stcal

Release 1.18.1.dev24+g4a4b38c72

STScI

Jun 11, 2026

CONTENTS

1 Package Documentation **1**

1.1 Package Index 1

2 Change Log **53**

2.1 Change Log 53

3 Indices and tables **73**

Python Module Index **75**

Index **77**

CHAPTER ONE

PACKAGE DOCUMENTATION

1.1 Package Index

1.1.1 Jump Detection

Algorithm

This routine detects jumps by looking for outliers in the up-the-ramp signal for each pixel. On output, the GROUPDQ array is updated with the DQ flag "JUMP_DET" to indicate the location of each jump that was found. In addition, any pixels that have non-positive or NaN values in the gain reference file will have DQ flags "NO_GAIN_VALUE" and "DO_NOT_USE" set in the output PIXELDQ array. The SCI array of the input data is not modified.

Jumps can be detected in two different ways. The primary way is the two-point difference method described below. The other way is by selecting `only_use_ints` as `True` and if there are enough integrations, then `sigma_clip` from the `astropy.stats` package will be used to detect jumps. The `sigma_clip` method will also be used if the total number of usable groups (number of groups per integration multiplied by the number of integrations) is above a minimum threshold.

The current implementation uses the two-point difference method described in Anderson & Gordon (2011).

Two-Point Difference Method

The two-point difference method is applied to each integration as follows:

1. Compute the first differences for each pixel (the difference between adjacent groups)
2. Compute the clipped median (dropping the largest difference) of the first differences for each pixel. If there are only three first difference values (four groups), no clipping is performed when computing the median.
3. Use the median to estimate the Poisson noise for each group and combine it with the read noise to arrive at an estimate of the total expected noise for each difference.
4. Compute the "difference ratio" as the difference between the first differences of each group and the median, divided by the expected noise.
5. If the largest "difference ratio" is greater than the rejection threshold, flag the group corresponding to that ratio as having a jump.
6. If a jump is found in a given pixel, iterate the above steps with the jump-impacted group excluded, looking for additional lower-level jumps that still exceed the rejection threshold.
7. Stop iterating on a given pixel when no new jumps are found or only one difference remains.
8. If there are only two differences (three groups), the smallest one is compared to the larger one and if the larger one is above a threshold, it is flagged as a jump.
9. If flagging of the 4 neighbors is requested, then the 4 adjacent pixels will have ramp jumps flagged in the same group as the central pixel as long as it has a jump between the min and max requested levels for this option.

10. If flagging of groups after a ramp jump is requested, then the groups in the requested time since a detected ramp jump will be flagged as ramp jumps if the ramp jump is above the requested threshold. Two thresholds and times are possible for this option.

Note that any ramp groups flagged as SATURATED in the input GROUPDQ array are not used in any of the above calculations and hence will never be marked as containing a jump.

1.1.2 Ramp Fitting

Description

This step determines the mean count rate, in units of counts per second, for each pixel by performing a linear fit to the data in the input file. The default method uses "ordinary least squares" based on the Fixsen fitting algorithm described by Fixsen et al. (2011).

The count rate for each pixel is determined by a linear fit to the cosmic-ray-free and saturation-free ramp intervals for each pixel; hereafter this interval will be referred to as a "segment."

Segments are determined using the 4-D GROUPDQ array of the input data set, under the assumption that the jump step will have already flagged CR's. Segments are terminated where saturation flags are found. Pixels are processed simultaneously in blocks using the array-based functionality of numpy. The size of the block depends on the image size and the number of groups.

There is also a new algorithm available for testing, the likelihood algorithm, implementing an algorithm based on the group differences of a ramp. See *likelihood algorithm*.

Output Products

There are two output products created by default, with a third optional product also available:

1. The primary output file ("rate") contains slope and variance/error estimates for each pixel that are the result of averaging over all integrations in the exposure. This is a product with 2-D data arrays.
2. The secondary product ("rateints") contains slope and variance/error estimates for each pixel on a per-integration basis, stored as 3-D data cubes.
3. The third, optional, output product contains detailed fit information for every ramp segment for each pixel.

RATE Product

After computing the slopes and variances for all segments for a given pixel, the final slope is determined as a weighted average from all segments in all integrations, and is written to the "rate" output product. In this output product, the 4-D GROUPDQ from all integrations is collapsed into 2-D, merged (using a bitwise OR) with the input 2-D PIXELDQ, and stored as a 2-D DQ array. The 3-D VAR_POISSON and VAR_RNOISE arrays from all integrations are averaged into corresponding 2-D output arrays. In cases where the median rate for a pixel is negative, the VAR_POISSON is set to zero, in order to avoid the unphysical situation of having a negative variance.

RATEINTS Product

The slope images for each integration are stored as a data cube in "rateints" output data product. Each plane of the 3-D SCI, ERR, DQ, VAR_POISSON, and VAR_RNOISE arrays in this product corresponds to the result for a given integration. In this output product, the GROUPDQ data for a given integration is collapsed into 2-D and then merged with the input 2-D PIXELDQ array to create the output DQ array for each integration. The 3-D VAR_POISSON and VAR_RNOISE arrays are calculated by averaging over the fit segments in the corresponding 4-D variance arrays.

FITOPT Product

A third, optional output product is also available and is produced only when the step parameter `save_opt` is True (the default is False). This optional product contains 4-D arrays called SLOPE, SIGSLOPE, YINT, SIGYINT, WEIGHTS, VAR_POISSON, and VAR_RNOISE, which contain the slopes, uncertainties in the slopes, y-intercept, uncertainty in the y-intercept, fitting weights, variance of the slope due to poisson noise, and the variance of the slope due to read noise for each segment of each pixel, respectively. The y-intercept refers to the result of the fit at an effective exposure time of zero. This product also contains a 3-D array called PEDESTAL, which gives the signal at zero exposure time for each pixel, and the 4-D CRMAG array, which contains the magnitude of each group that was flagged as having a CR hit.

By default, the name of this output file will have the product type suffix "_fitopt". In this optional output product, the pedestal array is calculated for each integration by extrapolating the final slope (the weighted average of the slopes of all ramp segments in the integration) for each pixel from its value at the first group to an exposure time of zero. Any pixel that is saturated on the first group is given a pedestal value of 0. Before compression, the cosmic-ray magnitude array is equivalent to the input SCI array but with the only nonzero values being those whose pixel locations are flagged in the input GROUPDQ as cosmic ray hits. The array is compressed, removing all groups in which all the values are 0 for pixels having at least one group with a non-zero magnitude. The order of the cosmic rays within the ramp is preserved.

Special Cases

If the input dataset has only one group in each integration (NGROUPS=1), the count rate for all unsaturated pixels in each integration will be calculated as the value of the science data in the one group divided by the group time. If the input dataset has only two groups per integration (NGROUPS=2), the count rate for all unsaturated pixels in each integration will be calculated using the differences between the two valid groups of the science data divided by the group time.

For datasets having more than one group in each integration (NGROUPS>1), a ramp having a segment with only one good group is processed differently depending on the number and size of the other segments in the ramp. If a ramp has only one segment and that segment contains a single group, the count rate will be calculated to be the value of the science data in that group divided by the group time. If a ramp has a segment with only one good group, and at least one other segment having more than one good group, only data from the segment(s) having more than one good group will be used to calculate the count rate.

For ramps in a given integration that are saturated beginning in their second group, the count rate for that integration will be calculated as the value of the science data in the first group divided by the group time, but only if the step parameter `suppress_one_group` is set to `False`. If set to `True`, the computation of slopes for pixels that have only one good group will be suppressed and the slope for that integration will be set to zero.

Slope and Variance Calculations

Slopes and their variances are calculated for each segment, for each integration, and for the entire exposure. As defined above, a segment is a set of contiguous groups where none of the groups is saturated or cosmic ray-affected. The appropriate slopes and variances are output to the primary output product, the integration-specific output product, and the optional output product. The following is a description of these computations. The notation in the equations is the following: the type of noise (when appropriate) will appear as the superscript 'R', 'P', or 'C' for readnoise, Poisson noise, or combined, respectively; and the form of the data will appear as the subscript: 's', 'i', 'o' for segment, integration, or overall (for the entire dataset), respectively.

It is possible for an integration or pixel to have invalid data, so usable slope data will not be available. If a pixel has an invalid integration, the value for that integration for that pixel will be set to NaN in the rateints product. Further, if all integrations for a given pixel are invalid the pixel value for the rate product will be set to NaN. An example of invalid data would be a fully saturated integration for a pixel.

Optimal Weighting Algorithm

The slope of each segment is calculated using the least-squares method with optimal weighting, as described by Fixsen et al 2000 and Regan 2007, JWST-STScI-001212. Optimal weighting determines the relative weighting of each sample when calculating the least-squares fit to the ramp. When the data have low signal-to-noise ratio S, the data are read noise dominated and equal weighting of samples is the best approach. In the high signal-to-noise regime, data are Poisson-noise dominated and the least-squares fit is calculated with the first and last samples. In most practical cases, the data will fall somewhere in between, where the weighting is scaled between the two extremes.

For segment k of length n, which includes groups $[g_k, ..., g_{k+n-1}]$, the signal-to-noise ratio S used for weighting selection is calculated from the last sample as:

$$S = \frac{data \times gain}{\sqrt{(read_noise)^2 + (data \times gain)}} \,,$$

where $data = g_{k+n-1} - g_k$.

The weighting for a sample i is given as:

$$w_i = \frac{[(i - i_{midpoint})/i_{midpoint}]^P}{(read_noise)^2} \,,$$

where $i_{midpoint} = \frac{n-1}{2}$ and $i = 0, 1, ..., n-1$.

is the the sample number of the midpoint of the sequence, and P is the exponent applied to weights, determined by the value of S. Fixsen et al. 2000 found that defining a small number of P values to apply to values of S was sufficient; they are given as:

Minimum S	Maximum S	P
0	5	0
5	10	0.4
10	20	1
20	50	3
50	100	6
100		10

Segment-specific Computations

The variance of the slope of a segment due to read noise is:

$$var^R_s = \frac{12 \; R^2}{(ngroups_s^3 - ngroups_s)(tgroup^2)(gain^2)} \,,$$

where R is the noise in the difference between 2 frames, $ngroups_s$ is the number of groups in the segment, and $tgroup$ is the group time in seconds (from the keyword TGROUP). The divide by gain converts to DN. For the special case where as segment has length 1, the $ngroups_s$ is set to 2.

The variance of the slope in a segment due to Poisson noise is:

$$var^P_s = \frac{slope_{est} + darkcurrent}{tgroup \times gain \; (ngroups_s - 1)} \,,$$

where $gain$ is the gain for the pixel (from the GAIN reference file), in e/DN. The $slope_{est}$ is an overall estimated slope of the pixel, calculated by taking the median of the first differences of the groups that are unaffected by saturation and cosmic rays, in all integrations. This is a more robust estimate of the slope than the segment-specific slope, which may be noisy for short segments. The contributions from the dark current are added when present; the value can be provided

by the user during the *jwst.dark_current.DarkCurrentStep*, or it can be specified in scalar or 2D array form by the dark reference file.

The combined variance of the slope of a segment is the sum of the variances:

$$var^C_s = var^R_s + var^P_s$$

Integration-specific computations

The variance of the slope for an integration due to read noise is:

$$var^R_i = \frac{1}{\sum_s \frac{1}{var^R_s}},$$

where the sum is over all segments in the integration.

The variance of the slope for an integration due to Poisson noise is:

$$var^P_i = \frac{1}{\sum_s \frac{1}{var^P_s}}$$

The combined variance of the slope for an integration due to both Poisson and read noise is:

$$var^C_i = \frac{1}{\sum_s \frac{1}{var^R_s + var^P_s}}$$

The slope for an integration depends on the slope and the combined variance of each segment's slope:

$$slope_i = \frac{\sum_s \frac{slope_s}{var^C_s}}{\sum_s \frac{1}{var^C_s}}$$

Exposure-level computations

The variance of the slope due to read noise depends on a sum over all integrations:

$$var^R_o = \frac{1}{\sum_i \frac{1}{var^R_i}}$$

The variance of the slope due to Poisson noise is:

$$var^P_o = \frac{1}{\sum_i \frac{1}{var^P_i}}$$

The combined variance of the slope is the sum of the variances:

$$var^C_o = var^R_o + var^P_o$$

The square-root of the combined variance is stored in the ERR array of the output product.

The overall slope depends on the slope and the combined variance of the slope of each integration's segments, so is a sum over integration values computed from the segments:

$$slope_o = \frac{\sum_i \frac{slope_i}{var^C_i}}{\sum_i \frac{1}{var^C_i}}$$

Error Propagation

Error propagation in the `ramp_fitting` step is implemented by carrying along the individual variances in the slope due to Poisson noise and read noise at all levels of calculations. The total error estimate at each level is computed as the square-root of the sum of the two variance estimates.

In each type of output product generated by the step, the variance in the slope due to Poisson noise is stored in the "VAR_POISSON" extension, the variance in the slope due to read noise is stored in the "VAR_RNOISE" extension, and the total error is stored in the "ERR" extension. In the optional output product, these arrays contain information for every segment used in the fitting for each pixel. In the "rateints" product they contain values for each integration, and in the "rate" product they contain values for the exposure as a whole.

Data Quality Propagation

For a given pixel, if all groups in an integration are flagged as DO_NOT_USE, then that pixel will be flagged as DO_NOT_USE in the corresponding integration in the "rateints" product. Note this does NOT mean that all groups are flagged as DO_NOT_USE. For example, slope calculations that are suppressed due to a ramp containing only one good group will be flagged as DO_NOT_USE in the first group, but not necessarily any other group, while only groups two and beyond are flagged as SATURATED. Further, only if all integrations in the "rateints" product are flagged as DO_NOT_USE, then the pixel will be flagged as DO_NOT_USE in the "rate" product.

For a given pixel, if any groups in an integration are flagged as SATURATED, then that pixel will be flagged as SATURATED in the corresponding integration in the "rateints" product. Furthermore, if any integration in the "rateints" product is flagged as SATURATED, then the pixel will be flagged as SATURATED in the "rate" product.

For a given pixel, if any group in an integration is flagged as JUMP_DET, then that pixel will be flagged as JUMP_DET in the corresponding integration in the "rateints" product. That pixel will also be flagged as JUMP_DET in the "rate" product.

Likelihood Algorithm Details

As an alternative to the OLS algorithm, a likelihood algorithm can be selected with the step argument `--ramp_fitting.algorithm=LIKELY`. This algorithm has its own algorithm for jump detection that augments anything identified by the regular jump detection step. The LIKELY algorithm requires a minimum of four (4) NGROUPS. If the LIKELY algorithm is selected for data with NGROUPS less than four, the ramp fitting algorithm is changed to OLS_C.

Each pixel is independently processed, but rather than operate on each group/resultant directly, the likelihood algorithm is based on differences of the groups/resultants $d_i = r_i - r_{i-1}$. The model used to determine the slope/countrate, a, is:

$$\chi^2 = (\mathbf{d} - a \cdot \mathbf{1})^T C^{-1} (\mathbf{d} - a \cdot \mathbf{1}) \,,$$

Differentiating, setting to zero, then solving for a results in

$$a = (\mathbf{1}^T C^{-1} \mathbf{d})(\mathbf{1}^T C^{-1} \mathbf{1})^{-1} \,,$$

The covariance matrix C is a tridiagonal matrix, due to the nature of the differences. Because the covariance matrix is tridiagonal, the computational complexity reduces from $O(n^3)$ to $O(n)$. To see the detailed derivation and computations implemented, refer to the links above. The Poisson and read noise computations are based on equations (27) and (28), in Brandt (2024).

For more details, especially for the jump detection portion in the likelihood algorithm, see Brandt (2024).

1.1.3 Alignment Utils

Description

This sub-package contains all the modules common to all missions.

WCS Info Dictionary

Many of the functions in this submodule require a `wcsinfo` dictionary. This dictionary contains information about the spacecraft pointing, and requires at least the following keys:

- `'ra_ref'`: The right ascension at the reference point in degrees.
- `'dec_ref'`: The declination at the reference point in degrees.
- `'v2_ref'`: The V2 reference point in arcseconds, with `'v3_ref'` maps to `'ra_ref'` and `'dec_ref'`.
- `'v3_ref'`: The V3 reference point in arcseconds, with `'v2_ref'` maps to `'ra_ref'` and `'dec_ref'`.
- `'roll_ref'`: Local roll angle associated with each aperture in degrees.
- `'v3yangle'`: The angle between V3 and North in degrees.
- `'vparity'`: The "V-parity" of the observation, which is 1 if the V3 axis is parallel to the detector Y-axis, and -1 if the V3 axis is parallel to the detector X-axis.

stcal.alignment Package

Functions

calc_rotation_matrix(roll_ref, v3i_yangle[, ...])	Calculate the rotation matrix.
combine_footprints(footprints)	Combine a list of footprints into one or more combined footprints using a Shapely union.
combine_sregions(sregion_list, det2world[, ...])	Combine s_regions from input models to compute the s_region for the resampled data.
compute_fiducial(wcslist[, bounding_box])	Calculate the world coordinates of the fiducial point from a list of WCS objects.
compute_s_region_imaging(wcs[, shape, center])	Update the `S_REGION` keyword using the WCS footprint.
compute_s_region_keyword(footprint)	Update the S_REGION keyword.
compute_scale(wcs, fiducial[, disp_axis, ...])	Compute the scale at the fiducial point on the detector..
sregion_to_footprint(s_region)	Parse the s_region string and return the footprint as an Nx2 array.
wcs_bbox_from_shape(shape)	Create a bounding box from the shape of the data.
wcs_from_sregions(footprints, ref_wcs, ...)	Create a WCS from a list of input s_regions or footprints.

calc_rotation_matrix

`stcal.alignment.`**`calc_rotation_matrix`**(*roll_ref: float*, *v3i_yangle: float*, *vparity: int = 1*) → list[float]

Calculate the rotation matrix.

Parameters

- **`roll_ref`** (*float*) – Telescope roll angle of V3 North over East at the ref. point in radians
- **`v3i_yangle`** (*float*) – The angle between ideal Y-axis and V3 in radians.
- **`vparity`** (*int*) – The x-axis parity, usually taken from the JWST SIAF parameter VIdlParity. Value should be "1" or "-1".

Returns

matrix – A list containing the rotation matrix elements in column order.

Return type

list

Notes

The rotation matrix is

$$PC =$$
$$beginbmatrixpc_{1,1}pc_{2,1}$$

$$pc_{1,2}pc_{2,2}$$
$$endbmatrix$$

combine_footprints

`stcal.alignment.combine_footprints(footprints)`

Combine a list of footprints into one or more combined footprints using a Shapely union.

Parameters

footprints (*list of np.ndarray*) – List of footprints, where each footprint is a 2-D array of shape (N, 2).

Returns

List of combined footprints, where each footprint is a 2-D array of shape (M, 2).

Return type

list of np.ndarray

combine_sregions

`stcal.alignment.combine_sregions(sregion_list, det2world, intersect_footprint=None)`

Combine s_regions from input models to compute the s_region for the resampled data.

Parameters

- **sregion_list** (*list[str] or list[np.ndarray]*) – List of s_regions from input models. If an element is a string, it will be converted to a footprint using *util.sregion_to_footprint*. If an element is already a footprint (2-D array of shape (N, 2)), it will be used directly.
- **det2world** (*~astropy.modeling.Model*) – WCS detector-to-world transform for the resampled data. Must take in exactly two inputs (x, y) and return exactly two outputs (RA, Dec). Must have a valid inverse transform.
- **intersect_footprint** (*np.ndarray, optional*) – Footprint of the output WCS in world coordinates, shape (N, 2). If provided, the combined footprint from the input s_region list will be intersected with this footprint.

Returns

The combined s_region.

Return type

str

Raises

ValueError – If there is no overlap between the input s_regions and the intersection footprint.

compute_fiducial

stcal.alignment.**compute_fiducial**(*wcslist: list*, *bounding_box: Sequence | None = None*) → np.ndarray

Calculate the world coordinates of the fiducial point from a list of WCS objects.

Calculates the world coordinates of the fiducial point of a list of WCS objects. For a celestial footprint this is the center. For a spectral footprint, it is the beginning of its range.

Parameters

- **wcslist** (*list*) – A list containing all the WCS objects for which the fiducial is to be calculated.
- **bounding_box** (*tuple, list, None, optional*) – The bounding box over which the WCS is valid. It can be a either tuple of tuples or a list of lists of size 2 where each element represents a range of (low, high) values. The bounding_box is in the order of the axes, axes_order. For two inputs and axes_order(0, 1) the bounding box can be either ((xlow, xhigh), (ylow, yhigh)) or [[xlow, xhigh], [ylow, yhigh]].

Returns

fiducial – A two-elements array containing the world coordinates of the fiducial point in the combined output coordinate frame.

Return type

np.ndarray

Notes

This function assumes all WCSs have the same output coordinate frame.

compute_s_region_imaging

stcal.alignment.**compute_s_region_imaging**(*wcs: gwcs.wcs.WCS*, *shape: Sequence | None = None*, *center: bool = True*) → str | None

Update the S_REGION keyword using the WCS footprint.

Parameters

- **wcs** (*WCS*) – The WCS object.
- **shape** (*tuple, None, optional*) – Shape of input model data array. Used to compute the bounding box if not provided in the WCS object, and required in that case. The default is None.
- **center** (*bool, None, optional*) – Whether or not to use the center of the pixel as reference for the coordinates, by default True

Returns

s_region – String containing the S_REGION object.

Return type

str

compute_s_region_keyword

stcal.alignment.**compute_s_region_keyword**(*footprint: ndarray*) → str | None

Update the S_REGION keyword.

Parameters

footprint – A 4x2 numpy array containing the coordinates of the vertices of the footprint.

Returns
s_region – String containing the S_REGION object.

Return type
str

compute_scale

`stcal.alignment.compute_scale(wcs: WCS, fiducial: tuple | ndarray, disp_axis: int | None = None, pscale_ratio: float | None = None) → float`

Compute the scale at the fiducial point on the detector..

Parameters

- **wcs** (*WCS*) – Reference WCS object from which to compute a scaling factor.
- **fiducial** (*tuple*) – Input fiducial of (RA, DEC) or (RA, DEC, Wavelength) used in calculating reference points.
- **disp_axis** (*int, None, optional*) – Dispersion axis integer. Assumes the same convention as `wcsinfo.dispersion_direction`
- **pscale_ratio** (*int, None, optional*) – Ratio of output pixel scale to input pixel scale.

Returns
scale – Scaling factor for x and y or cross-dispersion direction.

Return type
float

sregion_to_footprint

`stcal.alignment.sregion_to_footprint(s_region: str) → ndarray`

Parse the s_region string and return the footprint as an Nx2 array.

Parameters
s_region (*str*) – The S_REGION header keyword

Returns
footprint – A 2D array of the footprint of the region, shape (N, 2)

Return type
np.ndarray

wcs_bbox_from_shape

`stcal.alignment.wcs_bbox_from_shape(shape: Sequence) → tuple`

Create a bounding box from the shape of the data.

This is appropriate to attach to a wcs object

Parameters
shape (*tuple*) – The shape attribute from a *np.ndarray* array

Returns
bbox – Bounding box in x, y order.

Return type
tuple

wcs_from_sregions

`stcal.alignment.`**`wcs_from_sregions`**(*footprints: list[np.ndarray] | list[str], ref_wcs: gwcs.wcs.WCS, ref_wcsinfo: dict, transform:* `astropy.modeling.Model` *| None = None, pscale_ratio: float | None = None, pscale: float | None = None, rotation: float | None = None, shape: Sequence | None = None, crpix: Sequence | None = None, crval: Sequence | None = None*) → gwcs.wcs.WCS

Create a WCS from a list of input s_regions or footprints.

A fiducial point in the output coordinate frame is created from the footprints of all WCS objects. For a spatial frame this is the center of the union of the footprints. For a spectral frame the fiducial is in the beginning of the footprint range. If `refmodel` is None, the first WCS object in the list is considered a reference. The output coordinate frame and projection (for celestial frames) is taken from `refmodel`. If `transform` is not supplied, a compound transform is created using CDELTs and PC. If `bounding_box` is not supplied, the *bounding_box* of the new WCS is computed from *bounding_box* of all input WCSs.

Parameters

- **`footprints`** (*`list of np.ndarray or list of str`*) – If list elements are numpy arrays, each should have shape (N, 2) and contain (RA, Dec) vertices demarcating the footprint of the input WCSs. If list elements are strings, each should be the S_REGION header keyword containing (RA, Dec) vertices demarcating the footprint of the input WCSs.
- **`ref_wcs`** (*`WCS`*) – A WCS used as reference for the creation of the output coordinate frame, projection, and scaling and rotation transforms.
- **`ref_wcsinfo`** (*`dict`*) – A dictionary containing the WCS FITS keywords and corresponding values.
- **`transform`** (*`Model, None, optional`*) – A transform, the inverse of which is used to transform the *fiducial* to detecor coordinates. If not supplied *Scaling* | *Rotation* is computed from `refmodel`.
- **`pscale_ratio`** (*`float, None, optional`*) – Ratio of output pixel scale to input pixel scale. Ignored when either `transform` or `pscale` are provided.
- **`pscale`** (*`float, None, optional`*) – Absolute pixel scale in degrees. When provided, overrides `pscale_ratio`. Ignored when `transform` is provided.
- **`rotation`** (*`float, None, optional`*) – Position angle of output image's Y-axis relative to North. A value of 0.0 would orient the final output image to be North up. The default of *None* specifies that the images will not be rotated, but will instead be resampled in the default orientation for the camera with the x and y axes of the resampled image corresponding approximately to the detector axes. Ignored when `transform` is provided.
- **`shape`** (*`tuple of int, None, optional`*) – Shape of the image (data array) using `np.ndarray` convention (`ny` first and `nx` second). This value will be assigned to `pixel_shape` and `array_shape` properties of the returned WCS object.
- **`crpix`** (*`tuple of float, None, optional`*) – Position of the reference pixel in the image array. If `crpix` is not specified, it will be set to the center of the bounding box of the returned WCS object.
- **`crval`** (*`tuple of float, None, optional`*) – Right ascension and declination of the reference pixel. Automatically computed if not provided.

Returns

wcs_new – The WCS object corresponding to the combined input footprints.

Return type
WCS

1.1.4 TweakReg

Description

Overview

This step takes as input image catalogs of point-like sources, which are used to compute corrections to the WCS of the input images such that sky catalogs obtained from the image catalogs using the corrected WCS will align on the sky.

Source Catalogs

The input catalog must be in a format automatically recognized by *~astropy.table.Table.read*. The catalog must contain either `'x'` and `'y'` or `'xcentroid'` and `'ycentroid'` columns which indicate source *image* coordinates (in pixels). Pixel coordinates are 0-indexed. An optional column in the catalog is the `'weight'` column, which when present, will be used in fitting.

Relative Alignment

Relative alignment is performed by the *~stcal.tweakreg.relative_align* function. The source catalogs for each input image are compared to each other and linear (affine) coordinate transformations that align these catalogs are derived. This fit ensures that all the input images are aligned relative to each other. This step produces a combined source catalog for the entire set of input images as if they were combined into a single mosaic.

Absolute Alignment

Absolute alignment is performed by the *~stcal.tweakreg.absolute_align* function. If the parameter `abs_refcat` is set to 'GAIAREFCAT', 'GAIADR3', 'GAIADR2', or 'GAIADR1', an astrometric reference catalog then gets generated by querying a GAIA-based astrometric catalog web service for all astrometrically measured sources in the combined field-of-view of the set of input images. This catalog is generated from the catalogs available through the STScI MAST Catalogs and has the ability to account for proper motion to a given epoch. The epoch is computed from the observation date and time of the input data.

If `abs_refcat` is set to 'GAIADR3_S3' the HATS package will be used to access an S3 bucket containing a partitioned Gaia data release 3 catalog. When using the S3 catalog and a specified epoch, source positions are corrected internally by propagating the position to the given epoch using the Gaia proper motions and parallaxes (and throwing away any sources without computed proper motions and parallaxes). Parallaxes are computed as if the observer were located at the Earth barycenter.

The combined source catalog then gets cross-matched and fit to this astrometric reference catalog. The results of this one fit then gets back-propagated to all the input images to align them all to the astrometric reference frame while maintaining the relative alignment between the images.

For this part of alignment, instead of one of the above Gaia catalogs, users can supply an external reference catalog by providing a path to an existing file. A user-supplied catalog must contain `'RA'` and `'DEC'` columns indicating reference source world coordinates (in degrees). An optional column in the catalog is the `'weight'` column, which when present, will be used in fitting. The catalog must be in a format automatically recognized by *~astropy.table.Table.read*.

Grouping

Images taken at the same time (e.g., NIRCam images from all short-wave detectors) can be aligned together; that is, a single correction can be computed and applied to all these images because any error in telescope pointing will be identical in all these images and it is assumed that the relative positions of (e.g., NIRCam) detectors do not change.

Identification of images that belong to the same "exposure" and therefore can be grouped together is based on several attributes *change from here to end of section* described in *~jwst.datamodels.ModelContainer*. This grouping is performed automatically in the `tweakreg` step using the *~jwst.datamodels.ModelContainer.models_grouped* property, which assigns a group ID to each input image model in `meta.group_id`.

However, when detector calibrations are not accurate, alignment of groups of images may fail (or result in poor alignment). In this case, it may be desirable to align each image independently. This can be achieved either by setting the `image_model.meta.group_id` attribute to a unique string or integer value for each image, or by adding the `group_id` attribute to the `members` of the input ASN table - see *~jwst.datamodels.ModelContainer* for more details.

Note

Group ID (`group_id`) is used by both `tweakreg` and `skymatch` steps and so modifying it for one step will affect the results in another step. If it is desirable to apply different grouping strategies to the `tweakreg` and `skymatch` steps, one may need to run each step individually and provide a different ASN as input to each step.

WCS Correction

The linear coordinate transformation computed in the previous step is used to define tangent-plane corrections that need to be applied to the GWCS pipeline in order to correct input image WCS. This correction is implemented by inserting a `v2v3corr` frame with tangent plane corrections into the GWCS pipeline of the image's WCS.

Arguments

The `tweakreg` step has the following optional arguments:

Relative alignment parameters:

Parameters used for relative alignment between input images. These parameters are passed into the *~stcal.tweakreg.relative_align* function.

- `expand_refcat`: A boolean indicating whether or not to expand reference catalog with new sources from other input images that have been already aligned to the reference image. (Default=False)
- `minobj`: A positive *int* indicating minimum number of objects acceptable for matching. (Default=15)
- `searchrad`: A *float* indicating the search radius in arcsec for a match. (Default=2.0)
- `use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset. (Default=True)
- `separation`: Minimum object separation in arcsec. It **must be** at least `sqrt(2)` times larger than `tolerance`. (Default=1.0)
- `tolerance`: Matching tolerance for `xyxymatch` in arcsec. (Default=0.7)
- `xoffset`: Initial guess for X offset in arcsec. (Default=0.0)
- `yoffset`: Initial guess for Y offset in arcsec. (Default=0.0)
- `fitgeometry`: A *str* value indicating the type of affine transformation to be considered when fitting catalogs. Allowed values:
  - `'shift'`: x/y shifts only
  - `'rshift'`: rotation and shifts
  - `'rscale'`: rotation and scale
  - `'general'`: shift, rotation, and scale

  The default value is "rshift".

Note

Mathematically, alignment of images observed in different tangent planes requires `fitgeometry='general'` in order to fit source catalogs in the different images even if misalignment is caused only by a shift or rotation in the tangent plane of one of the images.

However, under certain circumstances, such as small alignment errors or minimal dithering during observations that keep tangent planes of the images to be aligned almost parallel, then it may be more robust to use a `fitgeometry` setting with fewer degrees of freedom such as `'rshift'`, especially for "ill-conditioned" source catalogs such as catalogs with very few sources, or large errors in source positions, or sources placed along a line or bunched in a corner of the image (not spread across/covering the entire image).

- `nclip`: A non-negative integer number of clipping iterations to use in the fit. (Default=3)
- `sigma`: A positive *float* indicating the clipping limit, in sigma units, used when performing fit. (Default=3.0)

Absolute Astrometric fitting parameters:

Parameters used for absolute astrometry to a reference catalog. These parameters are passed into the *~stcal.tweakreg.absolute_align* function.

- `abs_refcat`: String indicating what astrometric catalog should be used. Currently supported options: 'GAIAREFCAT', 'GAIADR1', 'GAIADR2', 'GAIADR3', 'GAIADR3_S3', a path to an existing reference catalog, *None*, or *''*. See *stcal.tweakreg.tweakreg.SINGLE_GROUP_REFCAT* for an up-to-date list of supported built-in reference catalogs.

  When `abs_refcat` is *None* or an empty string, alignment to the absolute astrometry catalog will be turned off. (Default= '')
- `abs_minobj`: A positive *int* indicating minimum number of objects acceptable for matching. (Default=15)
- `abs_searchrad`: A *float* indicating the search radius in arcsec for a match. It is recommended that a value larger than `searchrad` be used for this parameter (e.g. 3 times larger) (Default=6.0)
- `abs_use2dhist`: A boolean indicating whether to use 2D histogram to find initial offset. It is strongly recommended setting this parameter to *True*. Otherwise the initial guess for the offsets will be set to zero (Default=True)
- `abs_separation`: Minimum object separation in arcsec. It **must be** at least `sqrt(2)` times larger than `abs_tolerance`. (Default=1.0)
- `abs_tolerance`: Matching tolerance for `xyxymatch` in arcsec. (Default=0.7)
- `abs_fitgeometry`: A *str* value indicating the type of affine transformation to be considered when fitting catalogs. Allowed values:
  - `'shift'`: x/y shifts only
  - `'rshift'`: rotation and shifts
  - `'rscale'`: rotation and scale
  - `'general'`: shift, rotation, and scale

  The default value is "rshift". Note that the same conditions/restrictions that apply to `fitgeometry` also apply to `abs_fitgeometry`.
- `abs_nclip`: A non-negative integer number of clipping iterations to use in the fit. (Default=3)
- `abs_sigma`: A positive *float* indicating the clipping limit, in sigma units, used when performing fit. (Default=3.0)
- `save_abs_catalog`: A boolean specifying whether or not to write out the astrometric catalog used for the fit as a separate product. (Default=False)

Further Documentation

The underlying algorithms as well as formats of source catalogs are described in more detail at

https://tweakwcs.readthedocs.io/en/latest/

stcal.tweakreg.tweakreg Module

Functions

`relative_align`(correctors[, searchrad, ...])	
`absolute_align`(correctors, abs_refcat, ...)	
`filter_catalog_by_bounding_box`(catalog, ...)	Given a catalog of x,y positions, only return sources that fall inside the bounding box.

relative_align

`stcal.tweakreg.tweakreg.`**`relative_align`**(*correctors: list, searchrad: float = 2.0, separation: float = 1.0, use2dhist: bool = True, tolerance: float = 0.7, xoffset: float = 0.0, yoffset: float = 0.0, enforce_user_order: bool = False, expand_refcat: bool = False, minobj: int = 15, fitgeometry: str = 'rshift', nclip: int = 3, sigma: float = 3.0, clip_accum: bool = False*) → list

absolute_align

`stcal.tweakreg.tweakreg.`**`absolute_align`**(*correctors: list, abs_refcat: str, ref_wcs: gwcs.wcs.WCS, ref_wcsinfo: dict, epoch: str | astropy.time.Time, save_abs_catalog: bool = False, abs_catalog_output_dir: str | None = None, abs_searchrad: float = 6.0, abs_separation: float = 1.0, abs_use2dhist: bool = True, abs_tolerance: float = 0.7, abs_minobj: int = 15, abs_fitgeometry: str = 'rshift', abs_nclip: int = 3, abs_sigma: float = 3.0, clip_accum: bool = False, timeout: float = 30.0*) → list

filter_catalog_by_bounding_box

`stcal.tweakreg.tweakreg.`**`filter_catalog_by_bounding_box`**(*catalog: Table, bounding_box: list[tuple]*) → Table

Given a catalog of x,y positions, only return sources that fall inside the bounding box.

Also See:

Utility Functions

Currently, the `utils` module provides helpful functions for manually applying corrections to an imaging WCS.

stcal.tweakreg.utils Module

Functions

_wcsinfo_from_wcs_transform(wcs)	

_wcsinfo_from_wcs_transform

`stcal.tweakreg.utils._wcsinfo_from_wcs_transform(wcs)`

astrometric_utils

The `astrometric_utils` module provides functions for generating astrometric catalogs of sources for the field-of-view covered by a set of images.

stcal.tweakreg.astrometric_utils Module

Functions

compute_radius(wcs)	Compute the radius from the center to the furthest edge of the WCS.
create_astrometric_catalog(wcs, epoch[, ...])	Create an astrometric catalog that covers the inputs' field-of-view.
get_catalog(right_ascension, declination[, ...])	Extract catalog from VO web service.

compute_radius

`stcal.tweakreg.astrometric_utils.compute_radius(wcs)`

Compute the radius from the center to the furthest edge of the WCS.

create_astrometric_catalog

`stcal.tweakreg.astrometric_utils.create_astrometric_catalog(wcs, epoch, catalog='GAIADR3', output='ref_cat.ecsv', table_format='ascii.ecsv', num_sources=None, timeout=30.0)`

Create an astrometric catalog that covers the inputs' field-of-view.

Parameters

- **wcs** (*~astropy.wcs.WCS*) – WCS object specified by the user as generated by *resample.resample_utils.make_output_wcs*. This will typically have the same plate-scale and orientation as the first member in the list of input images to make_output_wcs. Fortunately, for alignment, this doesn't matter since no resampling of data will be performed.
- **epoch** (`float or None`) – Reference epoch used to update the coordinates for proper motion (in decimal year). When `None` no proper motion correction will be performed and all sources (even those without proper motion) will be returned. When not `None` only sources with proper motion will be returned.
- **catalog** (`str, optional`) – Name of catalog to extract astrometric positions for sources in the input images' field-of-view. Default: GAIADR3. Options available are documented on the catalog web page.
- **output** (`str, optional`) – Filename to give to the astrometric catalog read in from the master catalog web service. If None, no file will be written out.

- **num_sources** (*int*) – Maximum number of brightest/faintest sources to return in catalog. If *num_sources* is negative, return that number of the faintest sources. By default, all sources are returned.
- **timeout** (*float*) – Maximum time to wait (in seconds) for the catalog service to respond.

Notes

This function will point to astrometric catalog web service defined through the use of the ASTROMETRIC_CATALOG_URL environment variable.

Returns
ref_table – Astropy Table object of the catalog

Return type
~astropy.table.Table

get_catalog

`stcal.tweakreg.astrometric_utils.get_catalog(right_ascension, declination, epoch=2016.0, search_radius=0.1, catalog='GAIADR3', timeout=30.0)`

Extract catalog from VO web service.

Parameters

- **right_ascension** (*float*) – Right Ascension (RA) of center of field-of-view (in decimal degrees)
- **declination** (*float*) – Declination (Dec) of center of field-of-view (in decimal degrees)
- **epoch** (*float or None, optional*) – Reference epoch used to update the coordinates for proper motion (in decimal year). When `None` no proper motion correction will be performed and all sources (even those without proper motion) will be returned. When not `None` only sources with proper motion will be returned. Default: 2016.0
- **search_radius** (*float, optional*) – Search radius (in decimal degrees) from field-of-view center to use for sources from catalog. Default: 0.1 degrees
- **catalog** (*str, optional*) – Name of catalog to query, as defined by web-service. Default: 'GAIADR3'
- **timeout** (*float, optional*) – Timeout in seconds to wait for the catalog web service to respond. Default: 30.0 s

Returns
csv – CSV object of returned sources with all columns as provided by catalog

Return type
~astropy.table.Table

stcal.tweakreg Package

1.1.5 Outlier Detection Utils

Description

This sub-package contains functions useful for outlier detection.

stcal.outlier_detection.median Module

Compute median of large datasets in memory- and runtime-efficient ways.

Functions

nanmedian3D(cube[, overwrite_input])	Compute the nanmedian of a cube.

nanmedian3D

stcal.outlier_detection.median.**nanmedian3D**(*cube: ndarray*, *overwrite_input: bool = True*) → ndarray

Compute the nanmedian of a cube.

This produces identical results to np.nanmedian but is much more memory efficient for large cubes.

RuntimeWarnings reporting "All-Nan slice" will be ignored.

Parameters

- **cube** – 3-dimensional array. Will be modified in-place if *overwrite_input* is True
- **overwrite_input** – Passed to `np.nanmedian`, if True the input cube will be modified

Returns

2-dimensional computed median array

Return type

np.ndarray

Classes

MedianComputer(full_shape, in_memory[, ...])	Class to efficiently compute a median.

MedianComputer

class stcal.outlier_detection.median.**MedianComputer**(*full_shape: tuple*, *in_memory: bool*, *buffer_size: int | None = None*, *dtype: str | dtype = 'float32'*, *tempdir: str = ''*)

Bases: `object`

Class to efficiently compute a median.

Top-level class to treat median computation uniformly, whether in memory or on disk.

Initialize MedianComputer.

Parameters

- **full_shape** – The shape of the full input dataset.
- **in_memory** – Whether to perform the median computation in memory or using temporary files on disk to save memory.
- **buffer_size** – The buffer size for the median computation, units of bytes. Has no effect if *in_memory* is True.
- **dtype** – The data type of the input data.

- **tempdir** – The parent directory in which to create the temporary directory. Default is the current working directory.

Methods Summary

append(data[, idx])	Append data to the median computer.
evaluate()	Compute the median data from the input data.

Methods Documentation

append(*data: ndarray*, *idx: int | None = None*) → None

Append data to the median computer.

Parameters

- **data** – The data to append to the median computer. Must have shape full_shape[1:].
- **idx** – The index at which to append the data. Must be between 0 and full_shape[0]. Required if using in-memory median computation.

evaluate() → ndarray

Compute the median data from the input data.

Returns

The median data computed from the input data.

Return type

np.ndarray

Class Inheritance Diagram

MedianComputer

stcal.outlier_detection.utils Module

Utility functions for outlier detection routines.

Functions

medfilt(arr, kern_size)	Median filter.
compute_weight_threshold(weight, maskpt)	Compute the weight threshold for a single image or cube.
flag_crs(sci_data, sci_err, blot_data, snr)	Flag outliers.
flag_resampled_crs(sci_data, sci_err, ...)	Detect outliers (CRs) using resampled reference data.
gwcs_blot(median_data, median_wcs, ...[, ...])	Resample the median data to recreate an input image based on the blot wcs.

medfilt

`stcal.outlier_detection.utils.medfilt(arr, kern_size)`

Median filter.

scipy.signal.medfilt (and many other median filters) have undefined behavior for nan inputs. See: https://github.com/scipy/scipy/issues/4800.

Parameters

- **arr** (*numpy.ndarray*) – The input array
- **kern_size** (*list of int*) – List of kernel dimensions, length must be equal to arr.ndim.

Returns

filtered_arr – Input array median filtered with a kernel of size kern_size

Return type

numpy.ndarray

compute_weight_threshold

`stcal.outlier_detection.utils.compute_weight_threshold(weight, maskpt)`

Compute the weight threshold for a single image or cube.

Parameters

- **weight** (*numpy.ndarray*) – The weight array
- **maskpt** (*float*) – The percentage of the mean weight to use as a threshold for masking.

Returns

The weight threshold for this integration.

Return type

float

flag_crs

`stcal.outlier_detection.utils.flag_crs(sci_data, sci_err, blot_data, snr)`

Flag outliers.

Straightforward detection of outliers for non-dithered data since sci_err includes all noise sources (photon, read, and flat for baseline).

Parameters

- **sci_data** (*numpy.ndarray*) – "Science" data possibly containing outliers.
- **sci_err** (*numpy.ndarray*) – Error estimates for sci_data.
- **blot_data** (*numpy.ndarray*) – Reference data used to detect outliers.
- **snr** (*float*) – Signal-to-noise ratio used during detection.

Returns

cr_mask – Boolean array where outliers (CRs) are true.

Return type

numpy.ndarray

flag_resampled_crs

`stcal.outlier_detection.utils.flag_resampled_crs`(*sci_data*, *sci_err*, *blot_data*, *snr1*, *snr2*, *scale1*, *scale2*, *backg*)

Detect outliers (CRs) using resampled reference data.

Parameters

- **sci_data** (*numpy.ndarray*) – "Science" data possibly containing outliers
- **sci_err** (*numpy.ndarray*) – Error estimates for sci_data
- **blot_data** (*numpy.ndarray*) – Reference data used to detect outliers.
- **snr1** (*float*) – Signal-to-noise ratio threshold used prior to smoothing.
- **snr2** (*float*) – Signal-to-noise ratio threshold used after smoothing.
- **scale1** (*float*) – Scale used prior to smoothing.
- **scale2** (*float*) – Scale used after smoothing.
- **backg** (*float*) – Scalar background to subtract from the difference.

Returns

cr_mask – boolean array where outliers (CRs) are true

Return type

numpy.ndarray

gwcs_blot

`stcal.outlier_detection.utils.gwcs_blot`(*median_data*, *median_wcs*, *blot_shape*, *blot_wcs*, *pix_ratio=None*, *fillval=0.0*, *pixmap_stepsize=1*, *pixmap_order=1*)

Resample the median data to recreate an input image based on the blot wcs.

Parameters

- **median_data** (*numpy.ndarray*) – The data to blot.
- **median_wcs** (*gwcs.wcs.WCS*) – The wcs for the median data.
- **blot_shape** (*tuple of int*) – The target blot data shape.
- **blot_wcs** (*gwcs.wcs.WCS*) – The target/blotted wcs.
- **fillval** (*float, optional*) – Fill value for missing data.
- **pixmap_stepsize** (*int, optional*) – If `pixmap_stepsize>1`, perform the full WCS calculation on a sparser grid and use interpolation to fill in the rest of the pixels. This option speeds up pixel map computation by reducing the number of WCS calls, though at the cost of reduced pixel map accuracy. The loss of accuracy is typically negligible if the underlying distortion correction is smooth, but if the distortion is non-smooth, `pixmap_stepsize>1` is not recommended. Large `pixmap_stepsize` values are automatically reduced to no more than 1/10 of image size. Passed to *stcal.resample.utils.calc_pixmap*. Default 1.
- **pixmap_order** (*int, optional*) – Order of the 2D spline to interpolate the sparse pixel mapping if `pixmap_stepsize>1`. Supported values are: 1 (bilinear) or 3 (bicubic). This Parameter is ignored when `pixmap_stepsize <= 1`. Default 1.

Returns

- **blotted** (*numpy.ndarray*) – The blotted median data.

- **blot_img** (*datamodel*) – Datamodel containing header and WCS to define the 'blotted' image

1.1.6 Resample

Description

Classes
stcal.resample.Resample

Alias
resample

This routine will resample each input 2D image based on the WCS and distortion information, and will combine multiple resampled images into a single undistorted product.

This step uses the interface to the C-based cdriz routine to do the resampling via the drizzle method. The input-to-output pixel mapping is determined via a mapping function derived from the WCS of each input image and the WCS of the defined output product. This mapping function gets passed to cdriz to drive the actual drizzling to create the output product.

Error Propagation

The error associated with each resampled pixel can in principle be derived from the variance components associated with each input pixel, weighted by the square of the input user weights and the square of the overlap between the input and output pixels. In practice, the `cdriz` routine does not currently support propagating variance data alongside science images, so the output error cannot be precisely calculated.

To approximate the error on a resampled pixel, the variance arrays associated with each input model are resampled individually, then combined with a weighted sum. The process is:

1. For each input model, take the square root of each of the read noise variance arrays to make an error image.
2. Drizzle the read noise error image onto the output WCS, with drizzle parameters matching those used for the science data.
3. Square the resampled read noise to make a variance array.
   a. If the resampling *weight_type* is an inverse variance map (*ivm*), weight the resampled variance by the square of its own inverse.
   b. If the *weight_type* is the exposure time (*exptime*), weight the resampled variance by the square of the exposure time for the image.
4. Add the weighted, resampled read noise variance to a running sum across all images. Add the weights (unsquared) to a separate running sum across all images.
5. Perform the same steps for the Poisson noise variance and the flat variance. For these components, the weight for the sum is either the resampled read noise variance or else the exposure time.
6. For each variance component (read noise, Poisson, and flat), divide the weighted variance sum by the total weight, squared.

After each variance component is resampled and summed, the final error array is computed as the square root of the sum of the three independent variance components. This error image is stored in the `err` attribute in the output data model. Alternatively, the error array of the resampled image can be computed by resampling the error array associated with input data.

It is expected that the output errors computed in this way will generally overestimate the true error on the resampled data. The magnitude of the overestimation depends on the details of the pixel weights and error images. Note, however, that drizzling error images produces a much better estimate of the output error than directly drizzling the variance images, since the kernel overlap weights do not need to be squared for combining error values.

Context Image

In addition to image data, resample step also creates a "context image" stored in the `con` attribute in the output data model . Each pixel in the context image is a bit field that encodes information about which input image has contributed to the corresponding pixel in the resampled data array. The context image uses 32 bit integers to encode this information, and hence it can keep track of only 32 input images. The first bit corresponds to the first input image, the second bit corresponds to the second input image, and so on. If the number of input images is larger than 32, then it is necessary to have multiple context images ("planes") to hold information about all input images, with the first plane encoding which of the first 32 images contributed to the output data pixel, the second plane representing next 32 input images (number 33-64), etc. For this reason, context array is a 3D array of the type *numpy.int32* and shape `(np, ny, nx)` where `nx` and `ny` are the dimensions of the image data. `np` is the number of "planes" computed as `(number of input images - 1) // 32 + 1`. If a bit at position `k` in a pixel with coordinates `(p, y, x)` is 0, then input image number `32 * p + k` (0-indexed) did not contribute to the output data pixel with array coordinates `(y, x)` and if that bit is 1, then input image number `32 * p + k` did contribute to the pixel `(y, x)` in the resampled image.

As an example, let's assume we have 8 input images. Then, when `'CON'` pixel values are displayed using binary representation (and decimal in parenthesis), one could see values like this:

```
00000001 (1) - only first input image contributed to this output pixel;
00000010 (2) - 2nd input image contributed;
00000100 (4) - 3rd input image contributed;
10000000 (128) - 8th input image contributed;
10000100 (132=128+4) - 3rd and 8th input images contributed;
11001101 (205=1+4+8+64+128) - input images 1, 3, 4, 7, 8 have contributed
to this output pixel.
```

In order to test if a specific input image contributed to an output pixel, one needs to use bitwise operations. Using the example above, to test whether input images number 4 and 5 have contributed to the output pixel whose corresponding `'CON'` value is 205 (11001101 in binary form) we can do the following:

```
>>> bool(205 & (1 << (5 - 1)))  # (205 & 16) = 0 (== 0 => False): did NOT contribute
False
>>> bool(205 & (1 << (4 - 1)))  # (205 & 8) = 8 (!= 0 => True): did contribute
True
```

In general, to get a list of all input images that have contributed to an output resampled pixel with image coordinates `(x, y)`, and given a context array `con`, one can do something like this:

```
>>> import numpy as np
>>> np.flatnonzero([v & (1 << k) for v in con[:, y, x] for k in range(32)])
```

For convenience, this functionality was implemented in the `decode_context()` function.

References

A full description of the drizzling algorithm can be found in Fruchter and Hook, PASP 2002. A description of the inverse variance map method can be found in Casertano et al., AJ 2000, see Appendix A2. A description of the drizzle parameters and other useful drizzle-related resources can be found at DrizzlePac Handbook.

Also See:

Utility Functions

The `utils` module provides helpful functions for *Resample* such as creating image mask from model's DQ array, computing average pixel area, loading a custom WCS from an ASDF file, etc.

stcal.resample.utils Module

Functions

calc_pixmap(wcs_from, wcs_to[, shape, ...])	Calculate pixel coordinates of one WCS corresponding to the native pixel grid of another WCS.
build_driz_weight(model[, weight_type, ...])	Build drizzle weight map.
build_mask(dqarr, good_bits[, flag_name_map])	Build a bit mask from an input DQ array and a bitvalue flag.
compute_mean_pixel_area(wcs[, shape])	Compute mean pixel area.
get_tmeasure(model)	Get tmeasure from datamodel.
is_flux_density(bunit)	Differentiate between surface brightness and flux density data units.
is_imaging_wcs(wcs)	Return *True* if `wcs` is an imaging WCS and *False* otherwise.
resample_range(data_shape[, bbox])	

calc_pixmap

`stcal.resample.utils.`**`calc_pixmap`**(*wcs_from*, *wcs_to*, *shape=None*, *disable_bbox='to'*, *stepsize=1*, *order=1*)

Calculate pixel coordinates of one WCS corresponding to the native pixel grid of another WCS.

Note

This function assumes that output frames of `wcs_from` and `wcs_to` WCS have the same units.

Parameters

- **wcs_from** (*object*) – A WCS object representing the coordinate system you are converting from. This object's `array_shape` (or `pixel_shape`) property will be used to define the shape of the pixel map array. If `shape` parameter is provided, it will take precedence over this object's `array_shape` value.
- **wcs_to** (*object*) – A WCS object representing the coordinate system you are converting to.
- **shape** (*tuple, None, optional*) – A tuple of integers indicating the shape of the output array in the `numpy.ndarray` order. When provided, it takes precedence over the `wcs_from.array_shape` property.
- **disable_bbox** (*str, optional*) – Indicates whether to use or not to use the bounding box of either (both) `wcs_from` or (and) `wcs_to` when computing pixel map. Allowable values: "to", "from", "both", "none". When `disable_bbox` is "none", pixel coordinates outside of the bounding box are set to *NaN* only if `wcs_from` or (and) `wcs_to` sets world coordinates to NaN when input pixel coordinates are outside of the bounding box.
- **stepsize** (*int, optional*) – If `stepsize>1`, perform the full WCS calculation on a sparser grid and use interpolation to fill in the rest of the pixels. This option speeds up pixel map computation by reducing the number of WCS calls, though at the cost of reduced pixel

map accuracy. The loss of accuracy is typically negligible if the underlying distortion correction is smooth, but if the distortion is non-smooth, `stepsize>1` is not recommended. Large `stepsize` values are automatically reduced to no more than 1/10 of image size. Default 1.

- **order** (*int, optional*) – Order of the 2D spline to interpolate the sparse pixel mapping if stepsize>1. Supported values are: 1 (bilinear) or 3 (bicubic). This Parameter is ignored when `stepsize <= 1`. Default 1.

Returns

pixmap – A three dimensional array representing the transformation between the two. The last dimension is of length two and contains the x and y coordinates of a pixel center, respectively. The other two coordinates correspond to the two coordinates of the image the first WCS is from.

Return type

numpy.ndarray

Raises

`ValueError` – A *ValueError* is raised when output pixel map shape cannot be determined from provided inputs.

Notes

When `shape` is not provided and `wcs_from.array_shape` is not set (i.e., it is *None*), *calc_pixmap* will attempt to determine pixel map shape from the `bounding_box` property of the input `wcs_from` object. If `bounding_box` is not available, a *ValueError* will be raised.

build_driz_weight

`stcal.resample.utils.build_driz_weight`(*model*, *weight_type=None*, *good_bits=None*, *flag_name_map=None*)

Build drizzle weight map.

Create a weight map that is used for weighting input images when they are co-added to the output model.

Parameters

- **model** (*dict*) – Input model: a dictionary of relevant keywords and values.
- **weight_type** (*str or None, optional*) – The weighting type ("exptime", "ivm", or "ivm-sky") for adding models' data. For `weight_type="ivm"` and `weight_type="ivm-sky"`, the weighting will be determined per-pixel using the inverse of either the read noise (VAR_RNOISE) or sky variance (VAR_SKY) arrays, respectively. If the array does not exist, the weight is set to 1 for all pixels (i.e., equal weighting). If `weight_type="exptime"`, the weight will be set equal to the measurement time when available and to the exposure time otherwise for pixels not flagged in the DQ array of the model. The default value of *None* will set weights to 1 for pixels not flagged in the DQ array of the model. Pixels flagged as "bad" in the DQ array will have their weights set to 0.
- **good_bits** (*int, str, None, optional*) – An integer bit mask, *None*, a Python list of bit flags, a comma-, or `'|'`-separated, `'+'`-separated string list of integer bit flags or mnemonic flag names that indicate what bits in models' DQ bitfield array should be *ignored* (i.e., zeroed).

  See *Resample* for more information.
- **flag_name_map** (*astropy.nddata.BitFlagNameMap, dict, None, optional*) – A *~astropy.nddata.BitFlagNameMap* object or a dictionary that provides mapping from

mnemonic bit flag names to integer bit values in order to translate mnemonic flags to numeric values when `bit_flags` that are comma- or '+'-separated list of mnemonic bit flag names.

build_mask

`stcal.resample.utils.build_mask(dqarr, good_bits, flag_name_map=None)`

Build a bit mask from an input DQ array and a bitvalue flag.

In the returned bit mask, 1 is good, 0 is bad

compute_mean_pixel_area

`stcal.resample.utils.compute_mean_pixel_area(wcs, shape=None)`

Compute mean pixel area.

Computes the average pixel area (in steradians) based on input WCS using pixels within either the bounding box (if available) or the entire data array as defined either by `wcs.array_shape` or the `shape` argument.

Parameters

shape (*tuple, optional*) – Shape of the region over which average pixel area will be computed. When not provided, pixel average will be estimated over a region defined by `wcs.array_shape`.

Returns

pix_area – Pixel area in steradians.

Return type

float

Notes

This function takes the outline of the region in which the average is computed (a rectangle defined by either the bounding box or `wcs.array_shape` or the `shape`) and projects it to world coordinates. It then uses `spherical_geometry` to compute the area of the polygon defined by this outline on the sky. In order to minimize errors due to distortions in the `wcs`, the code defines the outline using pixels spaced no more than 15 pixels apart along the border of the rectangle in which the average is computed.

get_tmeasure

`stcal.resample.utils.get_tmeasure(model)`

Get tmeasure from datamodel.

Check if the measurement_time keyword is present in the datamodel for use in exptime weighting. If not, revert to using exposure_time.

Returns a tuple of (exptime, is_measurement_time)

is_flux_density

`stcal.resample.utils.is_flux_density(bunit)`

Differentiate between surface brightness and flux density data units.

Parameters

bunit (str or *~astropy.units.Unit*) – Data units, e.g. 'MJy' (is flux density) or 'MJy/sr' (is not).

Returns

True if the units are equivalent to flux density units.

Return type
bool

is_imaging_wcs

stcal.resample.utils.**is_imaging_wcs**(*wcs*)

Return *True* if wcs is an imaging WCS and *False* otherwise.

resample_range

stcal.resample.utils.**resample_range**(*data_shape*, *bbox=None*)

stcal.resample Package

Functions

compute_mean_pixel_area(wcs[, shape])	Compute mean pixel area.

compute_mean_pixel_area

stcal.resample.**compute_mean_pixel_area**(*wcs*, *shape=None*)

Compute mean pixel area.

Computes the average pixel area (in steradians) based on input WCS using pixels within either the bounding box (if available) or the entire data array as defined either by wcs.array_shape or the shape argument.

Parameters
shape (*tuple, optional*) – Shape of the region over which average pixel area will be computed. When not provided, pixel average will be estimated over a region defined by wcs.array_shape.

Returns
pix_area – Pixel area in steradians.

Return type
float

Notes

This function takes the outline of the region in which the average is computed (a rectangle defined by either the bounding box or wcs.array_shape or the shape) and projects it to world coordinates. It then uses spherical_geometry to compute the area of the polygon defined by this outline on the sky. In order to minimize errors due to distortions in the wcs, the code defines the outline using pixels spaced no more than 15 pixels apart along the border of the rectangle in which the average is computed.

Classes

Resample(output_wcs[, n_input_models, ...])	Base class for resampling images.
UnsupportedWCSError	Unsupported WCS Error.

Resample

class stcal.resample.**Resample**(*output_wcs*, *n_input_models=None*, *pixfrac=1.0*, *kernel='square'*, *fillval=0.0*, *weight_type='ivm'*, *good_bits=0*, *enable_ctx=True*, *enable_var=True*, *compute_err=None*, *propagate_dq=False*, *pixmap_stepsize=1*, *pixmap_order=1*)

Bases: `object`

Base class for resampling images.

The main purpose of this class is to resample and add input images (data, variance array) to an output image defined by an output WCS.

In particular, this class performs the following operations:

1. Sets up output arrays based on arguments used at initialization.
2. Based on information about the input images and user arguments, computes scale factors needed to convert resampled counts to fluxes.
3. For each input image, computes coordinate transformations (`pixmap`) from the coordinate system of the input image to the coordinate system of the output image.
4. Computes the weight image for each input image.
5. Calls `Drizzle` methods to resample and combine input images and their variance/error arrays.
6. Keeps track of total exposure time and other time-related quantities.

Initialize Resample.

Parameters

- **output_wcs** (*dict*) – Specifies output WCS as a dictionary with keys `'wcs'` (WCS object) and `'pixel_scale'` (pixel scale in arcseconds). `'pixel_scale'`, when provided, will be used for computation of drizzle scaling factor. When it is not provided, output pixel scale will be *estimated* from the provided WCS object.
- **n_input_models** (*int, None, optional*) – Number of input models expected to be resampled. When provided, this is used to estimate memory requirements and optimize memory allocation for the context array.
- **pixfrac** (*float, optional*) – The fraction of a pixel that the pixel flux is confined to. The default value of 1 has the pixel flux evenly spread across the image. A value of 0.5 confines it to half a pixel in the linear dimension, so the flux is confined to a quarter of the pixel area when the square kernel is used.
- **kernel** (*str, optional*) – The name of the kernel used to combine the input: "square", "gaussian", "point", "turbo", "lanczos2", or "lanczos3". The choice of kernel controls the distribution of flux over the kernel. The square kernel is the default.

> **Warning**
>
> The "gaussian" and "lanczos2/3" kernels **DO NOT** conserve flux.

- **fillval** (*float, None, str, optional*) – The value of output pixels that did not have contributions from input images' pixels. When `fillval` is either *None* or `"INDEF"` and `out_img` is provided, the values of `out_img` will not be modified. When `fillval` is either *None* or `"INDEF"` and `out_img` is **not provided**, the values of `out_img` will be initialized to

numpy.nan. If fillval is a string that can be converted to a number, then the output pixels with no contributions from input images will be set to this fillval value.

- **weight_type** (*str, optional*) – The weighting type ("ivm", "exptime", or "ivm-sky") for adding models' data. For weight_type="ivm" (the default), the weighting will be determined per-pixel using the inverse of the read noise (VAR_RNOISE) array stored in each input image. If the VAR_RNOISE array does not exist, the variance is set to 1 for all pixels (i.e., equal weighting). If weight_type="ivm-sky", the weighting will be determined per-pixel using the inverse of the sky variance (VAR_SKY) array stored in each input image. If the VAR_SKY array does not exist, the variance is set to 0 for all pixels (i.e., equal weighting). If weight_type="exptime", the weight will be set equal to the measurement time when available and to the exposure time otherwise.
- **good_bits** (*int, str, None, optional*) – An integer bit mask, *None*, a Python list of bit flags, a comma-, or '|'-separated, '+'-separated string list of integer bit flags or mnemonic flag names that indicate what bits in models' DQ bitfield array should be *ignored* (i.e., zeroed).

  When co-adding models using add_model(), any pixels with a non-zero DQ values are assigned a weight of zero and therefore they do not contribute to the output (resampled) data. good_bits provides a mean to ignore some of the DQ bitflags.

  When good_bits is an integer, it must be the sum of all the DQ bit values from the input model's DQ array that should be considered "good" (or ignored). For example, if pixels in the DQ array can be combinations of 1, 2, 4, and 8 flags and one wants to consider DQ "defects" having flags 2 and 4 as being acceptable, then good_bits should be set to 2+4=6. Then a pixel with DQ values 2,4, or 6 will be considered a good pixel, while a pixel with DQ value, e.g., 1+2=3, 4+8=12, etc. will be flagged as a "bad" pixel.

  Alternatively, when good_bits is a string, it can be a comma-separated or '+' separated list of integer bit flags that should be summed to obtain the final "good" bits. For example, both "4,8" and "4+8" are equivalent to integer good_bits=12.

  Finally, instead of integers, good_bits can be a string of comma-separated mnemonics. For example, for JWST, all the following specifications are equivalent:

  "12" == "4+8" == "4, 8" == "JUMP_DET, DROPOUT"

  In order to "translate" mnemonic code to integer bit flags, Resample.dq_flag_name_map attribute must be set to either a dictionary (with keys being mnemonc codes and the values being integer flags) or a *~astropy.nddata.BitFlagNameMap*.

  In order to reverse the meaning of the flags from indicating values of the "good" DQ flags to indicating the "bad" DQ flags, prepend '~' to the string value. For example, in order to exclude pixels with DQ flags 4 and 8 for computations and to consider as "good" all other pixels (regardless of their DQ flag), use a value of ~4+8, or ~4,8. A string value of ~0 would be equivalent to a setting of None.

  Default value (0) will make *all* pixels with non-zero DQ values be considered "bad" pixels, and the corresponding data pixels will be assigned zero weight and thus these pixels will not contribute to the output resampled data array.

  Set *good_bits* to *None* to turn off the use of model's DQ array.

  For more details, see documentation for *astropy.nddata.bitmask.extend_bit_flag_map*.
- **enable_ctx** (*bool, optional*) – Indicates whether to create a context image. If disable_ctx is set to *True*, parameters out_ctx, begin_ctx_id, and max_ctx_id will be ignored.
- **enable_var** (*bool, optional*) – Indicates whether to resample variance arrays.

- **compute_err** (*str or None, optional*) – Options are "from_var" or "driz_err":
  - "from_var": compute output model's error array from all (Poisson, flat, readout) resampled variance arrays. Setting compute_err to "from_var" will assume enable_var was set to *True* regardless of actual value of the parameter enable_var.
  - "driz_err": compute output model's error array by drizzling together all input models' error arrays.

  Error array will be assigned to 'err' key of the output model.

  Note

  At this time, output error array is not equivalent to error propagation results.

- **propagate_dq** (*bool, optional*) – Indicates whether to propagate DQ flags from input models to the output model. DQ flags are propagated by bitwise OR of all input DQ flags that contribute to a given output pixel. If *True*, output model will have a DQ array with the same shape as the output data array. If *False*, output model will not have a DQ array.
- **pixmap_stepsize** (*int, optional*) – If pixmap_stepsize>1, when computing pixel map used for resampling, perform the full WCS calculation on a sparser grid and use interpolation to fill in the rest of the pixels. This option speeds up pixel map computation by reducing the number of WCS calls, though at the cost of reduced pixel map accuracy. The loss of accuracy is typically negligible if the underlying distortion correction is smooth, but if the distortion is non-smooth, pixmap_stepsize>1 is not recommended. Large pixmap_stepsize values are automatically reduced to no more than 1/10 of image size. Default 1.
- **pixmap_order** (*int, optional*) – Order of the 2D spline to interpolate the sparse pixel mapping if pixmap_stepsize>1. Supported values are: 1 (bilinear) or 3 (bicubic). This parameter is ignored when pixmap_stepsize <= 1. Default 1.

Attributes Summary

compute_err	Indicates whether error array is computed and how it is computed.
dq_flag_name_map	
enable_ctx	Indicates whether context array is enabled.
enable_var	Indicates whether variance arrays are resampled.
error_from_variances	
group_ids	List of all group IDs of models resampled and added to the output model.
output_array_shape	Shape of the output model arrays.
output_array_types	
output_model	Output (resampled) model.
output_pixel_scale	Get pixel scale of the output model in arcsec.
output_wcs	WCS of the output (resampled) model.
pixel_scale_ratio	Get the ratio of the output pixel scale to the input pixel scale.
propagate_dq	Indicates whether DQ flags are propagated and output model has DQ array.
variance_array_names	

Methods Summary

add_model(model)	Resample a model.
add_model_hook(model, pixmap, ...)	Perform additional processing while resampling.
check_output_wcs(output_wcs[, ...])	Check output WCS.
create_output_model()	Create a new "output model": a dictionary of data and meta fields.
finalize()	Finalize computations.
finalize_resample_variance(output_model)	Finalize variance calculations.
finalize_time_info()	Perform final computations for the total time and update relevant fields of the output model.
get_input_model_pixel_area(model[, prefer_mean])	Compute input pixel area.
get_output_model_pixel_area(model)	Compute output pixel area.
init_time_counters()	Initialize variables/arrays needed to process exposure time.
init_variance_arrays()	Allocate arrays that hold co-added resampled variances and their weights.
is_finalized()	Check if `output_model` has been finalized.
resample_variance_arrays(model, pixmap, ...)	Resample variance arrays.
reset_arrays([n_input_models])	Reset intermediate arrays.
update_time(model)	Update time calculations.
validate_input_model(model)	Validate input model.

Attributes Documentation

compute_err

Indicates whether error array is computed and how it is computed.

dq_flag_name_map = None

enable_ctx

Indicates whether context array is enabled.

enable_var

Indicates whether variance arrays are resampled.

error_from_variances = None

group_ids

List of all group IDs of models resampled and added to the output model.

output_array_shape

Shape of the output model arrays.

output_array_types: defaultdict[str, DTypeLike] = {'con': <class 'numpy.int32'>, 'data': <class 'numpy.float32'>, 'err': <class 'numpy.float32'>, 'var_flat': <class 'numpy.float32'>, 'var_poisson': <class 'numpy.float32'>, 'var_rnoise': <class 'numpy.float32'>, 'wht': <class 'numpy.float32'>}

output_model

Output (resampled) model.

output_pixel_scale

Get pixel scale of the output model in arcsec.

output_wcs

WCS of the output (resampled) model.

pixel_scale_ratio

Get the ratio of the output pixel scale to the input pixel scale.

propagate_dq

Indicates whether DQ flags are propagated and output model has DQ array.

variance_array_names = ['var_rnoise', 'var_flat', 'var_poisson']

Methods Documentation

add_model(*model*)

Resample a model.

Resamples model image, variance data (if `enable_var` is *True*) , and error data (if `enable_err` is *True*), and adds them to the corresponding arrays of the output model using appropriate weighting. It also updates the weight array and context array (if `enable_ctx` is *True*) of the resampled data, as well as relevant metadata.

Whenever `model` has a unique group ID that was never processed before, the "pointings" value of the output model is incremented and the "group_id" attribute is updated. Also, time counters are updated with new values from the input `model` by calling `update_time()` .

Parameters

model (*dict*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines.

add_model_hook(*model*, *pixmap*, *pixel_scale_ratio*, *iscale*, *weight_map*, *xmin*, *xmax*, *ymin*, *ymax*)

Perform additional processing while resampling.

A hook method called by the `add_model()` method. It allows subclasses perform additional processing at the time the `model["data"]` array is resampled.

This method is called immediately after `model["data"]` is resampled.

Parameters

- **model** (*dict*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines.
- **pixmap** (*np.ndarray*) – A mapping (3D array) from input image (`data`) coordinates to resampled (`out_img`) coordinates. `pixmap` must be an array of shape `(Ny, Nx, 2)` where `(Ny, Nx)` is the shape of the input image. `pixmap[..., 0]` forms a 2D array of X-coordinates of input pixels in the output frame and `pixmap[..., 1]` forms a 2D array of Y-coordinates of input pixels in the output coordinate frame.
- **pixel_scale_ratio** (*float*) – Pixel scale ratio defined as the ratio of the output pixel scale to the first input model's pixel scale computed from `model` WCS at the fiducial point (taken as the `ref_ra` and `ref_dec` from the `wcsinfo` meta attribute of the first input image).
- **iscale** (*float*) – The scale to apply to the input variance data before drizzling.
- **weight_map** (*numpy.ndarray, None, optional*) – A 2D numpy array containing the pixel by pixel weighting. Must have the same dimensions as `data`.

When `weight_map` is *None*, the weight of input data pixels will be assumed to be 1.

- **xmin** (*float, optional*) – This and the following three parameters set a bounding rectangle on the input image. Only pixels on the input image inside this rectangle will have their flux added to the output image. Xmin sets the minimum value of the x dimension. The x dimension is the dimension that varies quickest on the image. If the value is zero, no minimum will be set in the x dimension. All four parameters are zero based, counting starts at zero.
- **xmax** (*float, optional*) – Sets the maximum value of the x dimension on the bounding box of the input image. If the value is zero, no maximum will be set in the x dimension, the full x dimension of the output image is the bounding box.
- **ymin** (*float, optional*) – Sets the minimum value in the y dimension on the bounding box. The y dimension varies less rapidly than the x and represents the line index on the input image. If the value is zero, no minimum will be set in the y dimension.
- **ymax** (*float, optional*) – Sets the maximum value in the y dimension. If the value is zero, no maximum will be set in the y dimension, the full x dimension of the output image is the bounding box.

check_output_wcs(*output_wcs*, *estimate_output_shape=True*)

Check output WCS.

Check that provided WCS has expected properties and that its `array_shape` property is defined. May modify `output_wcs`.

Parameters

- **output_wcs** (*gwcs.wcs.WCS*) – A WCS object corresponding to the output (resampled) image.
- **estimate_output_shape** (*bool, optional*) – Indicates whether to *estimate* output image shape of the `output_wcs` from other available attributes such as `bounding_box` when `output_wcs.array_shape` is *None*. If `estimate_output_shape` is *True* and `output_wcs.array_shape` is *None*, upon return `output_wcs.array_shape` will be assigned an estimated value.

create_output_model()

Create a new "output model": a dictionary of data and meta fields.

Returns

output_model – A dictionary of data model attributes and values.

Return type

dict

finalize()

Finalize computations.

Performs final computations from any intermediate values, sets output model values, and optionally frees temporary/intermediate objects.

`finalize` calls *finalize_resample_variance()* and *finalize_time_info()*.

> **Warning**
>
> Once the resample process has been finalized, adding new models to the output resampled model is not allowed.

finalize_resample_variance(*output_model*)

Finalize variance calculations.

Compute variance for the resampled image from running sums and weights. Free memory that holds these running sums and weights arrays.

output_model
[dict, None] A dictionary containing data arrays and other attributes that will be used to add new models to. When `accumulate` is *False*, only the WCS object of the model will be used. When `accumulate` is *True*, new models will be added to the existing data in the `output_model`.

finalize_time_info()

Perform final computations for the total time and update relevant fields of the output model.

get_input_model_pixel_area(*model*, *prefer_mean=True*)

Compute input pixel area.

Computes or retrieves pixel area of an input model. By default, this is the average pixel area of the input model's pixels within either the bounding box (if available) or the entire data array. Alternatively, this is the "nominal" pixel area as provided by the `"pixelarea_steradians"` keyword of the input model.

Subclasses can override this method to return the most appropriate pixel area value.

Parameters

- **model** (*dict, None*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines. In particular, it must have a keyword "wcs" and a WCS associated with it.
- **prefer_mean** (*bool, optional*) – If *True*, computes the mean pixel area of the model's pixels within either the bounding box (if available) or the entire data array. If this fails, it will fall back to the value of the `"pixelarea_steradians"` keyword of the input model. If *False*, returns the "nominal" pixel area as provided by the `"pixelarea_steradians"` keyword of the input model and if this is *None*, it will return mean pixel area.

Returns
pix_area – Pixel area in steradians.

Return type
float, None

get_output_model_pixel_area(*model*)

Compute output pixel area.

Computes or retrieves pixel area of the output model. Currently, this is the average pixel area of the model's pixels within either the bounding box (if available) or the entire data array.

Parameters
model (*dict, None*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines. In particular, it must have a keyword "wcs" and a WCS associated with it.

Returns
pix_area – Pixel area in steradians.

Return type
float

init_time_counters()

Initialize variables/arrays needed to process exposure time.

init_variance_arrays()

Allocate arrays that hold co-added resampled variances and their weights.

is_finalized()

Check if `output_model` has been finalized.

Indicates whether all attributes of the `output_model` have been computed from intermediate (running) values.

resample_variance_arrays(*model*, *pixmap*, *pixel_scale_ratio*, *iscale*, *weight_map*, *xmin*, *xmax*, *ymin*, *ymax*)

Resample variance arrays.

Resample and co-add variance arrays using appropriate weights and update total weights.

Parameters

- **model** (*dict*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines.
- **pixmap** (*np.ndarray*) – A mapping (3D array) from input image (`data`) coordinates to resampled (`out_img`) coordinates. `pixmap` must be an array of shape `(Ny, Nx, 2)` where `(Ny, Nx)` is the shape of the input image. `pixmap[..., 0]` forms a 2D array of X-coordinates of input pixels in the output frame and `pixmap[..., 1]` forms a 2D array of Y-coordinates of input pixels in the output coordinate frame.
- **pixel_scale_ratio** (*float*) – Pixel scale ratio defined as the ratio of the output pixel scale to the first input model's pixel scale computed from this model's WCS at the fiducial point (taken as the `ref_ra` and `ref_dec` from the `wcsinfo` meta attribute of the first input image).
- **iscale** (*float*) – The scale to apply to the input variance data before drizzling.
- **weight_map** (*numpy.ndarray, None, optional*) – A 2D `numpy` array containing the pixel by pixel weighting. Must have the same dimensions as `data`.

  When `weight_map` is *None*, the weight of input data pixels will be assumed to be 1.
- **xmin** (*float, optional*) – This and the following three parameters set a bounding rectangle on the input image. Only pixels on the input image inside this rectangle will have their flux added to the output image. Xmin sets the minimum value of the x dimension. The x dimension is the dimension that varies quickest on the image. If the value is zero, no minimum will be set in the x dimension. All four parameters are zero based, counting starts at zero.
- **xmax** (*float, optional*) – Sets the maximum value of the x dimension on the bounding box of the input image. If the value is zero, no maximum will be set in the x dimension, the full x dimension of the output image is the bounding box.
- **ymin** (*float, optional*) – Sets the minimum value in the y dimension on the bounding box. The y dimension varies less rapidly than the x and represents the line index on the input image. If the value is zero, no minimum will be set in the y dimension.
- **ymax** (*float, optional*) – Sets the maximum value in the y dimension. If the value is zero, no maximum will be set in the y dimension, the full x dimension of the output image is the bounding box.

reset_arrays(*n_input_models=None*)

Reset intermediate arrays.

Initialize/reset *Drizzle* objects, output model and arrays, and time counters and clears the "finalized" flag. Output WCS and shape are not modified from *Resample* object initialization. This method needs to be called before calling `add_model()` for the first time after `finalize()` was called.

Parameters
n_input_models (*int, None, optional*) – Number of input models expected to be resampled. When provided, this is used to estimate memory requirements and optimize memory allocation for the context array.

update_time(*model*)

Update time calculations.

A method called by the `add_model()` method to process each image's time attributes *only* when `model` has a new group ID.

Parameters
model (*dict*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines.

validate_input_model(*model*)

Validate input model.

Checks that `model` has all the required keywords needed for processing based on settings used during initialisation if the *Resample* object.

Parameters
model (*dict*) – A dictionary containing data arrays and other meta attributes and values of actual models used by pipelines.

Raises
KeyError – A *KeyError* is raised when `model` does not have a required keyword.

UnsupportedWCSError

exception stcal.resample.**UnsupportedWCSError**

Unsupported WCS Error.

Raised when provided output WCS has an unexpected number of axes or has an unsupported structure.

Class Inheritance Diagram

UnsupportedWCSError

Resample

1.1.7 Saturation

Description

Saturation Checking

The *flag_saturated_pixels()* routine flags pixels at or below the A/D floor or above the saturation threshold. Pixel values are flagged as saturated if the pixel value is larger than the defined saturation threshold. Pixel values are flagged as below the A/D floor if they have a value of zero DN or less.

The method loops over all integrations within an exposure, examining each one group-by-group, comparing the pixel values in the `data` array with defined saturation thresholds for each pixel. When it finds a pixel value in a given group that is above the saturation threshold (high saturation), it sets the "SATURATED" flag (as defined by the input `dqflags` dictionary) in the corresponding location of the input `gdq` array. When it finds a pixel in a given group that has a zero or negative value (below the A/D floor), it sets the "AD_FLOOR" and "DO_NOT_USE" flags in the corresponding location of the input `gdq` array. exposure. For the saturation case, it also flags all subsequent groups for that pixel as saturated. For example, if there are 10 groups in an integration and group 7 is the first one to cross the saturation threshold for a given pixel, then groups 7 through 10 will all be flagged for that pixel.

Pixels with thresholds set to NaN or flagged as "NO_SAT_CHECK" in the `sat_dq` array have their thresholds set above the 16-bit A-to-D converter limit of 65535 and hence will never be flagged as saturated. The "NO_SAT_CHECK" flag is propagated to the pixel data quality (`pdq`) array output to indicate which pixels fall into this category.

If the optional `read_pattern` input is provided, this method will use information about the read pattern to find pixels that saturated in the middle of grouped data. This can be particularly important for flagging data that saturated during the second group but did not trigger the normal saturation threshold due to the grouped data averaging. To trigger second group saturation in a pixel all three of the following criteria must be met:

1. The count rate estimated from the first group is not expected to saturate by the third group (as estimated by the difference between the first group counts and the superbias if available), which may occur for bright sources.
2. The difference in counts between the first and second group is larger than the remaining counts needed to saturate divided by the number of frames in the second group, i.e., the expected frame-averaged counts of a saturating signal that occurs in the last frame of the group.
3. The third group is saturated.

Charge Migration

There is an effect in IR detectors that results in charge migrating (spilling) from a pixel that has "hard" saturation (i.e. where the pixel no longer accumulates charge) into neighboring pixels. This results in non-linearities in the accumulating signal ramp in the neighboring pixels and hence the ramp data following the onset of saturation is not usable.

The *flag_saturated_pixels()* routine accounts for charge migration by flagging - as saturated - all pixels neighboring a pixel that goes above the saturation threshold. This is accomplished by first flagging all pixels that cross their saturation thresholds and then making a second pass through the data to flag neighbors within a specified region. The region of neighboring pixels is specified as a 2N+1 pixel wide box that is centered on the saturating pixel and N is set by the input parameter `n_pix_grow_sat`. The default value is 1, resulting in a 3x3 box of neighboring pixels that will be flagged.

stcal.saturation Package

Functions

flag_saturated_pixels(data, gdq, pdq, ...[, ...])	Flag saturated pixels.

flag_saturated_pixels

`stcal.saturation.flag_saturated_pixels`(*data*, *gdq*, *pdq*, *sat_thresh*, *sat_dq*, *atod_limit*, *dqflags*, *n_pix_grow_sat=1*, *zframe=None*, *read_pattern=None*, *bias=None*)

Flag saturated pixels.

Apply flagging for saturation based on threshold values stored in the saturation reference file data *sat_thresh* and A/D floor based on testing for 0 DN values. For A/D floor flagged groups, the DO_NOT_USE flag is also set.

Some input arrays may be provided as 2-D arrays, intended to match all integrations and all groups, or 4-D arrays that may have different values for each integration and group. 4-D arrays are intended to support multi-striping modes which may sample different detector regions in each integration.

Parameters

- **data** (*np.ndarray*) – 4-D science array (nint, ngroup, ny, nx).
- **gdq** (*np.ndarray*) – 4-D group dq array (nint, ngroup, ny, nx).
- **pdq** (*np.ndarray*) – 2-D pixel dq array matching the image dimensions (ny, nx). Alternately, a 4-D array may be provided, matching the data dimensions (nint, ngroup, ny, nx).
- **sat_thresh** (*np.ndarray*) – 2-D pixel-wise threshold for saturation, matching the image dimensions (ny, nx). Alternately, a 4-D array may be provided, matching the data dimensions (nint, ngroup, ny, nx).
- **sat_dq** (*np.ndarray*) – Data quality flags associated with *sat_thresh*.
- **atod_limit** (*int*) – Hard DN limit of 16-bit A-to-D converter.
- **dqflags** (*dict*) – A dictionary with at least the following keywords: DO_NOT_USE, SATURATED, AD_FLOOR, NO_SAT_CHECK providing a mapping between flag names and their integer values.
- **n_pix_grow_sat** (*int*) – Number of pixels that each flagged saturated pixel should be 'grown', to account for charge spilling. Default is 1.
- **zframe** (*np.ndarray*) – The ZEROFRAME 3-D array.
- **read_pattern** (*List[List[float or int]] or None*) – The times or indices of the frames composing each group.
- **bias** (*np.ndarray*) – 2-D superbias array (ny, nx) for use in group 2 saturation flagging for frame-averaged groups. Alternately, a 4-D array may be provided, matching the data dimensions (nint, ngroup, ny, nx).

Returns

- **gdq** (*np.ndarray*) – Updated 4-D group dq array
- **pdq** (*np.ndarray*) – Updated pixel dq array

1.1.8 Skymatch

Description

Overview

The *stcal.skymatch* module can be used to compute sky values in a collection of input images that contain both sky and source signal. The sky values can be computed for each image separately or in a way that matches the sky levels amongst the collection of images so as to minimize their differences. This operation is typically applied before doing cosmic-ray rejection and combining multiple images into a mosaic. When running *skymatch.skymatch.skymatch* in a

matching mode, it compares *total* signal levels in *the overlap regions* of a set of input images and computes the signal offsets either for each image *or a set/group of images* (see Image Groups section below) that will minimize – in a least squares sense – the residuals across the entire set. This comparison is performed directly on the input images without resampling them onto a common grid. The overlap regions are computed directly on the sky (celestial sphere) for each pair of input images. Matching based on total signal level is especially useful for images that are dominated by large, diffuse sources, where it is difficult – if not impossible – to find and measure true sky.

Note that the meaning of "sky background" depends on the chosen sky computation method. When the matching method is used, for example, the reported "sky" value is only the offset in levels between images and does not necessarily include the true sky level.

Note

Throughout this document the term "sky" is used in a generic sense, referring to any kind of non-source background signal, which may include actual sky, as well as instrumental (e.g. thermal) background, etc.

Sky background

Some components (e.g., in-field zodiacal light) result in reproducible background structures in all detectors when they are exposed simultaneously, while other components (e.g. stray light, thermal emission) can produce varying background from one exposure to the next exposure. The type of background structure that dominates a particular dataset affects the optimal way to group images for skymatch.

Image Groups

When computing and matching sky background on a set of input images, a *single sky level* (or offset, depending on selected `skymethod`) can be computed either for each input image or for groups of two or more input images.

When background is dominated by zodiacal light, images taken at the same time can be sky matched together; that is, a single background level can be computed and applied to all these images because we can assume that for the next exposure we will get a similar background structure, albeit with an offset level (common to all images in an exposure). Using grouped images with a common background level offers several advantages: more data are used to compute the single sky level, and the background level of images that do not overlap individually with any other images in other exposures can still be adjusted (as long as they belong to a group).

Algorithms

The *stcal.skymatch.skymatch* module provides several methods for constant sky background value computations.

The first method, called "local", essentially is an enhanced version of the original sky subtraction method used in older versions of AstroDrizzle. This method simply computes the mean/median/mode/etc. value of the sky separately in each input image. This method was upgraded to be able to use masks to remove bad pixels from being used in the computations of sky statistics.

In addition to the classic "local" method, two other methods have been introduced: "global" and "match", as well as a combination of the two – "global+match".

1. The "global" method essentially uses the "local" method to first compute a sky value for each image separately, and then assigns the minimum of those results to all images in the collection. Hence after subtraction of the sky values only one image will have a net sky of zero, while the remaining images will have some small positive residual.

2. The "match" algorithm computes only a correction value for each image, such that, when applied to each image, the mismatch between *all* pairs of images is minimized, in the least-squares sense. For each pair of images, the sky mismatch is computed *only* in the regions in which the two images overlap on the sky.

This makes the "match" algorithm particularly useful for equalizing sky values in large mosaics in which one may have only pair-wise intersection of adjacent images without having a common intersection region (on the sky) in all images.

Note that if the argument "match_down=True", matching will be done to the image with the lowest sky value, and if "match_down=False" it will be done to the image with the highest value (see *stcal.skymatch.skymatch* for full details).

3. The "global+match" algorithm combines the "global" and "match" methods. It uses the "global" algorithm to find a baseline sky value common to all input images and the "match" algorithm to equalize sky values among images. The direction of matching (to the lowest or highest) is again controlled by the "match_down" argument.

In the "local" and "global" methods, which find sky levels in each image, the calculation of the image statistics takes advantage of sigma clipping to remove contributions from isolated sources. This can work well for accurately determining the true sky level in images dominated by pure background such as sparse star fields with few extended sources that do not dominate the scene. The "local" method would likely struggle in scenes dominated by extended non-uniform sources such as nebulae, large/extended galaxies, etc. The "match" algorithm, on the other hand, compares the *total* signal levels integrated over regions of overlap in each image pair. This method can produce better results when there are no large empty regions of sky in the images. This method cannot measure the true sky level, but instead provides additive corrections that can be used to equalize the signal between overlapping images.

Examples

To get a better idea of the behavior of these different methods, the tables below show the results for two hypothetical sets of images. The first example is for a set of 6 images that form a 2x3 mosaic, with every image having overlap with its immediate neighbors. The first column of the table gives the actual (fake) sky signal that was imposed in each image, and the subsequent columns show the results computed by each method. All results are for the case where the argument `match_down = True`, which means matching is done to the image with the lowest sky value. Note that these examples are for the highly simplistic case where each example image contains nothing but the constant sky value. Hence the sky computations are not affected at all by any source content and are therefore able to determine the sky values exactly in each image. Results for real images will of course not be so exact.

Sky	Local	Global	Match	Global+Match
100	100	100	0	100
120	120	100	20	120
105	105	100	5	105
110	110	100	10	110
105	105	100	5	105
115	115	100	15	115

local
: finds the sky level of each image independently of the rest.

global
: uses the minimum sky level found by "local" and applies it to all images.

match
: with "match_down=True" finds the offset needed to match all images to the level of the image with the lowest sky level.

global+match
: with "match_down=True" finds the offsets and global value needed to set all images to a sky level of zero. In this trivial example, the results are identical to the "local" method.

The second example is for a set of 7 images, where the first 4 form a 2x2 mosaic, with overlaps, and the second set of 3 images forms another mosaic, with internal overlap, but the 2 mosaics do *NOT* overlap one another.

Sky	Local	Global	Match	Global+Match
100	100	90	0	86.25
120	120	90	20	106.25
105	105	90	5	91.25
110	110	90	10	96.25
95	95	90	8.75	95
90	90	90	3.75	90
100	100	90	13.75	100

In this case, the "local" method again computes the sky in each image independently of the rest, and the "global" method sets the result for each image to the minimum value returned by "local". The matching results, however, require some explanation. With "match" only, all of the results give the proper offsets required to equalize the images contained within each mosaic, but the algorithm does not have the information needed to match the two (non-overlapping) mosaics to one another. Similarly, the "global+match" results again provide proper matching within each mosaic, but will leave an overall residual in one of the mosaics.

Limitations and Discussions

As stated above, the best sky computation method depends on the nature of the data in the input images. If the input images contain mostly compact, isolated sources, the "local" and "global" algorithms can do a good job at finding the true sky level in each image. If the images contain large, diffuse sources, the "match" algorithm is more appropriate, assuming of course there is sufficient overlap between images from which to compute the matching values. In the event there is not overlap between all of the images, as illustrated in the second example above, the "match" method can still provide useful results for matching the levels within each non-contigous region covered by the images, but will not provide a good overall sky level across all of the images. In these situations it is more appropriate to either process the non-contiguous groups independently of one another or use the "local" or "global" methods to compute the sky separately in each image. The latter option will of course only work well if the images are not domimated by extended, diffuse sources.

The primary reason for introducing the `skymatch` algorithm was to try to equalize the sky in large mosaics in which computation of the absolute sky is difficult, due to the presence of large diffuse sources in the image. As discussed above, the `skymatch` code accomplishes this by comparing the sky values in the overlap regions of each image pair. The quality of sky matching will obviously depend on how well these sky values can be estimated. True background may not be present at all in some images, in which case the computed "sky" may be the surface brightness of a large galaxy, nebula, etc.

Here is a brief list of possible limitations and factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

1. Because sky computation is performed on *flat-fielded* but *not distortion corrected* images, it is important to keep in mind that flat-fielding is performed to obtain correct surface brightnesses. Because the surface brightness of a pixel containing a point-like source will change inversely with a change to the pixel area, it is advisable to mask point-like sources through user-supplied mask files. Values different from zero in user-supplied masks indicate good data pixels. Alternatively, one can use the `upper` parameter to exclude the use of pixels containing bright objects when performing the sky computations.

2. The input images may contain cosmic rays. This algorithm does not perform CR cleaning. A possible way of minimizing the effect of the cosmic rays on sky computations is to use clipping (`nclip` > 0) and/or set the `upper` parameter to a value larger than most of the sky background (or extended sources) but lower than the values of most CR-affected pixels.

3. In general, clipping is a good way of eliminating bad pixels: pixels affected by CR, hot/dead pixels, etc. However, for images with complicated backgrounds (extended galaxies, nebulae, etc.), affected by CR and noise, the clipping process may mask different pixels in different images. If variations in the background are too strong,

clipping may converge to different sky values in different images even when factoring in the true difference in the sky background between the two images.

4. In general images can have different true background values (we could measure it if images were not affected by large diffuse sources). However, arguments such as `lower` and `upper` will apply to all images regardless of the intrinsic differences in sky levels.

stcal.skymatch.skystatistics Module

skystatistics module provides statistics computation class.

Used by `skymatch()` and `SkyImage`.

Classes

`SkyStats`([skystat, lower, upper, nclip, ...])	Class built on top of `stsci.imagestats.ImageStats`.

SkyStats

class `stcal.skymatch.skystatistics.SkyStats`(*skystat='mean'*, *lower=None*, *upper=None*, *nclip=5*, *lsig=4.0*, *usig=4.0*, *binwidth=0.1*)

Bases: `object`

Class built on top of `stsci.imagestats.ImageStats`.

Deligates its functionality to calls to the `ImageStats` object. Compared to `stsci.imagestats.ImageStats`, *SkyStats* has "persistent settings" in the sense that object's parameters need to be set once and these settings will be applied to all subsequent computations on different data.

Initialize the SkyStats object.

Parameters

- **skystat** (*optional*) – possible values are 'mean', 'median', 'mode', 'midpt''. Sets the statistics that will be returned by *~stcal.skymatch.skystatistics.SkyStats.calc_sky*. The following statistics are supported: 'mean', 'mode', 'midpt', and 'median'. First three statistics have the same meaning as in stsdas.toolbox.imgtools.gstatistics while 'median' will compute the median of the distribution.
- **lower** (*float, None, optional*) – Lower limit of usable pixel values for computing the sky. This value should be specified in the units of the input image(s).
- **upper** (*float, None, optional*) – Upper limit of usable pixel values for computing the sky. This value should be specified in the units of the input image(s).
- **nclip** (*int, optional*) – A non-negative number of clipping iterations to use when computing the sky value.
- **lsig** (*float, optional*) – Lower clipping limit, in sigma, used when computing the sky value.
- **usig** (*float, optional*) – Upper clipping limit, in sigma, used when computing the sky value.
- **binwidth** (*float, optional*) – Bin width, in sigma, used to sample the distribution of pixel brightness values in order to compute the sky background statistics.

Methods Summary

__call__(data)	Call self as a function.
calc_sky(data)	Compute statistics on data.

Methods Documentation

__call__(*data*)

Call self as a function.

calc_sky(*data*)

Compute statistics on data.

Parameters

data (*numpy.ndarray*) – A numpy array of values for which the statistics needs to be computed.

Returns

statistics – A tuple of two values: (*skyvalue*, *npix*), where *skyvalue* is the statistics specified by the *skystat* parameter during the initialization of the *SkyStats* object and *npix* is the number of pixels used in computing the statistics reported in *skyvalue*.

Return type

tuple

Class Inheritance Diagram

SkyStats

stcal.skymatch.skyimage Module

The `skyimage` module contains algorithms that are used by `skymatch`.

Manage all of the information for footprints (image outlines) on the sky as well as perform useful operations on these outlines such as computing intersections and statistics in the overlap regions.

Classes

SkyImage(image, mask, wcs_fwd, wcs_inv, skystat)	Container that holds information about properties of a *single* image.
SkyGroup(images[, sky_id])	Collection of *SkyImage* objects.

SkyImage

class stcal.skymatch.skyimage.**SkyImage**(*image*, *mask*, *wcs_fwd*, *wcs_inv*, *skystat*, *sky_id=None*, *stepsize=None*, *meta=None*)

Bases: `object`

Container that holds information about properties of a *single* image.

Including:

- image data;
- WCS of the chip image;
- bounding spherical polygon;
- id;
- pixel area;
- sky background value;
- sky statistics parameters;
- mask associated image data indicating "good" (1) data.

Initialize the SkyImage object.

Parameters

- **image** (`numpy.ndarray`) – A 2D array of image data.
- **mask** (`numpy.ndarray`) – A 2D array that indicates which pixels in the input *image* should be used for sky computations (1) and which pixels should **not** be used for sky computations (`0`).
- **wcs_fwd** (`collections.abc.Callable`) – "forward" pixel-to-world transformation function.
- **wcs_inv** (`collections.abc.Callable`) – "inverse" world-to-pixel transformation function.
- **skystat** (`collections.abc.Callable, None, optional`) – A callable object that takes a either a 2D image (2D *numpy.ndarray*) or a list of pixel values (a Nx1 array) and returns a tuple of two values: some statistics (e.g., mean, median, etc.) and number of pixels/values from the input image used in computing that statistics.
- **sky_id** (`Any`) – The value of this parameter is simple stored within the *SkyImage* object. While it can be of any type, it is preferable that *id* be of a type with nice string representation.
- **stepsize** (`int, None, optional`) – Spacing between vertices of the image's bounding polygon. Default value of *None* creates bounding polygons with four vertices corresponding to the corners of the image.
- **meta** (`dict, None, optional`) – A dictionary of various items to be stored within the *SkyImage* object.

Methods Summary

calc_sky([overlap, delta])	Compute sky background value.
intersection(skyimage)	Compute intersection.

Methods Documentation

calc_sky(*overlap=None*, *delta=True*)

Compute sky background value.

Parameters

- **overlap** (SkyImage, SkyGroup, *None*, *optional*) – This parameter is used to indicate that sky statistics should computed only in the region of intersection of *this* image with the *SkyImage* or *SkyGroup* indicated by *overlap*. When *overlap* is *None*, sky statistics will be computed over the entire image.
- **delta** (*bool*, *optional*) – Should this function return absolute sky value or the difference between the computed value and the value of the sky stored in the *sky* property.

Returns

- **skyval** (*float, None*) – Computed sky value (absolute or relative to the *sky* attribute). If there are no valid data to perform this computations (e.g., because this image does not overlap with the image indicated by *overlap*), *skyval* will be set to *None*.
- **npix** (*int*) – Number of pixels used to compute sky statistics.
- **polyarea** (*float*) – Area (in srad) of the polygon that bounds data used to compute sky statistics.

intersection(*skyimage*)

Compute intersection.

Compute intersection of this *SkyImage* object and another *SkyImage* or *SkyGroup* object.

Parameters

skyimage (SkyImage, SkyGroup) – Another object that should be intersected with this *SkyImage*.

Returns

polygon – A SphericalPolygon that is the intersection of this *SkyImage* and *skyimage*.

Return type

SphericalPolygon

SkyGroup

class stcal.skymatch.skyimage.**SkyGroup**(*images*, *sky_id=None*)

Bases: object

Collection of *SkyImage* objects.

Holds multiple *SkyImage* objects whose sky background values must be adjusted together.

SkyGroup provides methods for obtaining bounding polygon of the group of *SkyImage* objects and to compute sky value of the group.

Attributes Summary

sky	Sky background value.

Methods Summary

calc_sky([overlap, delta])	Compute sky background value.

Attributes Documentation

sky

Sky background value. See *calc_sky* for more details.

Methods Documentation

calc_sky(*overlap=None*, *delta=True*)

Compute sky background value.

Parameters

- **overlap** (SkyImage, SkyGroup, *None*, *optional*) – This parameter is used to indicate that sky statistics should computed only in the region of intersection of *this* image with the *SkyImage* or *SkyGroup* indicated by *overlap*. When *overlap* is *None*, sky statistics will be computed over the entire image.
- **delta** (*bool*, *optional*) – Should this function return absolute sky value or the difference between the computed value and the value of the sky stored in the *sky* property.

Returns

- **skyval** (*float, None*) – Computed sky value (absolute or relative to the *sky* attribute). If there are no valid data to perform this computations (e.g., because this image does not overlap with the image indicated by *overlap*), *skyval* will be set to *None*.
- **npix** (*int*) – Number of pixels used to compute sky statistics.
- **polyarea** (*float*) – Area (in srad) of the polygon that bounds data used to compute sky statistics.

Class Inheritance Diagram

SkyImage

SkyGroup

stcal.skymatch.skymatch Module

A module that provides functions for matching sky in overlapping images.

Authors
Mihai Cara

Functions

skymatch(images[, skymethod, match_down, ...])	Compute and/or "equalize" sky background in input images.

skymatch

`stcal.skymatch.skymatch.skymatch`(*images*, *skymethod='global+match'*, *match_down=True*, *subtract=False*)

Compute and/or "equalize" sky background in input images.

Note

Sky matching ("equalization") is possible only for **overlapping** images.

Parameters

- **images** (*list of* SkyImage *or* SkyGroup) – A list of of *SkyImage* or *SkyGroup* objects.
- **skymethod** (*str, optional*) – Available methods: {'local', 'global+match', 'global', 'match'} Select the algorithm for sky computation:
  - **'local'** : independently compute the sky background for each input (either image or group). This method does not involve any special treatment of overlaps between inputs.

    Note

    This setting is recommended when regions of overlap between images are dominated by "pure" sky (as opposed to extended, diffuse sources).

  - **'global'** : similar to *local* first compute an independent sky background for each input (either image or group), then take the minimum of these computed sky values. This minimum will be assigned to the sky value of all inputs. This method *may* be useful when the input images have been already matched.
  - **'match'** : compute pair-wise differences in sky values between inputs that overlap. The reported sky values will be relative to one of the input images (which will have a sky value of 0). This setting will "equalize" sky values between the images in large mosaics.
  - **'global+match'** : first compute sky values using the above match method, then compute a global sky value after accounting for the match-computed sky values. The final sky values will be a combination of the two methods.

    Note

This is the *recommended* setting for images containing diffuse sources (e.g., galaxies, nebulae) covering significant parts of the image.

- **match_down** (*bool, optional*) – Specifies whether the sky *differences* should be subtracted from images with higher sky values (*match_down = True*) to match the image with the lowest sky or sky differences should be added to the images with lower sky values to match the sky of the image with the highest sky value (*match_down = False*).

  > **Note**
  >
  > This setting applies *only* when the *skymethod* parameter is either *'match'* or *'global+match'*.

- **subtract** (*bool*) – Subtract computed sky value from image data. (Default: *False*).

Raises

`TypeError` – The *images* argument must be a Python list of `SkyImage` and/or `SkyGroup` objects.

Notes

`skymatch()` provides new algorithms for sky value computations and enhances previously available algorithms used by, e.g., astrodrizzle.

Two new methods of sky subtraction have been introduced (compared to the standard `'local'`): `'global'` and `'match'`, as well as a combination of the two – `'global+match'`.

- The `'global'` method computes the minimum sky value across *all* input images and/or groups. That sky value is then considered to be the background in all input images.
- The `'match'` algorithm is somewhat similar to the traditional sky subtraction method (*skymethod = 'local'*) in the sense that it measures the sky independently in input images (or groups). The major differences are that, unlike the traditional method,

  1. `'match'` algorithm computes *relative* (delta) sky values with regard to the sky in a reference image chosen from the input list of images; *and*
  2. Sky statistics are computed only in the part of the image that intersects other images.

  This makes the `'match'` sky computation algorithm particularly useful for "equalizing" sky values in large mosaics in which one may have only (at least) pair-wise intersection of images without having a common intersection region (on the sky) in all images.

  The *'match'* method works in the following way: for each pair of intersecting images, an equation is written that requires that average surface brightness in the overlapping part of the sky be equal in both images. The final system of equations is then solved for unknown background levels.

  > **Warning**
  >
  > The current algorithm is not capable of detecting cases where some subsets of intersecting images (from the input list of images) do not intersect at all with other subsets of intersecting images (except for the simple case when *single* images do not intersect any other images). In these cases the algorithm will find equalizing sky values for each intersecting subset of images and/or groups of images. However since these subsets of images do not intersect each other, sky will be matched only within each subset and the "inter-subset" sky mismatch could be significant.

Users are responsible for detecting such cases and adjusting processing accordingly.

- The `'global+match'` algorithm combines the `'match'` and `'global'` methods in order to overcome the limitation of the `'match'` method described in the note above: it uses the `'global'` algorithm to find a baseline sky value common to all input images and the `'match'` algorithm to "equalize" sky values in the mosaic. Thus, the sky value of the "reference" image will be equal to the baseline sky value (instead of 0 in `'match'` algorithm alone).

Remarks:

- `skymatch()` works directly on *geometrically distorted* flat-fielded images thus avoiding the need to perform distortion correction on the input images.

  Initially, the footprint of a chip in an image is approximated by a 2D planar rectangle representing the borders of chip's distorted image. After applying distortion model to this rectangle and projecting it onto the celestial sphere, it is approximated by spherical polygons. Footprints of exposures and mosaics are computed as unions of such spherical polygons while overlaps of image pairs are found by intersecting these spherical polygons.

Limitations and Discussions:

Primary reason for introducing "sky match" algorithm was to try to equalize the sky in large mosaics in which computation of the "absolute" sky is difficult due to the presence of large diffuse sources in the image. As discussed above, `skymatch()` accomplishes this by comparing "sky values" in a pair of images in the overlap region (that is common to both images). Quite obviously the quality of sky "matching" will depend on how well these "sky values" can be estimated. We use quotation marks around *sky values* because for some image "true" background may not be present at all and the measured sky may be the surface brightness of large galaxy, nebula, etc.

In the discussion below we will refer to parameter names in `SkyStats` and these parameter names may differ from the parameters of the actual *skystat* object passed to initializer of the `SkyImage`.

Here is a brief list of possible limitations/factors that can affect the outcome of the matching (sky subtraction in general) algorithm:

- Since sky subtraction is performed on *flat-fielded* but *not distortion corrected* images, it is important to keep in mind that flat-fielding is performed to obtain uniform surface brightness and not flux. This distinction is important for images that have not been distortion corrected. As a consequence, it is advisable that point-like sources be masked through the user-supplied mask files. Values different from zero in user-supplied masks indicate "good" data pixels. Alternatively, one can use *upper* parameter to limit the use of bright objects in sky computations.

- Normally, distorted flat-fielded images contain cosmic rays. This algorithm does not perform CR cleaning. A possible way of minimizing the effect of the cosmic rays on sky computations is to use clipping (*nclip* > 0) and/or set *upper* parameter to a value larger than most of the sky background (or extended source) but lower than the values of most CR pixels.

- In general, clipping is a good way of eliminating "bad" pixels: pixels affected by CR, hot/dead pixels, etc. However, for images with complicated backgrounds (extended galaxies, nebulae, etc.), affected by CR and noise, clipping process may mask different pixels in different images. If variations in the background are too strong, clipping may converge to different sky values in different images even when factoring in the "true" difference in the sky background between the two images.

- In general images can have different "true" background values (we could measure it if images were not affected by large diffuse sources). However, arguments such as *lower* and *upper* will apply to all images regardless of the intrinsic differences in sky levels.

1.1.9 Multiprocessing Utils

stcal.multiprocessing Module

Functions

compute_num_cores(max_cores, nchunks, ...)	Compute the number of chunks to be created for multiprocessing.

compute_num_cores

stcal.multiprocessing.**compute_num_cores**(*max_cores*, *nchunks*, *max_available*)

Compute the number of chunks to be created for multiprocessing.

Parameters

- **max_cores** (*str*) – Number of cores to use for multiprocessing. If set to 'none' (the default), then no multiprocessing will be done. The other allowable values are 'quarter', 'half', and 'all' and string integers. This is the fraction of cores to use for multi-proc.
- **nchunks** (*int*) – The total number of chunks that will be processed. If more cores are requested than chunks, then the number of chunks will be used as the output to make sure that each process has some data.
- **max_available** (*int*) – This is the total number of cores available. The total number of cores includes the SMT cores (Hyper Threading for Intel).

Returns

number_slices – The number of slices for multiprocessing.

Return type

int

1.1.10 Velocity Aberration Utils

stcal.velocity_aberration Module

Utilities for velocity aberration correction.

Functions

compute_va_effects_vector(velocity_x, ...)	Compute velocity aberration effects scale factor.
compute_va_effects(velocity_x, velocity_y, ...)	Compute velocity aberration effects.

compute_va_effects_vector

stcal.velocity_aberration.**compute_va_effects_vector**(*velocity_x*, *velocity_y*, *velocity_z*, *u*)

Compute velocity aberration effects scale factor.

Computes constant scale factor due to velocity aberration as well as corrected `RA` and `DEC` values, in vector form.

Parameters

- **velocity_x** (*float*) – The components of the velocity of JWST, in km / s with respect to the Sun. These are celestial coordinates, with x toward the vernal equinox, y toward right ascension 90 degrees and declination 0, z toward the north celestial pole.

- **velocity_y** (*float*) – The components of the velocity of JWST, in km / s with respect to the Sun. These are celestial coordinates, with x toward the vernal equinox, y toward right ascension 90 degrees and declination 0, z toward the north celestial pole.
- **velocity_z** (*float*) – The components of the velocity of JWST, in km / s with respect to the Sun. These are celestial coordinates, with x toward the vernal equinox, y toward right ascension 90 degrees and declination 0, z toward the north celestial pole.
- **u** (*numpy.ndarray*) – The vector form (`[u0, u1, u2]`) of right ascension and declination of the target (or some other point, such as the center of a detector) in the barycentric coordinate system. The equator and equinox should be the same as the coordinate system for the velocity.

Returns

- **scale_factor** (*float*) – Multiply the nominal image scale (e.g., in degrees per pixel) by this value to obtain the image scale corrected for the "aberration of starlight" due to the velocity of JWST with respect to the Sun.
- **u_corr** (*numpy.ndarray*) – Apparent position vector (`[ua0, ua1, ua2]`) in the moving telescope frame.

compute_va_effects

`stcal.velocity_aberration.compute_va_effects(velocity_x, velocity_y, velocity_z, ra, dec)`

Compute velocity aberration effects.

Computes constant scale factor due to velocity aberration as well as corrected `RA` and `DEC` values.

Parameters

- **velocity_x** (*float*) – The components of the velocity of JWST, in km / s with respect to the Sun. These are celestial coordinates, with x toward the vernal equinox, y toward right ascension 90 degrees and declination 0, z toward the north celestial pole.
- **velocity_y** (*float*) – The components of the velocity of JWST, in km / s with respect to the Sun. These are celestial coordinates, with x toward the vernal equinox, y toward right ascension 90 degrees and declination 0, z toward the north celestial pole.
- **velocity_z** (*float*) – The components of the velocity of JWST, in km / s with respect to the Sun. These are celestial coordinates, with x toward the vernal equinox, y toward right ascension 90 degrees and declination 0, z toward the north celestial pole.
- **ra** (*float*) – The right ascension and declination of the target (or some other point, such as the center of a detector) in the barycentric coordinate system. The equator and equinox should be the same as the coordinate system for the velocity.
- **dec** (*float*) – The right ascension and declination of the target (or some other point, such as the center of a detector) in the barycentric coordinate system. The equator and equinox should be the same as the coordinate system for the velocity.

Returns

- **scale_factor** (*float*) – Multiply the nominal image scale (e.g., in degrees per pixel) by this value to obtain the image scale corrected for the "aberration of starlight" due to the velocity of JWST with respect to the Sun.
- **apparent_ra** (*float*) – Apparent star position in the moving telescope frame.
- **apparent_dec** (*float*) – Apparent star position in the moving telescope frame.

CHAPTER

TWO

CHANGE LOG

2.1 Change Log

2.1.1 1.18.1.dev24+g4a4b38c72 (2026-06-11)

ramp_fitting step

- Fix memory leak of orig_gdq. (#539)
- Use explicit read times for likelihood fits with JWST data, if available in the input model. This is intended to support new multistripe modes with repeated in-frame reads of the same detector area. (#546)

resample step

- Corrected the "level" and "subtracted" keys in the output model dictionary to show correct values for the resampled image. (#542)

2.1.2 1.18.0 (2026-03-25)

linearity step

- Reduce memory usage of linearity correction, particularly for read-level corrections. (#531)

ramp_fitting step

- Fix ambiguous behavior in jump detection for noiseless data, where the chi squared improvement can be the same when dropping one and two group differences. (#520)

saturation step

- Allow 4D input arrays in the saturation step, matching the dimensions of the input data. This is intended for support of striping modes that may cover different areas of the detector in each integration. (#526)

tweakreg step

- Fix handling of catalogs with all sources containing valid proper motion. (#519)
- Force anonymous S3 access when fetching S3 hosted GAIA data. (#535)

2.1.3 1.17.0 (2026-02-13)

Breaking Changes

- Removed `alignment.util.reproject`, `outlier_detection.utils.reproject`, and `outlier_detection.utils.calc_gwcs_pixmap`. For the latter, use `resample.utils.calc_pixmap` instead. (#488)

- Change the returned structures from ramp fitting from long tuples of ndarrays to dictionaries with names generally matching those of FITS extensions. Downstream uses of these functions will break if they assume tuples (as jwst and romancal have done). (#496)
- Remove unused reduce_memory_usage SkyImage mode. (#504)
- Significantly changed the linearity correction API to support read-level correction with new parameters for inverse linearity coefficients, read patterns, and saturation values. (#506)
- Deprecated argument 'pix_ratio' removed from 'gwcs_blot' function in `stcal.outlier_detection.utils` module. (#509)

alignment step

- Moved `calc_pixmap` to the `resample` submodule. Removed `util.reproject` method. (#488)

jump step

- Implement the correct behavior of jump detection in the presence of unevenly sampled data (e.g. from Roman). (#454)

linearity step

- Added read-level linearity correction that accounts for averaging of multiple reads into resultants. (#506)

outlier_detection step

- Removed `outlier_detection.utils.reproject`, and `outlier_detection.utils.calc_gwcs_pixmap`. For the latter, use `resample.utils.calc_pixmap` instead. (#488)
- Fix: Do not rescale blotted images by pixel area ratio in outlier detection. This rescaling could have resulted in flagging of valid sources as cosmic rays. (#509)

ramp_fitting step

- Change the returned structures from ramp fitting from long tuples of ndarrays to dictionaries with names generally matching those of FITS extensions. (#496)
- Expose chi squared as calculated in likelihood-based ramp fitting in the output dictionaries of ramp results. (#508)

resample step

- Fix a bug in resample due to which the "turbo", "gaussian", and "lanczos" resample kernels were not properly rescaled by pixel scale ratio. (#418)
- Enhanced `calc_pixmap` version from `drizzle.utils` by adding interpolation options to speed up resampling calculations. Coordinate transformations can dominate the runtime of `resample` and, for well-behaved distortion corrections, computing pixmap by interpolating over fewer points significantly reduce computing time with negligible loss of accuracy depending on chosen order and step size. (#488)
- Added support for propagating DQ flags in resampling. DQ flags are propagated by bitwise OR of all input DQ flags that contribute to a given output pixel. (#516)

skymatch step

- Remove unused reduce_memory_usage SkyImage mode and other unused code. (#504)

tweakreg step

- Fix `get_catalog` and `create_astrometric_catalog` to only return sources with proper motion when an epoch is provided. (#493)

Other Changes

- Allow gwcs 1.0.1 and higher. (#507)

2.1.4 1.16.0 (2026-01-22)

Breaking Changes

- Remove deprcated dqflags, dynamicdq and basic_utils modules. (#443)
- Remove deprecated `wcs_from_footprints`. (#470)
- Make saturation helper functions adjacent_pixels and plane_saturation private (#489)
- Rename `testing_helpers` to `_testing.memory_threshold`. (#490)

alignment step

- Add methods for computing complex S_REGION keywords for mosaics (#407)

jump step

- Replaced usage of `opencv-python` with analogous functionality from `scikit-image`. (#138)
- Fixed a bug where `point_inside_ellipse()` sometimes returning True for a point outside of the given ellipse. (#461)
- `JumpData.max_extended_radius` is renamed to `JumpData.max_extended_width` to accurately reflect its actual usage internally. (#474)
- Replace `warnings.resetwarnings` with `warnings.catch_warnings` to avoid global changes to warning filters. (#483)

ramp_fitting step

- Replace `warnings.resetwarnings` with `warnings.catch_warnings` to avoid global changes to warning filters. (#483)
- Add snowball flagging (`jump.expand_large_events`) to likelihood-based ramp fitting and jump detection. (#491)

saturation step

- Set saturation flag for any saturation. This reverts back to saturation flagging behavior prior to #125 (#421)
- Add saturation.saturation.flag_saturated_pixels to docs and public API, add description of saturation step to docs (#489)

tweakreg step

- Add support for 'GAIADR3_S3' catalog in tweakreg. (#447)

Other Changes

- Make new stcal.multiprocessing namespace with public compute_num_cores function (#431)
- Add missing `stsci.imagestats` dependency. (#433)
- Disable 3.14t wheel building. This package does not yet support 3.14t. (#451)
- Pin gwcs<1 to avoid errors for None input frame. (#500)

2.1.5 1.15.2 (2025-09-25)

Bug Fixes

- Removed Gaia-only option from create_astrometric_catalog. (#406)
- Removed extra item from `twopoint_difference.find_crs` early return, to match number of expected items. (#410)

General

- Made GAIAREFCAT the default astrometric reference catalog. (#406)

2.1.6 1.15.1 (2025-09-24)

Changes to API

- Add inverse of sky variance as an option to weight type for drizzle. (#359)

Bug Fixes

- Fix memory management in ramp fitting C-extension. (#399)

General

- The changes are: (1) handle "ivm" and "ivm-sky" cases identically: for "ivm", calculate 1/var_rnoise, setting non-finite to 0, and if missing or shape mismatch, return all ones (equal weighting); for "ivm-sky", calculate 1/var_sky, setting non-finite to 0, and if missing or shape mismatch, return all ones (equal weighting); (2) always multiply the result by the DQ mask; (3) the fallback for a missing variance array is an array filled with ones (equal weighting). (#359)
- Add upper pin not allowing python 3.14. (#396)
- Improve memory usage of jump detection. (#403)

2.1.7 1.15.0 (2025-08-14)

Bug Fixes

- Fix a bug in `resample`, first introduced in https://github.com/spacetelescope/jwst/pull/7894, due which the intensity of resampled images was incorrectly adjusted to minimize the error in flux computations while using an incorrect procedure. (#352)

General

- test with latest version of Python (#346)
- For the likelihood algorithm for ramp fitting, use a separately downlinked zeroframe if available, independent of the remaining first group data. (#372)
- Use `gwcs.FITSImagingWCSTransform` in the WCS of Level3 files. (#378)

- Add catalog timeout parameter from user-level funcs to utility funcs (#384)
- Add utility functions for calculating velocity aberration. (#385)

2.1.8 1.14.0 (2025-06-18)

Bug Fixes

- Scale drizzled errors and variances by the input-to-output pixel size ratio to ensure proper error propagation for non-standard output pixel scales in imaging mode. Apply a similar scaling for spectroscopic modes, but using the square root of the input-to-output pixel size ratio. (#370)

General

- Allow resample to resample arbitrary variance arrays. (#364)
- Extrapolate dark reference integration to match or exceed number of science file frames for cases when provided dark is not long enough to cover science data. (#368)

2.1.9 1.13.0 (2025-05-12)

Changes to API

- Added `add_model_hook()` method to `resample.Resample` that allows subclasses to perform additional processing while reusing quantities computed by `add_model()`.

  Modified the behavior of `resample.utils.build_driz_weight()`: any value of the `weight_type` argument other than "ivm", "exptime", or `None` will raise a `ValueError` exception (previous behavior was to treat them as `None`). (#342)

Bug Fixes

- Bugfix catalog parsing for tweakreg absolute refcat (#355)
- [saturation] Account for non-zero bias in group 2 saturation flagging in frame-averaged groups. (#356)

General

- Refactor Poisson variance calculation in ols_cas22/_ramp.pyx to use a cumulative variable instead of a nested for loop, change some single precision quantities in ols_cas22/_ramp.pyx to double precision. (#325)
- Removing old and unused OLS python and GLS code from ramp fitting. (#358)

2.1.10 1.12.0 (2025-03-18)

Changes to API

- Added `resample` submodule. (#320)
- Refactoring the jump module to improve memory consumption, readability, and maintenance. (#330)
- Removed all references to the unused ramp error array in dark, jump, and ramp fitting steps. (#334)
- Make sregion_to_footprint part of public API (#345)

Bug Fixes

- Fix a bug in `alignment.util.wcs_from_sregions()` and `alignment.util.wcs_from_footprints()` functions that was causing incorrect WCS bounding boxes when the `crpix` argument was provided. (#326)
- Allow tweakreg to parse absolute reference catalogs with skycoord objects in them (#333)

- Fix a bug in the jump step with counting the number of usable groups. (#338)
- Do not reset existing JUMP flags at the start of the likelihood ramp fitting algorithm. (#339)
- Fix a bug in `jump.twopointdifference.calc_med_first_diffs()` where `argmax` was being used instead of `nanargmax`. (#344)
- Changed multiprocessing from forkserver to spawn. (#350)

General

- Move common parts of skymatch shared by both jwst and romancal into stcal. (#310)
- Drop support for Python 3.10 and add support for Python 3.13. (#327)
- Performance improvements for saturation step (#331)
- Performance improvements for jump step targeting both runtime and memory consumption. Results are mostly identical, but there are some differences in the MIRI shower flagging (it now flags somewhat fewer showers). (#337)
- Refactoring twopoint_difference.py for the jump step. (#340)

2.1.11 1.11.0 (2024-12-20)

Changes to API

- Add maximum_shower_amplitude parameter to MIRI cosmic rays showers routine to fix accidental flagging of bright science pixels. (#306)

Bug Fixes

- Do not evaluate the inverse WCS transform within the bounding box in cases where a resampled WCS is computed. Do not pass table columns to the WCS Shared API. (#314)
- For *ramp_fitting*, the *CRMAG* element was not originally implemented in the C-extension for ramp fitting. It is now implemented. A bug in the read noise recalculation for CHARGELOSS when using the multiprocessing option has been fixed. Further, in *JWST* regression tests have been added to test for multiprocessing to ensure testing for anything that could affect multiprocessing. (#318)
- Update weight threshold calculation in outlier detection to work around numpy bug that introduces small numerical differences for a mean of a masked array. (#319)
- Change flagging of 'pre-saturation' grouped data to use DO_NOT_USE instead of SATURATION flag to avoid confusing the snowball routine downstream. (#321)

2.1.12 1.10.0 (2024-11-15)

Changes to API

- Add *outlier_detection* median calculators from jwst. (#292)
- Deprecate wcs_from_footprints. Use wcs_from_sregions instead. (#307)
- Add wcs_from_sregions function to compute a combined WCS from a list of s_regions. (#307)

Bug Fixes

- Fix *IntEnum* saturation flag issue with numpy 2+ for romancal. (#305)
- Fix abs_deriv handling of off-edge and nan values. (#311)

General

- Added fillval option to `gwcs_blot` utility. (#291)
- Update downstream tests for jwst and romancal to fix pytest configurations. (#297)
- Changed the default *ramp fitting* CI test algorithm to `OLS_C`. This also revealed a bug in control flow that allowed for the CHARGELOSS recalculation in error, which resulted in a crash while attempting to dereference a `NULL` pointer. Further, when creating the optional results product, the object creation was changed to *PyArray_ZEROS* to ensure invalid data was set to zero. The use of *PyArray_EMPTY* does not initialize memory, so junk data could be in used array elements. (#298)
- Add infrastructure for testing memory usage (#299)
- Preparing ramp fitting for the upgrade to python 3.13. In python 3.13, the C-API function `PyLong_AsLong` raises an exception if the object passed to it is `NoneType`. There are two integer attributes for the `RampData` class that can be `NoneType`, so a check for `NoneType` for these attributes was added. (#303)

2.1.13 1.9.0 (2024-09-19)

Changes to API

- [ramp_fitting] Add the likelihood algorithm to ramp fitting. (#278)

Bug Fixes

- [saturation] Add option for using the readout pattern information to improve saturation flagging in grouped data. (#283)

General

- Add clip_accum parameter to alignment methods. (#286)
- Improve handling of catalog web service connectivity issues. (#286)

2.1.14 1.8.2 (2024-09-10)

Bug Fixes

- Implement byteorder swap method that is forward-compatible with numpy 2.0 in jwst ramp_fitting. (#282)
- [jump] Fix a logical bug in the jump step for usage of > vs >= per JP-3689. (#285)

General

- [ramp_fitting] Moving the read noise recalculation due to CHARGELOSS flagging from the JWST ramp fit step code into the STCAL ramp fit C-extension. (#275)

2.1.15 1.8.1 (2024-09-08)

Bug Fixes

- Fixed memory leak in C-extension. (#281)

General

- use `towncrier` to handle changelog entries (#284)

2.1.16 1.8.0 (2024-08-14)

General

- Add TweakReg submodule. [#267]

ramp_fitting

- Move the CHARGELOSS read noise variance recalculation from the JWST step code to the C extension to simplify the code and improve performance.[#275]

Changes to API

- Add `outlier_detection` submodule with `utils` included from jwst. [#270]

2.1.17 1.7.3 (2024-07-05)

Bug Fixes

ramp_fitting

- Fix bugs in the C algorithm Poisson variance calculation when provided with an average dark current. [#269]
- When OLS_C was selected as the ramp fitting algorithm with multiprocessing, the C extension was not called. The old python code was called. This bug has been fixed, so the C extension is properly run when selecting multiprocessing. [#268]

2.1.18 1.7.2 (2024-06-12)

General

- build with Numpy 2.0 release candidate [#260]

Bug Fixes

jump

- Flag asymmetrical snowballs that are missed by the current code (JP-3638). This was changed to not require that the center of the snowball jump ellipse is a saturated pixel. [#261]

2.1.19 1.7.1 (2024-05-21)

Bug Fixes

jump

- Catch some additional warnings about all-NaN slices. [#258]

ramp_fitting

- Fix a bug in Poisson variance calculation visible when providing an average dark current value in which the specified dark current was not converted to the appropriate units for pixels with negative slopes. This resulted in incorrect SCI, ERR, and VAR_POISSON values. Also required revising the approach for catching all-zero variance cases when average dark current was not specified. [#255]
- Refactor ramp fitting using a C extension to improve performance. [#156]

2.1.20 1.7.0 (2024-03-25)

Changes to API

jump

- Switch multiprocessing method to `fork_server`. [#249]

ramp_fitting

- Switch multiprocessing method to `fork_server`. [#249]

Bug Fixes

jump

- Updated the shower flagging code to mask reference pixels, require a minimum number of groups to trigger the detection, and use all integrations to determine the median value. [#248]

ramp_fitting

- Changed the data type of three variables that are used in measuring the jump free segments of integrations. The variables were uint8 and they would yield wrong results for integrations with more than 256 groups. [#251]
- Use `sqrtf` instead of `sqrt` in ols_cas22 ramp fitting with jump detection to avoid small numerical errors on different systems due to a cast to/from double. [#252]

Other

jump

- Enable the use of multiple integrations to find outliers. Also, when the number of groups is above a threshold, use single pass outlier flagging rather than the iterative flagging. [#242]
- Use `sqrtf` instead of `sqrt` in ols_cas22 ramp fitting with jump detection to avoid small numerical errors on different systems due to a cast to/from double. [#252]

2.1.21 1.6.1 (2024-02-29)

Changes to API

ramp_fitting

- Add `average_dark_current` to calculations of poisson variance. [#243]

2.1.22 1.6.0 (2024-02-15)

Changes to API

jump

- Add in the flagging of groups in the integration after a snowball occurs. The saturated core of the snowball gets flagged as jump for a number of groups passed in as a parameter [#238]

Bug Fixes

jump

- Fixed the computation of the number of rows per slice for multiprocessing, which was causing different results when running the step with multiprocess [#239]
- Fix the code to at least always flag the group with the shower and the requested groups after the primary shower. [#237]

Other

jump

- Reorganize jump docs between the jwst and stcal repos. [#240]

ramp_fitting

- Reorganize ramp_fitting docs between the jwst and stcal repos. [#240]

2.1.23 1.5.2 (2023-12-13)

- non-code updates to testing and development infrastructure

2.1.24 1.5.1 (2023-11-16)

- re-release to publish source distribution

2.1.25 1.5.0 (2023-11-15)

Other

- Added `alignment` sub-package. [#179]
- Enable automatic linting and code style checks [#187]

ramp_fitting

- Refactor Casertano, et.al, 2022 uneven ramp fitting and incorporate the matching jump detection algorithm into it. [#215]
- Fix memory issue with Cas22 uneven ramp fitting [#226]
- Fix some bugs in the jump detection algorithm within the Cas22 ramp fitting [#227]
- Moving some CI tests from JWST to STCAL. [#228, spacetelescope/jwst#6080]
- Significantly improve the performance of the Cas22 uneven ramp fitting algorithm. [#229]

Changes to API

-

Bug Fixes

-

2.1.26 1.4.4 (2023-09-15)

Other

- small hotfix for Numpy 2.0 deprecations [#211]

2.1.27 1.4.3 (2023-09-13)

Changes to API

saturation

- Added read_pattern argument to flag_saturated_pixels. When used, this argument adjusts the saturation group-by-group to handle different numbers of frames entering different groups for Roman. When not set, the original behavior is preserved. [#188]

Bug Fixes

- Fixed failures with Numpy 2.0. [#210, #211]

Other

jump

- enable the detection of snowballs that occur in frames that are within a group. [#207]
- Added more allowable selections for the number of cores to use for multiprocessing [#183]
- Fixed the computation of the number of rows per slice for multiprocessing, which caused different results when running the step with multiprocess [#239]

ramp_fitting

- Added more allowable selections for the number of cores to use for multiprocessing [#183]
- Updating variance computation for invalid integrations, as well as updating the median rate computation by excluding groups marked as DO_NOT_USE. [#208]
- Implement the Casertano, et.al, 2022 uneven ramp fitting [#175]

2.1.28 1.4.2 (2023-07-11)

Bug Fixes

jump

- Added setting of number_extended_events for non-multiprocessing mode. This is the value that is put into the header keyword EXTNCRS. [#178]

2.1.29 1.4.1 (2023-06-29)

Bug Fixes

jump

- Added setting of number_extended_events for non-multiprocessing mode. This is the value that is put into the header keyword EXTNCRS. [#178]

1.4.1 (2023-06-29)

Bug Fixes

jump

- Added statement to prevent the number of cores used in multiprocessing from being larger than the number of rows. This was causing some CI tests to fail. [#176]

2.1.30 1.4.0 (2023-06-27)

Bug Fixes

jump

- Updated the jump detection to switch to using the numpy sigmaclip routine to find the actual rms across integrations when there are at least 101 integrations in the exposure. This still allows cosmic rays and snowballs/showers to be flagged without being affected by slope variations due to either brigher-fatter/charge-spilling or errors in the nonlinearity correction. Also added the counting of the number of cosmic rays and snowballs/showers that is then placed in the FITS header in the JWST routines. [#174]

ramp_fitting

- Changing where time division occurs during ramp fitting in order to properly handle special cases where the time is not group time, such as when ZEROFRAME data is used, so the time is frame time. [#173]
- Added another line of code to be included in the section where warnings are turned off. The large number of warnings can cause a hang in the Jupyter notebook when running with multiprocessing. [#174]

Changes to API

-

Other

-

2.1.31 1.3.8 (2023-05-31)

Bug Fixes

dark_current

- Fixed handling of MIRI segmented data files so that the correct dark integrations get subtracted from the correct science integrations. [#165]

ramp_fitting

- Correct the “averaging” of the final image slope by properly excluding variances as a part of the denominator from integrations with invalid slopes. [#167]
- Removing the usage of `numpy.where` where possible for performance reasons. [#169]

2.1.32 1.3.7 (2023-04-26)

Bug Fixes

ramp_fitting

- Correctly compute the number of groups in a segment to properly compute the optimal weights for the OLS ramp fitting algorithm. Originally, this computation had the potential to include groups not in the segment being computed. [#163]

Changes to API

- Drop support for Python 3.8 [#162]

2.1.33 1.3.6 (2023-04-19)

Bug Fixes

ramp_fitting

- The `meta` tag was missing when checking for `drop_frame1`. It has been added to the check. [#161]

Changes to API

-

Other

- Remove use of deprecated `pytest-openfiles` `pytest` plugin. This has been replaced by catching `ResourceWarning`. [#159]

2.1.34 1.3.5 (2023-03-30)

Bug Fixes

jump

- Updated the code for both NIR Snowballs and MIRI Showers. The snowball flagging will now extend the saturated core of snowballs. Also, circles are no longer used for snowballs preventing the huge circles of flagged pixels from a glancing CR. Shower code is completely new and is now able to find extended emission far below the single pixel SNR. It also allows detected showers to flag groups after the detection. [#144]

ramp_fitting

- During multiprocessing, if the number of processors requested are greater than the number of rows in the image, then ramp fitting errors out. To prevent this error, during multiprocessing, the number of processors actually used will be no greater than the number of rows in the image. [#154]

Other

- Remove the `dqflags`, `dynamicdq`, and `basic_utils` modules and replace them with thin imports from `stdatamodels` where the code as been moved. [#146]
- update minimum version of `numpy` to `1.20` and add minimum dependency testing to CI [#153]
- restore `opencv-python` to a hard dependency [#155]

2.1.35 1.3.4 (2023-02-13)

Bug Fixes

ramp_fitting

- Changed computations for ramps that have only one good group in the 0th group. Ramps that have a non-zero groupgap should not use group_time, but (NFrames+1)*TFrame/2, instead. [#142]

2.1.36 1.3.3 (2023-01-26)

Bug Fixes

ramp_fitting

- Fixed zeros that should be NaNs in rate and rateints product and suppressed a cast warning due to attempts to cast NaN to an integer. [#141]

Changes to API

dark

- Modified dark class to support quantities in Roman.[#140]

2.1.37 1.3.2 (2023-01-10)

Bug Fixes

ramp_fitting

- Changed a cast due to numpy deprecation that now throws a warning. The negation of a DQ flag then cast to a np.uint32 caused an over flow. The flag is now cast to a np.uint32 before negation. [#139]

2.1.38 1.3.1 (2023-01-03)

Bug Fixes

- improve exception handling when attempting to use ellipses without `opencv-python` installed [#136]

2.1.39 1.3.0 (2022-12-15)

General

- use `tox` environments [#130]

Changes to API

- Added support for Quantities in models required for the RomanCAL pipeline. [#124]

ramp_fitting

- Set values in the rate and rateints product to NaN when no usable data is available to compute slopes. [#131]

2.1.40 1.2.2 (2022-12-01)

General

- Moved build configuration from `setup.cfg` to `pyproject.toml` to support PEP621 [#95]

- made dependency on `opencv-python` conditional [#126]

ramp_fitting

- Set saturation flag only for full saturation. The rateints product will have the saturation flag set for an integration only if saturation starts in group 0. The rate product will have the saturation flag set only if each integration for a pixel is marked as fully saturated. [#125]

2.1.41 1.2.1 (2022-10-14)

Bug Fixes

jump

- Changes to limit the expansion of MIRI shower ellipses to be the same number of pixels for both the major and minor axis. JP-2944 [#123]

2.1.42 1.2.0 (2022-10-07)

Bug Fixes

dark_current

- Bug fix for computation of the total number of frames when science data use on-board frame averaging and/or group gaps. [#121]

jump

- Changes to flag both NIR snowballs and MIRI showers for JP-#2645. [#118]
- Early in the step, the object arrays are converted from DN to electrons by multiplying by the gain. The values need to be reverted back to DN at the end of the step. [#116]

2.1.43 1.1.0 (2022-08-17)

General

- Made style changes due to the new 5.0.3 version of flake8, which noted many missing white spaces after keywords. [#114]

Bug Fixes

ramp_fitting

- Updating multi-integration processing to correctly combine multiple integration computations for the final image information. [#108]
- Fixed crash due to two group ramps with saturated groups that used an intermediate array with an incorrect shape. [#109]
- Updating how NaNs and DO_NOT_USE flags are handled in the rateints product. [#112]
- Updating how GLS handles bad gain values. NaNs and negative gain values have the DO_NOT_USE and NO_GAIN_VALUE flag set. Any NaNs found in the image data are set to 0.0 and the corresponding DQ flag is set to DO_NOT_USE. [#115]

Changes to API

jump

- Added flagging after detected ramp jumps based on two DN thresholds and two number of groups to flag [#110]

2.1.44 1.0.0 (2022-06-24)

Bug Fixes

ramp_fitting

- Adding special case handler for GLS to handle one group ramps. [#97]
- Updating how one group suppression and ZEROFRAME processing works with multiprocessing, as well as fixing the multiprocessing failure. [#99]
- Changing how ramp fitting handles fully saturated ramps. [#102]

saturation

- Modified the saturation threshold applied to pixels flagged with NO_SAT_CHECK, so that they never get flagged as saturated. [#106]

Changes to API

ramp_fitting

- The tuple `integ_info` no longer returns `int_times` as a part of it, so the tuple is one element shorter. [#99]
- For fully saturated exposures, all returned values are `None`, instead of tuples. [#102]

saturation

- Changing parameter name in twopoint_difference from 'normal_rej_thresh' to rejection_thresh' for consistency. [#105]

Other

general

- Update CI workflows to cache test environments and depend upon style and security checks [#96]
- Increased required `Python` version from `>=3.7` to `>=3.8` (to align with `astropy`) [#98]

2.1.45 0.7.3 (2022-05-20)

Bug Fixes

jump

- Update `twopoint_difference.py` [#90]

ramp_fitting

- Updating the one good group ramp suppression handler works. [#92]

2.1.46 0.7.2 (2022-05-19)

Bug Fixes

ramp_fitting

- Fix for accessing zero-frame in model to account for Roman data not using zero-frame. [#89]

2.1.47 0.7.1 (2022-05-16)

Bug Fixes

jump

- Enable multiprocessing for jump detection, which is controlled by the 'max_cores' parameter. [#87]

2.1.48 0.7.0 (2022-05-13)

Bug Fixes

linearity

- Added functionality to linearly process ZEROFRAME data the same way as the SCI data. [#81]

ramp_fitting

- Added functionality to use ZEROFRAME data in place of group 0 data for ramps that are fully saturated, but still have good ZEROFRAME data. [#81]

saturation

- Added functionality to process ZEROFRAME data for saturation the same way as the SCI data. [#81]

2.1.49 0.6.4 (2022-05-02)

Bug Fixes

saturation

- Added in functionality to deal with charge spilling from saturated pixels onto neighboring pixels [#83]

2.1.50 0.6.3 (2022-04-27)

Bug Fixes

- Pin astropy min version to 5.0.4. [#82]
- Fix for jumps in first good group after dropping groups [#84]

2.1.51 0.6.2 (22-03-29)

Bug Fixes

jump

- Neighboring pixels with 'SATURATION' or 'DONOTUSE' flags are no longer flagged as jumps. [#79]

ramp_fitting

- Adding feature to use ZEROFRAME for ramps that are fully saturated, but the ZEROFRAME data for that ramp is good. [#81]

2.1.52 0.6.1 (22-03-04)

Bug Fixes

ramp_fitting

- Adding feature to suppress calculations for saturated ramps having only the 0th group be a good group. [#76]

2.1.53 0.6.0 (22-01-14)

Bug Fixes

ramp_fitting

- Adding GLS code back to ramp fitting. [#64]

jump

- Fix issue in jump detection that occurred when there were only 2 usable differences with no other groups flagged. This PR also added tests and fixed some of the logging statements in twopoint difference. [#74]

2.1.54 0.5.1 (2022-01-07)

Bug Fixes

jump

- fixes to several existing errors in the jump detection step. added additional tests to ensure step is no longer flagging jumps for pixels with only two usable groups / one usable diff. [#72]

2.1.55 0.5.0 (2021-12-28)

Bug Fixes

dark_current

- Moved dark current code from JWST to STCAL. [#63]

2.1.56 0.4.3 (2021-12-27)

Bug Fixes

linearity

- Let software set the pixel dq flag to NO_LIN_CORR if linear term of linearity coefficient is zero. [#65]

ramp_fitting

- Fix special handling for 2 group ramp. [#70]
- Fix issue with inappropriately including a flagged group at the beginning of a ramp segment. [#68]
- Changed Ramp Fitting Documentation [#61]

2.1.57 0.4.2 (2021-10-28)

Bug Fixes

ramp_fitting

- For slopes with negative median rates, the Poisson variance is zero. [#59]
- Changed the way the final DQ array gets computed when handling the DO_NOT_USE flag for multi-integration data. [#60]

2.1.58 0.4.1 (2021-10-14)

Bug Fixes

jump_detection

- Reverts "Fix issue with flagging for MIRI three and four group integrations. [#44]

2.1.59 0.4.0 (2021-10-13)

Bug Fixes

jump_detection

- Fix issue with flagging for MIRI three and four group integrations. [#44]

linearity

- Adds common code for linearity correction [#55]

ramp_fitting

- Global DQ variable removed [#54]

2.1.60 0.3.0 (2021-09-28)

Bug Fixes

saturation

- Adds common code for saturation [#39]

2.1.61 0.2.5 (2021-08-27)

Bug Fixes

jump

- added tests for two point difference [#37]

ramp_fitting

- Adds support for Roman ramp data. [#43] [#49]

2.1.62 0.2.4 (2021-08-26)

Bug Fixes

Workaround for setuptools_scm issues with recent versions of pip. [#45]

2.1.63 0.2.3 (2021-08-06)

Bug Fixes

jump

- documentation changes + docs for jump detection [#14]

ramp_fitting

- Fix ramp fitting multiprocessing. [#30]

2.1.64 0.2.2 (2021-07-19)

Bug Fixes

jump

- Move common `jump` code to stcal [#27]

ramp_fitting

- Implemented multiprocessing for OLS. [#30]
- Added DQ flag parameter to *ramp_fit* [#25]
- Reduced data model dependency [#26]

2.1.65 0.2.1 (2021-05-20)

Bug Fixes

ramp_fitting

- Fixed bug for median ramp rate computation in report JP-1950. [#12]

2.1.66 0.2.0 (2021-05-18)

Bug Fixes

ramp_fitting

- Added ramp fitting code [#6]

2.1.67 0.1.0 (2021-03-19)

- Added code to manipulate bitmasks.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

stcal.alignment, 7

m

stcal.multiprocessing, 50

o

stcal.outlier_detection.median, 18
stcal.outlier_detection.utils, 19

r

stcal.resample, 27
stcal.resample.utils, 24

s

stcal.saturation, 37
stcal.skymatch.skyimage, 43
stcal.skymatch.skymatch, 47
stcal.skymatch.skystatistics, 42

t

stcal.tweakreg, 17
stcal.tweakreg.astrometric_utils, 16
stcal.tweakreg.tweakreg, 15
stcal.tweakreg.utils, 15

v

stcal.velocity_aberration, 50

INDEX

Symbols

`__call__()` (*stcal.skymatch.skystatistics.SkyStats method*), 43
`_wcsinfo_from_wcs_transform()` (*in module stcal.tweakreg.utils*), 16

A

`absolute_align()` (*in module stcal.tweakreg.tweakreg*), 15
`add_model()` (*stcal.resample.Resample method*), 32
`add_model_hook()` (*stcal.resample.Resample method*), 32
`append()` (*stcal.outlier_detection.median.MedianComputer method*), 19

B

`build_driz_weight()` (*in module stcal.resample.utils*), 25
`build_mask()` (*in module stcal.resample.utils*), 26

C

`calc_pixmap()` (*in module stcal.resample.utils*), 24
`calc_rotation_matrix()` (*in module stcal.alignment*), 7
`calc_sky()` (*stcal.skymatch.skyimage.SkyGroup method*), 46
`calc_sky()` (*stcal.skymatch.skyimage.SkyImage method*), 45
`calc_sky()` (*stcal.skymatch.skystatistics.SkyStats method*), 43
`check_output_wcs()` (*stcal.resample.Resample method*), 33
`combine_footprints()` (*in module stcal.alignment*), 8
`combine_sregions()` (*in module stcal.alignment*), 8
`compute_err` (*stcal.resample.Resample attribute*), 31
`compute_fiducial()` (*in module stcal.alignment*), 9
`compute_mean_pixel_area()` (*in module stcal.resample*), 27
`compute_mean_pixel_area()` (*in module stcal.resample.utils*), 26
`compute_num_cores()` (*in module stcal.multiprocessing*), 50
`compute_radius()` (*in module stcal.tweakreg.astrometric_utils*), 16
`compute_s_region_imaging()` (*in module stcal.alignment*), 9
`compute_s_region_keyword()` (*in module stcal.alignment*), 9
`compute_scale()` (*in module stcal.alignment*), 10
`compute_va_effects()` (*in module stcal.velocity_aberration*), 51
`compute_va_effects_vector()` (*in module stcal.velocity_aberration*), 50
`compute_weight_threshold()` (*in module stcal.outlier_detection.utils*), 20
`create_astrometric_catalog()` (*in module stcal.tweakreg.astrometric_utils*), 16
`create_output_model()` (*stcal.resample.Resample method*), 33

D

`dq_flag_name_map` (*stcal.resample.Resample attribute*), 31

E

`enable_ctx` (*stcal.resample.Resample attribute*), 31
`enable_var` (*stcal.resample.Resample attribute*), 31
`error_from_variances` (*stcal.resample.Resample attribute*), 31
`evaluate()` (*stcal.outlier_detection.median.MedianComputer method*), 19

F

`filter_catalog_by_bounding_box()` (*in module stcal.tweakreg.tweakreg*), 15
`finalize()` (*stcal.resample.Resample method*), 33
`finalize_resample_variance()` (*stcal.resample.Resample method*), 33
`finalize_time_info()` (*stcal.resample.Resample method*), 34
`flag_crs()` (*in module stcal.outlier_detection.utils*), 20
`flag_resampled_crs()` (*in module stcal.outlier_detection.utils*), 21

flag_saturated_pixels() (*in module stcal.saturation*), 38

G

get_catalog() (*in module stcal.tweakreg.astrometric_utils*), 17
get_input_model_pixel_area() (*stcal.resample.Resample method*), 34
get_output_model_pixel_area() (*stcal.resample.Resample method*), 34
get_tmeasure() (*in module stcal.resample.utils*), 26
group_ids (*stcal.resample.Resample attribute*), 31
gwcs_blot() (*in module stcal.outlier_detection.utils*), 21

I

init_time_counters() (*stcal.resample.Resample method*), 34
init_variance_arrays() (*stcal.resample.Resample method*), 34
intersection() (*stcal.skymatch.skyimage.SkyImage method*), 45
is_finalized() (*stcal.resample.Resample method*), 35
is_flux_density() (*in module stcal.resample.utils*), 26
is_imaging_wcs() (*in module stcal.resample.utils*), 27

M

medfilt() (*in module stcal.outlier_detection.utils*), 20
MedianComputer (*class in stcal.outlier_detection.median*), 18
module
    stcal.alignment, 7
    stcal.multiprocessing, 50
    stcal.outlier_detection.median, 18
    stcal.outlier_detection.utils, 19
    stcal.resample, 27
    stcal.resample.utils, 24
    stcal.saturation, 37
    stcal.skymatch.skyimage, 43
    stcal.skymatch.skymatch, 47
    stcal.skymatch.skystatistics, 42
    stcal.tweakreg, 17
    stcal.tweakreg.astrometric_utils, 16
    stcal.tweakreg.tweakreg, 15
    stcal.tweakreg.utils, 15
    stcal.velocity_aberration, 50

N

nanmedian3D() (*in module stcal.outlier_detection.median*), 18

O

output_array_shape (*stcal.resample.Resample attribute*), 31
output_array_types (*stcal.resample.Resample attribute*), 31
output_model (*stcal.resample.Resample attribute*), 31
output_pixel_scale (*stcal.resample.Resample attribute*), 31
output_wcs (*stcal.resample.Resample attribute*), 32

P

pixel_scale_ratio (*stcal.resample.Resample attribute*), 32
propagate_dq (*stcal.resample.Resample attribute*), 32

R

relative_align() (*in module stcal.tweakreg.tweakreg*), 15
Resample (*class in stcal.resample*), 28
resample_range() (*in module stcal.resample.utils*), 27
resample_variance_arrays() (*stcal.resample.Resample method*), 35
reset_arrays() (*stcal.resample.Resample method*), 35

S

sky (*stcal.skymatch.skyimage.SkyGroup attribute*), 46
SkyGroup (*class in stcal.skymatch.skyimage*), 45
SkyImage (*class in stcal.skymatch.skyimage*), 44
skymatch() (*in module stcal.skymatch.skymatch*), 47
SkyStats (*class in stcal.skymatch.skystatistics*), 42
sregion_to_footprint() (*in module stcal.alignment*), 10
stcal.alignment
    module, 7
stcal.multiprocessing
    module, 50
stcal.outlier_detection.median
    module, 18
stcal.outlier_detection.utils
    module, 19
stcal.resample
    module, 27
stcal.resample.utils
    module, 24
stcal.saturation
    module, 37
stcal.skymatch.skyimage
    module, 43
stcal.skymatch.skymatch
    module, 47
stcal.skymatch.skystatistics
    module, 42
stcal.tweakreg

module, 17
stcal.tweakreg.astrometric_utils
module, 16
stcal.tweakreg.tweakreg
module, 15
stcal.tweakreg.utils
module, 15
stcal.velocity_aberration
module, 50

U

UnsupportedWCSError, 36
update_time() (*stcal.resample.Resample method*), 36

V

validate_input_model() (*stcal.resample.Resample method*), 36
variance_array_names (*stcal.resample.Resample attribute*), 32

W

wcs_bbox_from_shape() (*in module stcal.alignment*), 10
wcs_from_sregions() (*in module stcal.alignment*), 11