Sean Sellers

October 14, 2025

SSOSoft for ROSA and HARDcam

Contents

1 Introduction to the Code

SSOSoft [1] is the full-package facility reduction code for instruments currently in operation at the Dunn Solar Telescope. The module, `ssosoft.imagers` is designed to carry out all necessary calibrations to bring the raw, Level-0 data to Level-1.5 status, which includes flat-fielding, dark-correction, speckle-reconstruction, destretching, and optionally, alignment to solar coordinates.

Reductions are performed by setting up a configuration file (see Section 7), and then, in a python session, the full calibration stack can be run with the following code;

```
import ssosoft, os
rosa_filters = ["ROSA_GBAND", "ROSA_CAK", "ROSA_4170", "ZYLA"]
config_file = "config_rosazyla_20250903.ini"
base_directory = "/sunspot/solarchive/2025/09/03/level1"
for filt in rosa_filters:
    rz = ssosoft.rosaZylaCal(filt, config_file)
    rz.rosa_zyla_run_calibration()
    ks = ssosoft.kisipWrapper(rz)
    ks.kisip_despeckle_all_batches()
    os.chdir(base_directory)
ds = ssosoft.RosaZylaDestretch(rosa_filters, config_file)
ds.perform_destretch()
ds.register_rosa_zyla_observations()
```

At this point, the reduction process will begin. The pipeline should run without user intervention, provided no issues with the raw data (such as corrupted files). This process takes a significant amount of time, mostly due to the speckle reconstruction, through the destretch takes a non-negligable amount of time as well. When using all cores on SSOC, a typical 90-minue observing sequence will take upwards of a week to reduce. It cannot be sped up without either eliminating speckle (not recommended), or running the reductions on a different machine.

Note that there are three different classes used in the calibration routine: `rosaZylaCal`, `kisipWrapper`, and `rosaZylaDestretch`. These each perform a single section of the full calibration. The first two modules were originally written by Gordon MacDonald, who now works with APO, and modified somewhat by me. The last is entirely written by me, including the underlying destretch algorithm.

Once properly-configured, the ROSA/HARDcam calibration pipeline will perform the following corrections/reduction steps:

- **rosaZylaCal (detailed in 3):**
- Determination of average dark current

[1] https://github.com/sgsellers/SSOsoft

- Construction of average solar flat
- Correction of the plate scale using the linegrid images
- Writes binary files for kisip reconstruction
- **kisipWrapper (detailed in 4):**
- Writes configuration files for kisip
- Triggers kisip for each batch
- **RosaZylaDestretch (detailed in 5):**
- Dumps speckled files to FITS
- Destretches speckled files
- Corrects orientation of cameras relative to telescope optical axis
- Rigidly aligns different channels using Air Force resolution target images
- Aligns images to solar coordinates
- Writes context movie for each channel

The final data product will be a sequence of FITS files, each containing one frame of speckled and destretched data. Each data file will contain full WCS information, and be compatible with high-order frameworks, such as Sunpy maps.

2 General Data Properties

2.1 What is ROSA/HARDcam?

The **R**apid **O**scillations in the **S**olar **A**tmosphere instrument (with the **H**α **R**apid **D**ynamics **Cam**era) is a centrally-synced rapid imaging system that is the evolution of the **R**apid **D**ual **I**mager, which operated at the DST in the 1990's and early 2000's. At it's most basic, ROSA is a series of 1k×1k electron-multiplying CCDs with 8 μm square pixels (operated in non-multiplying mode). These are centrally-synced through a unique trigger box unit, built by Andor (I think) with multiple channel outputs for frame division. The ROSA cameras run through an ancient linux-based server that talks to each frame-grabber computer, each of which has its own internal storage. The das1 computer is the master channel, and all others are subordinate to it. The das1 computer sets the cadence, so for a 33.3 ms/30 fps cadence, the other computers will either grab at that cadence, or some integer division (e.g., 15 fps, 10 fps, 7.5 fps, 6 fps, 5 fps...) thereof. The HARDcam unit is a newer camera than the ROSA cameras. It is an Andor Zyla 2k×2k CMOS camera with 6.5 μm pixels. It is run through the Andor proprietary Solis data management system, which is slightly less-than-ideal overall. While ROSA writes 256-image extension FITS files, with timestamps, the buffering on Zyla means that it writes binary files with almost no metadata. The readout time for it, however, is 0 (rolling shutter mode), so as long as you know the exact starttime and exposure

time, you can reconstruct the time series. In order to ensure good tracking of starttime, HARDcam is set to wait for a ROSA trigger pulse before starting acquisition, so its starttime is the same as ROSA's, which can be read directly from the ROSA filenames.

HARDcam produces most of the facility's data. Since it writes uncompressed binary images with a 4 MP frame size, ~ 30 times a second, the data volumes climb quite quickly. If the storage on the server ever starts to fill up, one quick way to salvage some space back is to convert the binary files to FITS, and then use fpack to compress it, which is lossless for integer data[2].

ROSA produces the second-largest data volume, however, as it comes pre-packed in FITS with integer data, you can natively run fpack on the files using some variant of a `find ... -exec fpack ...` call straight from terminal. This was the standard practice when the NSO was hosting ROSA data, and a good portion of the SSOC data is compressed in this way as a result. The fpack and funpack commands are part of the cfitsio library, which can be either installed globally, or, if you're using conda for Python package management, having an environment with astropy, gcc, and gfortran should include cfitsio.

The ROSA system is highly configurable, and is frequently run with different camera lens focal lengths in order to either select for diffraction-limited imaging (800 mm lenses on ROSA), or wider field imaging ($\sim$ 300 mm to sample most of the DST FOV). My preference, and that of Dave Jess, ROSA's owner, is for a 500 mm lens on the ROSA cameras, which gives a $\sim$ 100" field size. This is slightly undersampled, but close to the diffraction limit, and large enough for the interesting sections of most active regions. HARDcam should only ever use a 450 mm lens, which actually slightly oversamples the field at 656 nm, but also images the entirety of the 173" field stop at the primary focal plane. A longer lens gets you nothing per the Nyquist condition, and a shorter lens just means you're imaging the edges of the field stop[3].

ROSA and HARDcam filters can be from several sources. Typically, the ROSA das1 camera uses a 9.2 Å G-band filter, the das2 camera uses a broadband ($\sim$ 30 Å) 417 nm continuum filter, and das3 uses a 1.2 Å Ca K filter. Prior to 2022, das4 was typically run with a 350 nm continuum filter, however, since the installation of FRANCIS, it now uses a 1 nm G-band filter to image the reflection from the FRANCIS fiber ferrule. Other configurations may include a selection of lines through the UBF, which has a 0.25 Å (ish) bandpass, either as a das4 channel, or as a replacement for das2. For practical reasons, mostly due to the length of the cables and position of the ROSA servers, it is usually easier to run a UBF channel with das2. Common UBF lines include Hβ (which is quite slow, < 6 fps), Na I D (which can usually hit 15 fps, assuming no 50/50 splitters in beam), and the Hα blue wing (which can usually get 30 fps). There are a number of prefilters for the UBF, and you're welcome to hunt through them for interesting bandpasses, but there are a couple things to keep in mind:

---

[2] Please note that fpack is *not* lossless on floating point data, i.e., reduced data. Compressing Level-1 data will not gain you very much extra space, and comes at a significant cost in terms of fidelity.

[3] DST's port4 can not, in fact, effectively use any camera with more than 2k pixels in either dimension (except in NUV applications, barely). In the IR, even less is needed. The math is left as an exercise to the reader. It's a 76.2 cm telescope.

- The UBF has a low, unknown (to me) throughput. I've heard it quoted as a couple percent. It's a Lyot filter, it's a ton of optical elements.
- The throughput is worse in the blue *and* the red. Its effective range is 380–700 nm, but anything short of 500 nm is quite poor.
- It's huge, and requires the user to set up a reference and signal diode in order to calibrate the central wavelength.
- The calibration drifts, so the observers will need to repeat it several times a week.
- While it can scan a spectral line, it's slow to change wavelength positions, and cannot be synced with ROSA easily. You'd be looking at several seconds to scan a line, minimum.

Between the above items, I almost never have the time, bench space, or light to set it up. There are, however, potential uses for it as a long-term Hα wing, Na I D, or Mg I b filter, and there's a way it could be set up (with das2) alongside FRANCIS, so long as the user is alright sacrificing the FRANCIS red bands. Contact me for details.

HARDcam uses a different Lyot filter, built by Zeiss in the 70s with a switchable 0.25/0.5 Å bandpass (though we don't use 0.5). It is also tunable (via the gray gear on the top of the case), and you should keep an eye on its tuning as well. I have a suspicion that the temperature stabilization is a bit off, and that's affecting the central wavelength. If you look at the indicator on the top, you'll notice I left it tuned about 0.25 Å off of the theoretical center. That was my best guess for the actual core of the line from some trial and error testing in April 2025. You could probably adjust the heater to get core in the correct spot, but it takes longer to wait for temperature equilibrium than it does to spin a dial, so keep that in mind as well.

There are a few other filter of interest in the Chief Observer's Office (aka Doug's office). I haven't found a use for any of them, but there are:

- A couple CN filters. These'll show similar structure to G-band, but further in the blue (and therefore in the telescope's worse transmission region). Plus these are older filters. You could use them to push the maximum resolution, but keep in mind that a wider filter bandpass will affect scintillation, and the AO is not as effective in the UV.
- Extra G-band and Ca K filters. The ones in use are the best, but these could be used for alternate setups or differential imaging with different bandpasses.
- A few narrowband 530.3 nm filters (∼a few Å). It's a coronal line, but who knows.
- An (I think) 854.2 nm etalon filter. It's some Ca II filter for sure. It's a temperature controlled unit in a wooden box. Never found the power cord, but you could bother Shane or someone to make one.
- A second powered etalon with a partially melted cord. It's probably Ca K, but I never managed to test it.

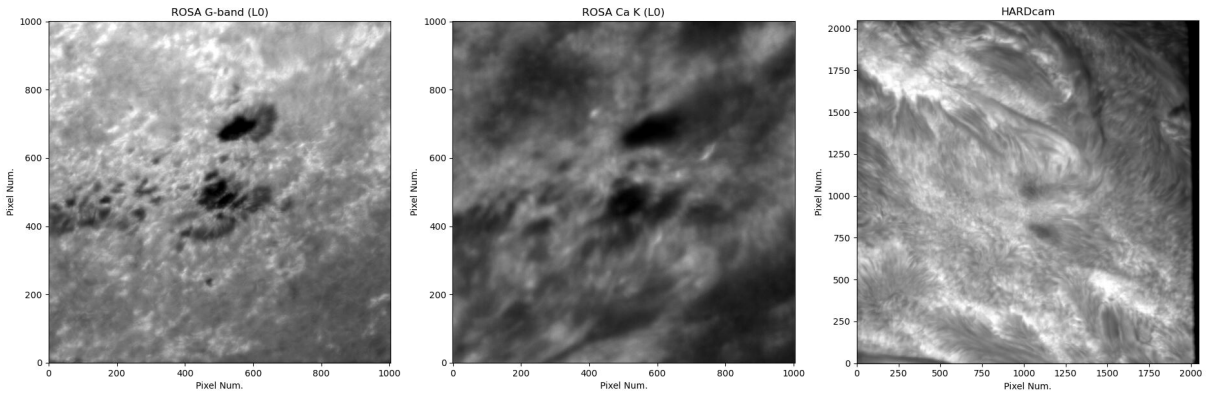


Figure 1: *Level-0 ROSA/HARDcam data example from 2025-09-03. These were the first images acquired in the day's observing sequence. Note that HARDcam channel is not in the same orientation as the ROSA channels. The various channels can all be in different orientations relative to each other. The dark bar on the right of the HARDcam image is the edge of the DST port4 field stop, which can creep into the frame, depending on the beam wobble and orientation of the telescope.*

- There's an etlon in the guider cam box for Hα. It's something like 0.75 Å. You'd have to put a different filter (and ND) in place of it, but in a pinch, you could use it for a science camera. We did that for the annular eclipse in 2023.

There is also a list of old Lyot filters we should have, but I've never been able to find. Doug Gilliam would know where they are, but odds are they're in the Evans somewhere. I doubt they'll be useful, but you never know.

2.2 Level-0 Data Properties

2.2.1 Level-0 ROSA data

ROSA raw data are stored in multi-extension FITS files. The 0th extension will always contain a header with some instrument configuration information, and no data. The following extensions will contain each a single image, and the `DATE` keyword in the extension header will have the starttime of the exposure. The raw files contain 256 image extensions, for a total of 257 extensions overall. A given observing sequence may have several hundred multi-extension FITS files. The data files are named `cam#_rosa_YYYY-MM-DD_HH.MM.SS_iiii.fit` [4]. Here, "CAM#" identifies the camera computer (das1–6), "YYYY-MM-DD" is the date of acquisition, "HH.MM.SS" is the starttime of the observing sequence in UTC (so all files in a sequence will have the same time, irrespective of the starttime of the first image in the file), and "iiii" is the number of the file in the sequence, starting at "0000". For some reason I never figured out, das3–6 observe daylight savings time, and the filenames from March through November will be off, as will the timestamps in the FITS headers. There's a switch in the code to correct this, but keep it in mind when you're setting the file patterns in your config file.

[4] note, ".fit", not ".fits". Only raw ROSA files are named ".fit" in the archive, and this can be an easy way to select raw ROSA data.

Flats and Darks have a modified naming scheme, `cam#_rosa_flat/dark...`, which allows you to pick these out easily. Target, linegrid, dotgrid, pinholes, and any other fiducial maps do not have an alternate naming scheme, and are named in the same way as regular data files. You'll have to consult the day's paper observing logs to determine which are which. Or open the actual files.

2.2.2 Level-0 HARDcam data

HARDcam data are spooled to binary files directly from the Andor Solis program. These files are named according to their number `iiiiiiiiiispool.dat`, where "i" is the 10-digit number of the frame. These are ordered by least significant digit first, e.g., `0000000000`, `1000000000`, `2000000000`, ..., `9000000000`, `0100000000`. There is a function in SSOSoft to reorder the filelist correctly, as well as to read the data from disk. These data are platform-native binary, and can be read with numpy:
`np.fromfile("0000000000spool.dat", dtype=np.uint16).reshape(2050, 2060)`
into a 2050×2060 pixel array. The SSOSoft function will also remove overscan rows and columns (which are the last two rows and columns), plus the bias columns to cast the data into a 2048×2048 array. The Solis program can also window the chip to save smaller regions of interest, in which case, the user will have to know the original data shape to accurately read the files. In practice, this functionality was only used briefly during the commissioning of the camera in 2019, and won't be particularly relevant most of the time.

Since the HARDcam data are not written with timestamps or any usable timing information whatsoever, keeping track of the exposure time and start of observing is vital. The observers will name the file directories with the starttimes, e.g., `250903_ZYLA/DBJ_data_141613_observation`, but these are the times the first file is *written* to disk, not the actual observing time. These may be different by a couple of seconds depending on the buffer. The best way to get the starttime is from ROSA. HARDcam is almost always (and should be always) operated in conjunction with the ROSA system, using one of the ROSA trigger lines to start the sequence. While it won't be centrally synced, the first file will have the same starttime as the first ROSA file. As HARDcam has a rolling shutter, the second file will have a starttime equal to the original starttime plus the exposure time, and so on. From this, the timestamps can be reconstructed.

Some HARDcam data, particularly during filament observations, were taken without ROSA coordination, with an odd cadence. During filament observations, the triggering was set up to use an Agilent signal generator, which produced a burst of 64 continuous images, waited a minute, then repeated. Reconstructing these timestamps is more difficult, but these images are usually used for context for FIRS observations of the filament, rather than rapid dynamics.

While the Andor Solis program is supposed to be able to write the data to FITS, with metadata including the starttime, in practice, this functionality does not work and crashes the computer. So we're stuck with the less-than-optimal solution.

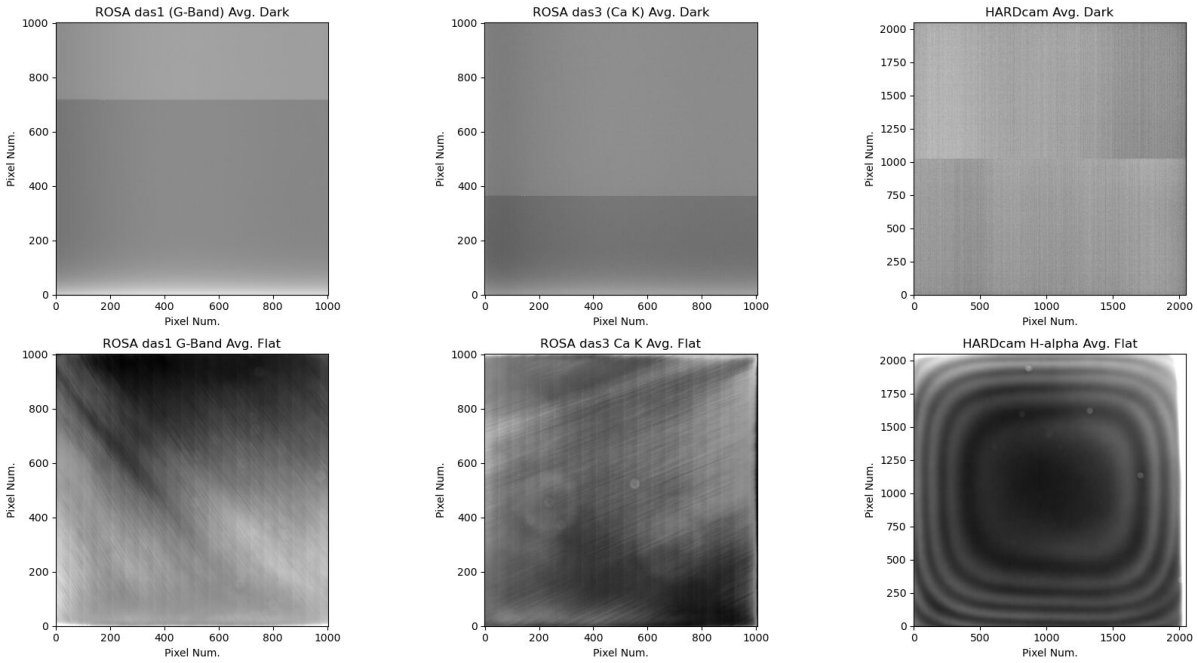


Figure 2: *Average dark and flat examples for two ROSA cameras and HARDcam. The variance on the dark frames is on the order of 10 counts. The splotches on the ROSA Ca K flat field are likely due to debris on the filter, which is mounted on the camera body. The rounded-square fringes on the HARDcam image are due to the window over the camera chip. They are stationary and completely corrected by the flat-fielding procedure, but can be quite noticeable on the raw images, particularly if the seeing is poor.*

2.3 Calibration Images

2.3.1 Flats and Darks

Darks and flats are typically taken both before and after the main observing sequence. Flats are taken with the telescope randomly slewing across a quiet region, usually near disk-center, although PIs may request flats taken at a similar μ-angle. During flat-fielding the deformable mirror in the AO is also set to "unflat" mode, which has the effect of defocusing the image further. The unflat mode is actually an alternating oscillation on all actuators, but it does the same thing as defocusing. If you are at the telescope during observing, keep an eye on HARDcam during flat fielding. There is an occasional issue where there are patches on the flat field, where you can see a localized ridge pattern on the camera. This indicates debris at a location near, but not at, a focal plane. Typically, it's on the exit window, and will need to daubed off with compressed air and lens cleaner.

Darks are usually taken with the dark slide in. The dark slide, along with the pinhole, linegrid, and target, is a physical slide in a filter wheel kept at the telescope prime focal plane. The dark slide is a milled block of aluminum that blocks the solar light. On cameras without internal shutters (SPINOR, HARDcam, FRANCIS), the dark frames also technically contain some stray light. In practice, I've never really seen any stray light contamination from this. ROSA cameras have an internal shutter, HARDcam does not. Typically, around a thousand dark frames are taken, and 2000 flats. That's the master cadence.

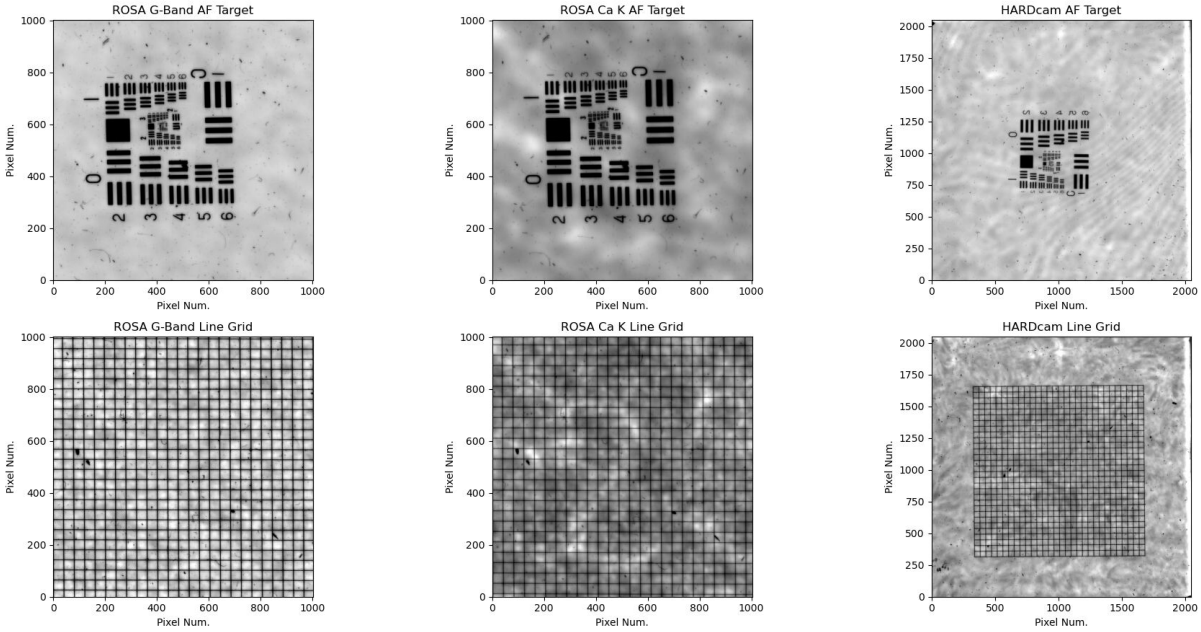


Figure 3: *Air Force Target and Line Grid Images from 2025-09-03. ROSA data from this date were taken with a* ~ 100*" FOV.*

Since ROSA does frame division, the slower channels will have fewer flat and dark frames than the faster ones.

ROSA cameras are Peltier-cooled to -70 C (or similar; some only get to -40, while HARDcam is liquid-cooled to -20 C. The dark signal is usually quite low.

2.3.2 Linegrids and Targets

These images are taken at the end of daily observations, and are used for various registration and calibration purposes. The slides for this are in the same filter wheel as the dark slide, which means they are in the shared beam, accessible to all instruments simultaneously.

The Air Force Resolution Target slide is meant to test the maximal resolution of an imaging system. When re-arranging the optics, it is used for checking focus. In the calibration pipeline, it is mostly used for determining the correct orientation of the cameras (which is *not* the same as the correct orientation of the slide, as the target is mounted upside-down in the filter wheel. See 4 for the correct orientation relative to the telescope optical axis.) It is also used in post-destretch registration to determine the relative offsets and rotations between cameras. A reference camera is chosen, usually G-Band, and offsets and rotations are determined relative to the reference channel.

The line grid slide is meant to fine-tune the plate scale of the cameras. The lines are spaces exactly 1 mm apart. Since the plate scale at the prime focal plane is ≈ 3.76"/mm, the number of pixels between lines can be used to determine the "/pixel of the camera in a more precise way than calculating it from magnification via the imaging lenses. During camera setups, I also use the line grid map to check for astigmatism in the system. Astigmatism is the optical aberration that manifests as a differing focal position in X vs Y. Since the grid is highly structured along X and Y, it is good for checking for astigmatism, which will

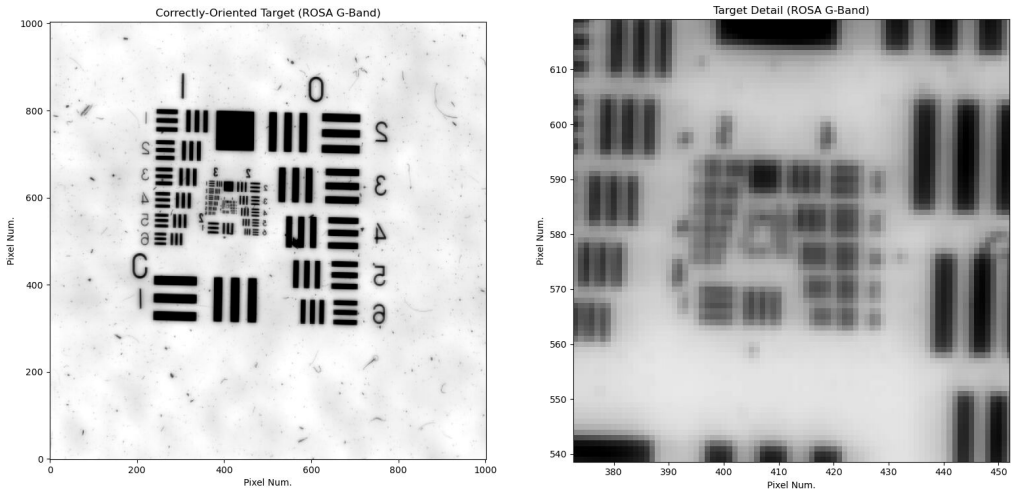


Figure 4: *Correct orientation of the AF Target to place images onto the telescope optical axis. If the telescope guider angle is at 13.3 (which is Solar-North), up in these images will correspond to Solar-North, left to Solar-East, etc.. The right panel shows a detail of the target image. Note that structure continues down to individual pixels. When you start seeing pixel-level structures, you've got pretty decent focus. Note that, if you lock the AO on the AF target, you can get slightly sharper and more stable. This is the second nested structure in the left-hand image, for reference.*

show up as ghosting along one direction or the other. Check the FRANCIS ferrule camera for what that looks like. Typically, astigmatism arises when the beam is not centered on one or more lenses, or if the lens is tilted relative to the beam. It is crucial to check both centering and back-reflection during setup to minimize astigmatism.

2.3.3 Dot Grids and Pinholes

These slides make up the last two positions in the port4 filter wheel (open slide for observations, line grid, target, dark slide, dot grid, pinhole). Neither are used for the current calibration pipeline. The dot grids are used for similar purposes as the line grid. The pinhole images are used by the observers to center the ROSA/HARDcam cameras before observing, as well as aligning the beam through the AO system. It would be tempting to use the pinhole for point-spread function deconvolution, but be warned that the pinhole is NOT a point source. It's a milled, round aperture with a 40 μm radius. When imaged by ROSA, using the 500 mm lenses that result in a $\sim$ 100" FOV, that's a bit more than 3 pixels for the pinhole, which is resolved. It's also not at infinity, it's internal to the optical system at the prime focal plane, which is after the entrance window, exit window, and primary mirror. It may be possible to use these images for a PSF estimation, but I've never tried it.

3 Initial Steps: rosaZylaCal

The initial step in the calibration pipeline performs the low-level corrections necessary to write the data bursts that are then fed into kisip speckle-burst reconstruction. This initial calibration is handled by

`ssosoft.imagers.rosaZylaCal`. The basic calibration routine determines the average dark and flat images, then uses these to create the gain image (normalized dark-subtracted flat). It then writes the burst files for kisip, which are binary files. Each pre-speckle burst contains a number of dark- and flat-corrected images, which are subjected to a triple correlation to recover a single image from the stack. We typically use 64-frame bursts, although 32 can be used to try and recover some extra cadence from the slower channels. 16-frame bursts are tricky, but I have successfully managed to reconstruct a few test image sets from such short series.

This module also writes a header file for each burst, which are used later to fill in the header information, such as the start and end time of the burst. As such, this module is also responsible for reconstructing the HARDcam timestamps. Since the ROSA and HARDcam systems are independant of the telescope, they also do not have access to the telescope logfiles natively. These contain such important information as the point, light level, scintillation values, and sundry other odds and ends, all of which are useful for performing science. Typically, the observers will package the raw data deliveries with both a scan of the paper observing logs they keep and the log of the telescope's positioning, called the DCSS Log. This can be provided to the code to backfill important telescope information to the header files. These are written to the disk as .txt format, and can be read by any text editor you like.

There is also optional functionality within the module to apply the dark and flat corrections to the average linegrid and target images, then write those to disk. It also fine-tunes the pixel scale from the linegrid image, and updates the values that the user provides. These updated values are also passed on to kisip, which results in a slightly better reconstruction.

Once the rosaZylaCal run is complete, the user should have a directory structure for the selected camera channel with:

- Directory: `preSpeckle`, containing speckle-burst files
- Directory: `hdrs`, containing proto-headers for final reduced files. These have corrected timestamps, and telescope parameters from the DCSS log (if available).
- Directory: `speckle`, currently empty, but will have speckle-reconstruction files written as kisip runs
- Directory: `postSpeckle`, for reconstructed files in FITS format, will be populated during destretch.
- Flat, dark, gain, and noise calibration files, named after the camera channel selected. For `ROSA_GBAND`, for example, these will be `ROSA_GBAND_flat.fits`
- (Optionally:) Average linegrid and target images for plate scale adjustment and channel co-alignment. Files named, e.g., `ROSA_GBAND_linegrid.fits` and `ROSA_GBAND_target.fits`.
- (Optionally:) The adjusted plate scale in the file `plate_scale.txt`. This can be read with, e.g., `numpy.loadtxt` with ordering x-scale, y-scale. The code does not currently use this, but it would be trivial to add a loadtxt call in the destretch module if the user wanted to alter this file themselves.
- The reductions log, named with the user-provided starttime.

Note that the `preSpeckle` directory is *not* cleaned up after speckle reconstruction. As such, the user should remove the directory after verifying their satisfaction with the run. Since the contained files are written in uncompressed binary, formatted for 32-bit floating point, the directory can be several TB in size. Failure to remove the directory can eat up a lot of valuable drive space.

4 Calibration Core: kisipWrapper

The core of the calibration pipeline is built around Friedrich Woeger's speckle-burst interferometry reconstruction code. The underlying code is written in C, and the version we use is precompiled along with the required dependancies in a single directory. The package is rather specialized, and was written quite some time ago. The pre-compiled version includes the very specific version of MPI that must be used to trigger the code in parallel. The code also includes a set of weights specifically calibrated for our AO system, which I do not understand and could not even begin to guess at. As such, please note that my knowledge of the inner workings of KISIP are very much "working" knowledge, not a true in-depth understanding.

For the most part, we've settled on a KISIP method that works well for typical DST data, using the triple-correlation method (rather than the Knoxville-Thompson algorithm). The parameters fed to KISIP are set in the main configuration file. There are a few items that are kept in each camera section of the config, and three additional sections of the config file dedicated to global KISIP params:

- `KISIP_METHOD`, which contains parameters written to the `init_method.dat` file. This contains information on the underlying setup and settings for the code. This is rarely altered.
- `KISIP_PROPS`, which contains parameters written to the `init_props.dat` file. This file contains information on the telescope and AO system, and is rarely altered, but there are a couple "cheats" to be aware of with these parameters. For one, the "kisipPropsTelescopeDiamm" sets the diameter of the telescope entrance window, which, for the DST is 762 mm. However, in cases of relatively poor seeing, KISIP has a tendancy to always reconstruct structures at the diffraction limit, even when no such structures are reconstructable. This can cause the data to tesselate. Setting the telescope size to a smaller diameter can mitigate this somewhat. Additionally, the "kisipPropsAoUsed" parameter should be set to 0 if the telescope was used with tip-tile support only (such as days where we used limb-tracking).
- `KISIP_ENV` controls where the package is triggered from. You should set it up to point to the `bin` and `lib` directories. The "kisipEnvMpiNproc" variable controls the number of threads to run KISIP on. SSOC has a paltry 24 threads, so it takes a while.

If KISIP fails on the first iteration, it is likely due to the "kisipMethodSubfieldArcsec" variable in each camera's config file section. At the moment, increasing this past 10 will cause the code to segfault, and I never had time to debug it.

Assuming everything is correctly set, the kisipWrapper class will handle writing the kisip-specific configuration files in the correct directories, and sheparding the data through reconstructions. Very occasionally, the code will quietly fail partyway through the loop. If this happens, you can alter the `init_file.dat` config file, and manually trigger the code to restart. This usually only happens if the server is overloaded.

Once the code is triggered, KISIP will begin reconstructing data in the `speckle` directory. Each image-burst will produce the following files:

- `.final`: The reconstructed image. This can be read into python with
  `np.fromfile(fname, dtype=np.float32).reshape(1002, 1004)`
  (or reshape to 2048, 2048 for HARDcam).
- `.subalpha`: The Fried parameter estimation for both the full image, and for each subfield KISIP used. This has shape (2, nsubfieldsx, nsubfieldsy).
- `.ct`: The correlation-tracking paramters for each subfield (x/y shifts to register each subfield). This has shape (burst number, 2, nsubfieldsx, nsubfieldsy)
- `.subrec`: The reconstructed data split into subfields, before recombining into a single image. This has shape (nsubfieldsx, nsubfieldsy, npixx, npixy), where the number of pixels is the number of pixels per subfield, usually a power of two.
- `.subamp`: Mean Fourier amplitudes of each subfield (I think). This has shape (nsubfieldsx, nsubfieldsy, npixx, npixy).
- `.subrsr`: Mean spectral ratio (I think) for each subfield. This has shape (nsubfieldsx, nsubfieldsy, ...), where the last dimension size depends on the data.

In practice, only the final and subalpha images are of use to us, and even then, only the first value in the subalpha image, as this is the mean Fried parameter for the frame, and can be used to determine which reconstructed image is the clearest. The other files are useful, I'm sure, to someone who is really familiar with KISIP. But that's not me.

This section of the reduction is by far the longest. For all cameras to run, an average dataset will spend almost a week in speckle-reconstructions.

5 Fine-Tuning and Cleaning

Once KISIP has run for each camera, the final phase of the calibration can be run. The destretch and optional co-alignment procedures are the last steps in the SSOSoft calibration pipeline, and any further analysis or reductions are solely the responsibility of the end user.

5.1 Destretch

The main calibration performed during this step is an iterative subfield-based destretch. The main module is based off an old IDL routine, `reg.pro`, which was used in most of the main calibration pipelines from the last thirty years, including the IBIS pipeline. In essence, this calibration compensates for two effects: the residual warping of an image due to residual motions not corrected by the AO system, and minor shifts induced frame-to-frame by the speckle reconstruction code. The first is easy enough to understand by watching the AO monitors during an observing sequence. Atmospheric motions are plainly visible with seconds-long timescales. The second is an artifact of the KISIP code. Since KISIP parallelized based on subfields, and since those subfields are co-aligned for a given image burst, there will be either a residual or an induced shift subfield-to-subfield along the reconstructed image stack. The destretch module attempts to blindly undo both effects and register each image in the final sequence to a common grid. This is done by a sequence of cross-correlations along subfields, with a warp grid determined by interpolating between each subfield.

This does have the unfortunate effect of supressing the magnitude of lateral solar flows. It generally does not erase them, but it may cause issues in, for example, loop oscillation studies. I have written a tack-on routine that attempts to determine the magnitude of solar flows and add them back in from the saved destretch vectors, however, I never got it to produce adequate results. Instead, it mostly induces lateral flows. Makes it look like the whole image is swimming. Switches for it are still there, and the underlying algorithm could be retooled if necessary.

The main `Destretch` class handles both the destretch and the registration, but the underlying workhorse is the `pyDestretch` class, which this module calls and depends on. That class handles the actual mechanics of the image registration and warping. It is generalized to work on pretty much any 2D data, and can be configured to return the destretch solutions, or apply a pre-existing solution. This funcitonality is not super useful for the ROSA or HARDcam cameras, as a portion of the warping is induced by KISIP instead of the atmosphere, but there are several possible implementations. For one, the FRANCIS ferrule imager cannot be speckled, but if the pre-speckle, post-flatfielding G-band data were destretched, those vectors could be applied to the ferrule camera in order to more rigidly align the fibers. Or, when combining multiple SPINOR channels into a single data product, e.g., for DeSIRe inversion, the 2D raster images from the subordiate channel can be destretched to the main channel, then applied across the wavelength axis to rigidly align chromospheric and photospheric structures.

Once the destretch routine is triggered, a directory is created in your workBase named `splineDestretch`[5]. After files are cast back into FITS format from the speckled files, the destretch routine runs and fills that directory. The destretch can be run either in "running" or "reference" mode. Running uses as a reference image the mean of the last "n" destretched images. This is the usual behaviour. I try to choose "n" such

[5] My... third?... iteration of the destretch algorithm explicitly constructed a B-spline for patching in the coordinate warp grid, which was differant than previous iterations which used affines[6] to fill. This is no longer an explicit step in the destretch, but the naming convention remains since all the archiving codes are geared to search for a splineDestretch directory.

that the reference image spans about a minute. A fixed reference can also be chosen to destretch the stack to a single image. This may be useful in shorter sequences or if there's a long section of the obs sequence with bad data (clouds, telescope issues, etc.).

The destretch is run on overlapping subfields. The side of these subfields is chosen by the `dstrKernel` attribute. This is given as a comma-separated list. My default is usually `0, 128, 64, 32, 12`. The leading "0" instructs the code to do a cross-correlation alignment for the full frame before attempting to align the subfields. The subfield sizes are given in pixels. 12 is usually the smallest I go. Below that, you're probably overcorrecting.

Speaking of overcorrecting, the "repairTolerance" keyword attempts to help with instances of overzealous shifting, setting the shifts of any subfield that exceed the tolerance back to 0. This is given as a fraction of the subfield, and defaults to 0.5, so for a 128-pixel subfield, any shift of 64+ pixels will be discounted as erroneous. If you start seeing subfields wiggling in your data, you may need to decrease this threshold.

The destretch routine can take anywhere from an hour to a day, depending on which camera and the length of the observing sequence. It is not parallelized, despite being a decent candidate for it. I did write a number of parallel versions, one using python's `multiprocessing` module, and one using `dask`, however, in every test I tried, the overhead required for the parallelization caused the result to run slower than the single-core version. This is mostly a limitation of SSOC, which has relatively few slow cores. A different machine with more or faster cores may be able to use the parallel version of the algorithm, and if you're interested, feel free to contact me. There may also be a way to speed it up with `numba`, which will pre-compile certain python functions for speed, but it's a very limited tool (you can't use any imports besides numpy), and can be pretty tricky to use. It is used in the crosstalk solver for spinorCal, but implementing it was such a nightmare that I haven't been able to convince myself to add it to the destretch.

By default, the code will save the per-subfield shifts in the "destretchVectors" directory in "workBase". These are the vectors loaded for the flow-preserving module, if used, and can be used in other applications, though it's not recommended for ROSA/HARDcam.

5.2 Registration and Alignment

Of course, a single fast-camera channel is not particularly useful on its own. However, co-aligning multiple channels is non-trivial. In order to do this, a reference channel must be chosen, ideally a photospheric channel, such as G-Band or the 417.0 nm continuum. For the channel co-registration, a reference image, usually the Air Force Focus Target must be used, but, in a pinch, you can set the target file name pattern to be any image you'd like. If you're observing a distinct sunspot, it'll work about the same. Since each camera has a different set of translations on the image according to the configuration of the relay optics, the "channelTranslation" keyword must be specified to translate each camera's target image to the correct orientation. Since all of the ROSA cameras and HARDcam are side-mounted, the sequence of

translations will probably include a 90-degree rotation. Once the target is correctly oriented, the oriented image is written to the "workBase" directory. From here, the `ssosoft.tools.alignmentTools` module takes over.

The alignmentTools module makes extensive use of Sunpy's Map class and functionality. As such, the headers on any file destined for alignment must contain a complete set of WCS cards that detail pointing and rotation. Once read into a Sunpy Map, reprojection functions handle interpolation and resizing in preparation for the actual alignment functions. The underlying alignments are mostly cross-correlation-based. Cross-correlations determine the subpixel offsets between a given image and a reference, in this case, between a single channel's AF Target image and the reference channel AF Target image. After shifting the channel's image to be aligned with the reference channel, the rotational offset is determined by performing a series of rotations on the channel image in small increments, and determining the linear correlation between the rotated and the reference image. The grid of rotation/correlation values is interpolated to find the rotation angle resulting in a maximum correlation. From this rprocedure, relative shifts between a channel and the reference, as well as the relative rotation, can be determined. These offsets are applied to the CRPIX1/2 and CROTA2 WCS keywords in the FITS headers. This is done instead of shifting and rotating each image in the reduced series. Doing the alignment via altering the WCS keywords means that we don't have to interpolate the reduced data again, which is particularly tricky with small rotations. However, it does mean that the data will not *look* aligned if a user were to simply overplot the two data arrays, as the alignment is entirely done from attributes in the file header. It is only aligned when it is read into the context of its WCS frame, e.g., into a Sunpy Map. Creating a Sunpy composite map from two co-aligned channels will show the data to be aligned. This does also mean that the CRPIX values are very important, and cannot be assumed to simply be the midpoint of the grid. They are instead the midpoint of the reference channel grid when projected to the working channel's WCS.

Alignment to the Sun is performed on the reference channel only. Aligning with Hα or Ca K is far more difficult due to the types of structure visible. Alignment using a photospheric channel allows the reference channel to be HMI intensity-grams, which are better for this than AIA, as AIA data at Level-1 are not rigidly co-aligned. HMI data are, which allows us to skip a step when getting a reference image map. The same basic flow as the channel-to-reference co-alignment is used for determining alignment to solar coordinates, but instead of saving the offsets to the CRPIX keywords, we take the center point of the aligned map, and adopt those coordinates as the CRVAL1/2 keywords, and the rotational offsets are added to the CROTA2 keyword. This works fairly well in most cases where structure is observed. As always, this will fail in quiet-Sun observations, in which case, contact me for advice, because there *may be* a way to do it, but it's a bit roundabout, and you'll have to manually adjust things in the code on the fly. The alignment is done on the clearest image in the reduced, destretched sequence, and the reference coordinate is differentially-rotated to the times in the rest of the files. The basic flow for the reference channel then becomes:

1. Find the clearest image from the speckle-reconstruction Fried parameter.

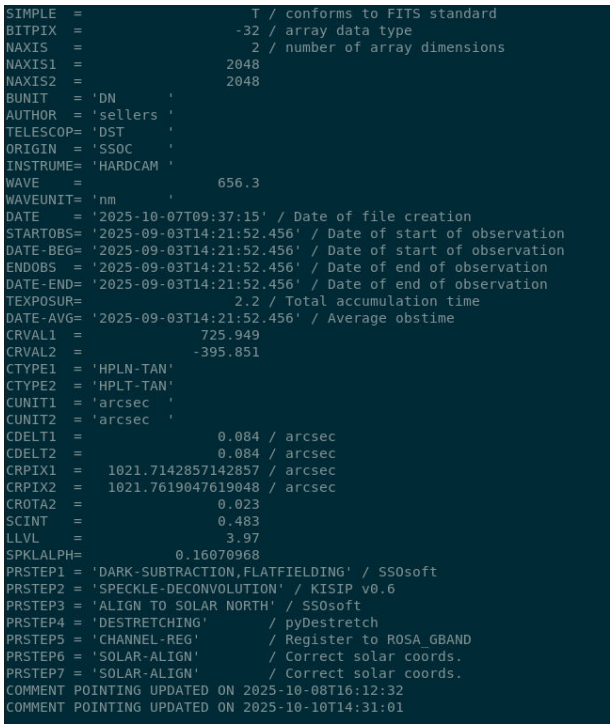
2. Fetch the HMI image closest in time to the clearest image.
3. Read the clearest image into a Sunpy map.
4. Reproejct the HMI map to the clear reference image map coordinate frame. This also interpolates and crops the HMI data.
5. Perform an alignment with 5 passes between the two, re-updating and reprojecting the original HMI map after each alignment attempt. This is done in case the original coordinates were very far off.
6. Once the alignment is done, determine the rotational offset.
7. Determine the center point of the aligned map.
8. For each image in the observing sequence, differentially-rotate the reference center point to the time of the image.
9. Update CRVAL1/2 with the differentially-rotated reference point, and add the rotational offsets.

For the subordinate channels, the flow is:

1. Determine the offsets and relative rotation between the subordinate channel and the reference channel using the target images.
2. For each image in the observing sequence, take the same reference center point and differentially rotate it to the time of the image.
3. Update CRVAL1/2 and CROTA2 with the differentially-rotated reference point, and the CROTA2 value of the reference channel.
4. Update CRPIX1/2 with the offsets between the subordiate and reference channel, and add the rotational offsets between the subordinate and reference channel.

This provides a decent alginemnt for all channels with a few caveats:

1. If the AO has come unlocked, or if the lock point was reset during the observing sequence, the reacquired lock may not be in the same position. As such the reference coordinate will be slightly incorrect.
2. If the observers re-centered the beam between the end of the observing sequence and acquiring the target image, the relative offsets between channels may be different than it was during observing.
3. Since the alignment is run on destretched data, the image grid may be warped relative to HMI. A better approach may be to detretch the aligned data to HMI, and apply that as a sort of master grid warp.
4. There is definite drift in the beam alignment during an observing sequence that will affect the offsets of each channel. Ideally, all would be affected in the same way, but I'm not entirely sure.
5. If the initial coordinates are very far off the true value, as happens when the observers haven't updated the telescope centering in a while, the inital guess for the alignemnt may be wrong. If



```
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -32 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                 2048
NAXIS2  =                 2048
BUNIT   = 'DN      '
AUTHOR  = 'sellers '
TELESCOP= 'DST     '
ORIGIN  = 'SSOC    '
INSTRUME= 'HARDCAM '
WAVE    =                656.3
WAVEUNIT= 'nm      '
DATE    = '2025-10-07T09:37:15' / Date of file creation
STARTOBS= '2025-09-03T14:21:52.456' / Date of start of observation
DATE-BEG= '2025-09-03T14:21:52.456' / Date of start of observation
ENDOBS  = '2025-09-03T14:21:52.456' / Date of end of observation
DATE-END= '2025-09-03T14:21:52.456' / Date of end of observation
TEXPOSUR=                  2.2 / Total accumulation time
DATE-AVG= '2025-09-03T14:21:52.456' / Average obstime
CRVAL1  =              725.949
CRVAL2  =             -395.851
CTYPE1  = 'HPLN-TAN'
CTYPE2  = 'HPLT-TAN'
CUNIT1  = 'arcsec  '
CUNIT2  = 'arcsec  '
CDELT1  =                0.084 / arcsec
CDELT2  =                0.084 / arcsec
CRPIX1  =   1021.7142857142857 / arcsec
CRPIX2  =   1021.7619047619048 / arcsec
CROTA2  =                0.023
SCINT   =                0.483
LLVL    =                 3.97
SPKLALPH=           0.16070968
PRSTEP1 = 'DARK-SUBTRACTION,FLATFIELDING' / SSOsoft
PRSTEP2 = 'SPECKLE-DECONVOLUTION' / KISIP v0.6
PRSTEP3 = 'ALIGN TO SOLAR NORTH' / SSOsoft
PRSTEP4 = 'DESTRETCHING'       / pyDestretch
PRSTEP5 = 'CHANNEL-REG'        / Register to ROSA_GBAND
PRSTEP6 = 'SOLAR-ALIGN'        / Correct solar coords.
PRSTEP7 = 'SOLAR-ALIGN'        / Correct solar coords.
COMMENT POINTING UPDATED ON 2025-10-08T16:12:32
COMMENT POINTING UPDATED ON 2025-10-10T14:31:01
```

Figure 5: *Header from a single HARDcam image acquired 2025-09-03*

this happens, the final coordinates can be very far off. This can be fixed by manually updating the CRVAL1/2 keywords to something closer to the correct values, and re-running the alignment. However, if you do this, you must make sure to set the "channelTranslation" keyword to None or 0 so that the translations aren't applied to the data a second time.

Once registration and alignment is complete, the last step in the pipeline is to create context movies for the speckled and destretched data for the purposes of visual quality tracking. These movies include GOES SXR lightcurves and a few parameters from the telescope, such as light level and scintillation.

6 Level-1 Data Structure

Unlike most other SSOC data products, reduced ROSA and HARDcam data are stored in single extension FITS files, one file per image. The only extension has a header with minimally-complete WCS information, instrument and observation parameters, and a description of the processing steps. See Figure 5 for an example header.

Regarding the header, a couple of things to note:

- There are a ton of keywords for acquisition date. This is partially to hew to both Solarnet and VSO best practices, and partially because Sunpy hasn't always been consistent on which keywords it likes.
- The TEXPOSUR keyword describes the total accumulation time, not the amount of time between the start and end of the burst. For HARDCam, it is frequently the same, but for the G-Band and

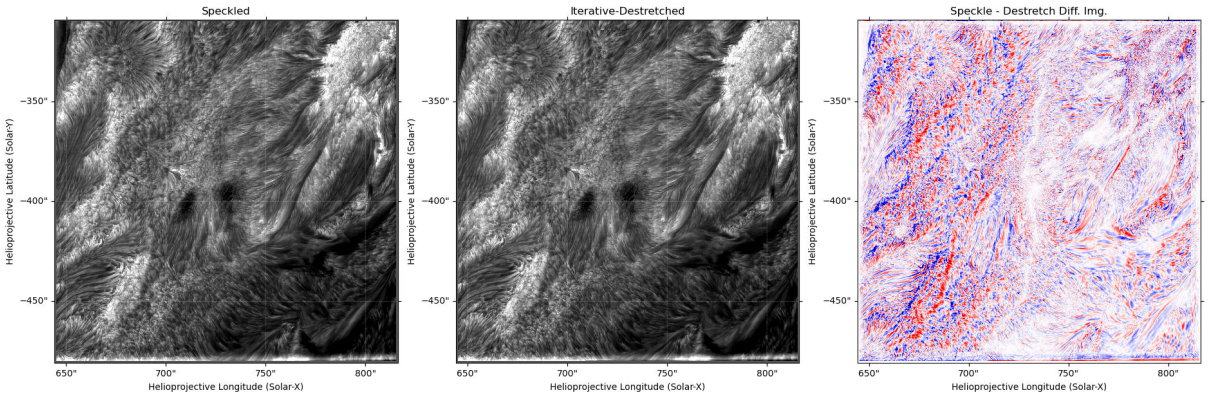


Figure 6: *Reduced HARDcam data from 2025-09-03. Left is a single speckle-reconstructed frame. Middle is the same frame after destretching (against a 1-minute running mean image). It should look visually very similar. Right is the difference image between the two, highlighting the destretch method. If the destretching is performing well, the changes will be around solar structures. A poor destretch or reconstruction will show up as a hard edge in the different image. These data have been placed into Sunpy maps after coordinate corrections, so the coordinates on the X/Y axes are correct.*

continuum channels, the usual single-frame exposure time is 10 ms, at 33.3 fps, which leaves gaps between exposures.

- CRPIX1/2 are not nice numbers, due to channel co-alignment with the G-Band channel. The channel used for co-alignment is in the comments of the CHANNEL-REG PRSTEP.
- When alignment to solar coordinates is done multiple times, a new PRSTEP is appended for each attempt, along with a new comment denoting when each attempt was made. This is done for tracking purposes.
- SCINT and LLVL are from the telescope DCSS logs. LLVL can be used for a first-order normalization. As the Sun rises (or sets) across an observing sequence, the brightness in each frame will change. LLVL tracks it, but not perfectly, as it's a voltage from the DST guider, and not chromatic in any way. SCINT likewise is measured in arcsec, but seeing is not as simple as reading the Seykora monitor value.
- SPKLALPH is the Fried parameter estimation from reconstructions. The highest value in the sequence is usually the best-reconstructed image.
- The header isn't fully WCS compliant, and Sunpy will yell at you for mapping them. Ignore it, it has everything it needs, it's just being a pain.

An example of destretched data, post-speckle data, and the difference between the two can be found in Figure 6

7 Allowed Config File Keywords (and what they do)

7.1 SHARED

Technically, all items in the `SHARED` section are optional, but it's good practice to include them if you're able. If no DCSS log is available, for example, a lot of the headers will be incomplete. In a pinch, you can manually backfill these from SPINOR, FIRS, or another facility instrument that talks directly to the DST ICC, but the DCSS logs are far easier to work with.

- DCSSLog: path, Location and name of DCSS or VTT log file.
- referenceChannel: str, Name of channel to use for alignment and registration. This name should appear later in the file as a section header.
- solarAlign: bool, If true, will attempt registration and solar alignment. Both are controlled by this switch; there's currently no way to just do the registration without alignment. If you are trying to registration but not alignment (i.e., quiet-Sun observations), you'll have to trigger the functions manually.
- verifyPointingUpdate: bool, if True, will allow the user to visually check the alignment quality before applying the coordinate update. It will spawn a figure of the alignment image from referenceChannel and overlay HMI contours to inspection, prompting the user for Y/n input, and exiting if "n" is provided. It's a bit hackier than I like, but you need to have an abort button for this. Alignment is more an art than a science.
- contextMovies: bool, if True, will write movies for both speckled and destretched data
- contextMovieDirectory: path, location for context movies. If not given, will place them in the working directory for each camera. I like to place them in their own directory.

7.2 Camera Sections

There's an allowed list of camera names. Holdover from the original code. The basic ones are `ZYLA`, `ROSA_GBAND`, `ROSA_CAK`, `ROSA_4170`, and `ROSA_3500`. The presence of ZYLA or ROSA in the camera name tells the code what the shape of the camera sensor is. If you're trying to reduce a different channel than those listed, I'd recommend just calling it `ROSA_4170` or `ROSA_3500`. Everything important to the naming and reduction is set further down in the keyword attributes, so it doesn't really matter what the section is called except for setting the chip size.

Allowed keywords are:

- darkBase, flatBase, targetBase, linegridBase, and dataBase [required-ish for rosaZylaCal]: path, locations to search for that type of file. If, like typical ROSA data, all files for the camera are in a single folder, only dataBase is required, and the rest can be fully omitted from the config file. For HARDcam, where different datatypes live in different directories, all are required. targetBase and

linegridBase are only required if other flags are set in the config file that tell the code to look for these files.

- workBase [required for all]: path, location to perform reductions in. Generally, I make a directory per observing sequence per camera, with a config file per observing sequence.
- burstNumber [required for rosaZylaCal]: int, Number of files to use in speckle-reconstructions
- obsDate [required for all]: int, Date of observations in format YYYYMMDD or similar.
- obsTime [required for all]: int, Time of observations in format HHMMSS or similar.
- expTimems [required for rosaZylaCal]: float, single-image exposure time in ms. Important for HARDcam timestamp reconstruction in particular.
- burstFileForm [required for rosaZylaCal]: str, Python-formattable string with four format sections to name pre-speckle files. Formatting sections are two `{:s}` sections for date and time, a `{:02d}` tag for the KISIP batch number (since KISIP will only loop from 000 to 999), and `{:03d}` tag for the burst number in the KISIP batch.
- speckledFileForm [required for rosaZylaCal]: str, Python-formattable string with four format sections to name post-speckle files. Format sections are same as burstFileForm.
- dataFilePattern, darkFilePattern, flatFilePattern, targetFilePattern, linegridFilePattern [required-ish for rosaZylaCal]: str, patterns of these filetypes. For, e.g., HARDcam, where all files are named `....spool.dat`, and separated by directory, only dataFilePattern is required, and can be set to `*spool.dat`. These are used to assemble the input file lists, using glob, so wildcards and other control patterns are valid here.
- noiseFile [required for rosaZylaCal]: str, filename to use for noise file pattern (KISIP input).
- wavelengthnm [required for rosaZylaCal]: float, in nm, the wavelength of observations. Used in final file headers and also passed to KISIP for diffraction limit calculations.
- kisipArcsecPerPixX, kisipArcsecPerPixY [required for rosaZylaCal, kisipWrapper]: float, approximate plate scale. Depending on switches elsewhere, this may be updated, but it is still important for an initial guess, and if the calculated version is more than 10% different, these values will be used as canonical instead.
- kisipMethodSubfieldArcsec [required for rosaZylaCal, kisipWrapper]: float, controls the size of tiles KISIP uses in reconstruction. If it's made too large, KISIP will segfault. I'm not actually convinced it does anything, and just falls back to 128-pixel tiles.
- correctPlateScale [optional, for rosaZylaCal]: bool, if True will attempt to fine-tune the plate scale using the linegrid map.
- createFiducialMaps [optional, for rosaZylaCal]: bool, if True will write average target and linegrid images to workBase.

- dstrFrom [required for RosaZylaDestretch]: str, either "fits" or "speckle". For destretch-compatibility with older-reduced datasets. You can set it up to take postSpeckle fits files reduced before 2023, when I wrote the module.
- dstrKernel [required for RosaZylaDestretch]: comma-separated int, sequence of destretch subfield sizes. Lead with a "0" to do an initial alignment. I get good results from "0, 128, 64, 32, 12".
- dstrMethod [required for RosaZylaDestretch]: str, either "running" or "reference". Running destretches to the running mean of previously-destretch images. Reference chooses a single image to destretch the stack to.
- dstrWindwo [required for RosaZylaDestretch]: int, if "running" is selected for dstrMethod, number of images to include in the average reference. If "reference" is selected for dstrMethod, the number set is the n-th post-speckled image to use as a reference.
- flowWindow [optional for RosaZylaDestretch]: int, if set to 0, turns off the flow-preserving module. You should keep it at 0. If set to another int, it defines the window to use as a baseline.
- repairTolerance [optional for RosaZylaDestretch]: float, between 0 and 1. Describes the maximum shift (as a fraction of the subfield size) for destretching. Higher allows for larger shifts. Defaults to 0.5. If you're seeing a lot of bulk subfield motions that shouldn't be there, you can lower it. Likewise, if you're getting subfields that aren't shifting in step with adject motions, you may need to increase it.
- channelTranslation [required for RosaZylaDestretch]: comma-separated str. Sequence of translations to get the target image in the correct orientation (and the rest of the files too). Allowed translations are "rot90" (90-degree rotation), "flipud" (flip up-down), "fliplr" (flip left-right), and "flip" (flip both axes).
- progress [optional for all]: bool, if True, will spawn progress bars for almost all major operations, including KISIP, for the user who wants to keep an eye on reduction status. Be warned that since a lot of the loops are quite long, resizing your terminal at any point will cause the progress bar not to erase properly, and may fill the terminal up with each iteration.

7.3 KISIP METHOD

These can usually be ignored, unless the user is very familiar with KISIP. These set up the reconstruction algorithmic parameters.

- kisipMethodMethod: int, 0 or 1. 0 uses the Knoxville-Thompson algorithm, 1 uses triple-correlation, which tends to perform better overall.
- kisipMethodPhaseRecLimit: int, 0-100. Limit for phase reconstruction, expressed as a limit of the diffraction limit. Typically set to 95. Lowering this can greatly speed up computation.

- kisipMethodUX: int, 0-100. Limit the x-component of the first bispectrum to this percentage of the diffraction limit. Usually uses 10.
- kisipMethodUV: int, 0-100. Limits the length of first and second bispetrum vector and their sum to this percentage of the diffraction limit. Should always be the same as kisipMethodUX. We usually use 10.
- kisipMethodMaxIter: int, Max iterations for least-squares fitting in kisip. Usually kept at 30.
- kisipMethodSNThresh: int, 0-100. Percentage of all bispectrum vectors to use in reconstruction. The best pecentage are kept for the calculation. Usually kept at 80.
- kisipMethodWeightExp: float, weighting for the SNR. Higher converges faster. Usually kept at 1.2 or 1.3.
- kisipMethodPhaseRecApod: int, 0-100. Percentage of the subfield to window with the apodisation function. Larger values reduce artifacts, but can cause issues in larger structures. We usually set it at 15, so 7% on each the top, bottom, and sides. I would suggest keeping it below 30.
- kisipMethodNoiseFilter: int, 0 or 1. Honestly, not sure what this does. Keep it at 1.

7.4 KISIP PROPS

These can usually be ignored, unless the user is very familiar with KISIP. These set up the properties of the data.

- kisipPropsHeaderOff: int, leave at 0. KISIP can technically read data from FITS files, and there's a parameter to skip this number of bytes for the file header. FITS headers are usually 2880 bytes, unless they're long or have HEIRARCH keywords, which ours either are or do. It's an untested feature and it's easier to just write the data in binary to disk, which is what we do. Don't mess with it.
- kisipPropsTelescopeDiamm: int, leave at 760 or 762 if you're feeling pedantic. This describes the clear aperture of the telescope. For the DST, this is 762 mm. If the seeing is poor, you can lower this to make kisip behave less agressively, but that's a last resort.
- kisipPropsAoLockX: int, either at -1 or the x-pixel coordinate of the AO lock position in the frame. A value of -1 tells KISIP to guess the lock position. It's better at this than you or I, so leave it at -1.
- kisipPropsAoLockY: Same as above, for Y-axis.
- kisipPropsAoUsed: 0 or 1. If the data were acquired with tip-tilt only, as in the case of limb-tracked data, or an unexpected AO failure, this can be set to 0. In all other cases, it should be 1.

7.5 KISIP ENV

Location of KISIP binaries. Set once and copy thereafter.

- kisipEnvBin: path to the "bin" folder of the KISIP install.
- kisipEnvLib: path to the "lib" folder of the KISIP install.
- kisipEnvMpiNproc: int, number of processors to feed to KISIP. SSOC has 24 available.
- kisipEnvMpirun: leave as "mpirun", selects the executable to use to trigger mpi in kisipEnvBin. It's set by the precompiled version we have, so no need to change.
- kisipEnvKisipExe: leave as "entry", selects the executable to use to trigger KISIP. It's set by the precompiled version we have, so leave it as "entry"

7.6 Logger Parameters

The logging software for the calibration run is configured to look at the config file for its setup parameters. It's not important to know what they all do, but the config file needs a section for "loggers", "handlers", "logger_root", "logger_RoHcLog", "handler_RoHcHand", and "formatter_RoHcForm". Honestly, just copy the bottom half of a template config file. It's a bit over-the-top.

8 Description of Class Methods

This section is not a complete API. Each class and all contained methods have extensive docstrings, type hinting, and comments. Reading the code is the absolute best way to understand what it's doing and why. This section is provided in case the user would like to isolate and run a single section of a given reduction procedure. For example, aligning a channel to solar coordinates without registering multiple cameras. Hopefully, this section enables you to get a sense of what steps are required, and which can be safely skipped to perform cetain reductions.

8.1 rosaZylaCal

The `rosaZylaCal` class (`ssosoft.imagers.rosaZylaCal.rosaZylaCal`) takes as arguments, a camera name and a config file name. The basic call structure is, e.g.,

```
import ssosoft
r = ssosoft.rosaZylaCal("ZYLA", "config.ini")
r.rosa_zyla_run_calibration()
```

The `rosa_zyla_run_calibration` method has the following (simplified) code:

```
def rosa_zyla_run_calibration(self, save_bursts=True):
```

```
self.rosa_zyla_configure_run()
self.rosa_zyla_get_file_lists()
self.rosa_zyla_order_files()
self.rosa_zyla_get_data_image_shapes(self.flat_list[0])
self.rosa_zyla_get_cal_images()
self.rosa_zyla_save_cal_images()
self.parse_dcss()
if self.correct_plate_scale:
    self.determine_grid_spacing()
if save_bursts:
self.rosa_zyla_save_bursts()
```

The `save_bursts` keyword can be quite useful if the user would like to, for example, write target and linegrid images to a dataset that has already been reduced without taking several hours to write the KISIP burst files.

These functions perform the following steps:

- `rosa_zyla_configure_run`: Reads your config file and sets up class variables.
- `rosa_zyla_get_file_lists`: Looks for files matching user-provided patterns in user-specified directories.
- `rosa_zyla_order_files`: For HARDcam, which numbers files by least significant digit, re-orders the file lists to be sequential.
- `rosa_zyla_get_data_image_shapes`: Reads the 0th flat file in the list assembled in the previous step to determine the camera chip size in pixels.
- `rosa_zyla_get_cal_images`: Reads from disk or creates necessary calibration images, which include the average dark, average flat, gain, average line grid (optionally), and average target (optionally).
- `rosa_zyla_save_cal_images`: Saves the calibration images generated during the previous step if they do not already exist. Allows the code to skip re-creating these files if the calibration has to be re-run.
- `parse_dcss`: If a DCSS logfile or a VTT logfile was provided in the config file, this function reads the file and pulls the relevant information from it, so the code can access it during header file creation.
- `determine_grid_spacing`: If there are linegrid files and the user has the correct flags set, this code attempts to fine-tune the plate scale of the camera from the average calibrated linegrid image. This is not a perfect determination, but is better than an estimate based on magnifcation via the imaging lenses.

- `rosa_zyla_save_bursts`: Assuming the `save_bursts` keyword argument is not set to "False", this function reads the data files for the observation sequence, applies dark and gain correction, then writes a stack of images to KISIP-formatted burst files, along with a header file in a separate directory, which will later be used in reconstructing a full FITS header.

8.2 kisipWrapper

The `kisipWrapper` class (`ssosoft.imagers.kisipWrapper.kisipWrapper`) takes as its argument a fully set-up instance of `rosaZylaCal`. The basic call structure is, e.g.,

```
import ssosoft
r = ssosoft.rosaZylaCal("ZYLA", "config.ini")
r.rosa_zyla_run_calibration()
k = ssosoft.kisipWrapper(r)
k.kisip_despeckle_all_batches()
```

If you're re-running a calibration, and would like to start `kisipWrapper` without the lengthy step of writing the per-speckle bursts, remember that this can be skipped with the `save_bursts` keyword argument in `rosaZylaCal.rosa_zyla_run_calibration()`!

The `kisip_despeckle_all_batches()` method has the following (simplified) code:

```
def kisip_despeckle_all_batches(self):
    self.kisip_configure_run()
    for batch in self.batchList:
        self.kisip_set_batch_start_end_inds(batch)
        self.kisip_set_environment()
        self.kisip_write_init_files()
        self.kisip_spawn_kisip()
```

These functions perform the following steps:

- `kisip_configure_run`: Reads user-provided config file, and sets up class instance variables.
- `kisip_set_batch_start_end_inds`: Sets the indices for the start and end of the current KISIP batch. At the moment, this always starts at 0, and ends at the number of files in the batch
- `kisip_set_environment`: Sets required environment variables to run KISIP.
- `kisip_write_init_files`: Writes KISIP configuration files in the camera channel's working directory.
- `kisip_spawn_kisip`: Starts the KISIP process and runs each batch. Note that it *does* change the working directory of the code, and does not change it back at the end of the KISIP run.

8.3 RosaZylaDestretch

The `RosaZylaDestretch` class (`ssosoft.imagers.destretch.RosaZylaDestretch`) takes as arguments, a camera name (or list thereof), and a configuration file. The basic call structure is, e.g.,

```
import ssosoft
d = ssosoft.RosaZylaDestretch("ZYLA", "config.ini")
d.perform_destretch()
d.register_rosa_zyla_observations()
```

Note that the class does not require either `rosaZylaCal` or `kisipWrapper` to have been run directly before calling it, just that those calibrations have been completed at some point. This is because the telescope did not have a destretch method from 2017-2023, and it is advantageous to be able to add this calibration to older-reduced data.

The `perform_destretch` method has the following (simplified) code:

```
def perform_destretch(self):
    # For cases where the code is given a list of cameras
    for channel in self.instruments:
        self.channel = channel
        self.configure_destretch()
        if self.dstr_from.lower() == "speckle":
            self.speckle_to_fits()
        self.destretch_observation()
        if self.flow_window > 0:
            self.remove_flows()
            self.apply_flow_vectors()
```

These functions perform the following steps:

- `configure_destretch`: Reads configuration file and sets up class variables.
- `speckle_to_fits`: Dumps post-KISIP ".final" files back to FITS format with their accompanying header information.
- `destretch_observation`: Triggers iterative destretch and writes destretched files to a new directory. Handles getting or calculating the reference image as well.
- `remove_flows`: Reads the saved destretch parameters, and attempts to isolate solar lateral flows from the vectors. Writes altered destretch vector files with the flows removed. Recommend not running.
- `apply_flow_vectors`: Calls the underlying destretch module with the flow-removed vectors, and applies these corrections to the post-speckle files. Recommend not running.

The `register_rosa_zyla_observations` method has the following (simplified) code:

```
def register_rosa_zyla_observations(self):
    for channel in self.instruments:
        self.channel = channel
        self.configure_destretch()
        self.apply_bulk_translations()
        if self.channel == self.reference_channel and self.solar_align:
            self.update_reference_frame()
            self.apply_solar_offsets()
        if self.channel != self.reference_channel:
            self.channel_offset_rotation()
            self.channel_apply_offsets()
            if self.solar_align:
                self.apply_solar_offsets()
        if self.create_context_movies:
            self.generate_context_movie()
```

Note that, for the registration and solar-alignment to work, the destretch class must be instatiated with a list of cameras, with the reference camera channel first in the list. Otherwise, there won't be a reference coordinate or channel to register the other channels to. The functions in this wrapper perform the following steps:

- `configure_destretch`: Sets up class instance variables from the configuration file
- `apply_bulk_translations`: Applies the user-provided translations to get the images in the correct orientation with respect to the telescope's angle of rotation from solar-North.
- `update_reference_frame`: For the reference channel only, finds the cleanest image reconstruction, and attempts an iterative cross-correlation between it and a cotemporal HMI image. Checks with the user that they're satisfied with the degree of alignment, and if they are, saves the reference coordinate within the class instance for further use.
- `apply_solar_offsets`: Loops through all reduced data, applying the new reference coordinate, differentially-rotated to the individual timestamp.
- `channel_offset_rotation`: For subordinate channels only, determines the relative offsets and rotations between correctly-oriented target images.
- `channel_apply_offsets`: Loops through all reduced data, applying the necessary shifts of the subordinate channel to its CRPIX1/2 header keywords, and the rotation offset to CROTA2.
- `generate_context_movie`: For both the post-speckle and destretched data (or whichever are available), generates a context movie from the reduced filelist.