FAUST Archival Imaging Pipeline

Release 0.1.0

Brian Svoboda, Claire Chandler

Sep 14, 2023

PAGES:

Welcome to the documentation for the FAUST Archival Imaging Pipeline! These web pages contain information on how to generate deconvolved image products for the ALMA FAUST Large Program. For instructions on how to download the pipeline script and setup your host environment in order to use the pipeline see the *Installation* page. Once all datasets and paths are in the appropriate locations, please refer to the *User Guide* for information on executing pipeline jobs. Detailed descriptions of the pipeline components and how to configure them may be found in the *Cookbook*.

The pipeline is authored by Brian Svoboda and Claire Chandler. The code is released under the MIT License. A copy of the license is supplied under the LICENSE file distributed with the software. The code may be found on GitHub.

CHAPTER

ONE

DOCUMENTATION

1.1 Pipeline installation and setup

The pipeline comprises a single Python file to be run using CASA. The pipeline script does not require installation per se, but must be executed when starting a new CASA process or session. The instructions below detail how to download the script, setup the correct pathes and directories, and execute the script.

1.1.1 CASA compatibility

The pipeline script can be used with the "monolithic" versions of CASA v5 (Python 2) or CASA v6 (Python 3). These are the versions of CASA downloaded from the main website. The pipeline has been most extensively tested under CASA v5.6 (the current Calibration Pipeline release), but has been tested to at least nominally work under CASA v6.2. We recommend using CASA v5.6 (or the latest v5 release) for consistency and accuracy. The "Standard Products" to be distributed to the ALMA Archive are made using CASA v5.6.

The script may also be run using the casatasks and casatools Python 3 library modules. The latter may be installed into the user's Python environment by running `pip install --user casatasks casatools`, preferably within a project-specific virtual environment.

1.1.2 Downloading the pipeline

The pipeline comprises a single Python source file, `faust_imaging.py`, that may be downloaded either by downloading the repository from GitHub as a Zip archive or by cloning the repository using Git:

```
git clone https://github.com/autocorr/faust_line_imaging
# this will create the directory "faust_line_imaging" in the current
# working directory.
```

Updates to the script from the main repository can be automatically merged by running `git pull` from the repository. To modify the source and have those changes reflected in the main repositry, please fork the above repository on GitHub and file a pull request.

1.1.3 Paths and directory setup

The pipeline script currently requires an explicit directory structure to read the measurement sets and write out the images. Essential global paths are set by the user in the configuration file `faust_imaging.cfg`. The contents of the template file included in the source repository are written below:

```
# This is the user configuration file for the FAUST archival imaging pipeline.
# Edit this file with settings appropriate for your host system and place it in
# the directory specified by PROD_DIR.

[Paths]
# Specify where the measurement set files/directories are located. The MSs
# for a field should follow the directory structure:
#   <DATA_DIR>/<FIELD>-Setup<N>/*.ms
# e.g.,
#   /scratch/faust/CB68-Setup1/*.ms
DATA_DIR=/lustre/aoc/users/USER/FAUST/2018.1.01205.L/completed_SBs/

# Specify where CASA is to be run and also the base-path where products
# are to be written.
PROD_DIR=/lustre/aoc/users/USER/faust/faust_alma/data/
```

Copy the template `faust_imaging.cfg` file from the repository into the directory where CASA is to be run from (`PROD_DIR`) and edit the paths appropriately for your host system. This file will be read when the pipeline script is executed in CASA. Sub-directories for images cubes, moments, and plots will be automatically generated.

Visualizing the directory structure as a tree, the file hierarchy would be organized:

```
/<PATH>/<TO>/DATA_DIR/   <- location on user's system
  - CB68-Setup1          <- MS files per target per setup
  - CB68-Setup2
  - ...
/<PATH>/<TO>/PROD_DIR/   <- location on user's system
  - images/              <- these will be made automatically
    + CB68/              <- image cubes and products per target
    + L1527/
    + ...
  - moments/
    + CB68/              <- moment maps per target
    + L1527/
    + ...
  - plots/               <- diagnostic and QA plots
```

Some CASA tasks do not work using absolute paths, so please ensure that the pipeline script is run from the directory specified by `PROD_DIR`. The measurement set files should be present in the directories:

```
$DATA_DIR/<TARGET>-Setup1/
$DATA_DIR/<TARGET>-Setup2/
$DATA_DIR/<TARGET>-Setup2/
```

where `<TARGET>` is the FAUST target field name, e.g. “CB68” or “L1527”. The calibrated measurement sets may be downloaded from RIKEN. The names may be found in the `ALL_TARGET_NAMES` global variable, I retrieved these values from the proposal, so they may be inconsistent for some targets. The paths above may modified directly by editing the format string attribute `DataSet.ms_fmt`.

1.1.4 Executing the script

Ensure that the script can be properly executed from within CASA by starting CASA from the directory set in `PROD_DIR`. The pipeline script can then be executed and all functions/symbols brought into scope using the `execfile` command:

```
# using a relative path
execfile('../<Path>/faust_line_imaging/faust_imaging.py')
# or alternatively using an absolute path
execfile('/<PATH>/<TO>/faust_line_imaging/faust_imaging.py')
```

This `execfile` command needs to be run whenever starting CASA or when the pipeline script source code is modified. Note that the `execfile` command can also be performed within "recipe" scripts that are themselves run with `execfile` in CASA.

1.1.5 Notes for MacOS users

MacOS users will likely need to run the command `ulimit -Sn 8000` from the shell before starting CASA. This command increases the maximum number of files that may be opened at once.

1.1.6 Next steps

Congratulations! Now that the environment is setup, please now refer to the *User Guide* or click the "Next" button for instructions on running the pipeline.

1.2 User Guide

Welcome to the pipeline User Guide! This guide details how to (1) run the pipeline interactively for combinations of targets and spectral windows, (2) how to trouble-shoot the pipeline products and results, and (3) run batch scripts for parallel processing. A table of contents tree may be found in the left side-bar.

1.2.1 Note on system resources

Imaging the full bandwidth cubes can require a large volume of disk space to store all of the intermediate image products. For example, the products can be greater than 100GB for a single SPW and several TB for a full Setup. After producing the final cubes, the intermediate products may be removed. Please ensure that the host system has sufficient resources to run the pipeline. If the host system has limited storage, we recommend imaging one spectral window (SPW) at a time and removing the intermediate products after it has finished.

1.2.2 Importing the pipeline

If you have not already done so, please see the *Installation* page for instructions on how to download and load the script in CASA. Please ensure that the paths are configured correctly in the `faust_imaging.cfg` file: (1) that `DATA_DIR` points to the directory containing the measurement sets and (2) that CASA is started from the directory set in `PROD_DIR`). Next, start CASA interactively from a shell prompt and execute the pipeline script using `execfile` (see the *Installation* page). To start an interactive session with an example products directory of `PROD_DIR="/mnt/scratch/faust"` and a pipeline script located at `/mnt/scratch/faust/faust_line_imaging/faust_imaging.py`, run:

```
$ cd /mnt/scratch/faust  # for example
$ casa
```

and then from the interactive CASA prompt:

```
CASA <1>: execfile('faust_line_imaging/faust_imaging.py')
```

1.2.3 Names and labels

Targets are referred to by their field names, e.g. "CB68" or "BHB07-11", as they appear in the measurement sets. All valid field names can be read from the global variable `ALL_FIELD_NAMES`:

```
print(ALL_FIELD_NAMES)
```

SPWs are assigned unique labels based on the principle molecular tracer targeted in the window. These labels are prefixed by the rest frequency of the transition (truncated to the nearest MHz) and followed by the simple formula name for the molecule. The CS (5-4) transition in Setup 2, for example, has the label "244.936GHz_CS". All valid SPW labels can be read from the global variable `ALL_SPW_LABELS`:

```
print(ALL_SPW_LABELS)
```

To image narrow-bandwidth "cut-outs" around lines without imaging the full bandpass of a SPW, see the Cookbook section *Imaging cut-out velocity windows*.

1.2.4 Running the pipeline

The primary user interface for running the pipeline tasks is through the Python class *faust_imaging.ImageConfig*, specifically the constructor *faust_imaging.ImageConfig.from_name()*. The preceding links point to the API documentation that further describe the calling convention and arguements functions and classes take. Most of this information can also be found in the docstrings as well, which can be read by running `help <ITEM>` or `<ITEM>?` from the CASA prompt. The *Pipeline task description* section of the Cookbook describes the individual processing steps of the pipeline in detail.

Once an instance of `ImageConfig` is created and set with the desired options, the pipeline can be be run using the *faust_imaging.ImageConfig.run_pipeline()* bound method. To run the pipeline for "CB68" and the SPW containing CS (5-4) in Setup 2 with the default pipeline parameters, run:

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS')
config.run_pipeline(ext='clean')
# The final products will be given a suffix based on `ext` above,
# the default is "clean". Different names may be used to avoid over-
# writing existing files.
```

The default parameters will use a Briggs robust *uv*-weighting of 0.5 and jointly deconvolve all array configurations (12m & 7m). We recommend imaging one "line of interest" first to test that the pipeline works and produces sensible results before moving to batched processing. The final pipeline products will have a suffix "_clean" before the CASA image extension (e.g., ".image" or ".mask"). In the above example, the final primary beam corrected image will be written to:

```
# Path relative to where CASA is run (also the value of `PROD_DIR`)
images/CB68/CB68_244.936GHz_CS_joint_0.5_clean.image.pbcor.common.fits

# Files are named according to the convention:
#   <FIELD>_<SPW_LABEL>_<ARRAY>_<WEIGHTING>_<SUFFIX>.<EXT>
# where
```

(continues on next page)

(continued from previous page)

```
#   FIELD     : source/field name
#   SPW_LABEL : rest-frequency and molecular tracer based name
#   ARRAY     : array configurations used; can be 'joint', '12m', '7m'
#   WEIGHTING : uv-weighting applied, e.g. 'natural' for natural weighting
#               or '0.5' Briggs robust of 0.5.
#   SUFFIX    : name to distinguish different files, 'clean' is used for
#               the default final pipeline products. Intermediate products
#               will also exist with suffixes including 'dirty', 'nomask',
#               etc.
#   EXT       : CASA image extension name, e.g. '.image' or '.mask'

# Other non-standard image extensions produced will include '.common' for
# images that have been smoothed to a common beam resolution, '.hanning' for
# hanning smoothed images, and '.pbcor' for images corrected for attenuation
# by the primary beam.
```

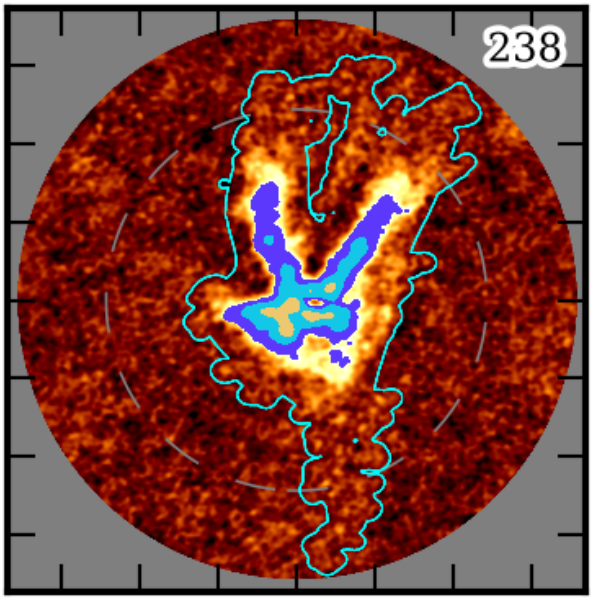
Please see the *API documentation* or docstring for further configuration information. For examples of more advanced uses of the pipeline please refer to the *Cookbook*.

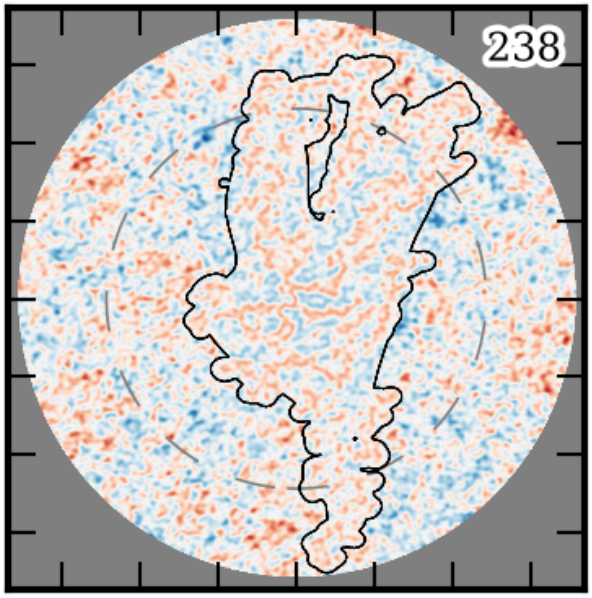
1.2.5 Quality assurance plots and moment maps

After the pipeline has been run, the next step is to validate the results. This is done by creating quality assurance plots for visual inspection and moment maps. The QA plots generate channel maps of the restored image and residual with the clean-mask overplotted. Only channels containing significant emission are plotted (regardless of whether the emission is masked). The default threshold to show such channels is 6-times the full-cube RMS. To create the quality assurance plots call the function *faust_imaging.make_all_qa_plots()* for the desired field and extension (e.g., "clean" as used above).

```
make_all_qa_plots('CB68', ext='clean', overwrite=False)
```

The `overwrite=False` keyword argument ensures that QA plots are only generated for images that do not already exist, so this function can be safely called after new pipeline jobs have been run. Plots will be written to `plots/` or the value of `PLOT_DIR`. Note that the creation of the plots is implemented inefficiently with `matplotlib` and `imshow`, and thus creating the plots for the Setup 3 SPWs may require >50 GB of memory.





The above figures show channel index number 238 of CB68 CS (5-4) for the restored image (**left**) and the residual image (**right**). For the restored image, color-scale ranges from -3 to 10 times the full-cube RMS and the filled contours are shown at increments of 10, 20, 40, and 80 times the RMS. The cyan contour shows the clean mask. For the residual image the color scale ranges from -5 to 5 times the RMS (negatives shown in blue) and the black contour shows the clean mask. The tick-marks show increments of 5 arcsec, the dashed line shows the half-power beamwidth of the primary beam, and imaged out to the 20%-power point of the primary beam.

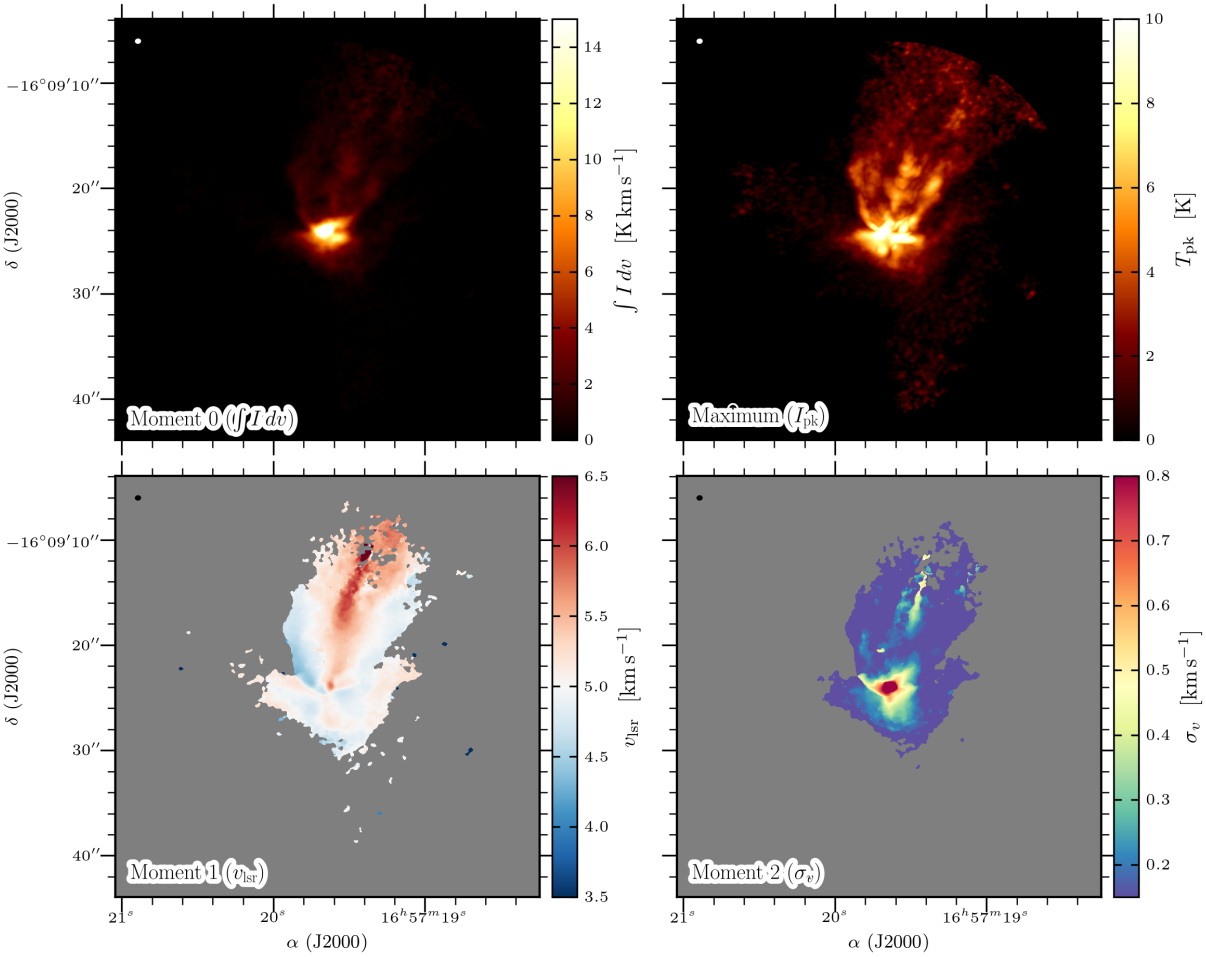
Further details on the QA plots may be found in the Cookbook section *Quality assurance plots*.

Moment maps may be generated by running the `make_all_moment_maps()` function for the desired field and extension (e.g., "clean" as used above).

```
make_all_moment_maps('CB68', ext='clean')
```

Maps based on the integrated intensity ("mom0"), maximum or peak intensity ("max"), centroid velocity ("mom1"), and velocity dispersion ("mom2") will be written to the `moments/` directory or the value of `MOMA_DIR`. The images

can be inspected with your FITS viewer of choice. The figure below shows an example matplotlib visualization of the moments for CB68 CS (5-4).



The above figures can be generated using the `util/moment_plotting.py` script under Python **v3** (currently undocumented; requires packages numpy, scipy, skimage, matplotlib, aplpy, radio_beam, and astropy).

1.2.6 Trouble-shooting

With the deconvolved image products and the quality assurance plots made, the next step is to inspect the results and resolve whether they are satisfactory for the science-goals of the Source Team. The following sub-sections describe common scenarios where the results are problematic and steps that may be taken to improve the imaging.

Extended negative emission

Are any strong negative-intensity artifacts or "bowls" masked? Extended emission that is not properly recovered due to missing short-spacings can introduce negative bowls that should not be cleaned and added to the source model. If this is observed, the auto-masking parameters may be tuned to limit the masking of negative emission.

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS')
config.autom_kwargs['negativethreshold'] = 8  # the default is 7
config.run_pipeline()
```

True absorption does frequently occur, however, towards the bright and compact continuum emission the central protostellar source(s). Because the visibility data is continuum subtracted, this absorption will appear negative in the restored images. This absorption should be masked and cleaned.

Overly permissive clean masks

Does the generated clean-mask appear to be overly permissive and include large areas without apparent emission? This effect has been known to appear in earlier iterations of the pipeline for certain fields with many execution blocks. The pipeline uses the auto-multithresh algorithm in `tclean` to procedurally generate the clean masks with an initial mask generated from a partially deconvolved version of the image. If the parameters of the auto-multithresh algorithm (Kepley et al. (2020)) are improperly tuned, the mask can undergo something similar to runaway growth yielding an "amoeba" like appearance, as can be seen in the following figure:

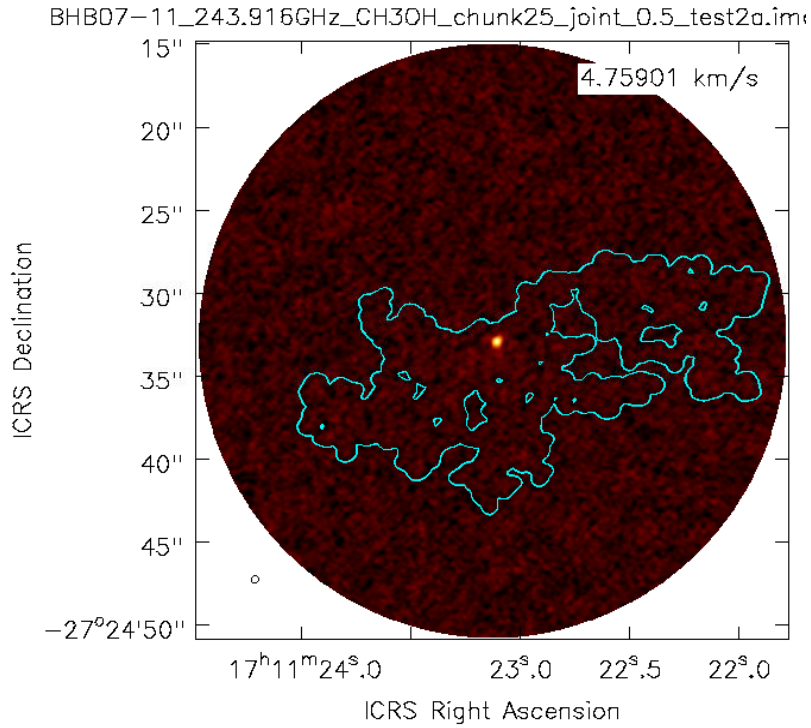


Fig. 1: The mask includes a large fraction of the field without apparent emission.

In some circumstances, all pixels in a channel may even be included in the mask. Note that such cases will appear to have no mask when using the `casaviewer` to plot a contour-diagram. This effect seems to be largely mitigated with the latest set of default parameters, but careful attention should be paid in case it appears. Spurious masking will have adverse effects on both the image quality and the moment maps. Over-cleaning within such a mask may corrupt the

noise statistics and include artifacts in the source model. The clean mask is also used for selecting pixels to use in creating the moment maps, and can produce poor results when large areas of effectively just noise are included.

The most straightforward solution is to raise the significance threshold used to "grow" the mask.

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS')
config.autom_kwargs['lownoisethreshold'] = 2.0  # the default is 1.5
config.run_pipeline()
```

Significant uncleaned emission

Has the automated masking left significant levels of emission unmasked, and thus uncleaned? This can frequently be diagnosed in the QA plots of the residual image. The investigator may use their discretion to decide whether such emission produces adverse affects and should be cleaned. Multiple methods exist to fix such images without re-running the full pipeline over again. The final clean may be restarted with:

1. auto-multithresh but with a lower 'lownoisethreshold'
2. auto-multithresh and manually adding regions to the existing mask
3. without using auto-multithresh and manually adding regions to the existing mask
4. an initial mask including fainter and more extended emission

The following example describes methods 1-3. The pipeline processes discrete image "chunks" in frequency to improve performance and ease memory constraints. Restarting thus requires operating on the chunk containing the offending emission. More information on manually restarting one chunk is described in the Cookbook *Restarting and manually cleaning a single chunk* section. In the following example, channel index number 238 is insufficiently cleaned and the offending chunk is restarted with the interactive cleaning.

```
# The standard, full instance will apply operations to the entire SPW,
# even if "under the hood" the processes are being applied to each
# sub-image or "chunk".
full_config = ImageConfig.from_name('CB68', '244.936GHz_CS')

# To get the frequency chunk with issues, we manually retrieve the
# `ImageConfig` instances for every chunk.
chunked_configs = full_config.duplicate_into_chunks()
problematic_config = chunked_configs.get_chunk_from_channel(238)

# (Method 1) Restart tclean non-interactively with a lower
# `lowernoisethreshold` for auto-multithresh.
#problematic_config.autom_kwargs['lownoisethreshold'] = 1.0
#problematic_config.clean_line(ext='clean', restart=True)

# (Method 2) Restart tclean interactively using the existing clean mask and
# model.
problematic_config.clean_line(ext='clean', restart=True, interactive=True)
# ^ The casaviewer will appear for manual masking. Identify the channel
#   with the offending emission (the channel indices will now be of the
#   chunk) and draw an addition to the mask. Often times it suffices to
#   select the "blue rightward arrow" icon immediately if the emission is
#   faint.

# (Method 3) Restart tclean interactively without auto-multithresh, using
```

(continues on next page)

(continued from previous page)

```
# a static mask that we can add to.
#problematic_config.clean_line(ext='clean', mask_method='fixed',
#            restart=True, interactive=True)

# Postprocess the results to reproduce the final full-cube products
chunked_configs.postprocess(ext='clean')
```

Alternatively, the procedure used to generate the initial "seed" mask can be modified in order to include larger scales or lower-significance emission (method 4). The final clean run without manual intervention. Following the same conventions as in the previous example:

```
full_config = ImageConfig.from_name('CB68', '244.936GHz_CS')
chunked_configs = full_config.duplicate_into_chunks()
problematic_config = chunked_configs.get_chunk_from_channel(238)

# Add a fourth scale to the seed mask generation using a Gaussian
# kernel with a FWHM of 5 arcsec. The default scales are 0 (unsmoothed),
# 1, and 3 arcsec.
problematic_config.mask_ang_scales = [0, 1, 3, 5]  # arcsec

# The default significance threshold applied to each scale is 5 sigma,
# here we use 4 sigma.
problematic_config.make_seed_mask(sigma=4.0)
# Re-run the deconvolution using the new seed mask.
problematic_config.clean_line(ext='clean')

chunked_configs.postprocess(ext='clean')
```

Inconsistent masking from varying noise

Are an unusual number of noise spikes masked at the band edges? The sensitivity as a function of frequency for some SPWs is affected by atmospheric lines. Examples include the "231.221_13CS" and "231.322_N2Dp" SPWs in Setup 1. An atmospheric ozone feature between these two windows increases the RMS by about 20% towards the respective band edge. In some circumstances, the use of a single RMS can lead to over-masking of many small noise spikes near the band edge. If this is the case, then using smaller image-chunk sizes should give more uniform results.

Divergences or negative edge-features

Do very strong, negative features appear at the edge of the mask or field? It is a known issue that the multiscale clean implementation in CASA can be unstable when also using clean masks. In some circumstances `tclean` can diverge at the edge of the clean mask or primary beam mask and insert spurious positive-intensity features into the model. These features are usually on large scales (often similar to the ACA synthesized beam) and produce strong negative-intensity features in the restored image.

The default parameters have been found to largely stabilize `tclean` by slowing the rate of convergence in the minor cycle. If these divergences appear, try running the pipeline with a lower `gain` and higher `cyclefactor`:

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS')
config.gain = 0.03  # default 0.05
config.cyclefactor = 2.5  # default 2.0
config.run_pipeline()
```

The above changes to the loop gain and cyclefactor may make `tclean` run much more slowly however. Alternative solutions are to reduce the size of the largest scale used by multiscale-clean or omit the largest scale altogether:

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS')
config.scales = [0, 15, 45]  # pix; default [0, 15, 45, 135]
# the cell size in arcsec can be read from `config.dset.cell`
config.run_pipeline()
```

1.2.7 Running the parallel pipeline

To run the pipeline in parallel, please refer to the *Parallelized computation with multiple CASA processes* section in the Cookbook. Example scripts are included for imaging a single SPW in parallel and also imaging all of the SPWs for a setup in parallel. On the NRAO NM postprocessing cluster, typical run-times are a few hours when imaging a single SPW in parallel and a few days for imaging all SPWs of a setup.

1.3 Pipeline Cookbook

The heuristics adopted in the pipeline appear to work well in the majority of cases, but there are invariably cases not adequately treated by the defaults and thus require custom processing. To aid in the processing of a FAUST target, the pipeline provides helper classes that abstract away the book-keeping aspects into individual tasks to be configured depending on the requirements of a particular field and SPW.

Note that because the pipeline is computationally intensive, even imaging a single SPW can take a day or more when CASA is run using single threaded execution, and still a substantial amount of time when executed in parallel.

1.3.1 Running the default pipeline

The *faust_imaging.ImageConfig* class provides the primary interface to `tclean` within CASA and encapsulates properties specific to a field, SPW, array configuration, and desired `tclean` parameters. Please refer to the *API Documentation* and the docstring for additional information on the calling convention of this class.

To run all tasks of the pipeline with default parameters, first create an instance of the *faust_imaging.ImageConfig* class and use the *faust_imaging.ImageConfig.run_pipeline()* method.

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS', weighting=0.5)
config.run_pipeline()
```

The full list of SPW labels may be found in the *ALL_SPW_LABELS* variable. The above command will generate the default pipeline image products for the target field CB68, for the CS (5-4) line of Setup 2, using a Briggs robust factor of `0.5`. The full calling convention is:

```
ImageConfig.from_name(
        '<FIELD>',  # field name, e.g., "CB68". See the global var `ALL_FIELDS`
        '<LABEL>',  # SPW label, e.g., "244.936GHz_CS"
        weighting=0.5,
        # ^ Use a number for Briggs robust or a string recognized by tclean,
        # e.g., 'uniform' or 'natural'. The default is 0.5.
        fullcube=True,
        # ^ Image the full bandwidth of the SPW or a narrow 20km/s window
        # centered on the primary line of interest. The default is True.
        kind='joint',
```

(continues on next page)

(continued from previous page)

```
        # ^ What combination of array configurations to use. Possible values
        # include ('joint', '12m', '7m'). The default is 'joint'.
)
```

1.3.2 Pipeline task description

The *faust_imaging.ImageConfig.run_pipeline()* method discussed above is primarily a wrapper for calling all of the pipeline tasks in sequence using the default parameters. For custom imaging, it is recommended that users create a recipe using the underlying tasks. As an example, this is the code that is executed by default:

```
config = ImageConfig.from_name(...)
ext = 'clean'  # extensionn name for final cleaned products
config.make_dirty_cube()
config.clean_line_nomask(sigma=4.5)
config.make_seed_mask(sigma=5.0)
config.clean_line(mask_method='seed+multithresh', ext=ext)
config.postprocess(ext=ext, make_fits=True)
```

The following sections describe the function of each pipeline step in detail. The start of each sub-section includes a link to the API documentation where a detailed description of the parameters may be found.

`make_dirty_cube`: Creating the dirty cube

faust_imaging.ImageConfig.make_dirty_cube(). A call to `tclean` is first made with `niter=0` to grid and transform the data. These data products have the default extension name of "dirty" which may be changed by modifying the global variable `DIRTY_EXT`. The dirty image products are used for computing the RMS and computing the common frequency coverage between different array configurations.

`clean_line_nomask`: Initial un-masked cleaning

faust_imaging.ImageConfig.clean_line_nomask(). The "auto-multithresh" method for automated clean mask generation in `tclean` determines noise characterstics per-plane, along with other heuristics based on the maximum negative residual, which can make cleaning extended emission where substantial filtering present problematic. For this reason an initial deconvolution is performed without masking to a relatively shallow depth set by the parameter `sigma` with a default value of 4.5 times the RMS. Files are generated using the default extension name "nomask" which can be set using the globa variable `NOMSK_EXT`. To avoid diverging in the unrestricted clean, large angular scales may be excluded from multiscale clean using the `scale_upper_limit` parameter.

`make_seed_mask`: Creating the threshold mask

faust_imaging.ImageConfig.make_seed_mask(). A threshold is applied to the restored image generated by `clean_line_nomask` to create an initial input mask to "seed" the mask created with auto-multithresh. This ensures that all channels, even those with strong spatial filtering, are masked. The threshold to apply may be set with the `sigma` parameter; the default is 5.0 times the RMS.

`clean_line`: Second, masked cleaning

faust_imaging.ImageConfig.clean_line(). A second, new round of deconvolution is performed using the thresholded mask generated in the previous steps as a "seed" for auto-multithresh. This combination of the seed mask and auto-multithresh is the recommended method and is the default masking method, and set as the default through the parameter `mask_method="seed+multithresh"`. If `mask_method="auto-multithresh"` is used then the prior two pipeline steps used to generate the "seed" mask are not required. The global clean threshold can be set with the `sigma` parameter. The parameter `ext` sets the string appended to the filename. The standard convention is to use "clean" for these final products, but other names may be used when experimenting with different parameters to avoid over-writing existing runs.

Runs can be restarted and cleaned interactively using the `restart` and `interactive` keyword arguments. See *Restarting* for further detail.

`postprocess`: Image cube postprocessing

faust_imaging.ImageConfig.postprocess(). Lastly, a few minor post processing steps are applied to the image products generated by `clean_line`. These include (1) checking whether the maximum value of the residual image exceeds 5.5-sigma, (2) correcting the restored image for the primary beam attenuation, (3) smoothing the image to a common-beam angular resolution, and (4) exporting the CASA image to a FITS image. The final FITS image names will be of the form:

```
# template form:
<PROD_DIR>/<FIELD>/<FIELD>_<LABEL>_<KIND>_<WEIGHT>_<EXT>.image.pbcor.common.fits
# example:
images/CB68/CB68_244.936GHz_CS_joint_0.5_clean.image.pbcor.common.fits
```

As an optional final step, quality assurance plots can be generated. These plots are useful for assessing whether further deconvolution is required. Making these plots is described in *QA Plots*.

1.3.3 Quality assurance plots

Quality Assurance (QA) plots are useful for quickly obtaining an overview of whether the deconvolved products are satisfactory. The function *faust_imaging.make_all_qa_plots()* can be used to generate channel maps of all restored images and residual images for a field where the peak restored image intensity exceeds 6-sigma. PDF and PNG files are written to the directory specified in `PLOT_DIR` (by default `<PROD_DIR>/plots`).

```
# to overwrite all existing plots, use the default overwrite=True
make_all_qa_plots('CB68', ext='clean')

# to skip plots that have already been made, set overwrite=False
make_all_qa_plots('CB68', ext='clean', overwrite=False)
```

To make an individual QA plot from an image path name use *faust_imaging.make_qa_plots_from_image()*:

```
# example filename for CB68 CS (5-4)
imagename = 'images/CB68/CB68_244.936GHz_CS_joint_0.5_clean.image'
make_qa_plots_from_image(imagename)
```

For developing custom plotting routines, the *faust_imaging.CubeSet* class may be of use.

1.3.4 Cleaning up and removing files

The pipeline produces a large number of intermediate image products. These products can be kept in order to restart unsatisfactory imaging results or removed if the final products are suitable.

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS', weighting='natural',
             kind='12m')

# The helper method ".remove_all_files()" will identify all files/images with
# names matching the configuration file stem, in this example:
#    images/CB68/CB68_244.936GHz_CS_12m_natural*
# but not others runs imaged with joint 12/7m data or other uv-weightings.
print(config.get_imagebase())
config.remove_all_files()
```

Intermediate image products that are unnecessary for the functioning of the pipeline or restarting image processes are removed during the run. If you wish to preserve these image products, such as for example the `.model` image to the un-masked clean run, then all intermediate products can be kept by setting the *faust_imaging.ImageConfig.preserve_all_intermediate_products* attribute:

```
config = ImageConfig.from_name(...)
config.preserve_all_intermediate_products = True
config.run_pipeline()
```

1.3.5 Imaging cut-out velocity windows

The default pipeline setting of `fullcube=True` will image the entire bandwidth of the SPW. These can be rather large, typically more than 470 channels or 90 km/s. If only a particular line is of interest, then a cut-out in frequency may be imaged to reduce run-time cost and disk space requirements.

If the line to be imaged was the primary target of the SPW, then no changes need to be made, e.g., Setup 1 C18O J=2-1 near 219.56 GHz. The default velocity bandwidth is 20 km/s (+/- 10 km/s about the system velocity) but may be set with the attribute *faust_imaging.ImageConfig.line_vwin*.

```
config = ImageConfig.from_name(..., fullcube=False)
# Set the window for +/- 2.5 km/s (full-width of 5 km/s) centered on the
# system velocity.
config.line_vwin = '5km/s'
config.run_pipeline()
```

The primary transition targeted by the SPW can be determined by comparing the value of `spw.mol_name` (primary molecule) to `spw.name` (full SPW name with transitions from the correlator configuration).

Imaging cut-outs that were not the primary transitions targeted by an SPW requires creating new instances of the classes *faust_imaging.Spw* and *faust_imaging.DataSet*. An instance of `ImageConfig` then must be created directly. This can be particularly useful for the continuum windows which are resource intensive to process with full bandwidth cubes.

```
# We wish to image the acetaldehyde CH3CHO 11(1,10)-10(1,9) transition
# also found in the Setup 1 SPW ID 27. The primary targeted line was
# deuterated ammonia NH2D 3(2,2)s-3(2,1)a. Create a copy of this window
# and change the molecule name (for files and paths) and the line
# rest frequency of the new transition.
spw = ALL_SPWS['216.563GHz_NH2D'].copy()
```

(continues on next page)

(continued from previous page)

```
spw.mol_name = 'CH3CHO'
spw.restfreq = '216.58193040GHz'  # from SLAIM
# Initialize the DataSet class, which contains information such as image
# and cell size, etc., with the same Setup as our SPW (Setup 1 in this
# example).
dset = DataSet('CB68', setup=spw.setup, kind='joint')
# Initialize the `ImageConfig` class directly from the instances and
# run the pipeline.
config = ImageConfig(dset, spw, fullcube=False)
config.line_vwin = '5km/s'  # default 20 km/s
config.run_pipeline(ext='5kms_clean')
# The final image products will be named as:
#   CB68_216.582GHz_CH3CHO_joint_0.5_5kms_clean.*
```

1.3.6 Modifying multi-scale parameters

To adjust the default parameters used by multi-scale CLEAN in the pipeline, modify the attributes listed in the example below:

```
config = ImageConfig.from_name('CB68', '244.936GHz_CS')
# The default scales, in pixels, are (0, 15, 45, 135) corresponding to
# approximately 0, 0.45, 1.35, 4.05 arcsec for a typical HPBW of 0.3as
# and an over-sampling factor of 10 (cell size of 0.03as). Here, we
# replace the 45 pixel scale with two new scales: 30 and 60 pixels.
config.scales = [0, 15, 30, 60, 135]  # pix
# We also increase the small-scale bias parameter from the pipeline
# default of -1.0 to -0.5.
config.smallscalebias = -0.5
config.run_pipeline()
```

1.3.7 Restarting `tclean` and manual masking

Some datasets can be difficult to clean satisfactorily with the default pipeline settings, particularly those with extended emission that suffers heavy spatial filtering (ex. C180, c-C3H2). Good results on these cubes may require manual intervention in the defining the clean masks. After inspecting the results of the pipeline products, a run can be restarted using the same settings but using the `restart` and `interactive` keyword arguments of *faust_imaging.ImageConfig.clean_line()*:

```
config = ImageConfig(...)
# The pipeline should already have been run previously and for this example
# there should exist images with names ending in the extension "clean".

# Now restart the deconvolution using the existing files and run it in
# interactive mode. It's likely a good idea to backup the `.model` and
# `.mask` files in the event the deconvolution fairs poorly. Also, if you
# wish to clean more deeply, one can set the `sigma` argument to a lower
# value here.
config.clean_line(ext='clean', restart=True, interactive=True)
# Note that an alias is included for this use, identical in arguments and
# calling convention as above, named:
```

(continues on next page)

(continued from previous page)

```
#   config.clean_line_interactive_restart(ext='clean')
# Now, if the results are satisfactory, apply the post-processing steps
# to finish the pipeline.
config.postprocess(ext='clean')
```

The above commands will pull up the CASA interactive viewer for creating the clean masks manually. It is recommended to set the `cycleniter` in the upper left to a low value, perhaps 100 iterations, and to use the green "recycling" arrow to execute a few iterations. The un-cleaned emission of interest is often present in channels with extended emission near the edge of the primary beam mask. These channels have a high-probability of diverging if left to finish the run using the blue "right arrow". Once the extended emission appears to have been cleaned satisfactorily, finish the run by clicking the red "stop sign" button.

Restarting `tclean` can also be performed without using the interactive mode. One example usage may be cleaning to a shallow depth, inspecting the results or applying a few tweaks, and then cleaning more deeply.

```
config = ImageConfig(...)
# ... pipeline has been run up to `.clean_line`
# First, clean relatively shallowly to 3.5-sigma and let it run
# automatically.
config.clean_line(ext='clean', sigma=3.5)
# Inspect the resulting cubes to see if the results are satisfactory. The
# QA plots could be made for the target, for example.
# Now, continue the deconvolution to a lower depth of 2.0-sigma
config.clean_line(ext='clean', restart=True, sigma=2.0)
config.postprocess(ext='clean')
```

1.3.8 Creating moment maps

Moment maps and other point estimators (e.g., maximum) may be generated from the data using the *faust_imaging.make_all_moment_maps()* function for all of the SPWs of a target or *faust_imaging.make_moments_from_image()* for a single SPW of a target.

```
# Create moment maps for all SPWs. The `vwin` parameter sets the velocity
# window half-width in km/s to calculate the moment over.
make_all_moment_maps('CB68', ext='clean', vwin=5)

# Generate a single set of moment maps
imagename = 'images/CB68/CB68_244.936GHz_CS_joint_0.5_clean.image'
make_moments_from_image(imagename, vwin=5)
```

The moments are calculated by masking pixels which are not (a) in the clean mask and (b) do not meet a significance cut on a Hanning smoothed cube. The moments are computed using the unsmoothed data.

1.3.9 Manually setting the RMS

By default the global RMS used for deriving thresholds is computed from the full dirty cube. For small windows where >50% of the channels may contain significant emission, this globally RMS value may not be appropriate. If the desired RMS value to use is known, the *faust_imaging.ImageConfig.rms* attribute may be set manually.

```
config = ImageConfig(...)
config.rms = 0.001  # in Jy
```

1.3.10 Frequency-chunked image processing

The memory requirements for imaging the full spectral windows using the `fullcube=True` are demanding, requiring several hundred gigabytes of RAM. In principle running `tclean` with the parameter `chankchunks=-1` serially processes individual frequency ranges to conserve memory, unfortunately problems persist. The most serious issues observed are that the final concatentation step in `tclean` can segfault, and that copying the internal mask files using `makemask` also frequently fails for large image cubes.

To relieve memory requirements, the pipeline may be run on individual frequency intervals or "chunks". By default the pipeline procedure *faust_imaging.ImageConfig.run_pipeline()* will chunk and concatenate the results:

```
config = ImageConfig(...)  # or `.from_name(...)`
config.run_pipeline(ext='clean')

# or explicitly
config.run_pipeline(ext='clean', use_chunking=True, nchunks=4)
```

The number of chunks can be controlled with the `nchunks` parameter. If left unset, then the number of chunks is chosen heuristically. The chunked configs may also be created from a normal instance using *faust_imaging.ImageConfig.duplicate_into_chunks()* and treated individually for more customized processing. This creates an instance of *faust_imaging.ChunkedConfigSet* which both encapsulates the chunked images and provides several helper methods.

```
# Initialize an image configuration instance with the desired properties.
full_config = ImageConfig(...)
# Create 4 chunks with properties inherited from the above, full instance.
# Note that if a ".sumwt" file does not exist, a dirty image will be
# made of a small field in order to calculate it first.
chunked_configs = full_config.duplicate_into_chunks(nchunks=4)
# Standard pipeline processing may now proceed on each chunked config.
for config in chunked_configs:
    config.run_pipeline(ext='clean')
# Post-process each chunked individually but using information from
# all of the runs (such as for determining the common beam)
chunked_configs.postprocess(ext='clean')
```

By default, most image extensions (e.g., '.image', etc.) are concatenated by *faust_imaging.ChunkedConfigSet.postprocess()*. Images may also be explicitly concatenated with *faust_imaging.ChunkedConfigSet.concat_cubes()*.

The pipeline procedures may also be run in different instances in CASA to process parts of the image in parallel. To do so, simply ensure that the same configuration options are applied in order to reproduce the equivalent `ImageConfig` instances, as below:

```
# In CASA instance 1, process chunks 0 and 1. Note that Python syntax for
# slicing (i.e., ":2") is inclusive on the lower limit but exclusive on
# the upper limit.
full_config = ImageConfig(...)
chunked_configs = full_config.duplicate_into_chunks(nchunks=4)
first_half = chunked_configs[:2]
for config in first_half:
    config.run_pipeline(ext='clean')

# In CASA instance 2, process chunks 2 and 3. Ensure that the same
# configuration options and modifications are also applied here as well!
full_config = ImageConfig(...)
chunked_configs = full_config.duplicate_into_chunks(nchunks=4)
second_half = chunked_configs[2:]
for config in second_half:
    config.run_pipeline(ext='clean')

# Now, in any CASA instance after the above two have finished running,
# merge the image products.
full_config = ImageConfig(...)
chunked_configs = full_config.duplicate_into_chunks(nchunks=4)
# Post-process each chunk separately and then concatenate the results.
chunked_configs.postprocess(ext='clean')
```

For specific example recipes, please refer to the *Parallel CASA* section.

1.3.11 Imaging Setup 3 SPWs with small chunk sizes

Due to the large field-of-view, small beam size, and large number of channels, imaging the Setup 3 SPWs poses a formidable data processing challenge (typical image dimensions of 3500x3500x1000). Individual Setup 3 cubes can also be about 10 to 40 times larger in file size (~40 to 200 GB for a single cube) which can cause significant issues in CASA usinng machines even with 500 GB RAM.

To effectively process these the Setup 3 SPWs, processing in small frequency interval "chunks" is required (see *Chunking*). Chunked images of only ~10 channels however may report biased image RMS values if emission is present over most channels (see *Set Rms*). Written below is a recipe for creating a few dirty cubes, calculating their RMS values, and then using that RMS value for all chunks.

```
# First calculate the frequency intervals of the chunks. For N2H+ with ~950
# channels and 100 chunks, a typical chunked image has 9 channels.
full_config = ImageConfig.from_name('CB68', '93.181GHz_N2Hp')
chunked_configs = full_config.duplicate_into_chunks(nchunks=100)
# Create dirty cubes and check the RMS values at the low-end, middle, and
# high-end of the band. The little helper function below is only for
# brevity. Note that the computing the RMS at the middle of the band is
# safe for N2H+ since the band center frequency is offset! This is not
# always the case.
def get_chunk_rms(config):
    config.make_dirty_cube()
    return config.rms
lo_rms = get_chunk_rms(chunked_configs[ 2])
md_rms = get_chunk_rms(chunked_configs[49])
```

(continues on next page)

(continued from previous page)

```
hi_rms = get_chunk_rms(chunked_configs[97])
# Check the RMS values here to see that they are pretty similar, if within
# ~10%, then it is reasonable to simply set it for all chunks.

# Now remake the config and run all chunks. By setting the RMS of the
# prototype instance it is propagated to all of the duplicate instances.
full_config = ImageConfig.from_name('CB68', '93.181GHz_N2Hp')
full_config.rms = md_rms  # or whatever
chunked_configs = full_config.duplicate_into_chunks(nchunks=100)
for config in chunked_configs:
    config.run_pipeline()
chunked_configs.postprocess(ext='clean')
```

Ozone lines are present near several SPWs that raise the RMS values appreciably (>30%) close to the band edge (e.g., "231.221GHz_13CS" and "231.322GHz_N2Dp"). For these SPWs setting a uniform RMS is not appropriate. It is straight forward however to increase the RMS for certain chunks or even set the value for each chunk individually using an interpolation function.

```
# Set the top 15 chunks (85-99) to have double the RMS.
rms = 0.003
full_config = ImageConfig.from_name('CB68', '93.181GHz_N2Hp')
full_config.rms = rms
chunked_configs = full_config.duplicate_into_chunks(nchunks=100)
for config in chunked_configs:
    if config.chunk.index > 84:
        config.rms = 2 * rms
    config.run_pipeline()
```

1.3.12 Restarting and manually cleaning a single chunk

Absorption and strong spatial filtering occassionally yield unsatisfactory results using the default heuristics of the pipeline. To manually restart and clean problematic chunks without re-running the full pipeline one can deconvolve and post-process chunks individually. For example, one can retrieve a specific image chunk by its channel number and then restart the deconvolution using interactive masking (as described in the *Restarting* section). The post-processing can be run individually for a chunk that is re-imaged without repeating the post-processing steps for all of the other chunks using the `use_existing_except` keyword parameter in *faust_imaging.ChunkedConfigSet.postprocess()*. The original concatenated products will then be replaced by new the ones containing the modified chunk(s).

```
# Create the configuration instances in the same way as they were set in the
# original run. For CB68 in CS (5-4), the default is 4 chunks:
full_config = ImageConfig.from_name('CB68', '244.936GHz_CS')
chunked_configs = full_config.duplicate_into_chunks()

# After reviewing the full, concatenated cubes of the pipeline run, it
# is found that channel index number 255 (i.e., indexed from 0, which is
# also the convention of the `casaviewer`) is insufficiently masked. For
# a cube with 477 channels in 4 chunks, this corresponds to chunk index 2.
# The chunk indices are also reflected in the image file names ("_chunk2_").
# Here we retrieve the configuration containing the desired channel index:
config_with_issues = chunked_configs.get_chunk_from_channel(255)
```

(continues on next page)

(continued from previous page)

```
# Clean the line interactively, restarting using the model and mask on disk.
config_with_issues.clean_line(ext='clean', interactive=True, restart=True,
        sigma=3)

# Remake the products for chunk2, but use the existing products from the
# other chunks. Concatenate all results together into new final cubes.
chunk_index = config_with_issues.chunk.index
chunked_configs.postprocess(ext='clean', use_existing_except=[chunk_index])
# Alternatively, just re-make everything.
#chunked_configs.postprocess(ext='clean')
```

The QA plots and final concatenated image products will report channel indices for the full cube and these differ from the channel indices of the chunks (as they start again from 0). To inspect or retrieve the corresponding channel indices from the full cube, one can access the *faust_imaging.ChunkConfig* class instance `faust_imaging.ImageConfig.chunk` to convert between channels of the full cube and channels of the chunk.

```
# Continuing from the previous example but renaming for brevity's sake:
config = config_with_issues

# For reference, print the chunk index number:
print(config.chunk.index)  # -> 2

# The channel index of interest is 255 in the full cube and corresponds
# to channel 17 in chunk 2. Note that these values are indexed from zero.
# One can calculate these values using the following methods:
print(config.chunk.convert_full_chan_to_chunk(255))  # -> 17
# Or do the reverse
print(config.chunk.convert_chunk_chan_to_full(17))   # -> 255

# A full list of which channels in the full cube the chunk channels
# correspond to can be found in:
print(config.chunk.fullcube_chan_indices)  # -> [238, 239, ..., 356]
```

1.3.13 Parallelized computation with multiple CASA processes

By default the pipeline will chunk the images and process each chunk in serial. Because this proceeds quickly for chunks that are mostly noise and reduces the gridding overhead on chunks with emission, this alone reduces the run-time. Helper scripts/templates are provided however to improve the performance further by running multiple instances of CASA simultaneously to process the image chunks in parallel. Example scripts may be found in the "pipe_scripts" directory in the pipeline GitHub repository. The shell files "run_pipe.sh" and "qsub_run_pipe.sh" may be lightly modified for job name and resources requested. The CASA Python files are tailored to specific example usages, such as parallelizing over SPWs (one SPW per job) or parallelizing over the chunks of a single SPW (multiple jobs per SPW). Recipes are included for:

- `run_pipe_cb68_setup1_continuum.py` Chunk the Setup 1 continuum SPW into 40 chunks (the default) and process batches of chunks in parallel. This considerably improves run-time performance compared to processing each chunk sequentially.
- `run_pipe_cb68_cs.py` Chunk the CS (5-4) SPW into 50 chunks (the default is 4) and process the chunks in parallel. This is useful for quickly processing one window that is well-characterized by a single RMS (i.e., not those near telluric lines).
- `run_pipe_cb68_all_setup1_cont_parallel.py` Process all of the narrow-band SPWs serially by chunk

but distribute the chunks of the continuum SPW among the CASA instances for parallel processing. This more efficiently balances the work-load among the CASA instances because the continuum SPW is substantially more computationally intensive to image. For 13 narrow-band SPWs and 8 CASA instances, each instance will process 1-2 narrow-band SPWs and ~5 chunks of the continuum SPW. This script is useful for processing a complete Setup in a straightforward way.

For correct usage with the given shell scripts, each Python script must implement the top-level functions `_preprocess()`, `_run_subset(<INDEX>)`, and `_postprocess()`. If they are not required, the `_preprocess` and `_postprocess` functions can simply execute a `pass` instruction for a no-op.

Personal machine

To run the pipeline in parallel on a user's personal machine or an interactive node on nmpost, first copy the "run_pipe.py" and "run_pipe.sh" files from the GitHub repository and place them in the directory specified by `PROD_DIR` (i.e., where you run CASA). The template files can be renamed to aid organization, simply ensure that the name of the Python script ("run_pipe.py") matches what is referenced in the shell script ("run_pipe.sh").

In the shell script, set the `NBATCHES` variable for the number of CASA processes/sessions used. From a limited number of experiments, dividing the number of available CPU cores by 2-3 provides reasonably good CPU utilization. The number of jobs set by this variable must be greater than or equal to the number of chunks or targets set in the Python script (each job must have at least one thing to process!).

Next in the Python script, modify the template function `_get_config` for the desired field, transition, and number of chunks. Global configuration settings can also be set here by modifying the attributes of `full_config` (such as the *faust_imaging.ImageConfig.rms* for example, or the auto-multithresh parameters). Note that the `_RUN_EXT` global variable may be changed to distinguish different runs, but intermediate products such as the "nomask" and "dirty" products will be over-written. For science cases that require per-chunk configuration, program logic may be added to `_run_subset`.

Finally, execute the shell script (`./run_pipe.sh`, say). Files for both the CASA log and the STDOUT terminal output will be created for each CASA process (one can monitor progress in real time with `tail -f file.log`).

If issues are discovered at certain frequencies/channels, such as divergences or insufficient masking, individual chunks may be restarted without re-running all of the others following the description in the *Restarting One Chunk* section.

Torque job submission

To run the pipeline in parallel using Torque on the NM or CV post-processing computing clusters ("nmpost"), copy the "run_pipe.py" and "qsub_run_pipe.sh" templates from the GitHub repository and place them in the `PROD_DIR` directory. Change the global variables as described above. The number of CPUs requested on the line:

```
#PBS -l nodes=1:ppn=16
```

should be more than the number of CASA instances/processes set by `NBATCHES`.

1.3.14 Applying a uv-taper

Source "IRAS_15398-3359" happens to have higher-than-requested angular resolution. To match the requested resolution (0.32 arcsec), we can apply a $1100\,\mathrm{k}\lambda$ *uv*-taper:

```
config = ImageConfig.from_name('IRAS_15398-3359', '244.936GHz_CS')
# Use a 1100 kilo-lambda (wavelength) circular taper.
config.uvtaper = ['1100klambda']
config.run_pipeline()
```

1.4 Developer documentation

This section is intended to given an overview of the pipeline script architecture. Contributions are welcome! Simply file a pull request on GitHub or post an issue for a bug or feature request.

The purpose of the pipeline script is to abstract away the details of imaging in CASA (calls to `tclean`, `immath`, etc.) and replace them with declarations to higher level pipeline functions specific to FAUST targets and spectral windows. Interferometric imaging, however, is a time-consuming and often error-prone process. Thus in practice the script is meant to be used as a toolkit that extends the functionality of an interactive session in CASA or a recipe to be run as a script or batch job. The *recipes* page details the usage of the *API*. Adding substantial new functionality, such as new pipeline tasks, will require extending and understanding the interfaces.

The pipeline code is organized mainly into the categories:

1. Global variables that contain default parameters.
2. Utility functions that operate directly on CASA image files. These functions are largely independent of project specific information.
3. Classes that encapsulate information specific to the FAUST project and provide methods for the execution of the main pipeline procedures. These are composed into the primary user facing class, *faust_imaging.ImageConfig*.
4. Utility functions that perform diagnostic tests, create moment maps, and quality assurance plots.

Extending the pipeline mainly requires understanding the four classes:

1. *faust_imaging.Target*: functionality specific to a FAUST target.
2. *faust_imaging.Spw*: functionality specific to a FAUST spectral window.
3. *faust_imaging.DataSet*: functionality specific to an ALMA configuration.
4. *faust_imaging.ImageConfig*: the composition of the above data classes that provides methods to perform the pipeline procedures and ultimately call `tclean`.

Adding new pipeline procedures is thus simply a matter of adding a new method to implement the functionality. For simple changes or changes intended to be new default behavior, `ImageConfig` should be directly modified with a new method. For experimental or significant changes that are not intended to be used as the default, `ImageConfig` should be sub-classed and extended, over-riding the *faust_imaging.ImageConfig.run_pipeline_tasks()* if necessary. The implementation of pipeline tasks should be factored such that operations on the "image level" are separated into distinct, small functions that are composed in the pipeline method/task. If new information or data is required that is not available in the existing data-classes (`Spw`, etc.), these should be extended to include it.

1.5 faust_imaging module

1.5.1 FAUST Archival Pipeline

This module contains the implementation for the CASA spectral line imaging pipeline for the archival products of the ALMA Large Program FAUST. Please see the README file for further information and the documentation, which may be found under the *docs/* directory or online at:

https://faust-imaging.readthedocs.io

This script may be run under CASA v5 (Python 2) or CASA v6 (Python 3). The current calibration pipeline release, CASA v5.6, is recommended as it has received the most extensive testing. To use the script interactively, execute the module by running at the CASA IPython prompt: `execfile('<PATH>/<TO>/faust_imaging.py')`.

Authors

Brian Svoboda and Claire Chandler.

class Target(*name*, *res*, *vsys*)

Bases: `object`

vlsr_from_window(*velo_width*)

Parameters
velo_width (*str*) – CASA quantity string, e.g. '10km/s'.

class DataSet(*field*, *setup=1*, *kind='joint'*)

Bases: `object`

Parameters

- **field** (*str*) – FAUST target field name.
- **setup** (*int*) –

  FAUST Setup index number:
  1 : Band 6, 220 GHz 2 : Band 6, 250 GHz 3 : Band 3, 90 GHz
- **kind** (*str*) – Datset descriptor for which measurement set files to use. Valid values include: ('joint', '12m', '7m'). Note that only 12m data is availabe for Setup 3.

low_freqs = {1: 217, 2: 245, 3: 93}

high_freqs = {1: 235, 2: 262, 3: 108}

property ms_fmt

property cell_12m

property cell_7m

property cell

property gridder

property pblimit

property imsize

Get the image size in pixels dynamically according to the lowest frequency SPW of the Setup and antenna diameter (i.e., 12m or 7m). The field size size is calculated as 10% larger than the full-width at the 10%-maximum point of the primary beam (approximated as $1.13 lambda / D$).

check_if_product_dirs_exist()

calc_res(*freq*, *diam*)

Note that no factor of 1.028 is applied for FWHM of Airy pattern, just l/D.

ms_contains_diameter(*ms_filen*, *diameter=None*, *epsilon=0.5*)

Determine whether an MS contains an antenna with a given diameter.

Parameters

- **diameter** (*number*) – Antenna diameter in meters.
- **epsilon** (*number, default 0.5*) – Fudge factor for querying the antennas min/max diameter.

class Spw(*setup*, *restfreq*, *mol_name*, *name*, *ms_restfreq*, *spw_id*, *ot_name*, *nchan*, *chan_width*, *tot_bw*)

Bases: object

Parameters

- **setup** (*int*) – Setup ID number (1, 2, 3)
- **restfreq** (*str*) – Rest frequency of primary targeted line in the SPW, e.g. "93.17GHz".
- **mol_name** (*str*) – Molecule name of the primary targeted line in the SPW.
- **name** (*str*) – Name of the spectral line.
- **ms_restfreq** (*str*) – Line rest frequency listed in the measurement set, e.g. "93.17GHz". Many of these values are not rest frequencies for specific molecular transitions but shifted values meant to center the bandpass.
- **spw_id** (*int*) – ID number of the spectral window
- **ot_name** (*str*) – Ending of the OT SPW label, e.g. "#BB_1#SW-01". These are useful for selecting the spectral window ID number if the correlator ID numbers are inconsistent across 7M/12M datasets.
- **nchan** (*int*) – Total number of channels (before flagging).
- **chan_width** (*number*) – Channel width in TOPO frame. **units**: kHz.
- **tot_bw** (*number*) – Total bandwidth in TOPO frame. **units**: MHz

property short_restfreq

property label

nchan_from_window(*velo_width*)

Parameters
velo_width (*str*) – Full velocity width of window. CASA quantity string, e.g. '10km/s'.

copy()

with_chunk(*chunk*)

Parameters
chunk (ChunkConfig, *None*) –

parse_spw_info_from_ms(*ms_filen*, *field='CB68'*, *out_filen=None*)

write_all_spw_info()

spw_list_to_dict(*spws*)

class ImageManager(*infile*, *cache=True*)

Bases: `object`

Safely open a CASA image by creating a context manager. The image tool and file will be closed even if an error occurs while manipulating the image tool. Note that multiple images can be opened in a nested sequence.

Parameters

infile (*str*) – CASA image file name, including any file suffixes.

Examples

The file is closed and the image tool destroyed when the scope of the context manager is exited. The following example shows how two images can be opened simulatenously in nested scopes.

```
>>> with ImageManager('foo.image') as tool1:
...     print(tool1.shape())
...     with ImageManager('bar.image') as tool2:
...         print(tool2.shape())
... print(tool1.shape())  # Error! The image tool is closed.
```

log_post(*msg*, *priority='INFO'*)

Post a message to the CASA logger, logfile, and stdout/console.

Parameters

- **msg** (*str*) – Message to post.
- **priority** (*str, default 'INFO'*) – Priority level. Includes 'INFO', 'WARN', 'SEVERE'.

check_delete_image_files(*imagename*, *parallel=False*, *preserve_mask=False*)

Check for and remove (if they exist) files created by clean such as '.flux', '.image', etc. NOTE this function has issues with deleting tables made by clean in parallel mode, probably because the directory structure is different.

Parameters

- **imagename** (*str*) – The relative path name to the files to delete.
- **parallel** (*bool, default False*) – rmtables can't remove casa-images made with parallel=True, they must be manually removed.
- **preserve_mask** (*bool, default False*) – Whether to preserve the *.mask* file extension

safely_remove_file(*filen*)

if_exists_remove(*filen*)

delete_all_extensions(*imagename*, *keep_exts=None*)

Parameters

- **imagename** (*str*) –
- **keep_exts** (*None, iterable*) – A list of extensions to keep, example: ['mask', 'psf']

delete_workdir(*imagename*)

replace_existing_file_with_new(*old_filen*, *new_filen*)

Replace an existing file with a new or temporary one. *old_filen* will be removed and replaced by *new_filen*.

remove_end_from_pathname(*filen*, *end='.image'*)

Retrieve the file name excluding an ending extension or terminating string.

Parameters

- **filen** (*str*) –
- **end** (*str*) – End to file name to remove. If an extension, include the period, e.g. ".image".

export_fits(*imagename*, *velocity=False*, *overwrite=True*)

concat_parallel_image(*imagename*)

Create a contiguous image cube file from a 'parallel image' generated from a multiprocess *tclean* run with *parallel=True*. The function replaces the original parallel image with the contiguous image.

concat_parallel_all_extensions(*imagebase*)

get_freq_axis_from_image(*imagename*)

Compute the frequency axis from an image's header coordinate system information.

Parameters

imagename (*str*) –

Returns

- *ndarray* – Frequency axis.
- *str* – Frequency axis unit (likely Hz).

calc_common_coverage_range(*imagebase*)

Calculate the common frequency coverage amongst a set of MSs/EBs. The *.sumwt* image extension is used from an existing "dirty" cube.

Parameters

imagebase (*str*) – Image name without the ".sumwt" extension.

Returns

- *str* – The start frequency in the LSRK frame.
- *int* – The number of channels from *start* at the native spectral resolution.

calc_chunk_freqs(*imagebase*, *nchunks=1*)

Divide the spectral axis of an image into continuous chunks in frequency. Returned frequencies specify the channel center frequency.

Parameters

- **imagebase** (*str*) – Image name base without the ".sumwt" extension.
- **nchunks** (*int*) – Number of contiguous chunks to divide the frequency range into.

Returns

(str) LSRK frequency of the bin left-edge of the first channel. **units**: Hz.

(int) Number of channels in the chunk.

Return type

[[str, int], . . .]

primary_beam_correct(*imagebase*)

hanning_smooth_image(*imagename*, *overwrite=True*)

Hanning smooth an image. Produces a new file with the extension ".hanning".

Parameters

- **filen** (*str*) –
- **overwrite** (*bool*) –

smooth_cube_to_common_beam(*imagename*, *beam=None*)

Use the `imsmooth` task to convert a cube with per-plane beams into one with a single common beam.

Parameters

- **imagename** (*str*) – Name of the image, including extension such as ".image".
- **beam** (*dict*) – Gaussian beam parameter dictionary (see `imsmooth` help).

copy_pb_mask(*imagename*, *pbimage*)

Copy primary beam T/F mask back into the image after being removed by a task such as `imsmooth`.

calc_rms_from_image(*imagename*, *chan_expr=None*)

Calculate RMS values from the scaled MAD of all channels (unless given a range) for the given full imagename.

Parameters

- **imagename** (*str*) – CASA Image name, e.g., "244.936GHz_CS_joint_0.5_dirty.image"
- **chan_expr** (*str, None*) – Channel selection expression passed to *imstat* parameter *chans*. If None, the full channel range is used.

Returns
RMS

Return type
float

create_mask_from_threshold(*infile*, *outfile*, *thresh*, *overwrite=True*)

Create a 1/0 mask according to whether pixel values in the input image are or are not above `thresh` in Jy.

Parameters

- **infile** (*str*) –
- **outfile** (*str*) –
- **thresh** (*number*) – **units**: Jy
- **overwrite** (*bool, default True*) –

effective_beamwidth_from_image(*imagename*)

Median of the geometric-mean beamwidths in an image with perplane beams. If the image has a single plane or single restoring beam, then the geometric mean of the major & minor axes is returned.

Parameters
imagename (*str*) –

make_multiscale_joint_mask(*imagename*, *rms*, *sigma=5.0*, *mask_ang_scales=(0, 1, 3)*)

Create a joint mask from multiple smoothed copies of an image. A file with the *.summask* extension is created from the union of an RMS threshold applied to each image. The RMS is re-computed for each smoothed cube.

Parameters

- **imagename** (*str*) – Image name base without the extension.
- **rms** (*number*) – RMS value of the unsmoothed image. **units**: Jy
- **sigma** (*number, Iterable(number)*) – Multiple of the RMS to threshold each image. If passed as a number, the same threshold is applied to all images (whether unsmoothed or smoothed). If passed as a list-like, then the sigma value is used for the image with the corresponding scale in *mask_ang_scales*.
- **mask_ang_scales** (*Iterable(number)*) – Gaussian kernel FWHM in units of arcseconds to convolve each image with (note: not in pixel units).

Returns

- *Files are written for each smoothing scale* – smoothed image: '<IMG>_smooth{.3f}.image' (excluding scale=0) masked image: '<IMG>_smooth{.3f}.mask'
- *and the joint or unioned mask file across scales is written to* – joint mask: '<IMG>.summask'

format_cube_robust(*weighting*)

format_rms(*rms*, *sigma=1*, *unit='Jy'*)

check_max_residual(*imagebase*, *sigma=5.5*)

Check the residual image cube for channels with deviations higher than a multiple of the global RMS.

Parameters

- **imagebase** (*str*) – Image name without extension.
- **sigma** (*number*) – Threshold to apply as a multiple of the RMS.

class ChunkConfig(*start*, *start_chan_ix*, *nchan*, *index*)

Bases: `object`

Parameters

- **start** (*str*) – CASA quantity string for the LSR frequency of the first channel of the chunk.
- **start_chan_ix** (*int*) – Channel index of the starting channel in the full cube.
- **nchan** (*int*) – Number of channels in the chunk.
- **index** (*int*) – Chunk index number, used as part of the file names: "_chunk0".

fullcube_chan_indices

Indices of the chunk channels in the full cube.

Type
[int], None

convert_chunk_chan_to_full(*ix*)

Parameters
ix (*int*) – Channel index of the chunked image cube.

convert_full_chan_to_chunk(*ix*)

Parameters
ix (*int*) – Channel index of the full image cube.

class ImageConfig(*dset*, *spw*, *fullcube=True*, *weighting=0.5*, *chunk=None*)

Bases: object

Configuration object for *tclean* related custom tasks. Parameters specify deconvolution and image-cube properties such as the weighting. See the docstring for *from_name()* for more complete parameter descriptions.

Parameters

- **dset** (DataSet) –
- **spw** (Spw) –
- **fullcube** (*bool*) –
- **weighting** (*str, number*) –
- **chunk** (ChunkConfig, *None*) –

scales

List of scales in pixels to be used for multi-scale clean. Will be passed to the scales keyword argument in *tclean*.

Type
Iterable

mask_ang_scales

FWHM sizes in arcsec for Gaussian smoothing kernels used in *faust_imaging.ImageConfig.clean_line_nomask()*.

Type
Iterable

autom_kwargs

Auto-multithresh keyword arguments passed to *tclean*.

Type
dict

smallscalebias

Type
number

gain

Type
number

cyclefactor

Type
number

uvtaper

Type
(list, None)

parallel

Is MPI enabled in CASA?

Type
bool

line_vwin

CASA quantity string for velocity window when *fullcube=False*.

Type
str

preserve_all_intermediate_products = False

smallscalebias = -1.0

gain = 0.05

cyclefactor = 2.0

uvtaper = None

parallel = False

line_vwin = '20km/s'

classmethod from_name(*field*, *label*, *kind='joint'*, ***kwargs*)

Create an *ImageConfig* object directly from the field name and SPW label without creating instances of auxillary class instances.

Parameters

- **field** (*str*) – Field name, e.g., "CB68"
- **label** (*str, None*) – SPW label, e.g., "216.113GHz_DCOp". If *None* then apply to all Setups and SPWs for target field.
- **kind** (*str*) – Datset descriptor for which measurement set files to use. Valid values include: ('joint', '12m', '7m').
- **fullcube** (*bool*) – Image the full spectral window or a small window around the SPWs defined rest frequency.
- **weighting** (*str, number*) – Either a string for a weighting method recognized by *tclean*, such as 'natural' or 'uniform'; or a number for a Briggs weighting robust value.
- **chunk** (ChunkConfig, *None*) – Chunk configuration instance for dividing the frequency axis into individual, smaller memory foot-print intervals in frequency. The default value of *None* will apply no chunking along the frequency axis.

property is_chunked

property dirty_imagebase

property nomask_imagebase

property tinyimg_imagebase

property rms

Image RMS. By default, determined using *imstat* over the entire dirty cube. The RMS can also be manually set.

property selected_start_nchan

property spw_ids

Determine spectral window ID numbers based on the (unique) Data Descriptor name found in the correlator configuration of the MS. This avoids indexing issues between mixed sets of 12m & 7m EBs.

duplicate_into_chunks(*nchunks=None*)

Duplicate the image configuration into multiple versions set to be chunked in frequency. This eases memory requirements because each smaller image cubes may be processed independently. A full-bandwidth ".sumwt" file must exist to compute the starting frequencies. If neither *faust_imaging.ImageConfig.make_dirty_cube()* nor *faust_imaging.ImageConfig.make_tiny_image_stamp()* have been executed, the latter will be run before calculating the chunk frequencies.

Parameters
nchunks (*int, None*) – Number of chunks to create. If unset then the number of chunks is chosen using a heuristic.

Returns
Configuration object encapsulating *ImageConfig* instances with properties set for chunking.

Return type
ChunkedConfigSet

get_imagebase(*ext=None*)

Get the relative path name for an image basename following

"images/<FIELD>/<LABEL>_<ARRAY>_<WEIGHTING>[_<EXT>]"

Note that this does not include image extensions such as ".image"

Parameters
ext (*str, Iterable, object*) – If *ext* is None, then compose path of setup, spw, array. If *ext* is an iterable, append each by underscores. If other object, must have a *__repr__* method for representation string.

mpicasa_cleanup(*imagename*)

Delete the ".workdirectory" folders that are occassionally generated and not removed in mpicasa. Also concatenate all image products into contiguous "serial" products using *ia.imageconcat*.

remove_all_files(*confirm=True*, *remove_chunks=True*)

Attempt to safely remove all files matching the image basename returned by *faust_imaging.ImageConfig.get_imagebase()*.

Parameters
confirm (*bool*) – Print the files to be removed to STDOUT and require keyboard confirmation before removing files.

make_tiny_image_stamp()

Image a small field in order to produce the ".sumwt" file without producing a full set of dirty image products (which may be very large). The default suffix set by `TINYIMG_EXT` is "_tinyimg".

make_dirty_cube()

Generate a dirty image cube and associated *tclean* products. Run with equivalent parameters as *clean_line* but with *niter=0*. The default suffix set by `DIRTY_EXT` is "_dirty".

clean_line_nomask(*sigma=4.5*, *scale_upper_limit=60*)

Deconvolve the image without using a mask to a depth of *sigma* and excluding multiscale size scales beyond *scale_upper_limit*.

Parameters

- **sigma** (*number*) – Global threshold to clean down to computed as multiple of the full-cube RMS.
- **scale_upper_limit** (*number, None*) – Restrict the scales (in pixels) used by multi-scale clean to values less than this limit. If *None*, use all scales.

make_seed_mask(*sigma=5.0*)

Create a mask based on a significance cut applied to the restored image of the unmasked run generated from *clean_line_nomask*. This mask is used to "seed" the mask generated using auto-multithresh in *clean_line* with *mask_method='seed+multithresh'*.

Parameters

sigma (*number*) – Limit to threhsold the restored image on.

clean_line(*mask_method='seed+multithresh'*, *sigma=3.0*, *ext=None*, *restart=False*, *interactive=False*)

Primary interface for calling *tclean* to deconvolve spectral windows. Multiple masking methods exist to automatically generate the clean masks used. The preferred method is *mask_method='seed+multithresh'*, which requires that *.clean_line_nomask* is run first.

Parameters

- **mask_method** (*str*) – Masking method to use in the deconvolution process. Available methods

  `"auto-multithresh"` use auto-multithresh automated masking.

  `"seed+multithresh"` generate initial mask from free clean and then use auto-multithresh for automatic masking.

  `"fixed"` use the existing clean mask but keep it fixed and do not apply auto-multithresh.

  `"taper"` use mask generated from a separate tapered run. Generating the tapared masks requires re-implementation.

- **sigma** (*number*) – Threshold in standard deviations of the noise to clean down to within the clean-mask. An absolute RMS is calculated from off-line channels in a dirty cube. The same value is applied to all channels.
- **ext** (*str*) – String of the form '_EXT' appended to the end of the image name.
- **restart** (*bool, default False*) – Restart using the existing model and mask files.
- **interactive** (*bool, default False*) – Begin tclean in interactive mode with *interactive=True*. This may be useful for touching-up some channels with particularly difficult to clean extended emission.

clean_line_interactive_restart(***kwargs*)

postprocess(*ext=None*, *make_fits=True*, *make_hanning=False*, *beam=None*)

Post-process image products. The maximum residual is logged to the CASA log file and STDOUT, the image is primary beam corrected, smoothed to a common beam, and the final FITS file is exported.

Parameters

- **ext** (*str*) –
- **make_fits** (*bool*) –
- **make_hanning** (*bool*) –
- **beam** (*dict, None*) – Gaussian beam parameter dictionary to use for common beam smoothing. If left as None, then the common beam of the image is used.

run_pipeline_tasks(*ext='clean'*, *nomask_sigma=4.5*, *seedmask_sigma=5.0*, *clean_sigma=3.0*)

Helper method to run the pipeline tasks in sequence with reasonable default parameters. See *run_pipeline()* for further description.

Parameters

- **ext** (*str*) –
- **nomask_sigma** (*number*) –
- **seedmask_sigma** (*number, Iterable(number)*) –
- **clean_sigma** (*number*) –

run_pipeline(*ext='clean'*, *use_chunking=True*, *nchunks=None*, *nomask_sigma=4.5*, *seedmask_sigma=5.0*, *clean_sigma=3.0*)

Run all pipeline tasks with default parameters. Custom recipes should call the individual methods in sequence. The final pipeline product will be named of the form:

"<FILENAME>_<EXT>.image.pbcor.common.fits"

or if `ext=None`:

"<FILENAME>.image.pbcor.common.fits"

For convenience, some significance thresholds ("sigma") may also be set for this method. For more customized pipeline procedures, calling the individual pipeline methods is recommended.

Parameters

- **ext** (*str*) – Extension name for final, deconvolved image products.
- **use_chunking** (*bool*) – If True then chunk the image into separate frequency intervals, process individually in serial, and then concatenate the final results.
- **nchunks** (*int, None*) – Number of approximately uniform frequency intervals to chunk image products into. If *None* then the number of chunks is chosen by heuristic.
- **nomask_sigma** (*number*) – Global clean threshold for the un-masked clean run.
- **seedmask_sigma** (*number, Iterable(number)*) – Significance with which to threshold the un-masked clean run to create the seed mask. If a list-like is passed, the corresponding significance is used for each smoothing scale (scales are set by *mask_ang_scales*).
- **clean_sigma** (*number*) – Global clean threshold of the final clean run.

class ChunkedConfigSet(*configs*)

Bases: `object`

Parameters

configs (*Iterable(ImageConfig)*) – List of chunked ImageConfig instances, such as generated by *ImageConfig.duplicate_into_chunks()*.

configs

Type
Iterable(*ImageConfig*)

nchunks

Type
int

nchan_total

Total number of channels summing across all chunks.

Type
int

get_imagebase(*ext=None*)

remove_all_files(*confirm=True*)

get_chunk_from_channel(*ix*)

Parameters
ix (*int*) – Channel index number in the full, concatenated cube.

Return type
ImageConfig

get_common_beam(*ext='clean'*)

Calculate the common beam (largest area) among all chunked images.

Parameters
ext (*str*) – Extension name suffix, e.g. 'clean', 'nomask', etc.

Returns
Beam parameters formatted as strings with unit labels: major axis (arcsec), minor axis (arcsec), position angle (deg).

Return type
(str, str, str)

Notes

A fudge factor of 0.3% is added to the beam size to account for discrepencies in how the common beam is calculated by CASA over the full cube compared to the largest-area common beam among the chunks. For typical ~0.35as HPBWs in the FAUST program, this multiplicatie factor corresponds to ~1 mas.

concat_cubes(*ext='clean'*, *im_exts=None*)

Use `ia.imageconcat` to concatenate chunked image files into single a cube with contiguous data.

Parameters

- **ext** (*str*) – Extension name suffix, e.g. 'clean', 'nomask', etc.
- **im_exts** (*Iterable(str), None*) – Names of image extensions, e.g. 'mask', 'image.pbcor.common', etc. If *None* then a default set of extensions is processed. The default extensions are "image", "mask", "model", "pb", "psf", "residual".

postprocess(*ext='clean'*, *use_existing_except=None*, *make_hanning=False*)

Parameters

- **ext** (*str*) –
- **use_existing_except** (*Iterable(int), None*) – If post-processing has already been run, but a few chunks have been modified (by manual cleaning for example), then only reprocess the chunks set by this variable. If left unset (None), then all chunks are processed.
- **make_hanning** (*bool, default False*) – Create the Hanning smoothed cube.

make_all_line_dirty_cubes(*dset*, *weighting=0.5*, *fullcube=True*)

postproc_all_cleaned_images(*dset*)

run_pipeline(*field*, *setup=None*, *weightings=None*, *fullcube=True*, *do_cont=True*, *chunked=True*)

Run all pipeline tasks over all setups, spectral windows, and uv-weightings.

Parameters

- **field** (*str*) –

- **setup** (*int, None*) – Setup number, if *None* run on all setups in sequence.
- **weightings** (*Iterable, default (0.5,)*) – List of uv-weightings to use in *tclean*, which may include the string "natural" or a number for the briggs robust parameter.
- **fullcube** (*bool, default True*) – Image the full spectral window or a small window around the SPWs defined rest frequency.
- **do_cont** (*bool, default True*) – Whether to image the continuum SPWs or not.
- **chunked** (*bool, default True*) – Process frequency intervals of each SPW in serial to reduce memory requirements.

test_perchanwt()

test_rename_oldfiles(*field*, *label=None*, *kind='joint'*, *weighting=0.5*)

Replace the rest frequency in each final image cube with the value given by Satoshi and then re-export the FITS cube.

Parameters

- **field** (*str*) – Field name, e.g., "CB68"
- **label** (*str, None*) – SPW label, e.g., "216.113GHz_DCOp". If *None* then apply to all Setups and SPWs for target field.
- **kind** (*str*) –
- **weighting** (*str, number*) –

class MomentMapper(*imagename*, *vwin=5*, *m1_sigma=4*, *m2_sigma=5*, *overwrite=True*)

Bases: object

Parameters

- **imagename** (*str*) – CASA image filename ending in '.image.common'.
- **vwin** (*number*) – Velocity window (half-width) to use for estimating moments over relative to the systemic velocity. **units**: km/s
- **m1_sigma** (*number*) – Multiple of the RMS to threshold the Moment 1 data on.
- **m2_sigma** (*number*) – Multiple of the RMS to threshold the Moment 2 data on.
- **overwrite** (*bool, default True*) – Overwrite moment maps and associated files if they exist.

get_cube_velocity_axis(*as_region_subim=False*, *with_image_axes=False*)

Velocity axis in units of km/s derived from the ".image.common" image file.

Parameters

- **as_region_subim** (*bool, default False*) – Compute velocity axis only over sub-image defined by the velocity window region.
- **with_image_axes** (*bool, default False*) – Reshape to use the same dimensions as the image cube (Stokes, RA, Dec, Chan). *False* returns as 1D.

get_top_right_corner_pixels()

get_region()

Create a full-field sub-cube restricted to a window in velocity about the source systemic velocity.

get_vlsr_pb_plane()

Retrieve the primary beam near the center of the velocity window and thus the source systemic velocity.

get_rms_map(*with_pb_atten=False*)

make_max()

Make the peak intensity map (maximum). Use all emission within the velocity window.

make_mom0()

Make the integrated intensity map (moment 0). Use all emission within the velocity window that is also in the CLEAN mask. Expressions for the moment calculations can be found in the help documentation to the `ia.moments` toolkit function.

$$M_0 = \Delta v I_i$$

make_mom0_err()

Make the error map of the integrated intensity (moment 0).

$$\delta M_0 = \Delta v \sqrt{N} \sigma_{\rm rms}$$

make_mom1()

Make the intensity weighted mean velocity map (moment 1). Use all emission within the velocity window that is within the CLEAN mask and also above a significance threshold in the Hanning smoothed cube.

$$M_1 =$$

rac{sum I_i v_i}{M_0}

make_mom1_err()

Make the error map of the intensity weighted mean velocity (moment 1).

rac{sigma_mathrm{rms} sum v_i^2}{M_1 M_0} ight)^2 + left(rac{delta M_0}{M_0} ight)^2}

make_mom2()

Make the intensity weighted mean velocity dispersion map (moment 2). Use all emission within the velocity window that is within the CLEAN mask and also above a significance threshold in the Hanning smoothed cube.

rac{sum I_i (v_i - M_1)^2}{M_0}}

make_moments(*remove_hanning=True*)

Create moment maps for the given image. The cube is masked using the associated clean mask.

Parameters

remove_hanning (*bool, default True*) – Delete the hanning-smoothed cube after use.

make_moments_from_image(*path*, *vwin=None*, *overwrite=True*)

make_all_moment_maps(*field*, *ext='clean'*, *vwin=None*, *ignore_chunks=True*, *overwrite=True*)

Generate all moment maps for images with the matching field ID name and extension. Moment maps will be written to the directory set in MOMA_DIR.

Parameters

- **field** (*str*) – Target field ID name
- **ext** (*str*) – Image extension name, such as 'clean', 'nomask', etc.
- **vwin** (*number, None*) – Velocity half-window to use for estimating moments over relative to the source systemic velocity (see value specified in ALL_TARGETS). If set to *None*, then a default half-window of 5km/s is used for all lines, except CCH, where a half-window of 1.25km/s used to accomodate the 2.5km/s spacing of the hyperfine satellite lines. If explicitly set, the half-window is used for all lines. **units**: km/s
- **ignore_chunks** (*bool*) – If 'True' ignore chunk image files.
- **overwrite** (*bool, default True*) – Overwrite moment maps files if they exist.

savefig(*filen*, *dpi=300*, *plot_exts=('png', 'pdf')*, *close=True*)

Save the figure instance to a file.

Parameters

- **filen** (*str*) – File name to write to, excluding the extension (e.g., ".pdf").
- **dpi** (*number*) –
- **plot_exts** (*Iterable*) – Extensions to create plots for, e.g., "pdf", "svg", "png", "eps", "ps".
- **close** (*bool*) – Close figure instance when finished plotting.

class CubeSet(*path*, *sigma=6*)

Bases: object

Parameters

- **path** (*str*) – Full image path, including extension.
- **sigma** (*number*) – Significance threshold used to select planes/channels to retrieve.

static get_chunk(*imagename*)

get_plane_from_image(*filen*, *ix*)

Parameters

- **filen** (*str*) – CASA image path, e.g. "images/CB68/CB68_<...>_clean.mask"
- **ix** (*int*) – Channel index number (zero indexed).

Returns

2D image plane of the selected channel.

Return type

ndarray

get_image_planes(*ix*)

iter_planes()

get_good_channels()

calc_tick_loc(*ang_tick=5*)

Parameters

ang_tick (*number*) – Tick interval in arcsec.

make_qa_plot(*cset*, *kind='image'*, *outfilen='qa_plot'*)

Generate Quality Assurance plots by visualizing each channel where significant emission occurs in the image cube of interest (i.e., *.image* or *.residual*).

Parameters

- **cset** (CubeSet) –

- **kind** (*str*) – Kind of image data to plot:

  'image' : restored/cleaned image

  'residual' : residual image

- **outfilen** (*str*) –

make_qa_plots_from_image(*path*, *plot_sigma=None*, *overwrite=True*)

Create a single set of Quality Assurance plots. Plots will be written to the directory specified in PLOT_DIR.

Parameters

- **path** (*str*) – Full path to imagename, including the ending ".image".
- **plot_sigma** (*number, None*) – Significance threshold to select a channel for plotting. If None, use the default set by *CubeSet*.
- **overwrite** (*bool*) – Overwrite plot files if they exist.

make_all_qa_plots(*field*, *ext='clean'*, *ignore_chunks=True*, *overwrite=True*)

Generate all Quality Assurance plots for all image cubes matching the given field ID name and extension. Plots will be written to the directory set in PLOT_DIR.

Parameters

- **field** (*str*) – Target field ID name
- **ext** (*str*) – Image extension name, such as 'clean', 'nomask', etc.
- **ignore_chunks** (*bool*) – If 'True' ignore chunk image files.
- **overwrite** (*bool, default True*) – Overwrite plot files if they exist. Setting to False will avoid the potentially large run-time cost of reading cubes into memory to re-make existing plots.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX