libskry — Lucky imaging implementation for the Stackistry project

Algorithms summary

Filip Szczerek

2019-10-20

Contents

1 Overview

This document describes the algorithms used by *libskry*[1], an image stacking implementation for the *Stackistry*[2] project.

Source code relevant for the discussed feature is shown in listing boxes:

```
size_t SKRY_get_img_count(const SKRY_ImgSequence *img_seq);
```

2 Input and output

The input to *libskry* is an *image sequence* (i.e. a video). The primary output is an *image stack* – a single image built from the video, with an improved signal-to-noise ratio and deformations removed.

Additional outputs are the *best fragments composite* – consisting of best-quality fragments of all images; and image quality data for the whole sequence.

3 Internal pixel formats

All processing phases except the final image stacking (shift-and-add summation) operate on 8-bit grayscale images. In the last phase, image stacking uses the same number of channels as the input video and produces an image stack with 32-bit floating-point channel values.

Raw-color images are demosaiced (debayered) using simple bilinear interpolation for all processing phases except image stacking, which uses high-quality linear interpolation[3].

Listing 1: Demosaicing functions (`demosaic.h/.c`)

```
enum SKRY_demosaic_method
void demosaic_8_as_RGB(...);
void demosaic_8_as_mono8(...);
void demosaic_16_as_RGB(...);
void demosaic_16_as_mono8(...);
enum SKRY_CFA_pattern translate_CFA_pattern(...);
```

4 Image quality

Image quality is defined for images (and image fragments) as the sum of absolute differences between pixel values of the image and its blurred version:

$$quality = \sum |I - I_{blur}|$$

In other words, it is the sum of pixel values of the image's high-frequency component. Higher values correspond to better quality.

[1] `https://github.com/GreatAttractor/libskry`

[2] `https://github.com/GreatAttractor/Stackistry`

[3] Rico Malvar, Li-wei He, Ross Cutler *High-Quality Linear Interpolation for Demosaicing of Bayer-Patterned Color Images*; International Conference of Acoustic, Speech and Signal Processing, May 2004. `https://www.microsoft.com/en-us/research/publication/high-quality-linear-interpolation-for-demosaicing-of-bayer-patterned-color-images/`

Image blurring is performed with a box filter applied three times, which gives results quite close to Gaussian blur and has short running time.

Listing 2: Image quality (`filters.h/.c, quality.h/.c`)

```
#define QUALITY_ESTIMATE_BOX_BLUR_ITERATIONS 3

SKRY_QualityEstimation* SKRY_init_quality_est(
    const SKRY_ImgAlignment* img_algn,
    unsigned estimation_area_size,
    /// Corresponds to box blur radius
    /// used for quality estimation.
    unsigned detail_scale
);

SKRY_quality_t estimate_quality(...);

struct SKRY_image* box_blur_img(...);
```

5 Block matching

Block matching is used during the image alignment and reference point alignment phases.

Given the position of a certain feature in an image, the aim is to find the position of the same feature in another image(s). It is done by comparing the feature's *reference block* (an image fragment around the feature's position) with other images, around the currently known position.

The reference block is overlayed onto the new image, at search positions surrounding the feature's last known position within a certain radius and spacing (*search step*). The initial search step is > 1, and is halved after each search iteration.

A search iteration involves calculating the sum of squared pixel value differences of the underlying image fragment and the reference block. The position with the smallest sum becomes the initial search position for the next iteration.

After the first iteration, the search radius equals the search step of the previous iteration. Once the search step reaches 1, the best position becomes the feature's new position.

Listing 3: Block matching (`match.h/.c`)

```
uint64_t calc_sum_of_squared_diffs(...);
void find_matching_position(...);
```

6 Processing phases

6.1 Image alignment (video stabilization)

Image alignment compensates any global image drift/shaking etc. Two modes are available: anchors and image centroid.

Anchor-based alignment performs block-matching around user- or automatically-selected anchor point(s). Automatic selection puts an anchor in the highest-quality fragment in the inner area of the first image of the input sequence.

Centroid-based alignment simply uses the image centroid as the reference position. It is best suited for planetary videos.

Listing 4: Image alignment (`img_align.h/.c`)

```
enum SKRY_img_alignment_method {...};
SKRY_ImgAlignment* SKRY_init_img_alignment(...);
enum SKRY_result SKRY_img_alignment_step(...);
```

Regardless of the mode used, the result of image alignment is a list of offsets, one for each image in the sequence. The offsets taken together define an intersection of all images, which is the region visible in the whole video.

```
struct SKRY_img_alignment
{
    // ...

    /// Set-theoretic intersection of all images after alignment
    /// (i.e. the fragment which is visible in all images).
    struct
    {
        /// Offset, relative to the first image's origin.
        struct SKRY_point offset;

        /// Coordinates of the bottom right corner
        /// (belongs to the intersection), relative
        /// to the first image's origin.
        struct SKRY_point bottom_right;

        unsigned width;
        unsigned height;
    } intersection;

    struct SKRY_point* img_offsets;
};
```

This "region of interest" (referred to as *images' intersection* throughout the code) is what the subsequent processing phases operate on.

6.2 Quality estimation

The goal of quality estimation is selecting image fragments for use by subsequent processing phases. The method of calculating the quality is given in 4.

The images' intersection (6.1) is divided into square *quality estimation areas* (plus any residual rectangles along the right and bottom border).

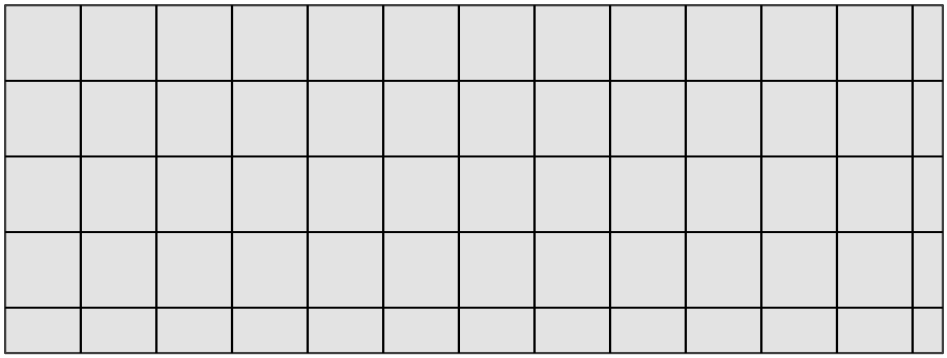

Figure 1: Quality estimation areas.

Each area's quality is calculated for each image in the sequence. After quality estimation completes, individual areas and whole images can be sorted by quality; additionally, a composite image containing the best image fragments (corresponding to quality estimation areas) of the whole sequence can be assembled (figs. 3, 4).

For each estimation area, its *reference block* is stored, which is the image fragment underlying this area and its neighboring (3x3) areas, taken from the image where the area has the best quality (fig. 2).

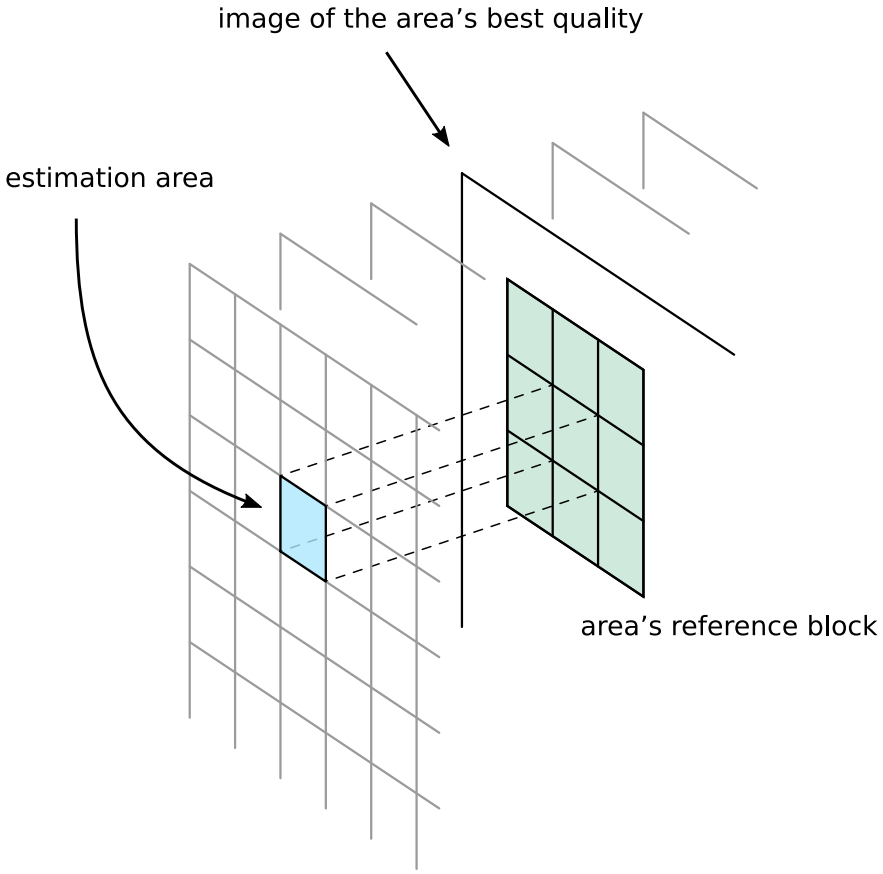

Figure 2: Quality estimation area and its reference block.

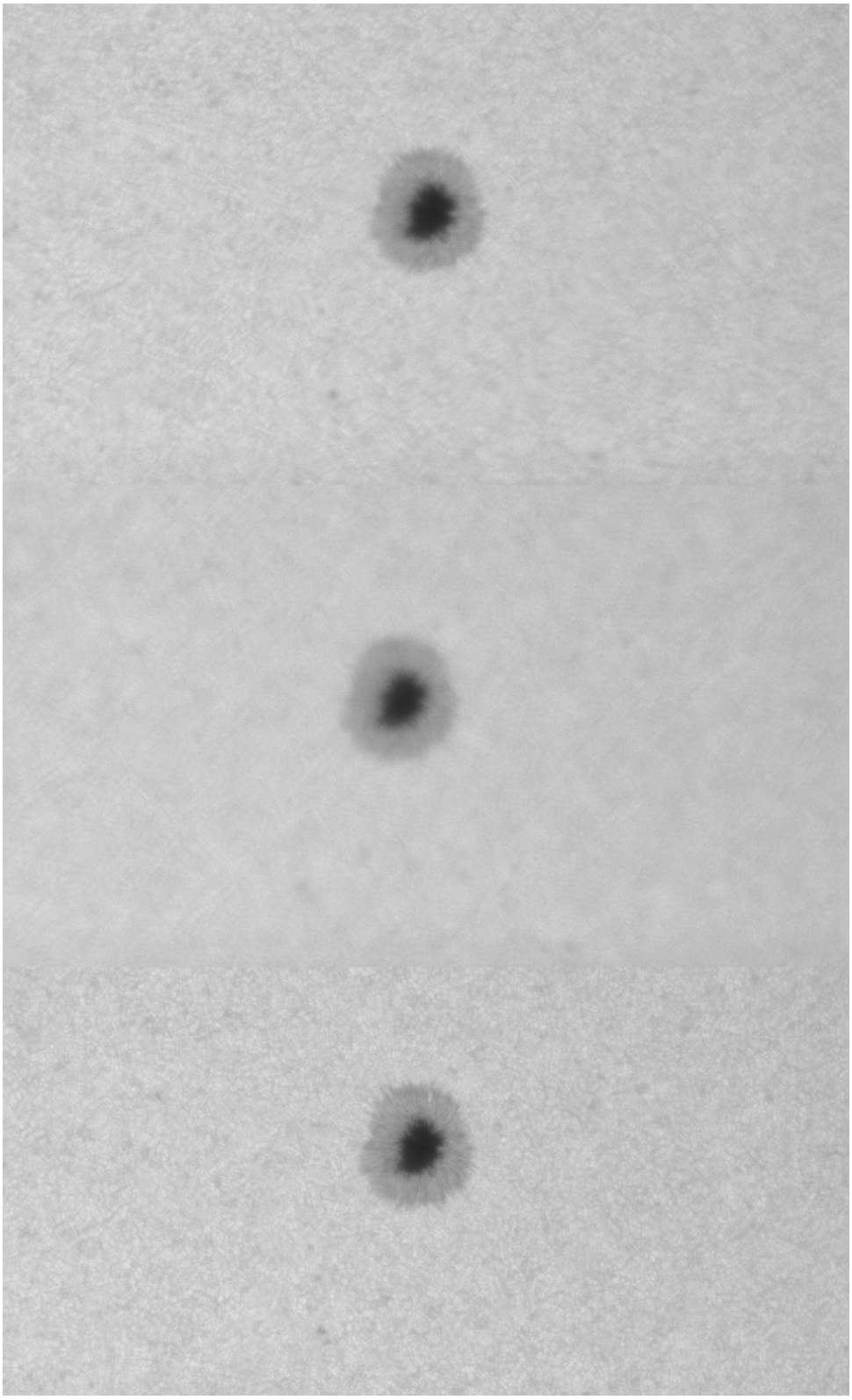

Figure 3: Results of quality estimation of a 1377-frame 60-second video of the Sun in white light (aperture: 180 mm). Top to bottom: the best image, the worst image, composite of best image fragments – note that in the composite, the granulation is visible everywhere. Estimation area size: 40x40 pixels.

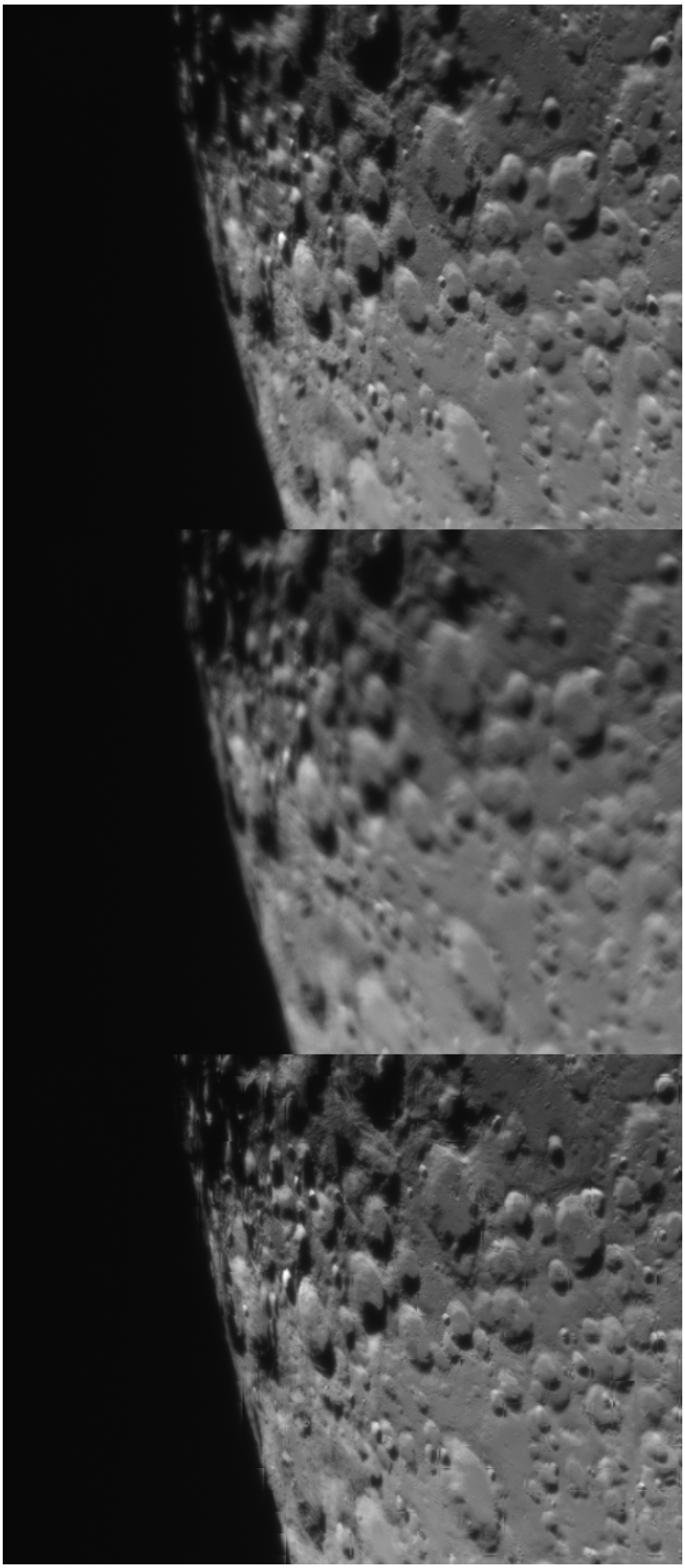

Figure 4: Results of quality estimation of a 500-frame 17-second video of the Moon (aperture: 300 mm). Top to bottom: the best image, the worst image, composite of best image fragments. Estimation area size: 40x40 pixels.

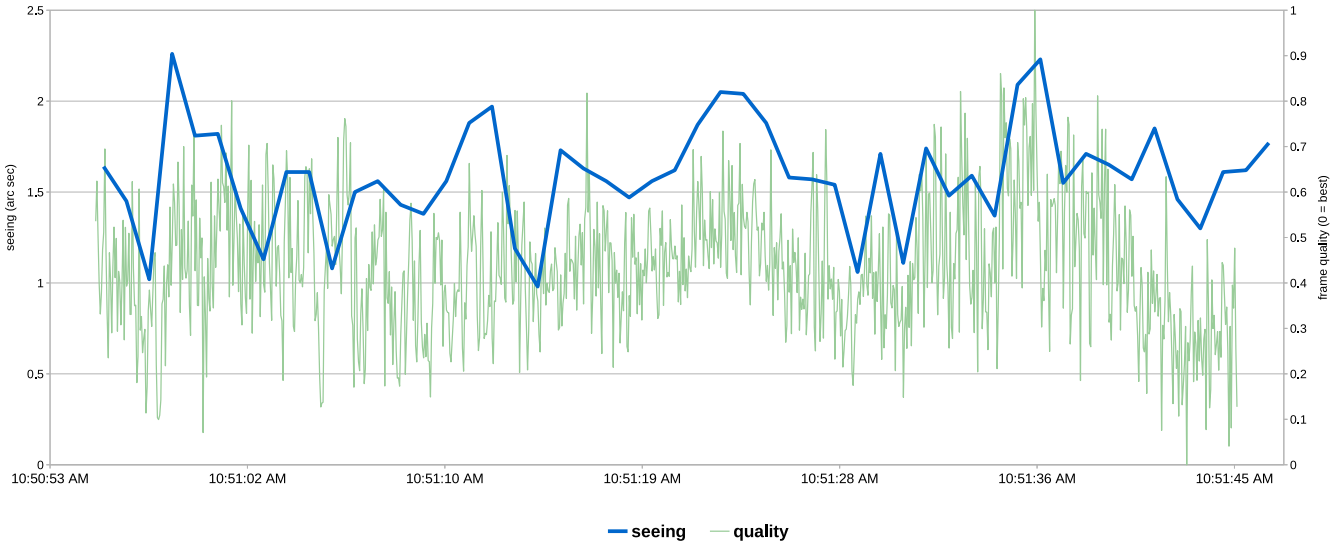

Figure 5: Comparison of the frame quality and the output of a *Solar Scintillation Seeing Monitor* for a 50-second 20-fps video of the Sun in Hα. The quality has been mapped to the range $[0, 1]$, where 0 = best, 1 = worst. Better (lower) seeing values in arc seconds generally correspond to better frame quality. Note, however, that an image deformed without blurring does not decrease the quality value, but may correspond to a stronger momentary scintillation.
EJ Seykora *Solar scintillation and the monitoring of solar seeing*
`http://adsabs.harvard.edu/full/1993SoPh..145..389S`.

6.3 Reference point alignment

Apart from inducing blurring, atmospheric seeing also distorts the images. These deformations have to be compensated for during final image stacking.

The distortion is tracked by placing a number of reference points within the images' intersection (6.1) and tracing their positions throughout the image sequence via block matching (5). Each reference point has an associated *reference block* (of user-specified size) copied from the underlying quality estimation area's reference block (fig. 2); i.e. it contains the best-quality fragment of the whole image sequence at this position.

Listing 5: Reference point alignment (`ref_pt_align.h/.c`)

```
SKRY_RefPtAlignment *SKRY_init_ref_pt_alignment(
    const SKRY_QualityEstimation *qual_est,
    /// Number of elements in 'points'; if zero, points will be
    /// placed automatically.
    size_t num_points,
    /// Reference point positions; if null, points will be
    /// placed automatically.
    /** Positions are specified within the images' intersection.
        The points must not lie outside it. */
    const struct SKRY_point *points,

    /// Criterion for updating ref. point position
    /// (and later for stacking).
    enum SKRY_quality_criterion quality_criterion,
```

```
    /// Interpreted according to 'quality_criterion'.
    unsigned quality_threshold,

    /// Size (in pixels) of reference blocks used for
    /// block matching.
    unsigned ref_block_size,

    /// Search radius (in pixels) used during block matching.
    unsigned search_radius,

    /// If not null, receives operation result.
    enum SKRY_result *result,

    // Parameters used if num_points==0 (automatic placement
    // of ref. points) ------------------------------------

    /// Min. image brightness that a ref. point can be placed at
    /// (values: [0; 1]).
    /** Value is relative to the image's darkest (0.0)
        and brightest (1.0) pixels. Used only during automatic
        placement. */
    float placement_brightness_threshold,

    /// Structure detection threshold; value of 1.2
    /// is recommended.
    /** The greater the value, the more local contrast
        is required to place a ref. point. */
    float structure_threshold,

    /** Corresponds to pixel size of smallest structures.
        Should equal 1 for optimally-sampled or undersampled
        images. Use higher values for oversampled (blurry)
        material. */
    unsigned structure_scale,

    /// Spacing in pixels between reference points.
    unsigned spacing
);
```

6.3.1 Initialization

The user may place the reference points themselves, or choose automatic placement. In case of the latter, the following parameters control the placement:

- *spacing*

  Defines the step of a square grid; in each grid cell at most one reference point is placed. An additional search for the best position within a grid cell is performed (see the function `SKRY_suggest_ref_point_positions()` in `quality.c`).

- *brightness threshold*

  A value from $[0, 1]$, where 0 corresponds to the darkest and 1 to the brightest pixel

of all quality estimation areas' reference blocks (fig. 2). A reference point can only be placed at a position whose brightness in the corresponding quality estimation area's ref. block is >= threshold.

- *structure threshold & scale*

  Determine how much structure (high-contrast features) must be present at a location to place a reference point. Since the points will be tracked using block matching, this is assessed by checking the sum of squared differences of pixel values of the image fragment surrounding the reference point and image fragments in two concentric shells around it (fig. 6) — higher sum is desirable. See `assess_ref_pt_location()` in `quality.c`.

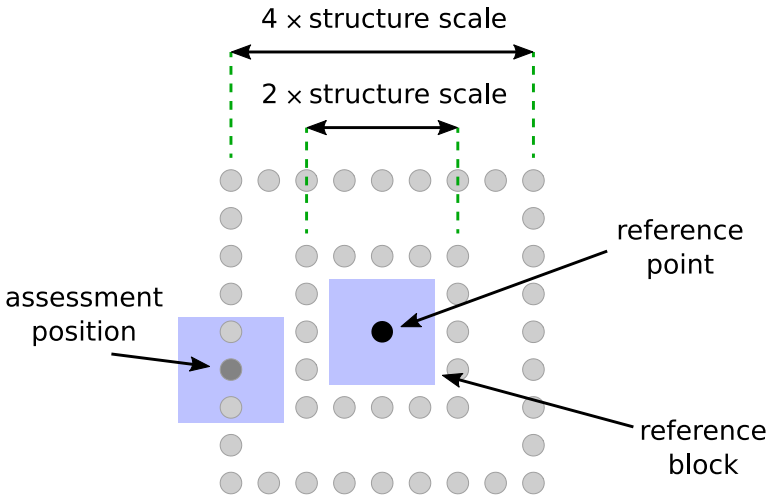

Figure 6: Assessment of the block-matching eligibility of a prospective reference point position. The reference block is compared to image fragments at surrounding locations in two concentric shells (gray dots). The higher the sums of squared differences of pixel values, the better the position (`assess_ref_pt_location()`).

An additional criterion for judging a reference point position is the distribution of gradient directions around it. If the histogram of gradient directions is insufficiently filled, the position is rejected. For instance, if the image is locally dominated by a single edge (e.g. the limb of overexposed solar disc, without prominences or resolved spicules), should block matching be performed, the tracked point would jump along the edge. See the function `assess_gradients_for_block_matching()` in `misc.c`.

6.3.2 Quality criteria

Finding of new positions of reference points in an image is performed only when the underlying quality estimation area has (in this image) quality that meets the user-specified criterion (see the enumeration `SKRY_quality_criterion`). It can be one of the following:

- *Percentage of best-quality fragments*

  Only use the specified % of the best-quality fragments.

- *Minimum relative quality percentage*

  Only use the fragments with quality above the specified threshold: % relative to $[min, max]$ of the corresponding quality estimation area.

- *Number of best-quality fragments*

  Only use the specified number of best fragment.

6.3.3 Triangulation and point tracking

The final processing phase, image stacking (6.4), operates on triangular patches of the region of interest (6.1). The triangles are created as follows (fig. 7):

- a number of additional fixed reference points are added outsite the region of interest, along its edges
- three fixed reference points are added, creating a triangle enveloping all the other points
- a Delaunay triangulation of the whole point set is determined

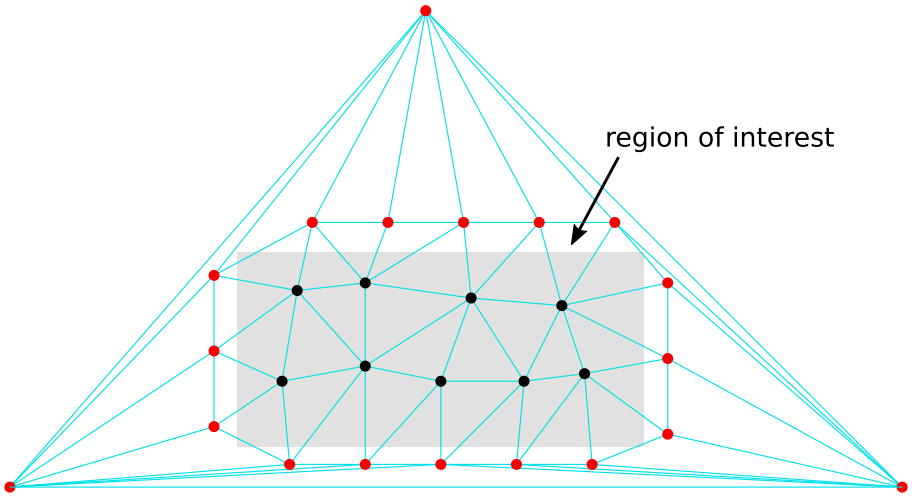

Figure 7: Triangulation of reference points. Black dots: regular (tracked) points, red dots: fixed points. (see `append_fixed_point()`).

Once the triangulation is known, for each input image, for each triangle, the new positions of its vertices (except for the fixed points) are found by block matching (5) iff the current sum of qualities of their underlying quality estimation areas meets the user-specified criterion (6.3.2). (A check is performed not to update a vertex more than once.)

This process produces a list of valid positions of each reference point (`ref_pt_align.c`):

```
struct reference_point
{
    //...

    struct
    {
```

```
        struct SKRY_point pos;
        int is_valid; ///< True if the quality criteria
                      /// for the image are met.
    } *positions; ///< Array of positions in every active image.
};
```

The per-image positions are then averaged to provide the final/undistorted position of each point (`SKRY_get_final_positions()`), later used for stacking. The averaging is important for making time-lapse animations (i.e. sequences of stacks made of a number of input videos); otherwise, if a randomly distorted reference point position were taken as the "true" position, each stack would have a slightly different geometry and the whole time lapse would be "wobbly" (showing deformations frame-to-frame).

6.4 Image stacking

Image stacking is the shift-and-add summation of high-quality (6.3.2) fragments, eventually producing a *stack* with improved signal-to-noise ratio and input video deformations removed.

6.4.1 Initialization

Listing 6: Stacking initialization (`stacking.h/.c`)

```
SKRY_Stacking *SKRY_init_stacking(
    const SKRY_RefPtAlignment *ref_pt_align,

    /// May be null; no longer used after the function returns.
    const SKRY_Image *flatfield,

    /// If not null, receives operation result.
    enum SKRY_result *result
);
```

Stacking operates on the triangular patches (6.3.3) which cover the region of interest (6.1). Each triangle is rasterized — a list of its pixels is created, together with their barycentric coordinates (fig. 8); the averaged point positions are used here. [4]

Then the stacking buffer (accumulator) is allocated; for each pixel of the region of interest, it contains an accumulated pixel value (single-precision floating-point value per channel) and a counter which stores the number of source pixels used to calculate the accumulated value.

6.4.2 Summation

The shift-and-add summation proceeds as follows:

- For each input image:
    - For each triangle, if the quality criterion is met (6.3.2):
        - For each pixel from the triangle's rasterization list (fig. 8):

[4] The triangulation was determined for the initial reference point positions; with the averaged positions, the Delaunay condition may be no longer met. This is of no consequence for stacking.

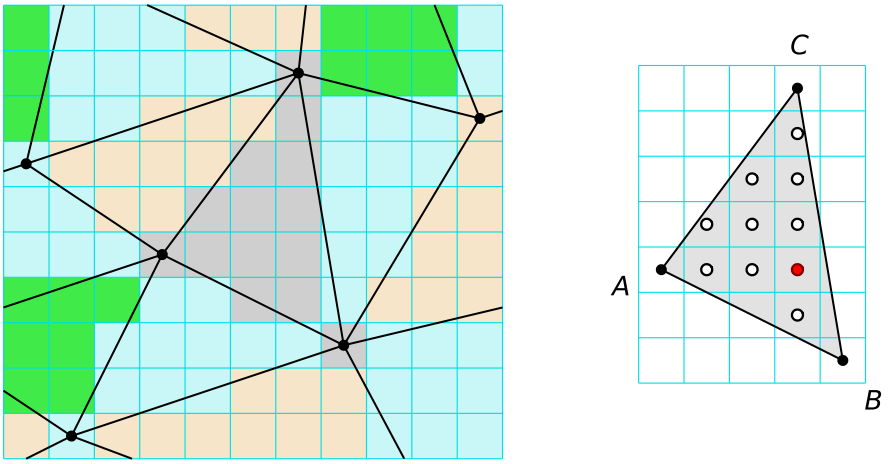

Figure 8: Left: rasterization of triangular patches (see `rasterize_triangle()`). Right: triangle's pixels (black, white and red dots). Barycentric coordinates of the pixel marked with red dot are $u = \frac{2}{11}, v = \frac{6}{11}$ (corresponding to $\vec{CA}$, $\vec{CB}$).

- Use the barycentric coordinates to find the (deformed) pixel coordinates in the input image, perform bilinear interpolation of pixel values in the input image, add the interpolated value to the stacking accumulator.

In addition, if the user has specified a *flat-field*[5], the source value is corrected by the local flat-field value (taking the image alignment offset (6.1) into account). The whole process is illustrated in detail in figs. 9, 10.

Finally, the complete stack is produced by dividing each accumulated pixel value by its corresponding counter. (Also, at each step of the stacking phase, a partial stack can be generated the same way.)

7 Post-processing

An image stack (6.4) is typically fuzzy (unless the input video was captured in perfect seeing conditions) and requires post-processing, which usually includes sharpening and tone mapping (fig. 11). The latter especially benefits from using a stacked image, which has a much improved signal-to-noise ratio; for instance, the darker areas can be "stretched" quite liberally before image noise becomes visible.

[5]An image, preferably captured by pointing the optical system at a uniformly lit background, which shows illumination non-uniformity caused by the optical train, e.g. vignetting, etalon "sweet-spot", etc.

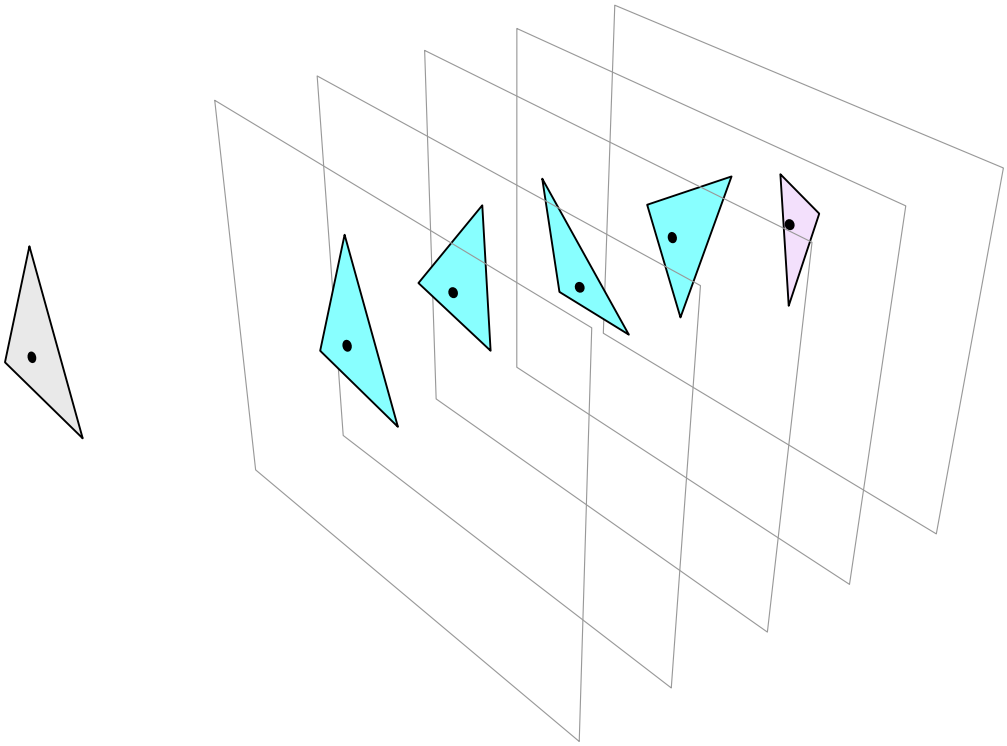

Figure 9: Left: triangle in the stacking accumulator, with one of its pixels marked. Right: the same location in the deformed triangles in input images.

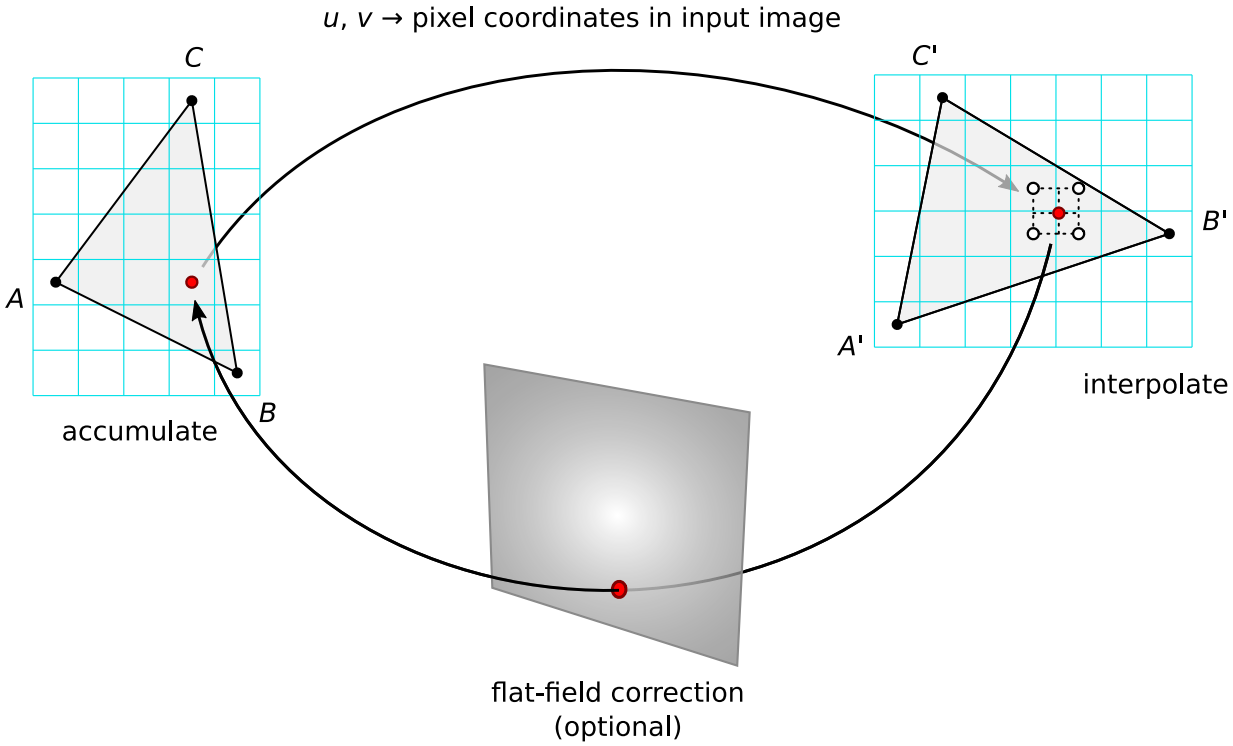

Figure 10: For each pixel of the stacked triangle $\triangle ABC$ (red dot), its position in input image is obtained by mapping barycentric coordinates u, v from $\triangle ABC$ to the deformed $\triangle A'B'C'$. After bilinear interpolation of the closest neighbors (white dots), the value is added to the stacking accumulator.

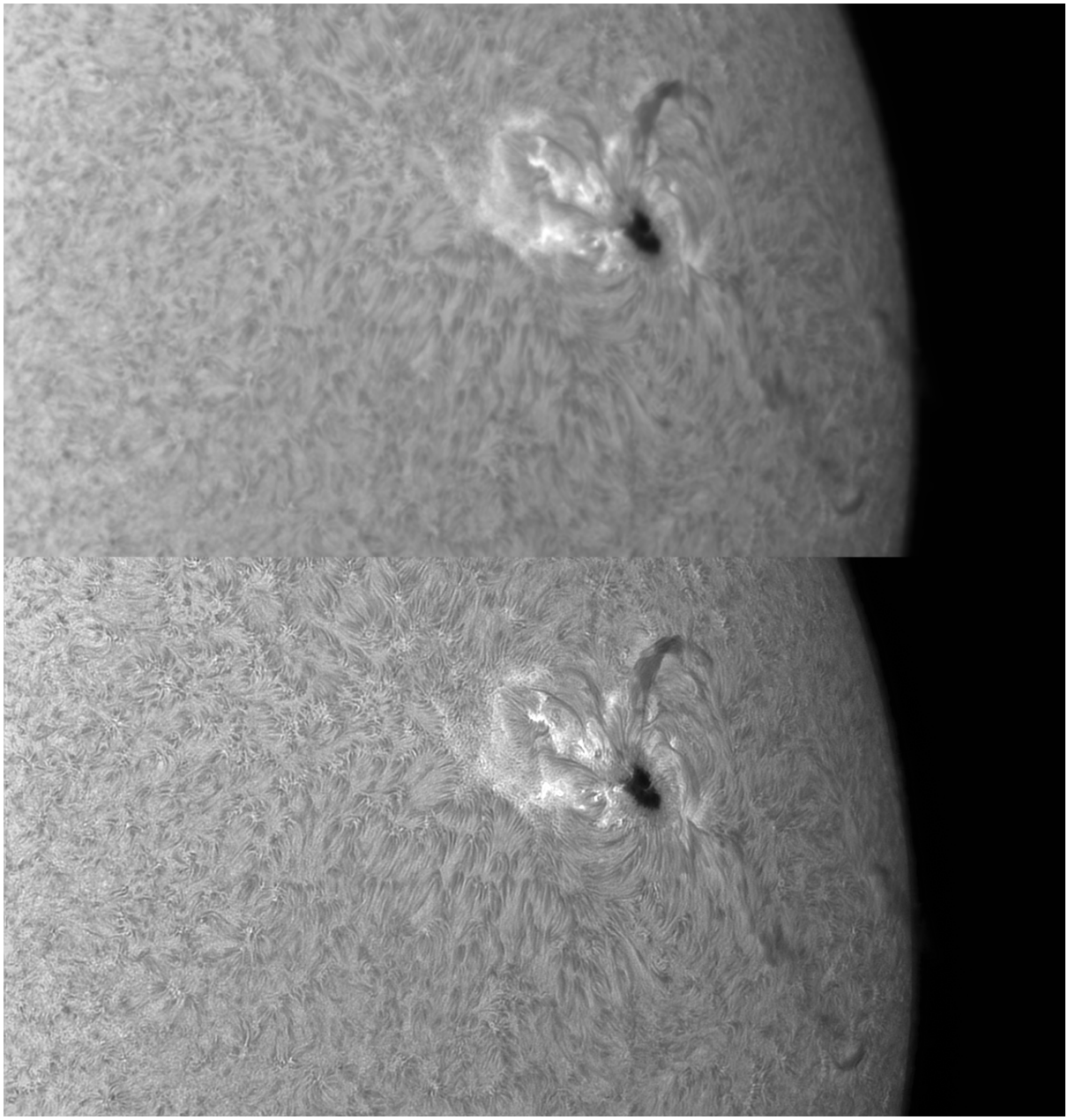

Figure 11: Top: image stack (Sun in Hα, aperture: 90 mm). Bottom: image stack with Lucy–Richardson deconvolution and unsharp masking applied in *ImPPG* (`https://greatattractor.github.io/imppg/`).