Summary of Algorithms used by PlanetarySystemStacker (PSS Version 0.8.22)

This document is an algorithmic guide through the open-source software “PlanetarySystemStacker”. It explains all the steps performed in stacking a video file or batch of single images. For every step it lists the data structures and configuration parameters (*cursive*) involved in the operation and states where in the code the step is performed.

Description of algorithmic step	Data strcuture(s) and configuration parameters involved	Module.Class.Method or Module.Function in Python source code
GUI control		

PlanetarySystemStacker comes with a GUI using the QT5 widget toolkit. The GUI is started with the main program in module “planetary_system_stacker.py”.
 	
The GUI starts a separate “workflow” thread which does all computations, especially those in batch mode.
 	
Individual processing phases of the workflow are triggered by the GUI using QT signals associated with slots in the Workflow object. The signals are emitted in method “work_next_task”.
 	
The processing steps are the same as described below for the version using a main program without GUI. That version is mostly used for debugging purposes. The processing details, therefore, in the following are described for the main program case only.
		`planetary_system_stacker.__main__`
		
		`planetary_system_stacker.PlanetarySystemStacker.init`
		
		`workflow.Workflow`
		
		`planetary_system_stacker.PlanetarySystemStacker.work_next_task`

Main control program (for debugging)		
All parameters controlling the program are set in the configuration object. Eventually these values are to be set by the GUI and maintained between executions in a configuration file.		`configuration.Configuration`

Set the names of files to process. This can either be the names of video files (type .avi / .ser / .mov) or names of directories containing image files (type .png /.tiff / .fits). In the latter case all files of one directory are taken as input for a single stacking operation. If those images differ in shape, an error is raised.
 	
If multiple video files or multiple directories are specified, multiple stacking operations are performed in batch mode in succession.
 	
The workflow for the entire stacking process for a video file or image directory is controlled by the “workflow” function.
	`input_names`
	
	`input_type = “video” or “image”`
		`main_program.__main__`
		
		`main_program.workflow`

Optional: Select a “region of interest” (ROI) in the form (y_low, y_high, x_low, x_high). If set to “None”, the full frames are used	`roi`	`main_program.__main__`
Optional: Select that input images should be converted to grayscale before processing. If set to “None”, color images are processed as three-channel RGB.	`convert_to_grayscale = True or False`	`main_program.__main__`
For performance measurements, code sections can be timed. If the same section is executed several times, counters can be incremented. A timer object is created at start of execution. Individual counters can be added later. In the end, a table with the accumulated times of all counters can be printed.	`my_timer`	`timer.timer`

Read frames and create derived images

PlanetarySystemStacker provides several buffering levels of data which are used more than once during a stacking job. For level 0, no data are buffered. Images are read from the input file when needed, and derived versions are re-computed. For level 4, frames are read only once, and all derived image versions are kept in memory.

The program uses four versions of the image data:

- The original image data, either read from a single .avi file, or from a directory with image files. The shape of a single frame is (pixels in y, pixels in x [, 3 in case of color]).
- 2D monochrome image (if the original frames were monochrome, this is just a pointer to the original frame)
- "Blurred" version of the monochrome image. It is computed by applying a Gaussian filter to the monochrome image. The width of the Gaussian is an input parameter.
- Laplacian of the Gaussian

Access to the four image versions is via methods of class "Frame". Depending on whether the data are buffered or not, a pointer to the object in memory is returned, or the image is read from the input file and/or computed by applying the appropriate filter.

The blurred image versions are used later in shift computations. This helps avoiding spurious local minima caused by pixel noise.

The Laplacians are used for ranking image quality. This happens

parameter: *global_parameters_buffering_level*

frames_original

frames_monochrome

frames_monochrome_blurred

frames_monochrome_blurred_laplacian

parameter: *frames_gauss_width*

frames.Frames.init

frames.Frames.frames

frames.Frames.frames_mono

frames.Frames.frames_mono_blurred

frames.Frames.frames_mono_blurred_laplacian

in two locations: First in ranking the overall frame quality for constructing the mean frame, and then when the frame quality in local areas around alignment points is computed in the stacking process.		
Global frame ranking		

Next all frames are ordered by their overall image quality. This is done by computing the amount of structure in the ("blurred" monochrome) images. Three methods can be selected for ranking, using one of the following expressions (greater value is better) on the local luminance $l_{j,i}$:
- "xy gradient": $\sum_{j,i=0}^{npixels-1} (l_{j+1,i} - l_{j,i})^2 + (l_{j,i+1} - l_{j,i})^2$
- "Laplace": $Var\ (\Delta\ l_{j,i}\ )$
- "Sobel": $\sum_{j,i=0}^{npixels-1} \sqrt{(G_{x\,j,i})^2 + (G_{y\,j,i})^2}$ with G_x and G_y being the horizontal / vertical Sobel operators.
 	
The values are normalized, so that the value for the best frame is 1.0. The index of the frame with the highest rank is computed. This frame is used as the reference for the computation of global frame shifts.
 	
A stride value can be specified. If set to a value > 1, the images are down-sampled by this value before computing the score. In typical video files setting a value of 2 usually gives a good ranking and saves compute time.
	`frame_ranks`
	
	`frame_ranks_max_index`
	
	`parameters:`
	
	`rank_frames_method`
	
	`rank_frames_pixel_stride`
		`rank_frames.RankFrames.frame_score`
		
		`methods:`
		- `local_contrast`
		- `local_contrast_laplace`
		- `local_contrast_sobel`
		
		`in class`
		
		`miscellaneous.Miscellaneous`

Usually the method “Laplace” is to be preferred. This is also the method used by the GUI version of PSS.		
Global frame alignment		
Next all frames are aligned with each other. The frame with the highest rank is used as reference (see above). Two alignment modes are available: “Surface” and “Planet”. - “Surface”: First find a rectangular patch with good structure as “alignment window”. The size of the patch is a fraction of the frame size, the scale factor being a configuration parameter. By setting a parameter “align_frames_border_width” the window can be kept away from the frame borders. This way, the window does not have to be moved as often when the object is drifting between frames. For all potential alignment patches in the frame the merit function $\min\left(\sum_{j,i=1,\ l_{j,i}>threshold}^{npixels-1} abs\left(\frac{l_{j+1,i}-l_{j-1,i}}{l_{j,i}}\right),\ \sum_{j,i=1,\ l_{j,i}>threshold}^{npixels-1} abs\left(\frac{l_{j,i+1}-l_{j,i-1}}{l_{j,i}}\right)\right)$ is executed. This way the patch is found where good vertical and horizontal structures are present. In order to ignore contributions by noise in dark areas, a brightness threshold is included. An ordered list of alignment patches with decreasing	Parameters: *align_frames_mode* *align_frames_rectangle_scale_factor* *align_frames_rectangle_black_threshold* *align_frames_search_width* *align_frames_border_width* *align_frames_sampling_stride* *align_frames_average_frame_percent* alignment_rect_qualities	miscellaneous.Miscellaneous.quality_measure_alternative align_frames.AlignFrames.

merit function is computed. Frame alignment (see below) is tried using the patch with the highest score. If it fails for some frame, the process is repeated using the next patch, and so on, until the alignment succeeds for all frames or there are no patches left.		`compute_alignment_rect`
		
		`align_frames.AlignFrames.`
		`select_alignment_rect`

Next all frames are compared with the reference frame at this patch, and the relative shift is computed. Four methods are available for this search:
 	
- "Translation" (not recommended): Phase-correlation
 	
- "MultiLevelCorrelation" (very stable and relatively fast, used by the GUI version of PSS since V0.7.0): The normalized cross correlation between the shifted alignment window in the current frame and the reference frame window is maximized in a two-level approach:
In the first phase, on a pixel grid around with stride 2 in both y and x with shifts [s_y, s_x] around the zero shift, with
abs(s_y) ≤ (align_frames_search_width – 4) / 2, and
abs(s_x) ≤ (align_frames_search_width – 4) / 2
the following expression is maximized:
 	
$$\frac{\sum_{j,i=0}^{window_{size}} l_{frame}{}_{j+s_y,i+s_x} * l_{reference}{}_{j,i}}{\sqrt{\sum_{j,i=0}^{window_{size}} l_{frame}{}_{j+s_y,i+s_x}{}^2 * \sum_{j,i=0}^{window_{size}} l_{reference}{}_{j,i}{}^2}}$$
 	
In the second phase, the expression is evaluated on the original (fine) pixel grid around the optimal shift point of the first phase, this time with a search radius of 4. The
	`frame_shifts`
	
	`Parameter:`
	`align_frames_search_width`
		`align_frames.`
		`AlignFrames.align_frames`
		
		`miscellaneous.Miscellaneous.`
		`translation`
		
		`miscellaneous.Miscellaneous.`
		`multilevel_correlation`

result of the second phase is accepted if in both phases the optimum was not attained on the border of the search area. Otherwise, the result is set to [0, 0] (zero shift), and the result is marked “unsuccessful”.
 	
- “RadialSearch” (stable but expensive): Search all positions for a local minimum of the expression
 	
$$\sum_{j,i=0}^{window_{size}} abs(l_{frame}{}_{j,i} - l_{reference}{}_{j,i})$$
 	
spiraling out from zero shift.
 	
- “SteepestDescent” (used exclusively by the GUI version up to V0.6.0): Same as “RadialSearch”, but not all positions around zero shift are evaluated, but only those in the direction of the steepest descent of the evaluation function. More precisely:
Starting with shift values [dy_min, dx_min] = [0, 0], the shifted alignment window in the current frame is compared with the reference frame window. For every search position, the match quality is computed with
 	
$$\sum_{j,i=0}^{window_{size}} abs(l_{frame}{}_{j,i} - l_{reference}{}_{j,i})$$
 	
The goal is to minimize this expression. A “sampling stride” parameter can be selected. In this case the summation indices are incremented using this stride (to save compute time).
		`miscellaneous.Miscellaneous.search_local_match`
		
		`miscellaneous.Miscellaneous.search_local_match_gradient`

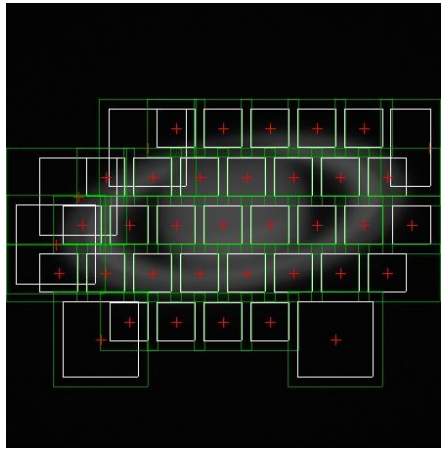
	First, for all positions with distance 1 in y or x the match quality is computed. If the minimal value is smaller than the best value so far, the corresponding position is taken as a new start point. For this new start point, again all positions with distance 1 are tested in search for a better minimum. The search ends when no improvement is found, or the maximum search width (parameter) is reached. In the latter case, the search is regarded as unsuccessful. Consecutive frames tend to have similar shifts. Therefore, the optimum found for one frame is taken as start value for the next one. The current shift is kept in the cumulative shifts [dy_min_cum, dx_min_cum]. All frame shifts are measured relative to the reference frame. To make it easier for the search algorithm described above, the computation starts for the frame captured just before the reference frame, and going backwards until the first frame is reached. In a second loop, the remaining frames are treated starting with the frame just after the reference frame and going to the end of the video. This way, the search always starts with a good approximation. If the object drift is too large, the alignment window can hit the frame border. Before that happens, the window is shifted by half the border width away from the border.	`[dy_min_cum, dx_min_cum]`	
-	"Planet": Only applicable if the object is surrounded by black space in all directions. In this case alignment is		

much easier. For each frame the “center of gravity” of the brightness distribution is computed. The differences in y and x give the relative shift.		
After aligning all frames, the pixel bounds of the intersection of all frames are computed: intersection_shape[y, x][low, high]	`intersection_shape`	
Mean frame computation		

Next, the average frame is computed by averaging the best frames, taking into account their relative global shifts. At this point no local warp effects can be corrected for. The percentage of the total number of frames is chosen via a parameter.
 	
From now on, pixel indexing of new image objects uses the shape of the average frame, given by the index bounds in the structure “intersection_shape”. The original frames, however, are not copied, so they keep their original indexing. The global shifts between the image frames and the new reduced-size images are stored in lists “dy” and “dx” in the “align_frames” object.
	`average_frame`
	
	`Parameter:`
	`align_frames_average_frame_percent`
	
	`dy / dx`
		`align_frames.AlignFrames.average_frame`

If a “region of interest” was selected, the intersection is reduced to this size and position in the frame. A new mean frame is computed with the new intersection shape.		`align_frames.AlignFrames.set_roi`
Alignment point creation		

Next, the alignment points (APs) for the “multi-point alignment” are defined. The methods are contained in class “alignment_points”. All alignment points are organized in a linear list. Each entry is a dictionary containing all information on a single point. The main variables in that dictionary are:
 	
- y, x pixel coordinates of center
- Lower and upper index bounds in y and x of the so-called “alignment box”. This is the area used for measuring the local (warp) shift against the mean frame.
- Lower and upper index bounds in y and x of the so-called “alignment patch”. The patch is somewhat larger than the box. It is the area used for stacking around this AP.
- The “reference box” with the section of the average frame at the location of the alignment box. If “MultiLevelCorrelation” is used for measuring the warp shifts, two reference boxes (“reference box first phase” with stride 2, and “reference box second phase” with stride 1) are created. They both cover the same area of the frame.
- The stacking buffer where frame contributions are accumulated during stacking for this alignment patch.
 	
Usually, first a grid of alignment points is created automatically, using the method “create_ap_grid”. The AP distance in y and x is specified with the “step_size” parameter. Additional points can be added or removed individually (“new_alignment_point”, “remove_alignment_points”). The following example shows the result of an automatic AP creation. The picture is created by method “show_alignment_points”. Red crosses show the (real) alignment points. White and green quadrats are the alignment boxes and patches around the APs, respectively.
	`alignment_points`
	
	`Parameters:`
	
	`alignment_points_half_box_width`
	
	`alignment_points_half_patch_width`
	
	`alignment_points_step_size`
		`alignment_points.AlignmentPoints.create_ap_grid`
		
		`alignment_points.AlignmentPoints.new_alignment_point`
		
		`alignment_points.`


 	
The automatic AP creation produces a staggered grid of points. In rows where the first and last points are farther away from the frame boundary, alignment patches are extended up to the frame boundary (in order to avoid holes).
 	
In AP grid creation, APs are put on the AP list only if they satisfy several conditions:
 	
- The alignment box must contain pixels brighter or equal to a given threshold (brightness condition).
- The difference between the brightest and dimmest pixels in the alignment box must be greater or equal to a second threshold (contrast condition).
- A “structure” value representing the amount of local structure in the alignment box must exceed a third threshold (structure threshold). The values are normalized such that it is 1. for the AP with maximum structure.
	alignment_points_brightness_threshold
	
	alignment_points_contrast_threshold
	
	alignment_points_structure_threshold
		`AlignmentPoints.remove_alignment_points`
		
		`alignment_points.AlignmentPoints.find_alignment_points`
		
		`alignment_points.AlignmentPoints.find_neighbor`
		
		`alignment_points.AlignmentPoints.show_alignment_points`

Special attention is given to APs in dark areas close to the moon or planet. If the AP satisfies the structure and brightness condition, the object must fill at least part of the alignment box. To find APs where this part is too small, the fraction of pixels brighter than the brightness threshold is computed. If it is smaller than parameter “dim_fraction_threshold”, the AP is moved towards the object. To this end, the “center of gravity” of the bright pixels inside the alignment box is computed, and the AP center is moved to this point.	`Parameter:` *`alignment_points_dim_fraction_threshold`*	
Finally, for each AP the amount of structure in the AP box is computed using the expression $s_y = \sum_{j,i=0}^{box\_size-1} abs(l_{j+1,i} - l_{j-1,i}) / \#pixels$ $s_x = \sum_{j,i=0}^{box\_size-1} abs(l_{j,i+1} - l_{j,i}) / \#pixels$ $Sharpness = \min(s_y, s_x)$ If the sharpness value is below the given threshold, the alignment point is added to the list of failed APs.	`Parameter:` *`alignment_points_structure_threshold`*	`miscellaneous.Miscellaneous.quality_measure`
Ranking frames at alignment points		

After all APs have been set, for each frame and each AP the image quality is computed, based on the alignment box around the AP. This is a very compute intensive operation. For computing the local frame qualities, as above three methods can	`alignment_point['frame_qualities']`
	
	`alignment_point['best_frame_indices']`
		`alignment_points.AlignmentPoints.compute_frame_qualities`

be chosen from (by selecting the “rank_method” parameter: - “xy gradient” - “Laplace” - “Sobel” For their definition, see above. The recommended choice is “Laplace”. It is used by the GUI version of PSS. As with frame ranking, a stride parameter can be set for down-sampling. If the “Laplace” method had been chosen to rank the frames, and now again “Laplace” is chosen for AP ranking, sampled-down Laplacians were stored for re-use. In this case the parameter “alignment_point_pixel_stride” is ignored, and the old parameter “rank_frames_pixel_stride” is re-used instead (because the Laplacians were computed with this stride). The qualities are stored for all frames in a list. The list is stored in the AP dictionary as “alignment_point[‘frame_qualities’]. A list of the best frame indices (up to the specified percentage of frames to be stacked) is computed and stored in the AP dictionary as “alignment_point[‘best_frame_indices’]. Note that these lists in general are different at different APs because of local seeing. To make the association of APs and best frames also accessible from the frame side, the APs are appended to the list of “used alignment points” of their corresponding frame objects. These lists are used in stacking below.	Parameters: *alignment_points_rank_method* *alignment_points_pixel_stride* *alignment_points_frame_percent* frames.used_alignment_points	
Frame stacking		

In preparation of frame stacking, method "prepare_for_stack_blending" is executed. It computes all auxiliary arrays and variables, so that in the remaining computations each video frame has to be loaded only once.

`stack_frames.number_single_frame_contributions`

`stack_frames.prepare_for_stack_blending`

First, for every AP an array with the size of the AP patch is computed. It is filled with "weights" between 0 and 1. Weights are 0 outside the patch rim and increase linearly to 1 at the AP center [j_{APc} , i_{APc}]. More precisely, for point [j, i] with

`alignment_point['weights_yx']`

$y_{low} \leq j < y_{high}$ *and* $x_{low} \leq i < x_{high}$:

$$hpw = \frac{y_{high} - y_{low}}{2}$$

$$weight_j = \begin{cases} \frac{j - y_{low} + 1}{hpw + 1} \; for \;\; y_{low} \leq j < j_{APc} \\ \frac{y_{high} - j}{hpw} \; for \;\; j_{APc} \leq j < y_{high} \end{cases}$$

$$weight_i = \begin{cases} \frac{i - x_{low} + 1}{hpw + 1} \; for \;\; x_{low} \leq i < i_{APc} \\ \frac{x_{high} - i}{hpw} \; for \;\; i_{APc} \leq i < x_{high} \end{cases}$$

$weight_{j,i}$ = min ($weight_j, weight_i$)

`stack_frames.StackFrames.one_dim_weight`

The weights for all points within the AP patch are stored with the AP at "alignment_point['weights_yx']". In both coordinate directions the weights ramp up linearly from a small value on the lower patch boundary to 1 at the patch center, and from there ramp down again to a small value on the upper patch boundary.

The helper function "one_dim_weight" computes the ramping

`stack_frames.one_dim_weight`

from zero to one and back to zero across a 1D line through the patch. Array “sum_single_frame_weights” accumulates the contributions from all APs. This array is used later for buffer normalization.
 	
If for every pixel in the frame there is at least one AP patch with a nonzero contribution, the whole frame is covered with APs and no additional background frame is required. To find out if this is the case, “number_stacking_holes” is computed as the total number of pixels where the accumulated weights are below a very small value (10^{-10}).
 	
If “number_stacking_holes” is zero, no background image is needed in stacking. In this case everything is set for stacking.
 	
If it is greater than zero, the stacked image contains holes, so it has to be blended with a background image. The background is computed as the average of the best frames. Only global shifts are applied, no warping. This image must be blended gradually with the stacked image.
 	
First the number of points where the background image is needed is computed. Because the background is to be blended gradually with the AP patches, this number is greater than the number of points where the accumulated patch weights are zero. More specifically, “points_where_background_used” is the number of points where the accumulated weights are greater than “stack_frames_background_blend_threshold”.
 	
If the fraction of the points where the background is required, as compared to the total number of pixels, is above the threshold “stack_frames_background_fraction”, it is decided to compute a full background image. If not, the entire frame is subdivided into rectangular patches, and for each patch the background is
	`sum_single_frame_weights`
	
	`number_stacking_holes,`
	
	`points_where_background_used`
	
	`backgroune_patches`
	
	`Parameters:`
	
	`stack_frames_background_blend_`	

computed if the patch contains pixels where the background is required. In this case, a list “backgroune_patches” of dictionaries is constructed. For each patch where the background image is to be computed during stacking, it specifies the bounds in y and x.	*threshold* *stack_frames_background_fraction*	
Frame stacking proceeds in a loop over all frames. For each frame there is a loop over all alignment points for which it was decided before that this frame is to be used (see “Rank frames at alignment points”).	`Frames.used_alignment_points`	`stack_frames.StackFrames.stack_frames`

First, the local shift of the frame at the AP is computed relative to the mean frame. Similar as with the computation of global frame shifts, four methods can be chosen from (see Section “Global frame alignment” for details):
 	
- “MultiLevelCorrelation”
- “Subpixel”
- “CrossCorrelation”
- “RadialSearch”
- “SteepestDescent
 	
The recommended version, i.e. the one used exclusively by the GUI version since V0.7.0, is “MultiLevelCorrelation”. It is most stable, reasonably fast and reliable. It works similarly as described in section “Global Frame Alignment” above. The only difference is that the expression which is maximized in the first phase (on the coarse pixel grid with stride 2) this time includes an additional penalty term
 	
$p(s_y, s_x) = 1 - f * ((\frac{s_y}{swf} - 1)^2 + (\frac{s_x}{swf} - 1)^2))$
 	
with
	`Parameter:`
	
	alignment_points_method
	
	`Parameters:`
		`alignment_points.AlignmentPoints.compute_shift_alignment_point`

$f =$ *alignment_points_penalty_factor* and $swf =$ *(alignment_points_search_width − 4) / 2* With this additional term, the expression maximized in the first phase is: $$\frac{p(s_y, s_x) * \sum_{j,i=0}^{window_{size}} l_{frame}{}_{j+s_y,i+s_x} * l_{reference}{}_{j,i}}{\sqrt{\sum_{j,i=0}^{window_{size}} l_{frame}{}_{j+s_y,i+s_x}{}^2 * \sum_{j,i=0}^{window_{size}} l_{reference}{}_{j,i}{}^2}}$$ In the second phase no penalty term is applied, so the expression to be maximized is the same as in section “Global Frame Alignment”. As in Section “global frame alignment”, when using the methods “RadialSearch” or “SteepestDescent”, a parameter “sampling_stride” can be set > 1 to speed up the process. In this case the optimal position of the local shift is still determined with 1 pixel accuracy, but the summation in the merit function only includes a coarser subset of pixels. A good match should still lead to a minimum of the merit function at the right place, in particular since the merit function is evaluated on the blurred monochrome images.	*alignment_points_penalty_factor* *alignment_points_search_width* Parameter: *alignment_points_sampling_stride*	

Next, the total shift at the AP is computed as the sum of the global frame shift and the local warp shift “[shift_y, shift_x]”. Using these shift values, function “remap_rigid” shifts the AP patch around the AP in the current frame and adds it to the AP’s stacking buffer. Here for the first time the original (color) frames are used, and not the blurred monochrome versions which had	`[shift_y, shift_x]`
	
	`[total_shift_y, total_shift_x]`
	
	`alignment_point[ ‘stacking buffer’]`
		`stack_frames. StackFrames.remap_rigid`

been the basis for all quality analyses and shift computations.		
Within the same loop over all frames, the (partial) background image is computed. Therefore, each frame has to be loaded only once.	`averaged_background`	
Merging alignment patches		

So far stacking was performed locally on the AP patches. Now those patches are blended into the global "stacked_image_buffer". This is done by method "merge_alignment_point_buffers".
 	
It is crucial at this step to avoid sharp transitions between patches. After all, they have been rigidly shifted, most likely using different shift values. Therefore, overlapping patches must be blended with each other. The difficulty is, however, that the program so far has no notion of AP neighborhood. This problem is solved by multiplying the AP patches with weight functions which smoothly go to zero on the patch rim.
 	
First, the "foreground image", i.e. the image in pixels covered by at least one AP, is computed by dividing the "stacked_image_buffer" by the accumulated AP patch weights given by "sum_single_frame_weights". Since the latter array was initialized with 1.E-30, there is no divide by zero in holes between AP patches.
 	
$$foreground\_image = \frac{stacked\_image\_buffer}{sum\_single\_frame\_weights}$$
 	
This "foreground_image" has the correct brightness within AP patches. If the whole image is covered with APs, the stacking is
	`stacked_image_buffer`
	
	`Alignment_point['weights_yx']`
	
	`sum_single_frame_weights`
		`stack_frames.StackFrames.`
		`merge_alignment_point_buffers`

completed at this point. Otherwise, the “foreground_image” is blended with the background in the next step.
 	
As a preparation of this blending process, the mask “foreground_weight” is computed as:
 	
$$foreground\_weight = \min\left(1, \frac{\text{ssfw}}{bt}\right) \quad with$$
 	
$$ssfw = sum\_single\_frame\_weights$$
$$bt = stack\_frames\_background\_blend\_threshold$$
 	
Using “foreground_weight” as a mask, the foreground and background images are blended with each other into the final result:
 	
$$result = fw * foreground\_image + (1 - fw) * background$$
 	
with:
 	
$$fw = foreground\_weight$$
$$background = averaged\_background$$
 	
In a final step, the image buffer “result” is converted from a 32bit floating point representation into 16bit unsigned int.
	`Parameter:`
	
	`stack_frames_background_blend_threshold`
	
	`stacked_image`	

Saving the final image		
Finally, the “stacked_image” is written to a file. At this point 16bit PNG, Tiff and FITS formats are supported.		`frames.Frames.save_image`
Postprocessing (Sharpening)		

PSS includes the option to postprocess the stacked image as the last step of the workflow. As an alternative, this step can be invoked directly for a given TIFF, PNG, or FITS image. All computations are performed in 32bit float arithmetic.
 	
Postprocessing is done in two steps:
 	
- RGB channel alignment (either automatic or manual)
- Sharpening / denoising (wavelets)
 	
Input to the RGB channel alignment is the “image_original”, i.e. the summation image from stacking. Channel alignment can be done either with 1 pixel resolution, or with sub-pixel resolution (0.5 or 0.25 pixel). If sub-pixel resolution is selected, the original image is enlarged first, using bicubic interpolation, to twice or four times the original image scale.
 	
Automatic alignment first aligns the red, and then the blue channel with respect to the green channel. The best match is determined using the same “MultiLevelCorrelation” algorithm as in the “Global frame alignment” step in stacking (see above). The entire frame is taken as the alignment window. For very large input images (DSLR images) this is a heavyweight computation, especially if sub-pixel resolution is selected. If RAM is not sufficient to store the interpolated images, sub-pixel resolution is restricted automatically to a level where they fit.
 	
As an alternative (or subsequent correction step) to automatic alignment, manual adjustments are possible, again in 1, 0.5, or 0.25 pixel steps. Implementation is complicated, because the GUI response would be extremely slow if after each correction step the whole sharpening pipeline had to be processed. Therefore, in “shift_image mode” the current postprocessed
	`image_original`
	
	`input_image`
	
	`postproc_version.`
	`images_uncorrected`
		`postproc_editor.`
		`ImageProcessor.`
		`recompute_selected_version`
		
		`miscellaneous.Miscellaneous.`
		`auto_rgb_align`
		
		`miscellaneous.Miscellaneous.`
		`measure_rgb_shift`
		
		`miscellaneous.Miscellaneous.`
		`shift_colors`

image is interpolated appropriately and stored in the "images_uncorrected" list. If then one of the correction buttons is pressed, the accumulated correction shifts ("correction_red" and "correction_blue") are applied to the "image_uncorrected".

The result of RGB alignment is stored in "input_image" as input to the wavelet pipleline. Sharpening / denoising uses a multi-level unsharp masking algorithm, using both Gaussian and Bilateral filters:

Starting with a given "input_image", a chain of up to ten (as currently set in the configurable module) postprocessing layers is applied:

$layer\_input_0 =$ input_image

$for\ i = 0, \dots, n_{layers} - 1$ (with $n\_layers \leq 10$):

$layer\_gauss_i = G_i(layer\_input_i))$

using a Gaussian low-pass filter G_i with "radius" $radius_i$,

$layer\_bilateral_i = B_i(layer\_input_i))$

using a bilateral filter B_i with "radius" $radius_i$ and "bilateral range" bi_range_i

$layer\_input_{i+1} = bi\_fraction_i * layer\_bilateral_i + (1 - bi\_fraction_i) * layer\_gauss_i$

$layer\_with\_noise_i = layer\_input_i - layer\_input_{i+1}$

$layer\_dn_i = dn_i * G_i(layer\_with\_noise_i) + (1 - dn_i) * layer\_with\_noise_i$

using the same Gaussian low-pass filter G_i with "radius" $radius_i$ as above, and the de-noise ratio dn_i

$$final\_image = layer\_input_{i+1} + \sum_{i=0}^{n\_layers-1} layer\_amount_i * layer\_dn_i$$

With $layer\_amount_i$ being the strength of the corresponding wavelet component.

Please note that this algorithm reduces to the one used by Registax 6 if $bi\_fraction_i$ is set to zero for all layers.

The resulting image "final_image" is converted to 16bit unsigned integer and written to a file, again using either the 16bit PNG, TIFF or FITS format.