the MADFlagger is used on the auto-correlations only. The resulting flags are applied to the cross-correlations, thus data are flagged where the corresponding auto-correlations are flagged.
An error is given if set to True, while the MS does not contain auto-correlations. | | <step>.blmin | integer | -1 | Minimum baseline length (in meters).
Only baselines with a length >= this minimum are flagged. If `applyautocorr=true`, the autocorrelations are applied to the matching baselines only. | | <step>.blmax | integer | 1e30 | Maximum baseline length (in meters). It is similar to minimum. | ==== PhaseShift ==== | <step>.type | string | | Case-insensitive step type; must be 'phaseshifter' (or 'phaseshift'). | | <step>.phasecenter | string vector | | The RA and DEC (in J2000) of the new phase center. If an empty vector (i.e. []) is given, the original phase center is used. The RA and DEC can be given in sexagesimal format or as a value followed by a unit (default rad). For example, `[12h31m34.5, 52d14m07.34]` or `[187.5deg, 52.45deg]` | ==== Demixer ==== | <step>.type | string | | Case-insensitive step type; must be 'demixer' (or 'demix'). | | <step>.baseline | string | “” | Baselines to demix. See Description of baseline selection parameters. | | <step>.blrange | double vector | “” | Baselines to demix. See Description of baseline selection parameters. | | <step>.corrtype | string | cross | Baselines to demix. Correlation type to match? Must be auto, cross, or an empty string. | | <step>.timestep | integer | 1 | Number of time slots to average when subtracting. It is truncated if exceeding the actual number of times. Note that the data itself will also be averaged by this amount. | | <step>.freqstep | integer | 1 | Number of channels to average when subtracting. It is truncated if exceeding the actual number of channels. Note that the data itself will also be averaged by this amount. | | <step>.demixtimestep | integer | timestep | Number of time slots to average when demixing. It is truncated if exceeding the actual number of times. It defaults to the averaging used for the subtract. | | <step>.demixfreqstep | integer | freqstep | Number of channels to average when demixing. It is truncated if exceeding the actual number of channels. It defaults to the averaging used for the subtract. | | <step>.ntimechunk | integer | #cores | Number of demix time slots (after averaging) that are processed jointly in as much a parallel way as possible. If subtract uses different time averaging, it has to fit integrally. | | <step>.skymodel | string | sky | The name of the SourceDB to use (i.e., the output of `makesourcedb`). | | <step>.instrumentmodel | string | instrument | The name of the ParmDB to use. The ParmDB does not need to exist. If it does not exist it will be created. | | <step>.subtractsources | string vector | | Names of the sources to subtract. If none are given, demixing comes down to averaging. The sources must exist as patches in the SourceDB. | | <step>.modelsources | string vector | [] | Names of sources with models to take into account when solving. the sources must exist as patches in the SourceDB. Note that the target should NOT be part of this parameter. If a model of the target has to be used, it has to be given in parameter `targetsource`. | | <step>.targetsource | string | “” | It can be used to specify the name of the source model of the target. If given, the target source model (its patch in the SourceDB) is

taken into account when solving; in this case parameter `othersources` cannot be given. It cannot be given if `ignoretarget=true`. If not given, the target is projected away or ignored (depending on parameter `ignoretarget`). | | <step>.ignoretarget | bool | false | false = project the target source away; true = ignore the target | | <step>.othersources | string vector | [] | Names of sources of which the direction is taken into account when demixing by projecting the directions away. The direction needs to be specified if the source is unknown (which is usually the case). It can be done using parameters `<step>.<sourcename>.phasecenter`. | | <step>.<sourcename>.phasecenter | string vector | Taken from SourceDB | The J2000 direction [ra,dec] of a source given above. | | <step>.propagatesolutions | bool | true | If set to true, solutions of a time slot are used as initial values for the next time slot. If set to false, the diagonal elements of the Jones matrix are initialized to one and the off-diagonal elements to zero. | | <step>.defaultgain | double | 1.0 | The default and initial gain for the directional gains that are computed internally. | | <step>.maxiter | int | 50 | Maximum number of iterations used in the LM solve | ==== SmartDemixer ==== | <step>.type | string | | Case-insensitive step type; must be 'smartdemixer' (or 'smartdemix'). | | <step>.baseline | string | "" | Baselines to demix. See Description of baseline selection parameters. | | <step>.blrange | double vector | "" | Baselines to demix. See Description of baseline selection parameters. | | <step>.corrtype | string | cross | Baselines to demix. Correlation type to match? Must be auto, cross, or an empty string. | | <step>.target.baseline | string | "CS*&" | Baselines to use in prediction of median target amplitude. See Description of baseline selection parameters. | | <step>.target.blrange | double vector | "" | Baselines to use in prediction of median target amplitude. See Description of baseline selection parameters. | | <step>.target.corrtype | string | cross | Baselines to use in prediction of median target amplitude. Correlation type to match? Must be auto, cross, or an empty string. | | <step>.timestep | integer | 1 | Number of time slots to average when subtracting. It is truncated if exceeding the actual number of times. Note that the data itself will also be averaged by this amount. | | <step>.freqstep | integer | 1 | Number of channels to average when subtracting. It is truncated if exceeding the actual number of channels. Note that the data itself will also be averaged by this amount. | | <step>.demixtimestep | integer | timestep | Number of time slots to average when demixing. It is truncated if exceeding the actual number of times. It defaults to the averaging used for the subtract. | | <step>.demixfreqstep | integer | freqstep | Number of channels to average when demixing. It is truncated if exceeding the actual number of channels. It defaults to the averaging used for the subtract. | | <step>.chunksize | integer | demixtimestep | Number of time slots in a chunk for which it is decided how to demix (which sources/stations to use and how to deal with the target). It has to be a multiple of parameter 'demixtimestep'. | | <step>.ntimechunk | integer | #cores | Number of time chunks that are processed jointly in as much a parallel way as possible. Preferably it is a multiple of the number of cores. Note that for a typical LOFAR observation the data of a single time slot is about 4 MB. A typical chunk size can be 2 minutes, thus 120 time slots per core. For 24 cores this amounts to about 11 GB!! | | <step>.ateam.skymodel | string | | The detailed sky model of the A-team sources used to solve for the complex gains. It is the name of the SourceDB to use (i.e., the output of makesourcedb). | | <step>.estimate.skymodel | string | "" | The coarse sky model of the A-team sources used to estimate the visibilities when deciding how to demix a chunk. It is the name of the SourceDB to use (i.e., the output of `makesourcedb outtype=blob`).

**If no name is given, the detailed A-team model will be used.
The SourceDB must contain the same sources as the detailed model at about the same position. The order can be different though. | | <step>.target.skymodel | string | | The sky model of the target. It is the name of the SourceDB to use (i.e., the output of makesourcedb). | | <step>.target.delta | double | 60 | Angular distance uncertainty (in arcsec) to determine if an A-team source is at the same position as a target source. | | <step>.instrumentmodel | string | instrument | The name of the ParmDB to use. The ParmDB does not need to exist. If it does not exist it will be created. Note that the ParmDB is created after the output MS is created, so it can be a subdirectory of the output MS.| | <step>.sources | string vector | “” | Names of the A-team sources to use. If none are given, all sources in the A-team sky model will be used. | | <step>.ateam.threshold | double | 50 for LBA
5 for HBA | Take a source/baseline into account if its maximum estimated amplitude > threshold. | | <step>.minnbaseline | integer | 6 | Solve a source/station if the station occurs in at least 'minnbaseline' baselines with amplitude > ateam.threshold. | | <step>.minnstation | integer | 5 | Solve a source if at least 'minnstation' stations are solvable for the source. | | <step>.target.threshold | double | 200 for LBA
100 for HBA | Include the target in the solve if its maximum estimated amplitude > threshold. | | <step>.ratio1 | double | 5 | Include the target in the solve if the estimated amplitude ratio Target/max(Ateam) > ratio1. | | <step>.distance.threshold | double | 60 | Distance threshold (in degrees). The target is close to the A-team if the angular distance (scaled with freq) < threshold for any A-team source (thus angdist*obsfreq/reffreq < threshold). | | <step>.distance.reffreq | double | 60e6 | The 'reffreq' frequency used above. | | <step>.ratio2 | double | 0.25 | Include the target in the solve if the target is close to the A-team and the estimated amplitude ratio Target/min(Ateam) > ratio2. | | <step>.maxiter | integer | 50 | Maximum number of iterations to use in the solve. | | <step>.propagatesolutions | bool | true | If set to true, solutions of a time slot are used as initial values for the next time slot. If set to false, the diagonal elements of the Jones matrix are initialized to one and the off-diagonal elements to zero. However, solutions will not be transferred between chunks processed in parallel. | | <step>.defaultgain | double | 1e-3 | The default gain to use for the real part of the diagonal Jones elements for the unsolvable sources/stations. Take into account that the scale of the raw visibilities changed when COBALT was adopted. In the case of data correlated with BG/P, this parameter should be tuned down (1e-8). | | <step>.verbose | int | 0 | 0 = only show basic demix statistics
1 = show for each time chunk how target is handled, which sources are solvable, and how many stations.
>10 = various levels of debugging output. | | <step>.solveboth | bool | false | Mainly for test purposes. True means that in the solve only the baselines are used for which both stations are solvable. Usually this gives worse results. | | <step>.targethandling | integer | 0 | Mainly for test purposes. It enforces the target handling. 1=include, 2=deproject, 3=ignore, else=use smart way. | | <step>.applybeam | bool | true | Mainly for test purposes. Apply the station beam in the estimate, solve, and subtract steps? | | <step>.subtract | bool | true | Mainly for test purposes. False means that the subtract step is not done, thus only a solve of the gains is done. | ==== Averager ==== | <step>.type | string | | Case-insensitive step type; must be 'averager' (or equivalent 'average' or 'squash'). | | <step>.timestep | integer | 1 | Number of time slots to average. It is truncated if exceeding the actual number of times. | | <step>.freqstep | integer | 1 | Number of channels to average. It is truncated if exceeding the actual number of channels. | | <step>.minpoints | integer | 0 | If**

number of averaged unflagged input points < minpoints, the averaged point is flagged. | | <step>.minperc | float | 0 | Like minpoints, but expressed as a percentage of timestep*freqstep. | | <step>.timeresolution | float | 0 | Target time resolution, in seconds. If this is given, and bigger than zero, it overrides <step>.timestep | | <step>.freqresolution | float | 0 | Target frequency resolution, in Hz (or append "MHz" or "kHz" to specify it in those units). If this is given, and bigger than zero, it overrides <step>.freqstep | ==== StationAdder ==== | <step>.type | string | | Case-insensitive step type; must be 'stationadder' (or equivalent 'stationadd'). | | <step>.stations | record | | One or more names of new stations each followed by the list of stations it consists of. A station name in the list can be a glob-like pattern. Optionally such a pattern can be negated by a ! or ^ meaning that names matching that pattern are excluded from the selection so far. For example:
`stations={ST6:'CS00[2-7]*'}` can be used to form the superstation from all superterp stations.
`{ST6:['CS00[2-7]*','!CS005*']}` is similar, but excludes CS005.
`{ST001:[CS001,CS002,CS003], ST002:[CS004,CS005,CS006]}`
defines 2 new stations ST001 and ST002 consisting of the stations in the lists following their names. | | <step>.minpoints | int| 1 | Flag a new data point if number of unflagged data points added is less than `minpoints`. | | <step>.useweights | bool | true | Use the input data weights? False means all input visibilities have weight 1. | | <step>.average | bool | true | Is a visibility of a new station the weighted average of its input visibilities and its UVW the weighted average of the input UVWs? | | <step>.autocorr | bool | false | Form new auto-correlations? | | <step>.sumauto | bool | true | Sum auto- or cross-correlations to form new auto-correlations? | ==== ScaleData ==== | <step>.type | string | | Case-insensitive step type; must be 'scaledata'. | | <step>.stations | string vector | [] | Zero or more glob-like patterns defining the stations for which the corresponding coefficient vector has to be used. The coefficients of the first matching pattern are used. Default coefficients (determined by Adam Deller for LBA and HBA) are used for stations not given. For example:
`stations=[CS*, RS*, *]` | | <step>.coeffs | double vector | [] | Zero or more vectors of coefficients defining a polynomial in frequency (MHz). For example:
`coeffs=[ [1.5, 0.7, 0.04], [1.7, 0.65], [1.2, 0.8] ]`
The first vector results in a scale factor of `1.5 + 0.7*f + 0.04*f*f` where f is the channel frequency in MHz.
Note that an extra scaling can be applied taking into account the number of used dipoles/tiles of a station (see next parameter). | | <step>.scalesize | bool | | This parameter determines if an extra scaling has to be applied to correct for the number of tiles/dipoles actually used in a station. By default this will be done for the stations using the default coefficients, because those coefficients have been determined for an LBA station with 48 dipoles and HBA station with 24 tiles. By default it will not be done for explicitly given coefficients, because it is supposed they are determined specifically for that station.
Note that giving `stations=* coeffs=1 scalesize=true` will correct for station size only. | ==== PreFlagger ==== | <step>.type | string | | Case-insensitive step type; must be 'preflagger' (or 'preflag'). | | <step>.count.save | bool | false | If true, the flag percentages per frequency are saved to a table with extension `.flagfreq` and percentages per station to a table with extension `.flagstat`. The basename of the table is the MS name (without extension) followed by the stepname and extension. | | <step>.count.path | string | "" | The directory where to create the flag percentages

table. If empty, the path of the input MS is used. | | <step>.mode | string | set | Case-insensitive string telling what to do with the flags of the data matching (or not matching) the selection criteria given in the other parameters.
'set' means set the flags for the matching data. This is the default mode.
'clear' means clear the flags for the matching data. However, flags of invalid data (NaN or zero) are always set.
'setcomplement' or 'setother' means set flags for NON-matching data.
'clearcomplement' or 'clearother' means clear flags for NON-matching (valid) data. | | .expr | string | [] | Expression of preflagger keyword sets (see above). Operators AND, OR, and NOT are possible (or their equivalents &&,&, ||, |, and !). Parentheses can be used to change precedence order. For example: `c1 and (c2 or c3)`
Take care that the name of the set is used as an extra prefix in the PreFlagger parameter names. | | .timeofday | time vector | [] | Ranges of UTC time-of-day given as st..end or val+-delta. Each value must be given as 12:34:56.789, 12h34m56.789, or as a value followed by a unit like h, min, or s. | | .abstime | date/time vector | [] | Ranges of absolute UTC date/time given as st..end or val+-delta. Each value (except delta) must be given as a date/time in casacore MVTime format, for instance 12-Mar-2010/11:31:00.000. A delta value must be given as a time (for instance 1:30:0 or 20s). | | .reltime | time vector | [] | Ranges of times (using .. or +-) since the start of the observation. A time can be given like 1:30:0 or 20s. | | .timeslot | integer vector | [] | Time slot sequence numbers. First time slot is 0. st..end means end inclusive. | | .lst | time vector | [] | Ranges of Local Apparent Sidereal Times like `1:30:0 +-20min`. The LST of a time slot is calculated for the array position, thus not per antenna. | | .azimuth | direction vector | [] | Ranges of azimuth angles given as st..end or val+-delta. Each value has to be given as a casacore direction like 12:34:56.789 or 12h34m56.789, 12.34.56.789 or 12d34m56.789, or a value followed by a unit like rad or deg. | | .elevation | direction vector | [] | Ranges of elevation angles (similar to azimuth). For example: `0deg..10deg` | | .baseline | baseline vector | “” | See Description of baseline selection parameters. | | .corrtype | string | “” | Correlation type to match? Must be auto, cross, or an empty string. | | .blmin | double | -1 | If blmin > 0, baselines with length < blmin meter will match. | | .blmax | double | -1 | If blmax > 0, baselines with length > blmax meter will match. | | .uvmmin | double | -1 | If uvmmin > 0, baselines with UV-distance < uvmmin meter will match. Note that the UV-distance is the projected baseline length. | | .uvmmax | double | -1 | If uvmmax > 0, baselines with UV-distance > uvmmax meter will match. | | .freqrange | string vector | [] | Channels in the given frequency ranges will match. Each value in the vector is a range which can be given as start..end or start+-delta. A value can be followed by a unit like KHz. If only one value in a range has a unit, the unit is also applied to the other value. If a range has no unit, it defaults to MHz. For example: `freqrange=[1.2 .. 1.4 MHz, 1.8MHz+-50KHz]` flags channels between 1.2MHz and 1.4MHz and between 1.75MHz and 1.85MHz. The example shows that blanks can be used at will. | | .chan | string vector | [] | The given channels will match (start counting at 0). Channels exceeding the number of channels are ignored. Similar to `msin`, it is possible to specify the channels as an expression of `nchan`. Furthermore, .. can be used to specify ranges. For example: `chan=[0..nchan/32-1, 31*nchan/32..nchan-1]` to flag the first and last 2 or 8 channels (depending on 64 or 256 channels in the observation). | | .amplmin | float vector | -1e30 | Correlation data with amplitude < amplmin will match. It can be given per correlation. For example, `amplmin=[100,,,100]` matches data points with XX or YY amplitude < 100. The non-specified amplitudes get the default value.
It is also possible to give a single value (without brackets) meaning that it is used as

the minimum for all correlations. | | .amplmax | float vector | 1e30 | Correlation data with amplitude > amplmax will match. | | .phasemin | float vector | -1e30 | Correlation data with phase < phasemin (in radians) will match. | | .phasemax | float vector | 1e30 | Correlation data with phase > phasemax (in radians) will match. | | .realmin | float vector | -1e30 | Correlation data with real complex part < realmin will match. | | .realmax | float vector | 1e30 | Correlation data with real complex part > realmax will match. | | .imagmin | float vector | -1e30 | Correlation data with imaginary complex part < imagmin will match. | | .imagmax | float vector | 1e30 | Correlation data with imaginary complex part > imagmax will match. | ==== ApplyCal ==== | <step>.type | string | | Case-insensitive step type; must be 'applycal' (or 'correct'). | | <step>.parmdb | string | | Path of parmdb in which the parameters are stored. This can also be an H5Parm file, in that case the filename has to end in '.h5' | | <step>.correction | string | gain | Type of correction to perform, can be one of 'gain', 'tec', 'clock', '(common)rotationangle' / 'rotation', '(common)scalarphase', '(common)scalaramplitude' or 'rotationmeasure' (create multiple ApplyCal steps for multiple corrections). When using H5Parm, specify the name of the soltab here; the type will be deduced from the metadata in that soltab. | | <step>.direction | string | "" | If using H5Parm, the direction of the solution to use | | <step>.updateweights | bool | false | Update the weights column, in a way consistent with the weights being inverse proportional to the autocorrelations (e.g. if 'autoweights' was used before). | | <step>.invert | bool | true | Invert the corrections, to correct the data. Default is true. If you want to corrupt the data, set it to 'false' | | <step>.timeslotsperparmupdate | int | 100 | Number of time slots to handle after one read of the parameter file. Optimization to prevent spurious reading from the parmdb. | | <step>.steps | list | [] | (new in version 3.1) ApplyCal substeps, e.g. [myApplyCal1, myApplyCal2]. Their parameters can be specified through e.g. <step>.myApplyCal1.correction=tec. If a parameter is not given for the substep, it takes the value from `<step>.`. | ==== GainCal ==== | <step>.type | string | | Case-insensitive step type; must be 'gaincal' or 'calibrate'. | | <step>.caltype | string | | The type of calibration that needs to be performed, can be one of 'fulljones', 'diagonal', 'phaseonly', 'scalarphase'. Experimental values are 'amplitude' or 'scalaramplitude', 'tec', 'tecandphase' | | <step>.parmdb | string | | Path of parmdb in which the computed parameters are to be stored. If the parmdb already exists, it will be overwritten. Note**: You cannot use this parmdb in an applycal step in the same run of DPPP. To apply the solutions of the gaincal directly, use 'gaincal.applysolution' (see below).** New in LOFAR 3.1: **if the parmdb name ends in `.h5` , an H5Parm will be written.| | <step>.blrange | vector | | Vector of baseline lengths to use for calibration. See Description of baseline selection parameters. New in version 2.20 | | <step>.uvlambdamin | double | 0 | Ignore baselines / channels with UV < uvlambdamin wavelengths.** Note**: also all other variants of uv flagging described in UVWFlagger (uvmmin, uvmrange, uvlambdarange, etc) are supported (New in 3.1)| | <step>.baseline | string | | Baseline selection filter for calibration. See Description of baseline selection parameters. New in version 2.20 | | <step>.applysolution | bool | false | Apply the calibration solution to the visibilities. Note that you should always also inspect the parmdb afterwards to check that the solutions look reasonable. | | <step>.solint | int | 1 | Number of time slots on which a solution is assumed to be constant (same as CellSize.Time in BBS). 0 means all time slots. Note that for larger settings of solint, and specially for solint = 0, the memory usage of gaincal will be large (all visibilities for a solint should fit in memory).| | <step>.nchan | int | 0 | Number of channels on which a solution is assumed to be constant (same as**

CellSize.Freq in BBS). 0 means all channels. When caltype = 'tec' or 'tecandphase', the default is 1, meaning that a TEC will be fitted through a phase for each channel. | | <step>.usemodelcolumn | bool | false | Use model column. The model column name can be specified with msin.modelcolumn (default MODEL_DATA) | | <step>.applybeamtomodelcolumn | bool | false | Apply the beam model (at the phase center) to the visibilities in the model column. If this option is true, all options from applybeam are valid as well (except .invert, since the model data will always be corrupted for the beam)| | <step>.propagatesolutions | bool | true | Use solutions of one time interval as a starting value for the next time interval | | <step>.maxiter | int | 50 | Maximum number of iterations of stefcal | | <step>.detectstalling | bool | true | Detect if the iteration does not converge anymore and then stop iterating even if maxiter is not reached | | <step>.tolerance | float | 1.e-5 | Tolerance to which the model should match the data | | <step>.minblperant | int | 4 | If an antenna has less than minblperant unflagged data points for a given solution slot, it is not used for calibration | | <step>.timeslotsperparmupdate | int | 500 | Number of solution intervals after which the parmdb is updated | | <step>.debuglevel | int | 0 | Debugging. If debuglevel==1, then a file debug.h5 is created containing all iterands. This file will be very large; you can use it to check the convergence speed etc. | | <step>.sourcedb | | | Same as in Predict **step | | <step>.sources | | | Same as in** Predict **step | | <step>.usebeammodel | | | Same as in** Predict **step | | <step>.operation | | | Same as in** Predict **step | | <step>.applycal.* | | | ApplyCal sub-step, same as in** Predict **step | | <step>.onebeamperpatch | | | Same as in** ApplyBeam **step | | <step>.usechannelfreq | | | Same as in** ApplyBeam **step | | <step>.beammode | | | Same as in** ApplyBeam **step | ==== DDECal ==== | <step>.type | string | | Case-insensitive step type; must be 'ddecal'. | | <step>.sourcedb | string | | Sourcedb (created with `makesourcedb`) with the sky model to calibrate on. | | <step>.directions | list | [] | List of directions to calibrate on. Every element of this list should b a list of facets. Default: every facet is a direction. | | <step>.maxiter | int | 50 | Maximum number of iterations. | | <step>.detectstalling | bool | true | Stop iterating when no improvement is measured anymore (after a minimum of 30 iterations). | | <step>.stepsize | double | 0.2 | stepsize between iterations. | | <step>.h5parm | string | | Filename of output H5Parm (to be read by e.g. losoto). If empty, defaults to `instrument.h5` within the measurement set. | | <step>.solint | int | 1 | Solution interval in timesteps. | | <step>.usebeammodel | bool | false | use the beam model. All beam-related options of the Predict step are also valid. | | <step>.mode | string | complexgain | Type of constraint to apply. Options are scalarcomplexgain, scalarphase, scalaramplitude, tec, tecandphase. Modes in development are fulljones, complexgain, phaseonly, amplitudeonly, rotation, rotation+diagonal. | | <step>.tolerance | double | 1e-5 | Controls the accuracy to be reached: when the normalized solutions move less than this value, the solutions are considered to be converged and the algorithm finishes. Lower values will cause more iterations to be performed.| | <step>.propagatesolutions | bool | false | Initialize solver with the solutions of the previous time slot. | | <step>.approximatetec | bool | false | Uses an approximation stage in which the phases are constrained with the piece-wise fitter, to solve local minima problems. Only effective when mode=tec or mode=tecandphase. | | <step>.maxapproxiter | int | maxiter/2 | Maximum number of iterations during approximating stage. | | <step>.approxchunksize | int | 0 | Size of fitted chunksize during approximation stage in nr of channels. With approxchunksize=1 the constraint is disabled during the approx stage (so channels are solved for independently). Once converged, the solutions are constrained and more iterations are performed until that has converged too. The default is**

approxchunksize=0, which calculates the chunksize from the bandwidth (resulting in 10 chunks per octave of bandwidth). | | <step>.approxtolerance | double | tolerance*10 | Tolerance at which the approximating first stage is considered to be converged and the second full-constraining stage is started. The second stage convergences when the tolerance set by the 'tolerance' keyword is reached. Setting approxtolerance to lower values will cause more approximating iterations. Since tolerance is by default 1e-5, approxtolerance is by default 1e-4. | | <step>.nchan | int | 1 | Number of channels in each channel block, for which the solution is assumed to be constant. The default is 1, meaning one solution per channel (or in the case of constraints, fitting the constraint over all channels individually). 0 means one solution for the whole channel range. If the total number of channels is not divisable by nchan, some channelblocks will become slightly larger. | | <step>.coreconstraint | double | 0 | Distance in meters. When unequal to 0, all stations within the given distance from the reference station (0) will be constraint to have the same solution. | | <step>.statfilename | string | "" | File to write the step-sizes to. Form of the file is: "<iterationnr> <normalized-stepsize> <unnormalized-stepsize>", and all solution intervals are concatenated. File is not written when this parameter is empty. | | <step>.uvlambdamin | double | 0 | Ignore baselines / channels with UV < uvlambdamin wavelengths. Note**: also all other variants of uv flagging described in UVWFlagger (uvmmin, uvmrange, uvlambdarange, etc) are supported (New in 3.1)| **==== Predict ==== | <step>.type | string | | Case-insensitive step type; must be 'predict' | | <step>.sourcedb | string | | Path of sourcedb in which a sky model is stored (the output of makesourcedb)| | <step>.sources | string vector | [] | Patches to use in the predict step of the calibration | | <step>.usebeammodel | bool | false | Use the LOFAR beam in the predict part of the calibration | | <step>.operation | string | replace | Should the predicted visibilities replace those being processed (`replace`, default), should they be subtracted from those being processed (`subtract`) or added to them (add) | | <step>.applycal.* | | | Set of options for applycal to apply to this predict. For this applycal-substep, .invert is off by default, so the predicted visibilities will be corrupted with the parmdb | | <step>.onebeamperpatch | | | Same as in** ApplyBeam **step | | <step>.usechannelfreq | | | Same as in** ApplyBeam **step | | <step>.beammode | | | Same as in** ApplyBeam **step | ==== H5ParmPredict ==== | <step>.type | string | | Case-insensitive step type; must be 'h5parmpredict' | | <step>.sourcedb | string | | Path of sourcedb in which a sky model is stored (the output of makesourcedb)| | <step>.parmdb | string | | Path of the h5parm in which the corruptions are stored | | <step>.applycal.correction | string | | SolTab which contains the directions to be predicted. The names of the directions need to look like `[dir1,dir2]`, where `dir1` and `dir2` are patches in the sourcedb. | | <step>.directions | string vector | [] | List of directions to include. Each of those directions needs to be in the h5parm soltab. If empty, all directions in the soltab are predicted. || | <step>.usebeammodel | bool | false | Use the LOFAR beam in the predict part of the calibration | | <step>.operation | string | replace | Should the predicted visibilities replace those being processed (`replace`, default), should they be subtracted from those being processed (`subtract`) or added to them (add) | | <step>.applycal.* | | | Set of options for applycal to apply to this predict. For this applycal-substep, .invert is off by default, so the predicted visibilities will be corrupted with the parmdb | | <step>.onebeamperpatch | | | Same as in** ApplyBeam **step | | <step>.usechannelfreq | | | Same as in** ApplyBeam **step | | <step>.beammode | | | Same as in** ApplyBeam **step | ==== ApplyBeam ==== | <step>.type | string | | Case-insensitive step type; must be 'applybeam' | | <step>.onebeamperpatch | bool |**

true | Compute the beam only for the center of each patch (saves computation time, but you should set this to false for large patches. This option is only useful if the beam is applied as part of a predict step. | | <step>.usechannelfreq | bool | true | **Compute the beam for each channel of the measurement set separately. This is useful for merged / concatenated measurement sets. For raw LOFAR data you should set it to false, so that the beam will be formed as in the station hardware. Also, setting it to false is faster. | | <step>.updateweights | bool | false | Update the weights column, in a way consistent with the weights being inverse proportional to the autocorrelations (e.g. if 'autoweights' was used before). | | <step>.invert | bool |** true** | Invert the beam. When applying the beam to transfer calibration solutions, this should be true. In other words: `invert=true` means correcting for the beam, `invert=false` means corrupting with the beam. When using the beam in a predict (or gaincal) step, this option defaults to `false` (so it will corrupt for the beam). |

| <step>.beammode | string | “default” | Beam mode to apply, can be “array_factor”, “element” or “default”. Default is to apply both the element beam and the array factor. |

UVWFlagger

<step>.type	string		Case-insensitive step type; must be 'uvwflagger' or 'uvwflag'.
<step>.count.save	bool	false	If true, the flag percentages per frequency are saved to a table with extension `.flagfreq` and percentages per station to a table with extension `.flagstat`. The basename of the table is the MS name (without extension) followed by the stepname and extension.
<step>.count.path	string	“”	The directory where to create the flag percentages table. If empty, the path of the input MS is used.
<step>.uvmrange	string vector	[]	Flag baselines with UV within one the given ranges (in meters). Delimiters .. and +- can be used to specify a range. E.g., `uvmrange = [20..30, 40+-5]` flags baselines with UV in range 20-30 meter and 35-45 meter.
<step>.uvmmin	double	0	Flag baselines with UV < uvmmin meter.
<step>.uvmmax	double	1e15	Flag baselines with UV > uvmmax meter.
<step>.umrange	string vector	[]	Flag baselines with U within one of the given ranges (in meters).
<step>.ummin	double	0	Flag baselines with U < ummin meter.
<step>.ummax	double	1e15	Flag baselines with U > ummax meter.
<step>.vmrange	string vector	[]	Flag baselines with V within one of the given ranges (in meters).
<step>.vmmin	double	0	Flag baselines with V < vmmin meter.
<step>.vmmax	double	1e15	Flag baselines with V > vmmax meter.
<step>.wmrange	string vector	[]	Flag baselines with W within one of the given ranges (in meters).
<step>.wmmin	double	0	Flag baselines with W < wmmin meter.
<step>.wmmax	double	1e15	Flag baselines with W > wmmax meter.

<step>.uvlambdarange	string vector	[]	Flag baselines/channels with UV within one the given ranges (in wavelengths). Delimiters .. and +- can be used to specify a range. E.g., `uvlambdarange = [20..30, 40+-5]` flags baselines/channels with UV in range 20-30 wavelengths and 35-45 wavelengths.
<step>.uvlambdamin	double	0	Flag baselines/channels with UV < uvlambdamin wavelengths
<step>.uvlambdamax	double	1e15	Flag baselines/channels with UV > uvlambdamax wavelengths
<step>.ulambdarange	string vector	[]	Flag baselines/channels with U within one the given ranges (in wavelengths).
<step>.ulambdamin	double	0	Flag baselines/channels with U < ulambdamin wavelengths
<step>.ulambdamax	double	1e15	Flag baselines/channels with U > ulambdamax wavelengths
<step>.vlambdarange	string vector	[]	Flag baselines/channels with V within one the given ranges (in wavelengths).
<step>.vlambdamin	double	0	Flag baselines/channels with V < vlambdamin wavelengths
<step>.vlambdamax	double	1e15	Flag baselines/channels with V > vlambdamax wavelengths
<step>.wlambdarange	string vector	[]	Flag baselines/channels with W within one the given ranges (in wavelengths).
<step>.wlambdamin	double	0	Flag baselines/channels with W < wlambdamin wavelengths
<step>.wlambdamax	double	1e15	Flag baselines/channels with W > wlambdamax wavelengths
<step>.phasecenter	string vector	[]	If given, use this phase center to calculate the UVW coordinates to flag on. The vector can consist of 1, 2 or, 3 values. If one value is given, it must be the name of a moving source (e.g. SUN or JUPITER). Otherwise the first two values must contain a source position that can be given in sexagesimal format or as a value followed by a unit. The third value can contain the direction type; it defaults to J2000. Possible types are GALACTIC, ECLIPTIC, SUPERGAL, J2000, B1950 (as defined in the casacore `Measures` system).

Description of baseline selection parameters

Parameters to select on baseline can be used in the steps preflagger and filter. The step msin only supports .baseline. The parameters are described in the table below.

Parameter	type	default	description
.corrtype	string	“”	Correlation type to match? Must be auto, cross, or an empty string (= all).
.blrange	double vector	[]	Zero or more ranges of physical baseline lengths (in m). A baseline matches if its length is within one of the ranges. E.g., `blrange=[0,10000, 100000, 1e30]`

Parameter	type	default	description
.baseline	baseline vector	“”	Names of baselines to be matched. It can be given as either a vector of vectors or as a casacore MSSelection string. These two methods are mutually exclusive. When in doubt, use the second syntax. 1. If given as a vector, a vector element can be a vector of two names giving the stations forming a baseline. For example: `baseline=[ [CS001,RS003], [CS002,RS005] ]` selects baselines CS001-RS003 and CS002-RS005. Each name can be a shell-type pattern (with wildcards * ? [] or {}). Thus `baseline=[ [CS*,RS*] ]` selects all baselines between core and remote stations. Note that the wildcard characters {} mean OR. They can be used to pair groups of stations (quotes are needed). For example: `baseline=[ [“{CS001,CS002}”,“{RS003,RS005}”] ]` selects baselines CS001-RS003, CS001-RS005, CS002-RS003, and CS002-RS005. Besides giving a baseline, it is also possible to give a single station name (possibly wildcarded) meaning that all baselines containing that station will be selected. For example: `baseline=[RS*,CS*]` selects all baselines containing remote or core stations. Please note that an extra bracket pair is needed to specify baselines between RS and CS like in `baseline=[ [RS*,CS*] ]` It is a bit hard to select international stations using this syntax. 2. The casacore MSSelection baseline syntax is described in this note and Casacore note 263. The advantage of this syntax is that it is more concise and that besides a station name pattern, it is possible to give a station number. The examples above can be expressed as: `baseline=CS001&RS003;CS002&RS005` for baseline CS001-RS003 and CS002-RS005 `baseline=CS001,CS002&RS003,RS005` for CS001-RS003, CS001-RS005, CS002-RS003, and CS002-RS005 `baseline=RS*&&CS*` for baselines (also auto-corr) between RS and CS stations. `baseline=8&12` baseline between station number 8 and 12. Note that & means cross-correlations, && means cross and auto, &&& means auto only. International stations can be selected most easily using negation. E.g. use `baseline=^[CR]S*&&*` to select all baselines containing an international station. use `baseline=^[CR]S*&&` to select baselines containing ONLY international stations. Sometimes the baselines between the HBA ears of the same station should be deselected, which can be done using the following string `^/(.*)HBA0&\1HBA1/` Without the up-arrow it will select such baselines. Note: in the `msin` step only the second way is possible. Also note that, currently, only the first way works properly when selecting baselines after a station has been added. The reason is that the second way looks in the original ANTENNA table to find matching station names, thus will not find the new station.