# Summary of Algorithms used by PlanetarySystemStacker

This document is a 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) involved** | **Module.Class.Method or Module.Function in Python source code** |
|---|---|---|
| **Main control program** | | |
| 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) or names of directories containing image files (type .tiff). 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.<br><br>If multiple video files or multiple directories are specified, multiple stacking operations are performed in batch mode in succession.<br><br>The workflow for the entire stacking process for a video file or image directory is controlled by the “workflow” function. | `input_names`<br><br>`input_type = “video” or “image”` | `main_program.__main__`<br><br>`main_program.workflow` |

| | | |
|---|---|---|
| Eventually this is to be replaced by GUI control. | | |
| 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** | | |
| Read all frames, either from a single .avi file, or from a directory into a list structure. The shape of a single frame is (pixels in y, pixels in x [, 3 in case of color]). | `frames, shape` | `frames.Frames.init` |
| Parallel to the list of frames, three lists with different versions of the frames are computed:<br>- 2D monochrome image (if the original frames were monochrome, this list points to the original frame list)<br>- “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.<br>- Laplacian of the Gaussian | `frames_mono,`<br>`frames_mono_blurred`<br>`frames_mono_blurred_laplacian`<br>`parameter: frames_gauss_width` | `frames.Frames.init`<br>`frames.Frames.add_monochrome` |

| | | |
|---|---|---|
| The blurred image versions are used later in shift computations. This helps avoiding spurious local minima caused by pixel noise.<br><br>The Laplacians are used for ranking image quality. This happens 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 $l_{j,i}$:<br><br>- "xy gradient": $\sum_{j,i=0}^{npixels-1} (l_{j+1,i} - l_{j,i})^2 + (l_{j,i+1} - l_{j,i})^2$<br>- "Laplace": $Var\,(\Delta\, l_{j,i}\,)$<br>- "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.<br><br>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 | `frame_ranks`<br><br>`frame_ranks_max_index`<br><br>`parameters:`<br><br>*`rank_frames_method`*<br><br>*`rank_frames_pixel_stride`* | `rank_frames.RankFrames.frame_score`<br><br>`methods:`<br><br>- `local_contrast`<br>- `local_contrast_laplace`<br>- `local_contrast_sobel`<br><br>`in class`<br><br>`miscellaneous.Miscellaneous` |

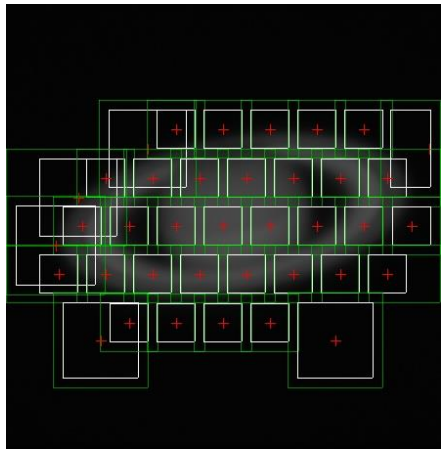
| | | |
|---|---|---|
| computed. This frame is used as the reference for the computation of global frame shifts.<br><br>A stride value can be specified. If set to a value > 1, the images are down-sampled by this value before computing the score. In typical video files setting a value of 2 usually gives a good ranking and saves compute time.<br><br>Usually the method “Laplace” is to be preferred. | | |
| **Global frame alignment** | | |
| Next all frames are aligned with each other. The frame with the highest rank is used as reference (see above). It is planned to implement two alignment algorithms: “Surface” and “Planet”.<br><br>- “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.<br><br>For all potential alignment patches in the frame the merit function<br>$\min( \sum_{j,i=1,\ l_{j,i}>threshold}^{npixels-1} abs\left(\frac{l_{j+1,i}-l_{j-1,i}}{l_{j,i}}\right),$ | `Parameters:`<br><br>*`align_frames_rectangle_scale_factor`*<br><br>*`align_frames_rectangle_black_threshold`*<br><br>*`align_frames_search_width`*<br><br>*`align_frames_border_width`*<br><br>*`align_frames_sampling_stride`*<br><br>*`align_frames_average_frame_percent`* | `align_frames.AlignFrames.select_alignment_rect`<br><br>`miscellaneous.Miscellaneous.quality_measure_alternative` |

| | | |
|---|---|---|
| $$\sum_{j,i=1,\ l_{j,i}>threshold}^{npixels-1} abs(\frac{l_{j,i+1}-l_{j,i-1}}{l_{j,i}}))$$ 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.<br><br>Next all frames are compared with the reference frame at this patch, and the relative shift is computed. Three methods are available for this search:<br>- "Translation" (not recommended): Cross-correlation<br>- "RadialSearch" (stable but expensive): Search all positions for a local minimum of the expression given below, spiraling out from zero shift.<br>- "SteepestDescent" (recommended): 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:<br><br>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<br><br>$$\sum_{j,i=0}^{window_{size}} abs(l_{frame}{}_{j,i} - l_{reference}{}_{j,i})$$<br><br>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 | `frame_shifts` | `align_frames.`<br>`AlignFrames.align_frames`<br><br>`miscellaneous.Miscellaneous.`<br>`translation`<br><br>`miscellaneous.Miscellaneous.`<br>`search_local_match`<br><br>`miscellaneous.Miscellaneous.`<br>`search_local_match_gradient` |

| | | |
|---|---|---|
| (to save compute time).<br><br>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.<br><br>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.<br><br>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].<br><br>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.<br><br>- "Planet": Not implemented yet | `[dy_min_cum, dx_min_cum]` | |
| 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 | `average_frame` | `align_frames.AlignFrames.average_frame` |

| | | |
|---|---|---|
| point no local warp effects can be corrected for. The percentage of the total number of frames is chosen via a parameter.<br><br>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. | `Parameter:`<br>*`align_frames_average_frame_percent`*<br><br>`dy / dx` | |
| 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:<br><br>- y, x pixel coordinates of center<br>- 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.<br>- 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. | `alignment_points`<br><br>`Parameters:`<br><br>*`alignment_points_half_box_width`*<br><br>*`alignment_points_half_patch_width`* | |

| | | |
|---|---|---|
| - The “reference box” with the section of the average frame at the location of the alignment box.<br>- The stacking buffer where frame contributions are accumulated during stacking for this alignment patch.<br><br>Two lists, included with each alignment point, deserve particular attention:<br><br>- The list “low_structure_neighbors” contains alignment point objects for points which during AP construction did not satisfy the structure condition (i.e. they do not show enough structure to be used in shift computations, and therefore are not included in the alignment point list). Using this list, those “failed” points are registered at the closest “real” alignment point. The idea is that in stacking a real AP, the alignment patches of the failed neighbors are stacked using the shifts determined at the “real” point. This way, holes in the stacked frame are avoided.<br>- In analogy to the above list, the list “dim_neighbors” contains failed AP objects which did not satisfy the brightness or contrast condition. Again, the reason is that at these points a shift cannot be computed with reasonable accuracy.<br><br>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_point”). 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 | *alignment_points_structure_threshold*<br><br>*alignment_points_brightness_threshold*<br><br>*alignment_points_contrast_threshold*<br><br>*alignment_points_step_size* | `alignment_points.AlignmentPoints.create_ap_grid`<br><br>`alignment_points.AlignmentPoints.new_alignment_point`<br><br>`alignment_points.` |

| | | |
|---|---|---|
| boxes and patches around the APs, respectively.<br><br>  <br><br>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).<br><br>When in AP grid creation a point does not fulfill the brightness or structure condition, it is appended to one of the failed AP lists “alignment_points_dropped_dim” or “alignment_points_dropped_structure”, respectively. Later, shortly before stacking, points on those lists are assigned to “real” APs in their neighborhood. Stacking for the failed APs then uses the shifts measured at the “real” neighbor. | `alignment_points_dropped_dim`<br><br>`alignment_points_dropped_structure` | `AlignmentPoints.remove_alignment_point`<br><br>`alignment_points.AlignmentPoints.find_alignment_points`<br><br>`alignment_points.AlignmentPoints.find_alignment_point_neighbors`<br><br>`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 | `Parameter:`<br><br>*`alignment_points_dim_fraction_threshold`* | |

| | | |
|---|---|---|
| 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.<br><br>To avoid that moving the AP opens a gap in the stacked image, the AP patch width is increased such that it covers the whole are of the original AP patch. | | |
| Finally, for each AP the amount of structure in the AP box is computed using the expression<br><br>$s_y = \sum_{j,i=0}^{box\_size-1} abs(l_{j+1,i} - l_{j-1,i}) / \#pixels$<br><br>$s_x = \sum_{j,i=0}^{box\_size-1} abs(l_{j,i+1} - l_{j,i}) / \#pixels$<br><br>$Sharpness = \min(s_y, s_x)$<br><br>If the sharpness value is below the given threshold, the alignment point is added to the list of failed APs. | Parameter:<br><br>*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 be chosen from (by selecting the “rank_method” parameter: | `alignment_point['frame_qualities']`<br><br>`alignment_point['best_frame_indices']` | `alignment_points.AlignmentPoints.compute_frame_qualities` |

| | | |
|---|---|---|
| - “xy gradient”<br>- “Laplace”<br>- “Sobel”<br><br>For their definition, see above. The recommended choice is “Laplace”.<br><br>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).<br><br>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.<br><br>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:`<br><br>*`alignment_points_rank_method`*<br><br>*`alignment_points_pixel_stride`*<br><br>*`alignment_points_frame_percent`*<br><br>`frames.used_alignment_points` | |
| **Frame stacking** | | |
| For every “failed” AP (in lists “alignment_points_dropped_dim” and | `alignment_points_dropped_dim` | `stack_frames.StackFrames.stack_frames` |

| | | |
|---|---|---|
| "alignment_points_dropped_structure") the closest real alignment point is determined. The failed alignment point object is appended to one of the lists alignment_point['dim_neighbors'] or alignment_point['low_structure_neighbors'] contained in the dictionary of the real AP.<br><br>The neighbor search is still done in an $n^2$ loop. It was not optimized yet because this step is very fast, anyway.<br><br>Associating all "failed" APs with "real" ones is important for the following reason: In the general case the "real" APs do not cover the entire frame completely. Only stacking those APs, therefore, would leave holes in the stacked image. By including the failed APs, full frame coverage is guaranteed. Local frame shifts which cannot be computed at failed APs are copied from their real neighbor. | `alignment_points_dropped_structure`<br><br>`alignment_point['dim_neighbors']`<br><br>`alignment_point['low_structure_neighbors']` | `alignment_points.AlignmentPoints.find_alignment_point_neighbors` |
| 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 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):<br><br>- "Subpixel"<br>- "CrossCorrelation"<br>- "RadialSearch"<br>- "SteepestDescent"<br><br>The recommended version is "SteepestDescent". It is most | *shift_pixel*<br><br>Parameters:<br><br>*alignment_points_method* | `alignment_points.AlignmentPoints.compute_shift_alignment_point` |

| | | |
|---|---|---|
| stable, reasonably fast and reliable.<br><br>As in Section “global frame alignment”, 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_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_pixel”. 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.<br><br>Using the same total shift values, patches are accumulated in the same way for all “failed” APs associated with this real one. Here for the first time the original (color) frames are used, and not the blurred monochrome versions which had been the basis for all quality analyses and shift computations. | `[total_shift_y, total_shift_x]`<br><br>`alignment_point[ 'stacking buffer']`<br><br>`alignment_point[ 'dim_neighbors']`<br><br>`alignment_point[ 'low_structure_neighbors']` | `stack_frames. StackFrames.remap_rigid` |
| **Merging alignment patches** | | |
| So far stacking was performed locally on the AP patches (both for “real” and “failed” APs). Now those patches are blended into the global “stacked_image_buffer”. This is done by method “merge_alignment_point_buffers”.<br><br>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 | `stacked_image_buffer` | `stack_frames.StackFrames. merge_alignment_point_buffers` |

| | | |
|---|---|---|
| must be blended with each other. The difficulty is, however, that the program so far has no notion of AP neighborhood (except for the weak association of “failed” with “real” APs. Therefore, a trick is used to perform the blending:<br><br>For every patch, an array with the same shape as the AP patch is created. It is filled with “weights” between 0 and 1. Weights are 0 on the patch rim and increase linearly to 1 on the box rim. Within the box the weight is set constant to 1. The helper function “one_dim_weight” computes the ramping from zero to one and back to zero across a 1D line through the patch.<br><br>Parallel to the accumulation of the global image buffer, the sum of the weights over all contributing images is accumulated pixel-wise in the buffer “single_frame_contributions”. (To speed up this accumulation, the fact is exploited that for any given AP patch the weights of all stacked frames are the same. So, the sum of their contributions is just a single weight times the number of stacked frames.)<br><br>When the stacking is done, the “single_frame_contributions” buffer for each pixel contains the sum of weights over all patches and frames. As the last step, the global image buffer (“stacked_image_buffer”) is divided pixel-wise by the “single_frame_contributions” buffer. As a result of this “normalization” the overall brightness values of the “stacked_image_buffer” correspond to the mean brightness of the stacked images. In particular, neighboring patches are blended with each other with the same overall weight at every pixel. | `weights_yx`<br><br>`single_frame_contributions` | `stack_frames.StackFrames.one_dim_weight` |
| **Saving the final image** | | |

| | | |
|---|---|---|
| Finally, the “stacked_image_buffer” is written to a file. At this point only 16bit Tiff is supported. | `stacked_image_buffer` | `frames.Frames.save_image` |