Jupiter Horizon Investigation
Image Analysis 30330

Authors

Andrey Galkin - s252898
Vojtech De Coninck - s252918

December 19, 2025

Contents

Chapter 1

Introduction

1.1 JUNO Mission

JUNO is a NASA spacecraft dedicated to the investigation of Jupiter's interior, atmosphere, and magnetospheric environment from a polar, highly elliptical orbit. This trajectory creates short perijove windows with rapidly changing observation geometry, separated by long arcs far from the planet. An overview of the orbit evolution and close-approach passes is shown in Fig. 1.1.

JUNO is spin-stabilized, which simplifies attitude dynamics but introduces image motion for instruments integrating during spacecraft rotation. The original mission baseline envisioned two initial 53-day orbits followed by a period reduction to 14 days. However, due to concerns in the main-engine feed system (check-valve behavior), this maneuver was not executed and the mission remained in the longer-period orbit, reducing operational risk while preserving the perijove science geometry [1].

Following completion of the prime objectives, NASA approved an extended mission in which the orbit continues to evolve and enables additional science, including targeted encounters in the Jovian system [2, 3]. At end-of-mission, the spacecraft is planned to be disposed by a controlled entry into Jupiter's atmosphere to satisfy planetary protection requirements [2].

The spacecraft carries four DTU Advanced Stellar Compass (ASC) star trackers, primarily providing precise attitude information to the magnetometer system [4]. During

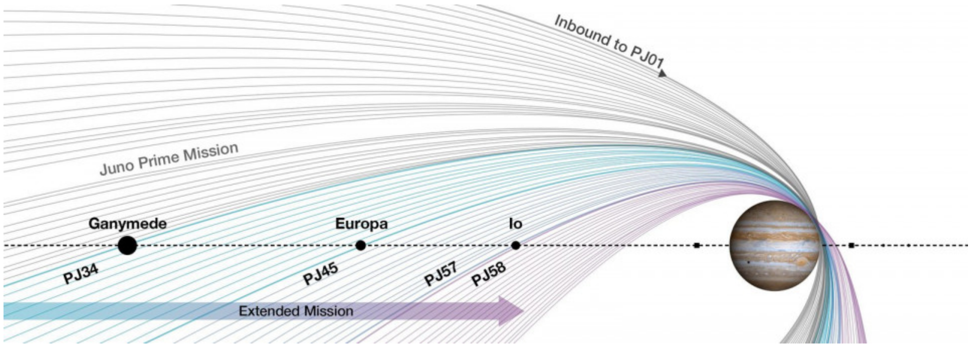


Figure 1.1: Overview of JUNO's orbit evolution and perijove passes around Jupiter (prime and extended mission). [2]

selected perijove intervals, these cameras are commanded to acquire imagery containing Jupiter's limb. Although the ASC is not designed as a science imager, these horizon views provide insights about Jupiter's atmosphere.

1.2 Problem statement

The ASC does not output a fully synchronous frame. Due to the staggered readout scheme (Section 2.1), each acquisition contains two interleaved fields captured at different times, which introduces intra-frame misalignment during spacecraft spin. In addition, the orbit environment and sensor characteristics generate strong outliers and background structure that bias direct feature extraction.

The dataset therefore presents two coupled challenges. First, the raw frames must be converted into a calibrated consistent representation. Second, the signals of interest (a short limb arc and a sparse star field) must be extracted robustly. The required corrections and their implementation are treated in Chapter 3.

1.3 Research goals

The assignment specification was intentionally open-ended. The work therefore focuses on the aspects of the dataset that are most relevant for geometric reconstruction and atmospheric interpretation. The objectives are:

- Construct an image processing sequence that converts raw ASC acquisitions into calibrated, denoised, and geometrically corrected frames (Chapter 3).

- Compensate the odd/even readout misalignment by despinning and merging both fields into a single consistent image (Section 3.3).
- Extract Jupiter's apparent horizon from the corrected frames and validate the result against a SPICE-derived reference (Chapter 4).
- Detect and match background stars to an inertial catalogue to estimate camera pointing from the same frames (Chapter 5).
- Combine limb geometry and star-derived attitude to recover an image-constrained spacecraft position estimate, and derive basic horizon-referenced atmospheric observables where supported by the data (Chapter 6).

Chapter 2

Background and Data

2.1 Juno ASC's camera

The Advanced Stellar Compass (ASC) is an instrument providing attitude measurements to Juno's magnetometers. It consists of two pairs of Camera Head Units (CHUs) placed at the end of one of Juno's 3 solar arrays, 8 and 10 meters from the spacecraft's body to mitigate electromagnetic disturbance. Each CHU is a star tracker monochrome 8-bit camera with a 19°x13.5° field of view and a 752x580 pixel CCD sensor. For optimal performance in a high radiation environment, the sensor lowers its integration time by using the built-in electronic shutter. The full integration time is divided evenly between two image fields : the odd-numbered rows are read out in the first field, followed by the even rows in the second field after a fixed delay Δt. Due to Juno's spin during this delay, the result consists of two clearly misaligned row sets, which must be accounted for during image processing [4].

In addition to the odd-even row misalignment, the ASC operates under low-light conditions with short exposure times, making the images inherently photon-limited. As a result, shot noise and readout noise contribute significantly to the background intensity fluctuations. This noise floor noticeably limits the detectability of faint stars.

Furthermore, the sensor is affected by spatial non-uniformities, including pixel-to-pixel gain variations and fixed-pattern noise. These artefacts can introduce bias in local background estimation and star flux measurements. These effects should likewise be considered

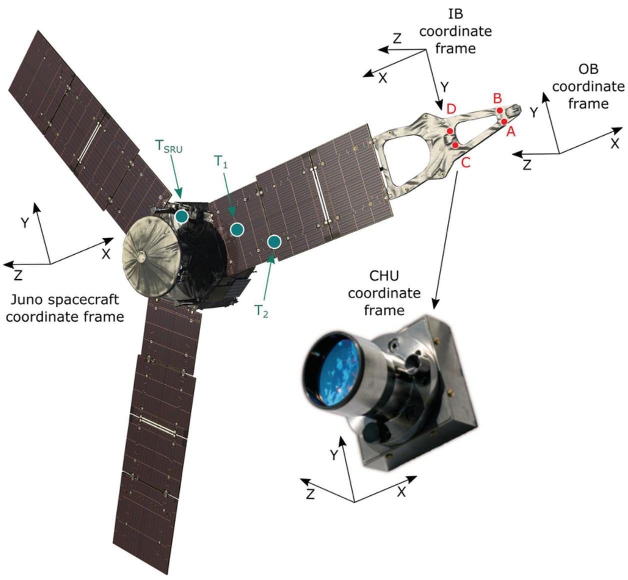


Figure 2.1: CHU disposition on Juno [5]

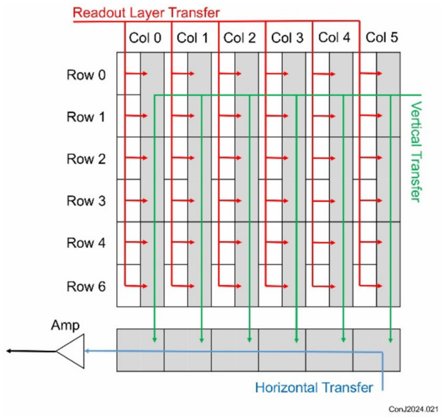


Figure 2.2: Principle of CCD operation. Photosensitive integration layer in white, shuttered layer in gray. Sketch shows integration of first image field [6]

during processing (e.g. using reference images).

Scenes containing bright objects, such as Jupiter or its illuminated limb, can induce intense stray light and scattering within the lens system. This results in extended intensity gradients and halo structures that are not associated with point sources. Such features distort the local background and can suppress or falsely trigger star detections if not explicitly masked.

Finally, residual geometric distortions introduced by the lens lead to systematic deviations from an ideal pinhole camera assumption.

2.2 Dataset description

All of this paper's work and conclusions are derived from 6 images captured by the ASC's Camera Head Unit D. They were all taken during the rare perijove passage (when the orbit has the lowest altitude) over Jupiter's North Pole. During this window, the Jovian horizon is observed at a very high slant angle, providing imaging of the upper atmosphere for a short time, before returning to deep space star field observation.

The dataset consists of 3 pairs of images taken 30 seconds apart. Within a pair of images, they are are taken exactly 500ms apart. Due to the spacecraft's high speed and rotation, the observation geometry drastically changes from one pair to the next. All acquisitions were made on September 12 2019 from 3:22:25.714 to 3:23:26.714 UTC. The filename of each acquisition encodes a timestamp, formatted as a floating-point scalar. This value denotes the number of seconds elapsed since the J2000 epoch (January 1, 2000, 12:00:00 UTC), allowing for sub-millisecond synchronization with the spacecraft's trajectory data. [7]

The first pair is right before Jupiter enters the camera's field of view. Naturally, they provide little interest in analyzing Juno's atmosphere.

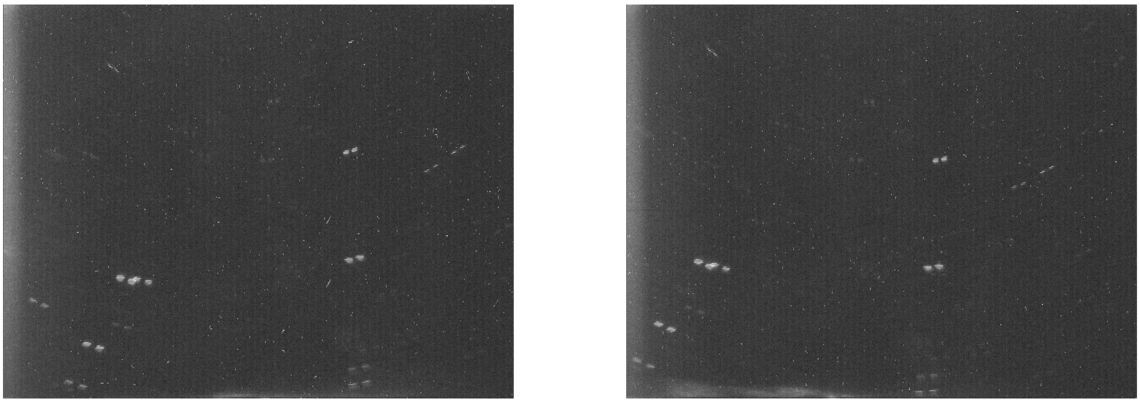


Figure 2.3: Pre-encounter observation (gained x4)

The second pair is right after the planet enters the camera's field of view. A very short and dark limb of the night side if the planet is observed. The primary features of interest in these frames are the atmospheric intensity drop-off and an aurora visible in the bottom right corner.

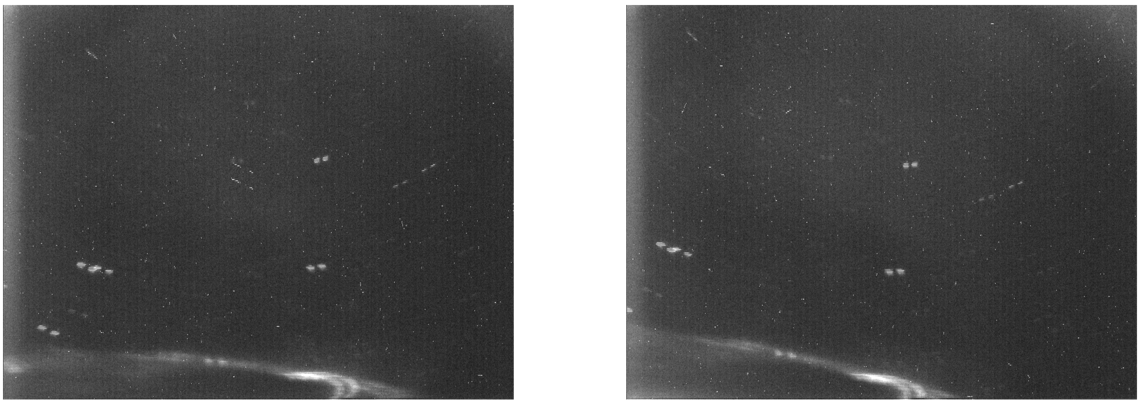


Figure 2.4: Atmospheric encounter (gained x4)

Finally, the third pair presents the greatest scientific interest. Visually, the image appears to capture a terminator line separating a bright region (day side) from a dimmer one (night side). However, this feature is not the true day/night terminator, but rather an effect of atmospheric scattering of sunlight. The terminator line is out of bounds, positioned below the image frame.

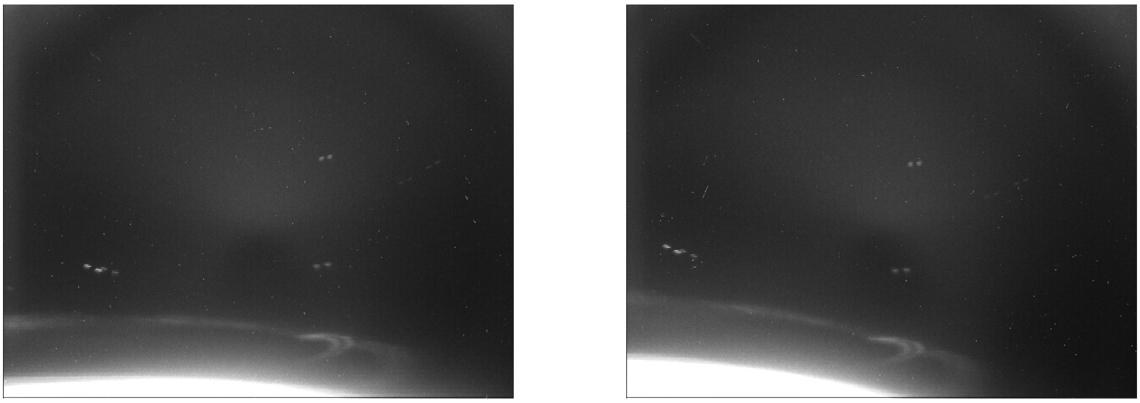


Figure 2.5: Atmospheric scattering

2.3 SPICE toolkit

To accurately interpret the 2D imagery acquired by the ASC, it is necessary to reconstruct the precise 3D observation geometry at the exact moment of capture. This was achieved using the SPICE toolkit, a library provided by NASA's Navigation and Ancillary Information Facility. SPICE stands for Spacecraft, Planet, Instrument, C-matrix and Events. It is used to handle planetary mission geometry, time conversions, and reference frame transformations. It relies on data files known as "kernels," which model the Juno mission environment:

- SPK (Spacecraft Planet Kernel): Provides the trajectory of the Juno spacecraft and the position of the Sun and Jupiter
- CK (C-matrix Kernel): Provides the attitude of the spacecraft, allowing to determine the pointing vector of the camera
- PCK (Planetary Constants Kernel): Provides the physical properties of Jupiter

In this study, CSPICE was used for these critical tasks:

- Surface intercept and horizon determination : By combining the spacecraft's position vector with the camera's pointing vector, each pixel's line-of-sight was projected onto Jupiter. This maps the 2D image plane to the 3D planetary surface, allowing to determine the "true" boundary between the planet and space on the image. This gives a comparison metric for the horizon detection algorithm.

- Distance and scale validation: The toolkit computes the exact slant range, i.e the distance between the spacecraft and the horizon line, which is about 60,000 km for the dataset. This measurement, alongside camera parameters, serves as a baseline to derive spatial resolution, required to convert pixels into physical kilometers.

As handling these tools is complex enough and not the primary focus of this study, LLMs were used for code generation in MATLAB to retrieve geometric data. These measurements were double-checked for plausibility and consistency given the mission's constraints.

Chapter 3

Image Processing

3.1 Radiometric Calibration

The first image processing step is to isolate the signal from sensor-intrinsic artifacts. This process addresses two categories of noise: stochastic thermal noise (dark current) and systematic readout structures (fixed pattern noise).

3.1.1 Dark frame subtraction

The primary calibration step involves the subtraction of a dark reference frame, generated by averaging multiple exposures taken with the shutter closed at the same operating temperature and integration time as the science data. This process serves a dual purpose:

1. Thermal correction: it removes the stochastic dark current noise. This is a thermal phenomenon where silicon atoms spontaneously release electrons, mimicking light signals. They accumulate during exposure and appear as signal in the image, making dark current noise extremely temperature and exposure dependent. Since it is purely additive, the reference dark frame can be subtracted from the raw image.

2. Fixed pattern noise removal: The ASC sensor exhibits a vertical oscillatory pattern caused by gain variations in the column readout amplifiers. Since this pattern is systematic and stable over time, it is captured in the dark frame.

By subtracting the reference frame from the raw image, both the thermal baseline and the vertical stripes are effectively eliminated in a single operation:

$$I_{corr}(x, y) = I_{raw}(x, y) - I_{dark}(x, y)$$

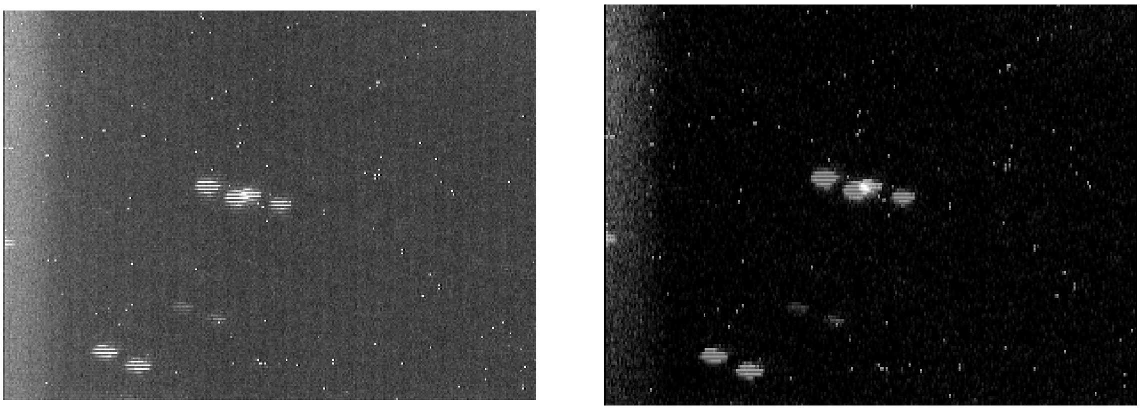


Figure 3.1: Before and after dark frame subtraction (images gained x10). The background is much darker and vertical stripes are removed.

Alternatively, the oscillatory pattern and the DC component could be filtered in the frequency domain. Since the readout noise is a vertical periodic oscillation, its spectrum is concentrated on the horizontal axis of the 2D Fourier transform (see 3.2). Notching the peak at frequency 0 would reduce overall brightness. Notching the secondary peaks would remove the periodic noise. However, this approach is only of theoretical interest and suboptimal compared to dark frame subtraction. Indeed, notching introduces ringing artifacts near high-frequency features such as stars, degrading the signal. In addition, FFT is computationally far heavier than simple integer substraction.

3.1.2 Residual sensor artifacts

Following the dark subtraction, residual systematic artifacts persist in the calibrated image. The most prominent feature is a localized, additive brightness gradient visible on the left edge of the sensor. Physically, this phenomenon (known as amplifier glow) is a consequence of the sensor's architecture. The power supply and readout electronics are located along the left side of the silicon die. During the readout process, these components dissipate power, leading to localized heating. This power dissipation creates a thermal gradient across the silicon substrate. Since dark current generation is exponentially depen-

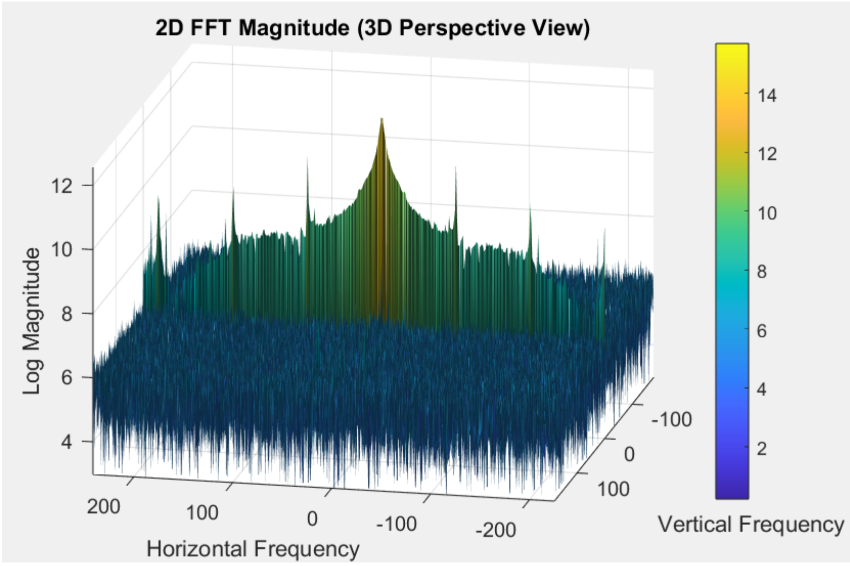


Figure 3.2: 2D Fourier transform of dark frame

dent on temperature, pixels in close proximity to the warm power supply rails accumulate significantly more thermal charge than those in the cooler center of the array.

To eliminate these column-correlated artifact, a column-median subtraction filter is employed. This method relies on the fact that the majority of field of view captures deep space. Consequently, the statistical median of any column provides an estimate of the background bias level for that column, independent of bright features such as stars or the planetary disk.

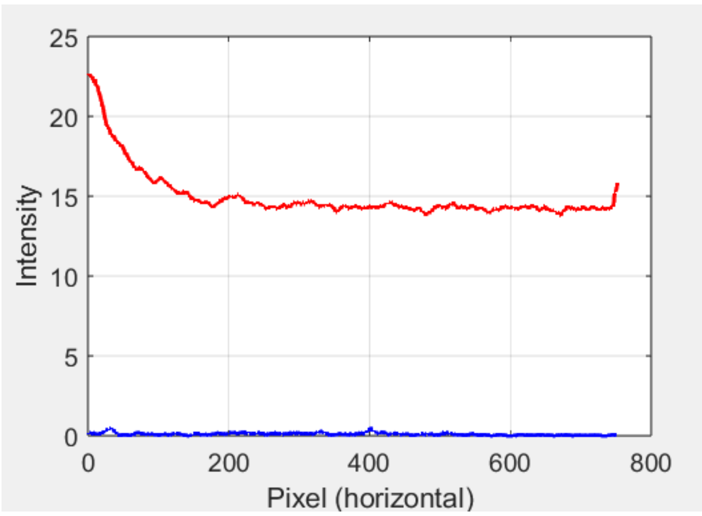


Figure 3.3: Line scan of raw (in red) and calibrated (in blue) background intensities (moving average). The calibration successfully removes amplifier glow (on the left-hand side) and background bias.

3.2 External Noise Mitigation

3.2.1 Radiation noise characterization

Jupiter's radiative environment The Juno spacecraft operates in a harsh radiation environment, primarily dominated by the Jovian radiation belts. As it approaches the poles, the flux of particles in the magnetosphere increases by several orders of magnitude. The CCD is exposed to ionizing radiation flux, specifically high-energy electrons. For a star tracker, which relies on identifying stable constellations, this creates a false depiction of the sky by adding thousands of transient bright spots, interpreted as stars. Therefore, removing these bright features is not an enhancement step : it is absolutely necessary to retrieve scientific data from the image.

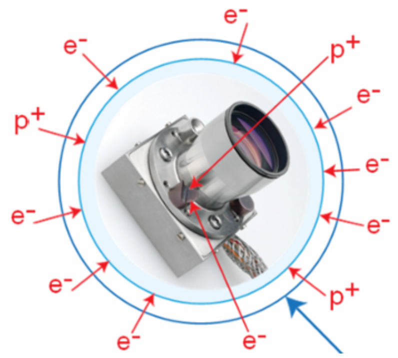


Figure 3.4: Sensor-particle interaction [6]

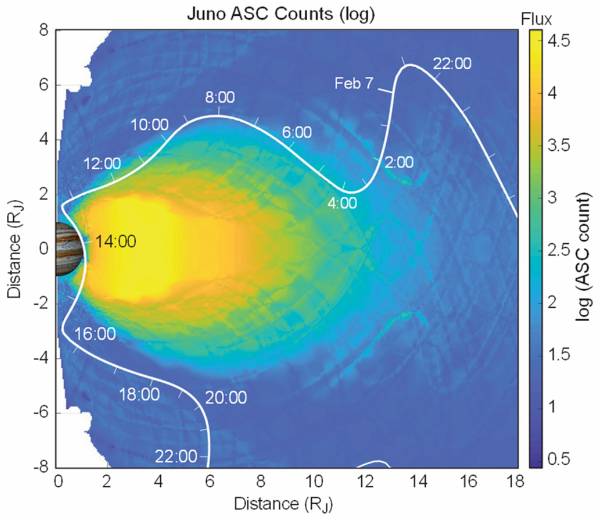


Figure 3.5: Jovian magnetosphere particle flux [6]

Particle-sensor interaction The ASC sensor is equipped with radiation shielding that significantly reduces the flux of particles reaching the CCD. While it efficiently blocks protons and electrons <10MeV, higher energy particles have enough momentum to penetrate the shielding enclosure. The particles that do reach the CCD liberate charge that is collected for a single frame. The noise is therefore transient : it appears in one frame and is completely uncorrelated in the next one.

In the image, an electron appears as a very high intensity isolated pixel because it deposits a large amount of kinetic energy in a single potential well of the CCD. This "impulse" characteristic makes noise distinguishable from other light sources. The optical system is designed to spread out focal objects (i.e. stars) over a Point Spread Function with an area larger than 4 pixels [6] (see 3.7).

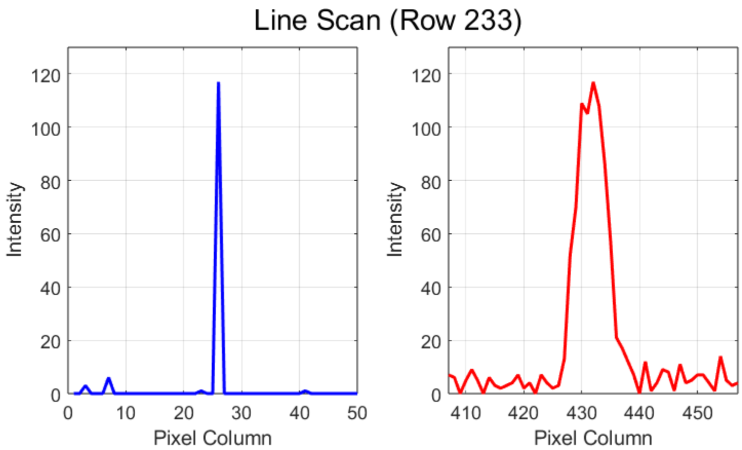


Figure 3.6: Blue line indicates radiation hit, characterized by a pixel-wide impulse (Dirac delta function). Red line represents a star, displaying a broader Gaussian distribution consistent with the optical PSF.

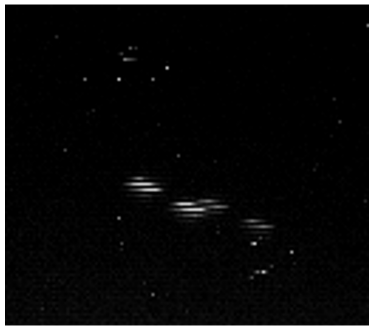


Figure 3.7: Radiation noise (dots) and stars from calibrated image

3.2.2 Gradient-based filtering

The algorithm determines which pixels are valid data and which are radiation speckles. To preserve the edges of the planets and the star field, the chosen topology is a gradient-based switching median filter. Unlike a standard median filter that processes every pixel (potentially blurring fine details), this method is conditional, meaning it only modifies pixels identified as defective based on a gradient threshold. The detection logic relies on the morphological difference between the sharp profile of a radiation speckle and the smoother gradient of a Point Spread Function (cf. 3.6).

Step A : Gradient detection The main challenge is to wisely choose the threshold. If it is set too low, false detections will occur. Stars will be treated as detected particles and removed. If set too high, radiation speckles will be interpreted as stellar candidates. There are 2 cases to separate :

- Light sources : The optics are calibrated to have a finite Point Spread Function. The signal from a star is normally distributed.

$$I_{star}(r) \propto e^{-r^2/2\sigma^2}$$

- Radiation noise : a particle enters and exit the same pixel sensor (in most cases). This results in a high-intensity impulse signal. The gradient between the speckle and the neighbor is nearly equal to the amplitude of the hit itself/

$$I_{rad}(x, y) = A \cdot \delta(x - x_0, y - y_0)$$

Therefore, by setting a gradient threshold T_{grad} such that:

$$\max(\nabla I_{star}) < T_{grad} < \min(\nabla I_{rad})$$

the filter can effectively separate the two sets.

Step B : Median Replacement Pixels flagged by the gradient check are replaced using a kernel restricted to the nearest orthogonal neighbors (top bottom left right). This method is preferred to a box or Gaussian filter to preserve the edge of Jupiter's limb.

$$P_{new}(x, y) = \text{median}\{P(x, y - 1), P(x, y + 1), P(x - 1, y), P(x + 1, y)\}$$

This approach is supported by literature [8], where Van Dokkum demonstrated that cosmic rays can be robustly distinguished from stars by analyzing the sharpness of their edges (using Laplacian or gradient derivatives) rather than just their intensity.

3.3 Despinning

Due to the ASC staggered readout (cf. 2.1), a single "image" actually contains two interleaved fields captured at different times. During the fixed delay Δt between the odd- and even-row readouts, Juno rotates, causing a systematic duplication and misalignment of all scene features. Despinning compensates this intra-frame motion by mapping the even field back to the odd-field acquisition time, after which both fields are merged into a single geometrically consistent frame.

3.3.1 Odd/even field decomposition

Let $I(u,v)$ denote the raw 752×580 image, with u the column index and v the row index. The two half-height fields are extracted by row decimation:

$$I_{\text{even}}(u,y) = I(u,2y+1) \tag{3.1}$$

$$I_{\text{odd}}(u,y) = I(u,2y) \qquad y \in [0,289] \tag{3.2}$$

I_{even} and I_{odd} seem to be inverted because the index starts with 0. Line number 1 (odd) corresponds to $I(u,0)$.

3.3.2 Rotation alignment and merge

For accurate alignment, the motion between both fields is modeled as a 3D rotation of the camera's line-of-sight vectors. The process begins by transforming the spacecraft's angular velocity vector $\boldsymbol{\omega}_{SC}$, which is defined in the spacecraft body frame, into the camera's reference frame (see figure 3.8). This is achieved using the fixed mounting rotation matrix $R_{SC\to cam}$:

$$\boldsymbol{\omega}_{cam} = R_{SC\to cam}\,\boldsymbol{\omega}_{SC} \tag{3.3}$$

From this angular velocity vector, the unit rotation axis $\mathbf{k}$ and the scalar rotation angle θ are defined as :

$$\mathbf{k} = \frac{\boldsymbol{\omega}_{cam}}{\|\boldsymbol{\omega}_{cam}\|}, \quad \theta = \|\boldsymbol{\omega}\|\Delta t \tag{3.4}$$

Using $\omega = 11.967°/\text{s}$ and $\Delta t = 125\,\text{ms}$ yields $\Delta\theta = 1.496°$

Each pixel (u,v) is assigned a 3D vector which represents a pinhole ray using the intrinsic camera parameters :

$$\mathbf{r} = \begin{bmatrix} (u - c_x)\, p_x / f \\ (v - c_y)\, p_y / f \\ 1 \end{bmatrix} \tag{3.5}$$

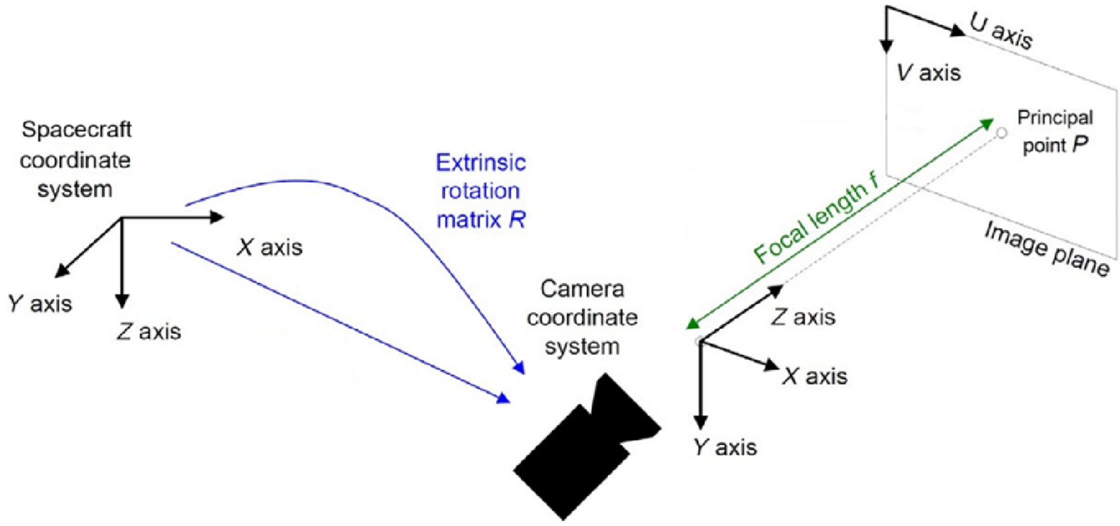


Figure 3.8: Rotation between SC and camera coordinate systems [9].

The rotation from a frame to another is applied using Rodrigues' rotation formula 3.6, allowing to spin a vector around an arbitrary axis $\mathbf{k}$ [10]. For a target pixel in the even field, its corresponding 3D vector $\mathbf{r}$ is rotated around the axis $\mathbf{k}$ by angle θ to align it with the odd-field timestamp (see figure 3.9). The rotated vector $\mathbf{r}'$ is given by:

$$\mathbf{r}' = \mathbf{r}\cos\theta + (\mathbf{k} \times \mathbf{r})\sin\theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{r})(1 - \cos\theta) \tag{3.6}$$

Finally, this rotated vector $\mathbf{r}'$ is reprojected onto the 2D sensor plane to determine the motion-compensated pixel coordinates (u', v'), allowing the even field to be merged seamlessly with the odd field. Since the even field has half the vertical sampling, the corresponding even-field row index is

$$y' = \frac{v' - 1}{2}$$

The intensity is obtained by bilinear interpolation of $I_{\text{even}}(u', y')$. Pixels whose back-

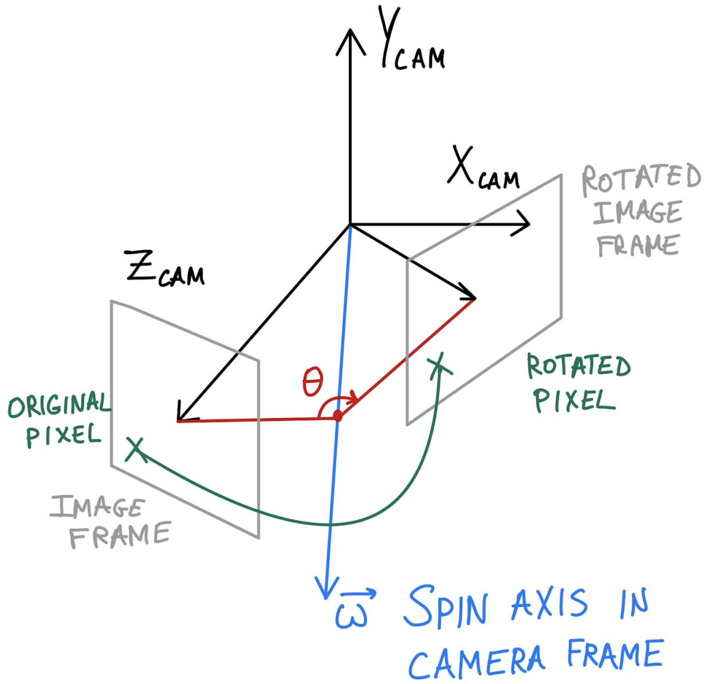
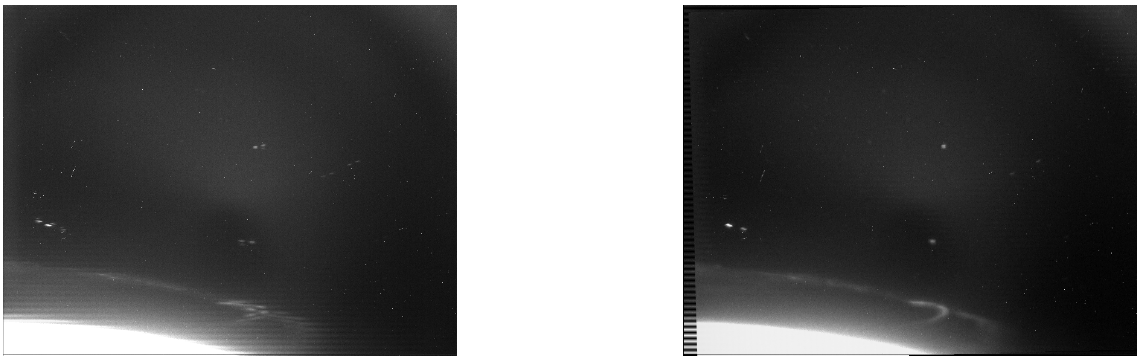


Figure 3.9: Rotation of the camera's pointing vector around the SC's spin axis (expressed in camera frame).

projected coordinates fall outside the even-field bounds are assigned zero, which explains the residual border artifacts observed after despinning.

The two fields are finally merged into a single full-resolution image by interlacing. The odd rows are taken directly from I_{odd}, while the even rows are filled with the motion-compensated samples from $I_{\text{even}\to\text{odd}}$ evaluated on the even-row grid. This preserves the original 752×580 sampling. The total despinning result is shown in Fig. 3.10.



(a) Raw ASC image

(b) Despinned image

Figure 3.10: Effect of the despinning procedure.

3.4 Lens Distortion Correction

The preceding analysis implicitly relied on the pinhole camera model, which assumes the lens is a single point without geometry. Lens distortion is a deviation from the ideal projection in the pinhole model. It is a form of optical aberration where straight lines appear bent in the CCD due to lens curvature (cf 3.11).

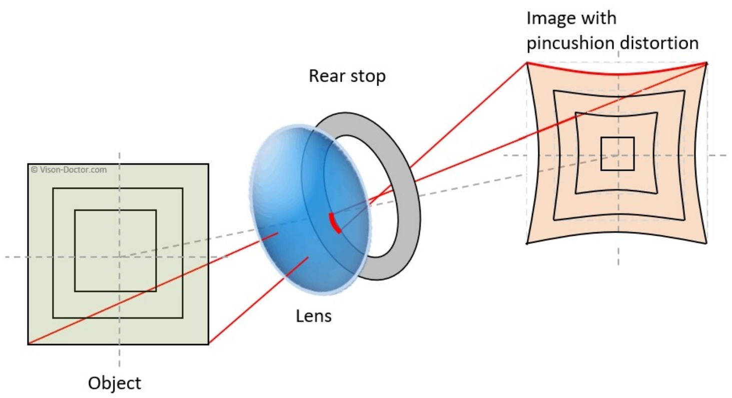


Figure 3.11: Lens distortion origin [11]

3.4.1 Mathematical model

The image can be corrected using a simplified Brown-Conrady transformation, considering only the first-order radial distortion term. To handle the sensor's non-square pixel geometry ($dx = 8.6\mu m, dy = 8.3\mu m$), a pixel aspect ratio $\alpha = dy/dx$ is introduced.
The transformation relates a point in the undistorted image space to the distorted image space:

$$\begin{cases} x = u_n - u_c \\ y = \alpha(v_n - v_c) \\ u_d = u_c + x(1 + \kappa r^2) \\ v_d = v_c + y(1 + \kappa r^2)/\alpha \end{cases} \tag{3.7}$$

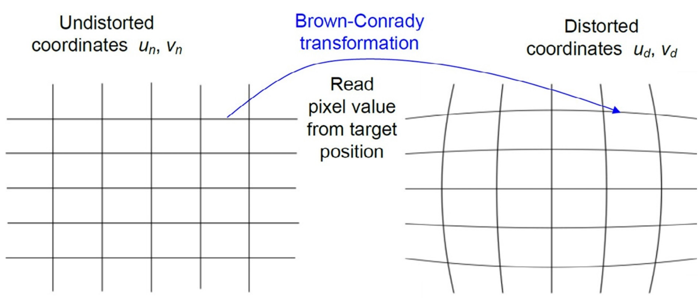


Figure 3.12: Transformation in image spaces [9]

where (u_n, v_n) are the coordinates in the undistorted image, (u_d, v_d) are the coordinates in the distorted image, $r = \sqrt{x^2 + y^2}$ is the distance from optical center, (u_c, v_c) the coordinates of the optical center, and κ a distortion coefficient.. A positive coefficient indicates a pincushion distortion, i.e the lines bend inwards [9].

It is important to note that these equations 3.7 solve the inverse problem : they solve for the input (distorted image) based on the output (corrected image). An inverse mapping is therefore implemented : instead of iterating over the input image and projecting pixels forward, the algorithm iterates over every coordinate (u_n, v_n) in the target image, computes the corresponding (u_d, v_d) and samples the correct intensity value.

3.4.2 Distortion magnitude determination

To assess the scale of the distortion in the Juno data, a simulation was run with the optical parameters of the ASC's camera:

- $\kappa = 3.3 \cdot 10^{-8}$
- $(u_c, v_c) = (383, 257)$
- (W,H)=(752,580)
- (DX,DY)=(8.6,8.3) µm
- α=DY/DX=0.965

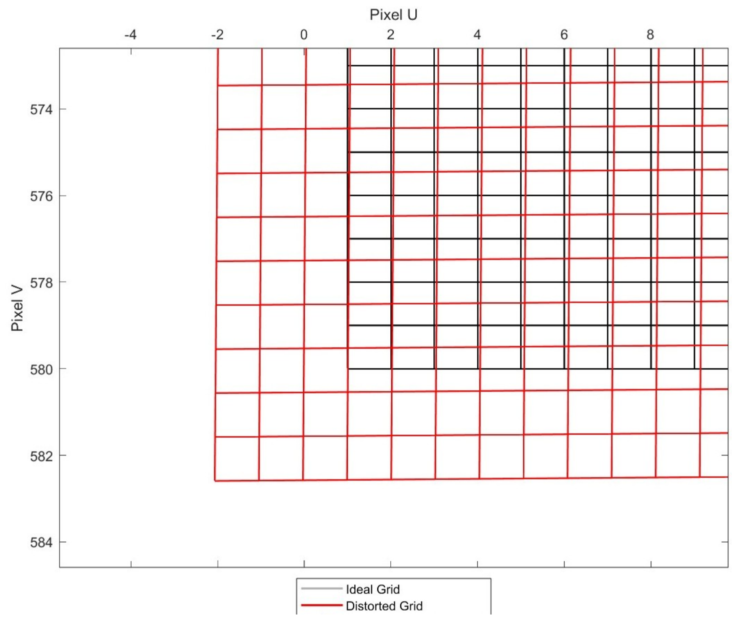


Figure 3.13: Lens distortion simulation (zoomed on corner)

In the simulation 3.13, the distorted grid (red) is projected over the ideal grid of pixels (gray). The distortion is radially dependent, reaching its maximum at the image corners, . In these peripheral regions, the displacement is 3-4 pixels. While visually unnoticeable, it is absolutely non-negligible in the context of atmospheric investigation. Neglecting those few pixels can translate in a geolocation error of hundreds of kilometers.

3.5 Performance Benchmark

This section evaluates the computational resources needed to execute the image processing sequence. The feasibility of running the process on the onboard µASC is assessed given the hardware specifications. One crucial detail is that only the star tracking needs to be performed in flight. If a processing step is numerically too intensive to determine attitude, it can be left out if it does not affect the result too much. The atmospheric investigation is ground-based, therefore computational resources are not a limitation.

3.5.1 Implementation

The most computationally demanding stages of the pipeline are the geometric transformations: despinning and lens correction. To assess their viability on onboard hardware, they were implemented in C++. The logic relies on a standard optimization technique that decouples time-invariant map generation from real-time pixel processing.

Pre-computation Since the lens distortion and rotation parameters are constant for the ASC instrument, the coordinate transformation map is time-invariant. Therefore, the computationally heavy fields are computed only once during initialization. Two floating-point matrices (Look-Up Tables) are generated and stored in memory.

Remapping In flight, the correction is applied to incoming raw frames by referencing the pre-computed LUTs. Because the calculated source coordinates rarely align with integer pixel centers, the algorithm employs bilinear interpolation to resample the pixel intensity. This separation allows the heavy trigonometric and polynomial calculations to be front-loaded, leaving only memory lookups and interpolation for the real-time loop.

3.5.2 Onboard feasibility analysis

The analysis in table 3.1 demonstrates that the full image processing sequence is infeasible for the onboard μASC microcomputer due to two primary constraints :

- Processing Latency: The ASC operates on a 250 ms cycle (4 Hz) [4]. While the geometric correction takes only 22 ms on a 3.6 GHz laptop, onboard processors (op-

Processing Step	Laptop Benchmark	Onboard Feasibility
Memory (RAM)		
Input Image (8-bit raw)	0.42 MB	Feasible
Output Image (8-bit processed)	0.42 MB	Feasible
Despin Maps (2× 32-bit float)	3.33 MB	Inefficient
Lens Correction Maps (2× 32-bit float)	3.33 MB	Inefficient
Total RAM Required	**∼7.5 MB**	Borderline
Processing Time		
Initialization & Map Computing	102.4 ms	N/A (One-time)
Dark Frame Calibration	4.6 ms	Feasible
Column-Median Subtraction	72.1 ms	Critical
Speckle Removal	49.3 ms	High Load
Geometric Correction (Despin + Lens)	22.4 ms	**Critical (FPU Load)**
Total Latency per Frame	**∼148.4 ms**	**Infeasible**

Table 3.1: Feasibility analysis: processing time and memory requirements for onboard execution (single frame)

erating at <100 MHz with limited Floating-Point Unit capabilities) would execute these floating-point interpolations orders of magnitude slower. The estimated onboard execution time would exceed 1 second per frame, causing a processing timeout that would disrupt the star tracker's critical attitude determination function.

- RAM Efficiency: Storing the high-precision LUTs for geometric correction requires approximately 7.5 MB of RAM. While technically possible within the hardware limits, allocating so much of the available memory to static maps is highly inefficient and competes with critical buffers required for star catalogs.

In conclusion, the image processing sequence is NOT feasible in its current form on the µASC due to strict 250ms time budget. This was expected, as the star-tracking microcontroller was not optimized for image cleaning, but sparse feature extraction (analyzing star shapes) rather than transforming the full 436,000 pixel array. The hardware specifications (RAM, CPU clock speed) are constrained to the bare minimum required for this primary function. The µASC should remain focused on lightweight tasks such as simple particle enumeration (thresholding), while the high-load image analysis must be reserved for the ground segment. Further work could analyze the feasibility of skipping certain numerically heavy steps, such as lens correction and median subtraction, and still keeping acceptable tolerance for attitude determination. That would leave only crucial steps such as bias subtraction and de-rotation in the lighter processing sequence.

sta

3.6 Final Output

The image processing chain has successfully transformed raw, noisy sensor data into calibrated scientific imagery. As summarized in figure 3.14, the process systematically addressed the environmental and hardware-specific challenges of the JUNO mission. The

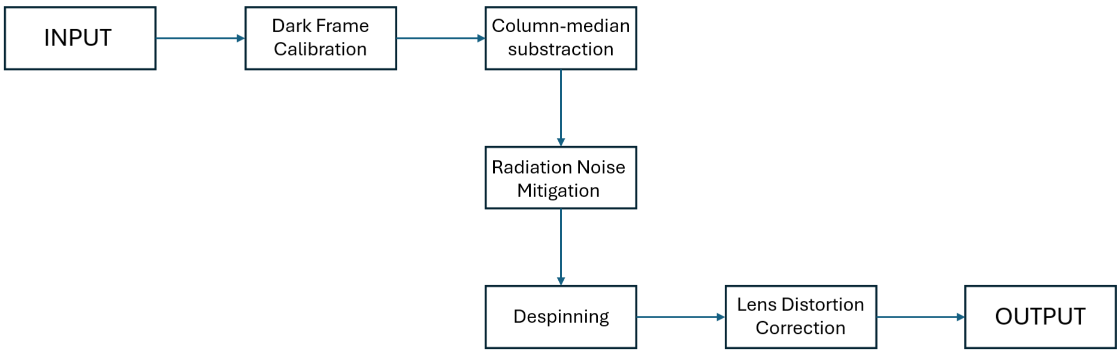


Figure 3.14: Image processing flowchart

final output exhibits an improved SNR and important upgrades :

- Dark calibration has normalized background intensity to nearly zero.
- Gradient-based filtering proved effective in mitigating the high-energy particle speckles due to harsh radiation environment. By selectively targeting high-frequency impulse noise without applying a global smoothing kernel, atmospheric and stellar features are preserved.
- Lens correction and despinning steps have aligned the image features, merging the rotation-induced duplicates into one single object.

While the core features of the atmosphere are now distinct, residual artifacts remain, specifically at the periphery of the frame. The left-hand side of the image exhibits edge-effects from the despinning. These artifacts are attributed to the lack of overlapping data for interpolation at the image border. Some vignetting is still present in the top right corner.

Despite these minor localized issues, the central region of interest is calibrated and geometrically corrected, rendering the dataset suitable for the intended scientific analysis.

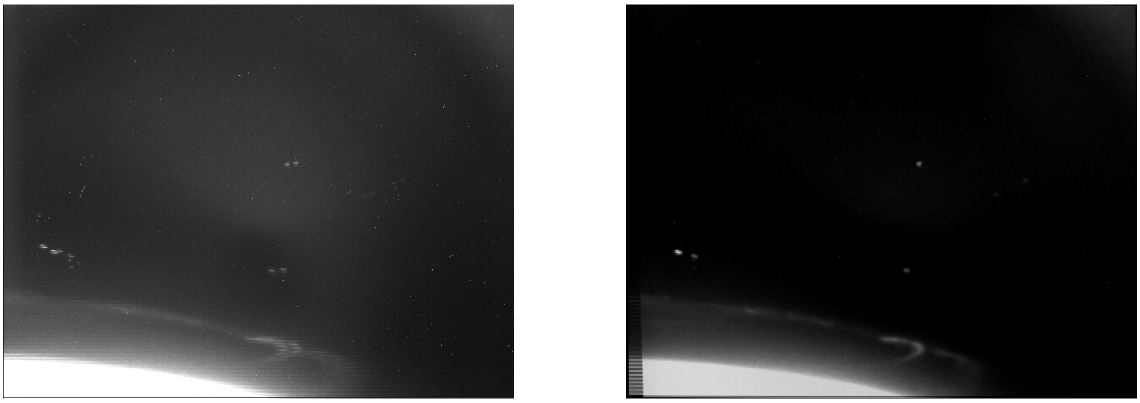


Figure 3.15: Pre & Post-Processing

Chapter 4

Horizon Detection

4.1 Problem statement & Motivation

This chapter investigates the feasibility of extracting Jupiter's horizon purely based on the dataset. Combined with star mapping, the spacecraft's position may be recovered. Accurate detection of the planetary horizon is a required for this 2 purposes :

- Optical navigation
- Atmospheric investigation

Fitting a curve to such partial data presents a mathematical challenge. Traditional algebraic methods, such as the standard least-squares fit, are known to be statistically inconsistent when applied to short arcs. They underestimate the curvature radius, leading to substantial errors in estimating the planet's center and the spacecraft's relative position.

The motivation for this study is to implement and validate a robust fitting algorithm capable of overcoming these limitations.

4.2 Gradient-based Edge Tracking

The goal of this section is to extract Jupiter's horizon purely from the dataset, independently of trajectory kernels. In an ideal case, the horizon is defined as the set of points where the intensity gradient is maximized. Standard edge detection methods, such as the Canny Detector, prove unsuitable for the ASC images. Due to the faintness of the limb (thus low SNR), these methods misidentify high contrast features such as stars, auroras and atmospheric haze as part of the horizon.

To overcome this limitation, a custom algorithm was made just for this purpose. Unlike global edge detectors, this algorithm enforces spatial continuity, tracking the horizon across adjacent columns to reject isolated noise. While generally robust, the method remains sensitive to high-intensity transient features, such as auroras, which can intersect the limb and bias the method. The detection logic is implemented as follows:

- Scanning: Iterate through columns j.
- Gradient Computation: Calculate the vertical gradient vector ∇I_j for the column.
- Thresholding: Calculate an adaptive threshold $T_j = \mu_j + k \cdot \sigma_j$, where μ_j and σ_j are the mean and standard deviation of the gradients in column j.
- Peak Finding: If a gradient $\nabla I_{i,j} > T_j$ is found, verify it is the local maximum within a window $[i-3, i+3]$. This step ensures the data is continuous. Update the horizon row $R_j = i$.
- Zero-Order Hold: If no gradient in the window exceeds T_j, the algorithm maintains the previous horizon row: $R_j = R_{j-1}$, assuming the limb has not moved.

This method does not work as is on the last pair of images 2.5. The detected "horizon" is of course the separation between the bright and dark regions, even though in reality it's an effect of sunlight scattering. These regions need to be explicitly masked out for the algorithm to work.

4.3 Circle Fitting Method

The edge tracking yielded a set of vaguely connected points. The goal is now to fit a smooth curve that approximates best this discontinuous line.

4.3.1 Validity of local circle approximation

Jupiter is an oblate spheroid, with a semi-major axis (equatorial radius) and semi-minor axis (polar radius) measuring respectively 71492km and 66854km. These values correspond to the point of 1 bar of pressure. This gives a flattening ratio of :

$$f = \frac{a-b}{a} = 6.49\%$$

This makes Jupiter the second most flattened planet in the Solar System after Saturn, approximating it as a sphere would yield a non-negligible error.

However, the spacecraft is looking only at an arc of $\sim 16°$ of Jupiter (data derived from SPICE). A simulation was performed to evaluate whether a local spherical approximation is valid (see 4.1). The outcome shows that the curvature of the limb within the narrow FOV is indistinguishable from a spherical arc. The simulation indicated that fitting a circle to the true ellipsoid segment yielded a radial deviation of 12.68 km, which is less than the spatial resolution of a single pixel. An important distinction is that the radius of the fitted circle is not the radius of Jupiter, but rather a local radius of curvature.

In conclusion, fitting a circle rather than an ellipse is valid given the narrow view angle.

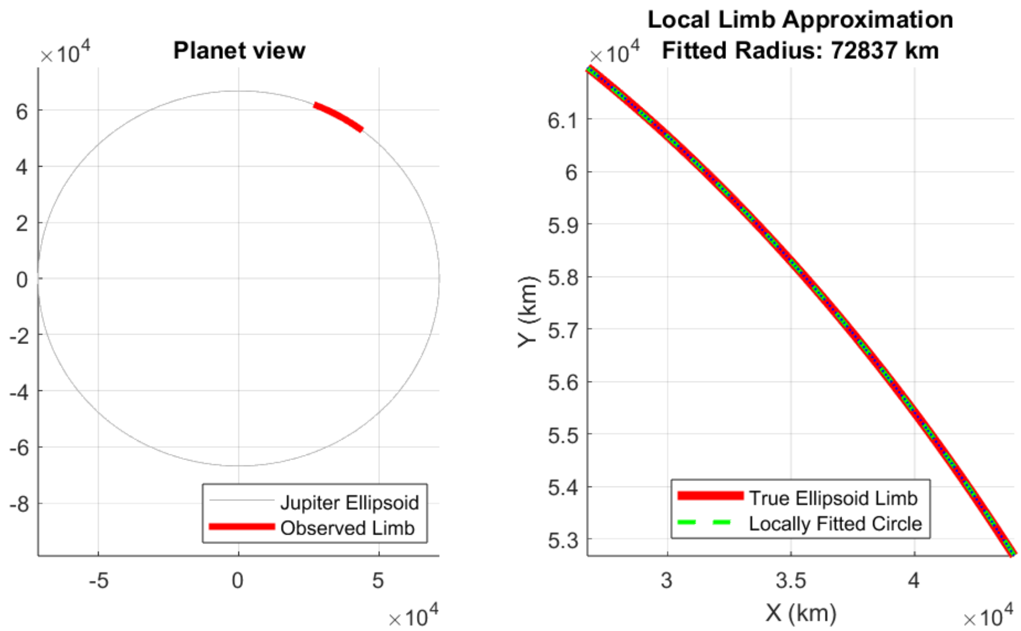


Figure 4.1: Simulation of Jupiter's ellipsoid and locally fit circle

4.3.2 Algorithm selection and validation

Fitting a circle to a set of 2D points is a common problem in image analysis. The most classic approach is performing Kåsa's fit. It is a simple least-square problem where the squared distance between the estimated circle and the set of points is minimized. It performs well if the set of points is at least half a circle. It tends to be heavily biased towards smaller circles when the set is a short arc, which is precisely the case for Juno's dataset.

For this study's purpose, the most accurate method is a Hyperaccurate (Hyper) algebraic circle fit proposed by Al-Sharadqah and Chernov [12]. This method minimizes the algebraic error for the general circle equation defined by the parameter vector $\mathbf{A} = (A, B, C, D)^T$:

$$A(x^2 + y^2) + Bx + Cy + D = 0$$

The method seeks to minimize the mean square algebraic distances subject to a constraint that prevents the trivial solution $\mathbf{A} = 0$. This is formulated as a generalized eigenvalue problem:

$$M\mathbf{A} = \eta N\mathbf{A}$$

where M is the matrix of data moments, η is the eigenvalue, and N is the constraint matrix, which for centered data is defined as:

$$N = \begin{pmatrix} 8\bar{z} & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$$

where $\bar{z}$ is the mean of $z = x^2 + y^2$. By selecting the eigenvector corresponding to the smallest positive eigenvalue, this constraint neutralizes the geometric bias to the order of $O(n^{-2})$, ensuring a stable estimation of the radius even when the visible horizon represents a short limb. Algorithm implemented in C++ with use of AI tools.

Unsurprisingly, the Hyper method outperforms the basic least-square algorithm (see 4.4 for more details). Due to the short visible limb and therefore large circles, the uncertainty on the radius is significant. The radii yielded by Kåsa's and Hyper methods are respectively 1141 and 2248 pixels, which is about 100% difference. This was expected, as the first one is biased towards shorter circles. Still, the efficiency of any circle fitting method is limited

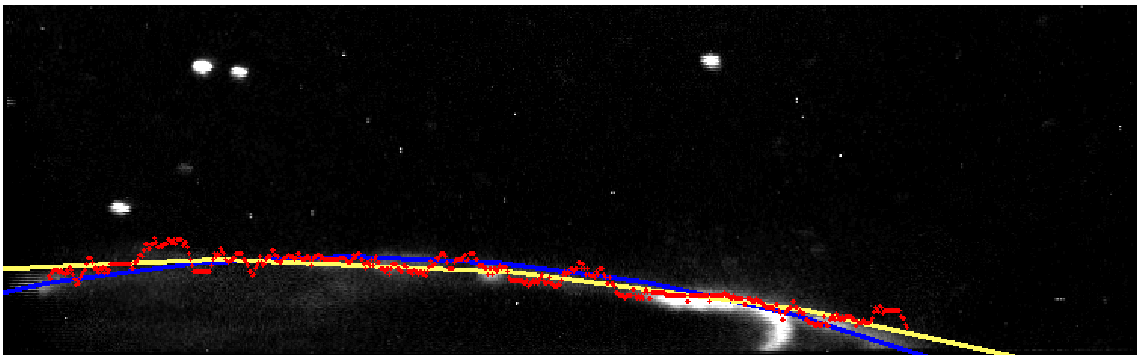


Figure 4.2: Red discontinuous data points show the output of edge-tracking. Blue line is Kåsa's fit. Yellow line is Hyper fit. Image gained x6.

by the quality of the edge tracking algorithm, and ultimately the data itself.

4.4 Comparison with SPICE output

Only by specifying the exact time of the image capture, the SPICE-based program computes the exact location and field of view of the camera. It can then represent where the 1-bar planet edge is on the image without even analyzing it. This gives a baseline for comparing horizon detection algorithms.

The evaluation metrics for both methods is the error area (total number of mismatched pixels) and the deviation from the radius estimate.

Algorithm	**Error Area (px)**	**Radius (px) (% error)**
SPICE (kernel data)	0	2458.16 (+0%)
Kåsa	7937	1141.39 (-54%)
Hyper	7261	2248.52 (-8%)

The Hyper fit outperforms the least-square in both metrics. The Kåsa fit exhibits a massive bias towards smaller circles, underestimating the radius by 54%. Conversely, the Hyper fit maintains the curve shape, recovering the radius with an 8% deviation despite processing only a 16° arc.

However, a persistent error area of approximately 7,000 pixels remains for both methods. This discrepancy is not algorithmic but physical. The edge-tracking detects the limb based on the strongest gradient. This corresponds to high-altitude stratosphere, scattered sunlight, auroral emissions, rather than the 1-bar pressure level defined by the SPICE

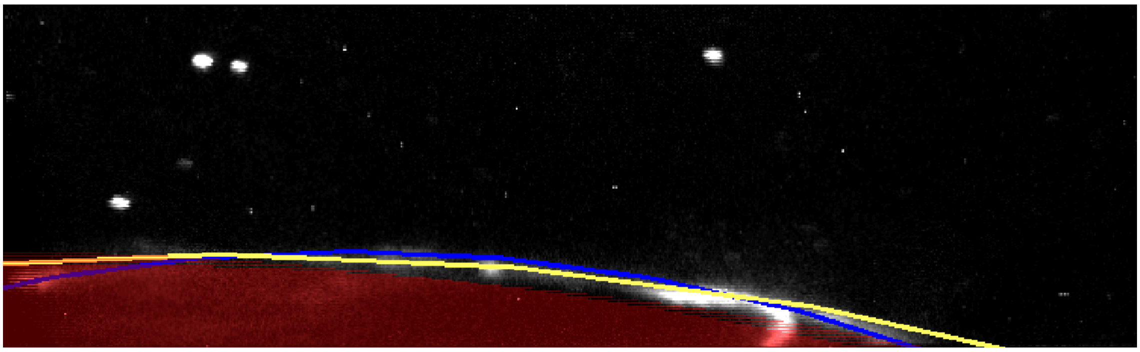


Figure 4.3: Red area represents the 1-bar planet edge extracted from SPICE. Blue line is Kåsa's fit. Yellow line is Hyper fit. Image gained x6.

model. As a consequence, the visible limb detected by the camera is offset from the physical limb provided by the kernels.

4.5 Conclusions

The investigation of horizon detection on Juno images reveals the discrepancies between physical reality and optical observations. A horizon in a planetary image is not a binary boundary but a continuous gradient of atmospheric scattering, especially when observed from close. While SPICE kernels provide an exact mathematical model of the 1-bar edge, the camera observes the cloud tops and stratospheric hazes. Consequently, any image-based edge detection will inherently deviate from the kernel data. This atmospheric offset is a physical reality of the dataset, not a failure of the fitting algorithm.

For scientific analysis where the spacecraft's trajectory is well-defined, SPICE kernels should be the primary source for geometry. Image-based horizon detection should be reserved for scenarios where the spacecraft position is the unknown variable (e.g., autonomous navigation). In such cases, the algorithm must combine the Hyper fit with an atmospheric model to account for the offset between the visible atmosphere and the 1-bar edge.

Chapter 5

Star Mapping

The JUNO images contain a limited number of background stars that appear as compact intensity 'blobs', distributed over a small number of pixels. By computing the center-of-mass for each blob, every star can be reduced to a single point vector in space.

Using these vectors, the relative spatial configuration of these stars can be determined. This set can then be compared to the relative angular separations of stars listed in an existing celestial catalogue. Once the correct mapping between the image - catalogue is found, the orientation of the star set with respect to the image plane can be computed.

From this information, the sensor attitude can be determined accurately. Since the sensor is fixed on the spacecraft, this directly yields the spacecraft orientation in inertial space.

5.1 Star Detection

In the ASC images, stars appear with strongly varying intensities. On the one hand, a small number of stars are recorded as highly concentrated, near-saturated intensity blobs. These starts are clearly distinguishable from the background, and can be detected reliably by applying a high absolute intensity threshold. On the other hand, the majority of stars are significantly fainter and closer to the noise floor, which makes them way harder to detect. To attempt to detect both groups of stars, a hybrid approach is implemented in `star_tracker.py`:

5.1.1 Dual mask construction

Let $I(x, y)$ be the grayscale image (after denoising, see Chapter 3). A binary detection mask is formed as

$$M_{\text{final}} = M_{\text{sat}} \ \vee \ M_{\text{faint}}, \tag{5.1}$$

where M_{sat} flags saturated/near-saturated pixels and M_{faint} targets small-amplitude point sources.

(1) Saturated blobs. Saturated or near-saturated stars are detected directly from the raw intensity image using a fixed saturation threshold, ensuring robust localisation of the brightest point sources. A saturation threshold T_{sat} is applied directly on I:

$$M_{\text{sat}}(x, y) = 1\{I(x, y) \geq T_{\text{sat}}\}. \tag{5.2}$$

This preserves large bright stars even when adaptive statistics (mean/std) are dominated by Jupiter glow or background gradients.

(2) Faint stars via background subtraction. In parallel, faint stars are detected through background subtraction followed by adaptive thresholding, which enhances small-amplitude point sources while remaining locally robust to background variations. A smooth background estimate $B(x, y)$ is computed with a large median filter kernel (e.g. 61×61 in our default preset):

$$B = \text{MedianFilter}(I, k_{\text{bg}}), \qquad I'(x, y) = \max(I(x, y) - B(x, y), 0). \tag{5.3}$$

Optionally, I' is normalized to $[0, 1]$ by its maximum. An adaptive threshold is then computed using the global mean and standard deviation:

$$T = \mu(I') + \alpha\, \sigma(I'), \tag{5.4}$$

where α is the `threshold_multiplier` (e.g. $\alpha \approx 1.4$ in the default settings). The faint-star mask is

$$M_{\text{faint}}(x, y) = 1\{I'(x, y) \geq T\}. \tag{5.5}$$

Optionally, a morphological opening removes isolated pixels and small artifacts.

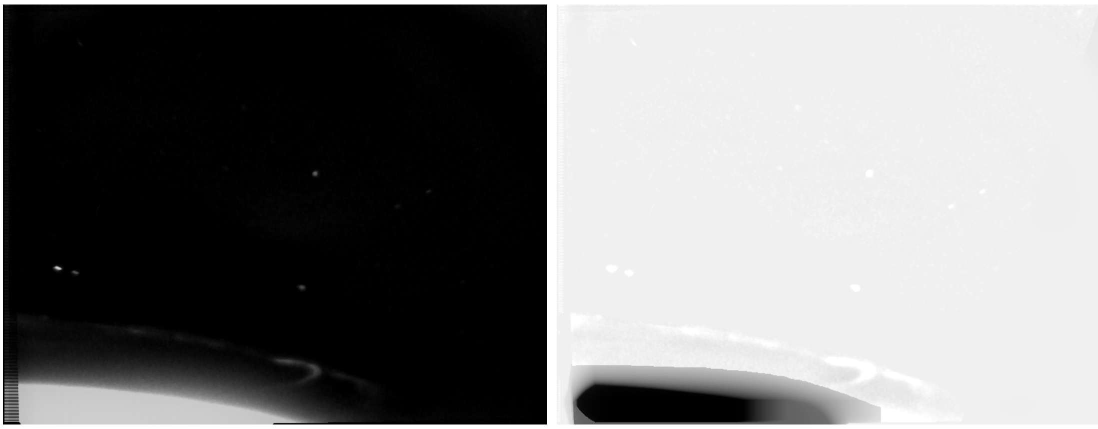


Figure 5.1: (Left) Input frame used for star detection. (Right) Saturated mask

5.1.2 Centroid extraction

Connected components are extracted from M_{final}. For each connected component Ω, we compute:

- its area $A = |\Omega|$ (used to reject oversized artifacts), and
- an intensity-weighted centroid $(\hat{x}, \hat{y})$ using the original (non-binary) image intensities.

Let $w(x, y) = I(x, y)$ for $(x, y) \in \Omega$. The weighted centroid is:

$$\hat{x} = \frac{\sum_{(x,y)\in\Omega} x\, w(x, y)}{\sum_{(x,y)\in\Omega} w(x, y)}, \qquad \hat{y} = \frac{\sum_{(x,y)\in\Omega} y\, w(x, y)}{\sum_{(x,y)\in\Omega} w(x, y)}. \tag{5.6}$$

The sum $\sum w(x, y)$ is used as a flux proxy to rank detections; the pipeline keeps only the top-N brightest detections (e.g. $N = 400$), because the subsequent matching is designed around bright catalogue stars.

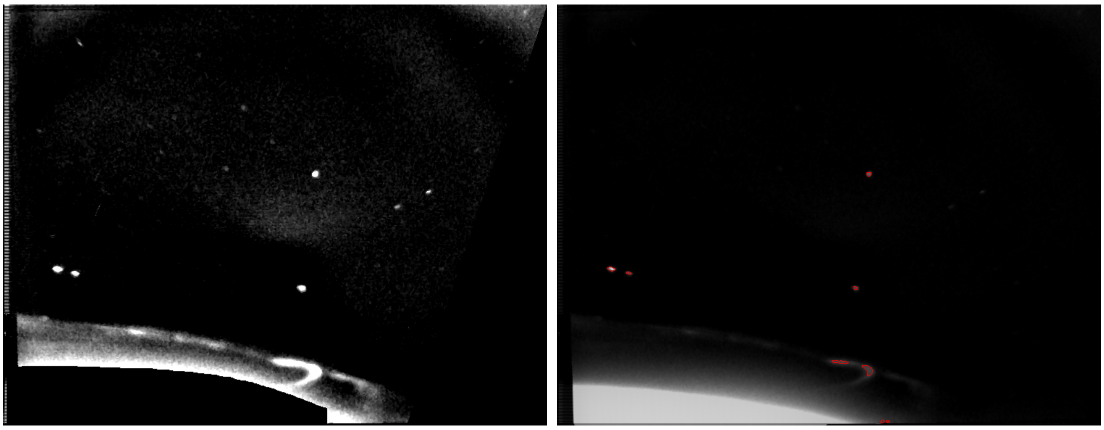


Figure 5.2: (Left) Mask-subtracted image M_{sat}. (Right) Detected star candidates

Figure 5.3: Detected star candidates overlaid on the image. Each blob is reduced to a subpixel centroid via Eq. (5.6).

5.2 Star Catalogue

5.2.1 Catalogue selection

A practical limitation is that not every catalogue is ideal across the full magnitude range of interest:

- **Bright stars:** a HIP/Bright-Star-Catalogue [13] style list is reliable for very bright objects (roughly mag $\in [-2, 4]$).
- **Fainter stars:** The Gaia DR3 catalogue [14] provides dense coverage and accurate astrometry, but a given curated export may miss some of the extremely bright stars or require special handling of identifiers.

Therefore we use a mixed approach: HIP-based entries for the brightest stars, and Gaia entries beyond that range, with a crossmatch/name-lookup layer for consistent labeling (HIP numbers, Gaia IDs, and friendly names/IAU where available).

5.2.2 Negligibility of Parallax effects

Catalogue coordinates are provided as inertial directions from the Solar System barycentric perspective (J2000/ICRF). For star-tracking, we treat stars as fixed points on the celestial sphere. The error from observing from Jupiter instead of Earth is dominated by stellar parallax:

$$\theta_{\text{parallax}} \approx \frac{B}{D}, \tag{5.7}$$

where B is the observer baseline (Earth–Jupiter distance, order of a few AU) and D is the star distance (typically many parsecs, i.e., light-years). Even for very nearby stars, the resulting angular difference is on the order of arcseconds, which is negligible compared to the matching tolerance used in our robust solver (order of 10^{-1} degrees). Hence, using the same inertial star directions is sufficiently accurate for this study.

5.2.3 Inertial vector conversion

Each catalogue star is represented by its equatorial coordinates (α, δ) in radians (RA, Dec). The corresponding inertial unit vector $\mathbf{k}$ is:

$$\mathbf{k}(\alpha, \delta) = \begin{bmatrix} \cos\delta\, \cos\alpha \\ \cos\delta\, \sin\alpha \\ \sin\delta \end{bmatrix}, \qquad \|\mathbf{k}\| = 1. \tag{5.8}$$

In the code, these are stored as `vec_inertial` per catalogue entry and are the fundamental primitives used for angular comparisons and attitude estimation.

5.3 Attitude Estimation

5.3.1 Pattern ambiguity

If we only consider relative geometry between stars, the key invariant is the angle between two lines of sight:

$$\theta_{ij} = \arccos\left(\mathbf{u}_i^\top \mathbf{u}_j\right), \tag{5.9}$$

where $\mathbf{u}_i$ and $\mathbf{u}_j$ are unit direction vectors.

With three stars, there are three pairwise angles; with four stars, there are six. In a large catalogue (thousands to tens of thousands of stars), a triple of angles may still occur for multiple different triples, especially under measurement noise. Adding a fourth star provides additional constraints (six angles total), and the probability of accidental collisions becomes extremely small.

However, even if the identity of the stars is known, the attitude is still not obtained from angles alone: the same 4-star shape can appear at different locations on the sensor, which corresponds to different boresight directions. In other words, inter-star angles constrain the pattern but not the absolute pointing. The absolute pixel locations must be mapped to absolute rays in the camera frame, and these rays must be aligned with inertial catalogue vectors.

In our implementation, this disambiguation is achieved by (i) generating hypotheses from angle-consistent pairs, and (ii) validating them against many additional stars (RANSAC inliers). This plays the same role as "adding the 4th star and beyond": it collapses the remaining ambiguity to a unique attitude.

5.3.2 Camera ray projection

For each detected centroid $(\hat{x}, \hat{y})$ (in image pixel coordinates), we compute a unit ray $\mathbf{v}_{\text{cam}}$ using a pinhole camera model with the ASC intrinsics. Let (f_x, f_y) be focal lengths in pixels and (c_x, c_y) the principal point. Then:

$$x_n = \frac{\hat{x} - c_x}{f_x}, \qquad y_n = -\frac{\hat{y} - c_y}{f_y}, \tag{5.10}$$

where the minus sign accounts for the image y-axis pointing downward while the camera frame convention uses $+y$ upward.

A small radial distortion term is optionally applied using a coefficient κ:

$$r^2 = x_n^2 + y_n^2, \qquad \begin{bmatrix} x_d \\ y_d \end{bmatrix} = (1 - \kappa r^2) \begin{bmatrix} x_n \\ y_n \end{bmatrix}. \tag{5.11}$$

Finally, the (unnormalized) ray is $\tilde{\mathbf{v}} = [x_d, y_d, 1]^\top$ and the unit ray is

$$\mathbf{v}_{\text{cam}} = \frac{\tilde{\mathbf{v}}}{\|\tilde{\mathbf{v}}\|}. \tag{5.12}$$

Note on resizing. Since figures in this report may be exported/rescaled, the code rescales

(f_x, f_y, c_x, c_y) by the ratio of the current image dimensions to the hardware sensor resolution before applying Eq. (5.10).

5.3.3 Fast candidate search

To efficiently propose catalogue correspondences, angles between pairs of the N brightest catalogue stars are precomputed:

$$\theta_{ab}^{\text{cat}} = \arccos\left(\mathbf{k}_a^\top \mathbf{k}_b\right). \tag{5.13}$$

In classical star trackers, Mortari's Pyramid star pattern recognition algorithm [15] introduces the so-called K-vector technique to accelerate this search. All catalogue pair-angles are stored in a single sorted array. A small auxiliary index (the K-vector) then maps a range query

$$\theta_{\text{obs}} \pm \varepsilon$$

to (approximately) the corresponding lower and upper indices in that sorted list. This converts an exhaustive scan over all pairs into a fast range lookup over a short contiguous interval.

A discretized equivalent is used here: the catalogue angles are grouped into bins of width $\Delta\theta$ (e.g. $0.01°$), yielding a dictionary

$$\text{bin} \;\rightarrow\; \{(a, b, \theta_{ab}^{\text{cat}})\}.$$

Given an observed detection pair-angle, only the corresponding bin (and a small neighborhood of adjacent bins to account for measurement noise) is queried. This reduces the candidate set by several orders of magnitude, while preserving the same core objective as a K-vector range search: limiting hypothesis generation to catalogue pairs with compatible inter-star angles.

5.3.4 Robust solver (RANSAC/Wahba)

Let $\{\mathbf{v}_i\}_{i=1}^{M}$ be the detected camera rays (Eq. (5.12)) and $\{\mathbf{k}_j\}_{j=1}^{K}$ the catalogue inertial vectors (Eq. (5.8)). We seek the rotation matrix $\mathbf{R} \in SO(3)$ such that

$$\mathbf{k}_{\pi(i)} \approx \mathbf{R}\,\mathbf{v}_i, \tag{5.14}$$

where $\pi(i)$ is the catalogue index matched to detection i.

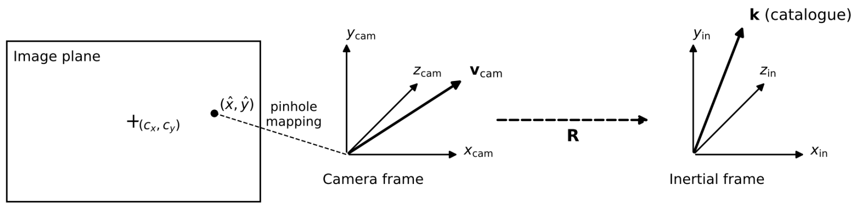


Figure 5.4: From Image Plane Coordinates to Inertial Frame Vectors

(1) RANSAC hypothesis generation from one angle-consistent pair. We precompute all detection-pair angles θ_{ij}^{det} via Eq. (5.9). Each RANSAC iteration:

1. randomly selects a detection pair (i, j) with measured θ_{ij}^{det},
2. retrieves candidate catalogue pairs (a, b) whose θ_{ab}^{cat} lies within the corresponding bin neighborhood,
3. builds a candidate attitude $\mathbf{R}$ using the TRIAD method from the two vector correspondences.

TRIAD [16] construction. Given two non-collinear camera rays $(\mathbf{v}_1, \mathbf{v}_2)$ and their inertial matches $(\mathbf{k}_1, \mathbf{k}_2)$, TRIAD forms orthonormal bases:

$$\mathbf{t}_1 = \mathbf{v}_1, \quad \mathbf{t}_2 = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{\|\mathbf{v}_1 \times \mathbf{v}_2\|}, \quad \mathbf{t}_3 = \mathbf{t}_1 \times \mathbf{t}_2, \tag{5.15}$$

and similarly $(\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3)$ from $(\mathbf{k}_1, \mathbf{k}_2)$. Writing

$$\mathbf{C}_{\text{cam}} = [\mathbf{t}_1\ \mathbf{t}_2\ \mathbf{t}_3], \qquad \mathbf{C}_{\text{in}} = [\mathbf{s}_1\ \mathbf{s}_2\ \mathbf{s}_3],$$

the attitude mapping camera $\to$ inertial is

$$\mathbf{R} = \mathbf{C}_{\text{in}}\, \mathbf{C}_{\text{cam}}^{\top}. \tag{5.16}$$

(2) Inlier finding. For a candidate $\mathbf{R}$, every detected ray is projected into the inertial frame:

$$\hat{\mathbf{k}}_i = \mathbf{R}\, \mathbf{v}_i. \tag{5.17}$$

We assign it to the catalogue star with maximum dot product:

$$j^*(i) = \arg\max_j\ \mathbf{k}_j^{\top} \hat{\mathbf{k}}_i, \tag{5.18}$$

and declare an inlier if the angular error is below a tolerance ε:

$$\arccos\Big(\mathbf{k}_{j^*(i)}^{\top} \hat{\mathbf{k}}_i\Big) \leq \varepsilon. \tag{5.19}$$

The hypothesis with the highest inlier count is kept.

(3) Wahba [17] refinement using Davenport's Q-method [18]. Once a consensus set of inliers is found, the attitude is refined by solving Wahba's problem:

$$\mathbf{R}^* = \arg\min_{\mathbf{R}\in SO(3)} \sum_{i\in\mathcal{I}} w_i \left\|\mathbf{k}_{\pi(i)} - \mathbf{R}\mathbf{v}_i\right\|^2, \tag{5.20}$$

where $\mathcal{I}$ is the inlier set and w_i are weights (unity in our default configuration). The implementation uses Davenport's Q-method to obtain the optimal rotation via a quaternion eigenproblem, then converts the quaternion to $\mathbf{R}^*$.

5.3.5 Boresight calculation

The final output of the attitude solver is $\mathbf{R}$ mapping camera-frame rays to inertial directions. The camera boresight is the camera z-axis:

$$\mathbf{b}_{\mathrm{cam}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathbf{b}_{\mathrm{in}} = \mathbf{R}\mathbf{b}_{\mathrm{cam}} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \tag{5.21}$$

The corresponding pointing angles are:

$$\delta = \arcsin(z), \qquad \alpha = \operatorname{atan2}(y, x), \tag{5.22}$$

with α wrapped to $[0, 2\pi)$ and both angles reported in degrees. This yields the final **RA/Dec of the optical axis** for the considered frame.

5.3.6 Mapping results

After the refined attitude is obtained, every detection is matched again using the nearest-by-dot criterion (Eq. (5.18)) within a slightly looser angular threshold (e.g. 0.3°). The image (Fig. 5.5) is then annotated with detected centroids, matches star names/identifiers (HIP / Gaia / friendly name when available) and the estimated boresight RA/Dec.

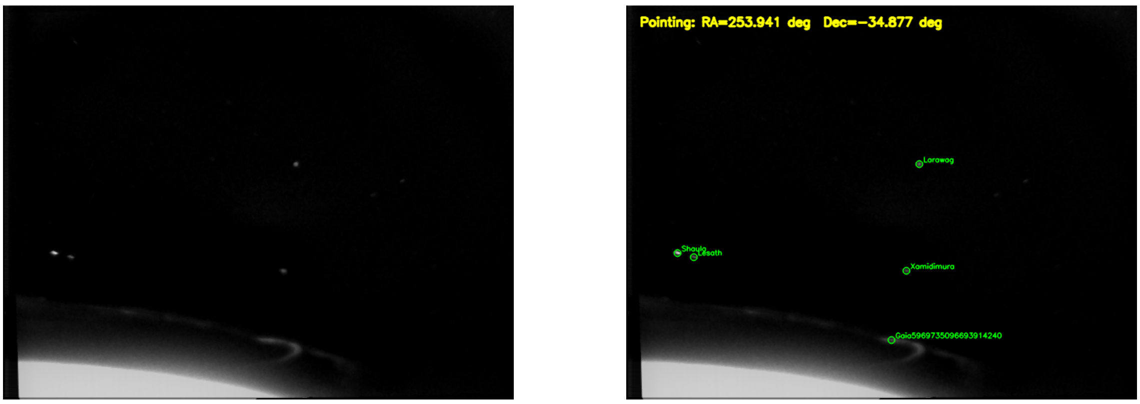


Figure 5.5: Final star mapping result. Matched stars are labeled in the image plane, and the estimated camera pointing (α, δ) is printed.

Chapter 6

Applications

6.1 Juno's position vector

As defined in the research objectives (Chapter 1), the final goal of this study was to combine the limb geometry and the star-derived attitude to recover an image-constrained spacecraft position estimate. Ideally, this would close the loop on the geometric reconstruction.

However, the investigation in Chapter 4 demonstrated that the optical horizon detected in the ASC images does not correspond to the physical 1-bar reference surface provided by the SPICE kernels. Instead, it represents a diffuse boundary of stratospheric haze and scattered sunlight, introducing a systematic offset that cannot be fully corrected without a complex atmospheric radiative transfer model.

Consequently, applying the navigation algorithm to this specific dataset yields a biased position fix. Therefore, the following subsection focuses on the **theoretical derivation** of the horizon navigation method. This explains the mathematical logic intended to be applied, demonstrating how the position vector is constructed in a scenario where the visual limb is detected reliably.

6.1.1 Constructing the position vector

Limb pixels to camera-frame rays

Let (x_i, y_i) be pixel coordinates sampled from the detected horizon curve, where $i = 1, \ldots, N$ indexes valid limb points. Because the images are already lens-corrected and despun (Chapter 3), a pinhole model is sufficient for ray construction.

Using the camera intrinsics (f_x, f_y, c_x, c_y) in pixels, each pixel is mapped to a normalized direction in the camera frame:

$$x_{n,i} = \frac{x_i - c_x}{f_x}, \qquad y_{n,i} = -\frac{y_i - c_y}{f_y}, \tag{6.1}$$

where the minus sign accounts for the downward image y-axis convention. The corresponding (unnormalized) ray is

$$\tilde{\mathbf{v}}_i = \begin{bmatrix} x_{n,i} \\ y_{n,i} \\ 1 \end{bmatrix}, \qquad \mathbf{v}_i = \frac{\tilde{\mathbf{v}}_i}{\|\tilde{\mathbf{v}}_i\|}. \tag{6.2}$$

Camera intrinsics. For the JUNO ASC camera, the focal length and principal point are known from calibration and are used directly. The focal lengths in pixel units are given by

$$f_x = \frac{f}{D_X}, \qquad f_y = \frac{f}{D_Y}, \tag{6.3}$$

with $f = 20006$ μm and $(D_X, D_Y) = (8.6, 8.3)$ μm. The principal point is fixed at $(c_x, c_y) = (383, 257)$ pixels.

Camera rays to inertial rays

Let $\mathbf{R}_{I \leftarrow C}$ be the rotation matrix from camera frame to inertial frame estimated by the star mapping solver (Chapter 5). Each limb ray becomes an inertial unit vector:

$$\mathbf{u}_i = \mathbf{R}_{I \leftarrow C}\, \mathbf{v}_i, \qquad \|\mathbf{u}_i\| = 1. \tag{6.4}$$

Spherical horizon constraint (tangent-ray cone)

Assume Jupiter is locally approximated as a sphere of radius R (as motivated in Chapter 4). Let $\mathbf{r}$ be the unknown spacecraft position vector in a Jupiter-centered inertial frame (origin at Jupiter's center, axes aligned with J2000/ICRF). A limb ray from the spacecraft has the parametric form:

$$\mathbf{p}_i(s) = \mathbf{r} + s\,\mathbf{u}_i, \qquad s \geq 0. \tag{6.5}$$

Tangency to the sphere $\|\mathbf{p}\| = R$ occurs when the quadratic in s has zero discriminant. Expanding $\|\mathbf{r} + s\mathbf{u}_i\|^2 = R^2$ yields:

$$s^2 + 2s\,(\mathbf{r}^\top \mathbf{u}_i) + (\|\mathbf{r}\|^2 - R^2) = 0. \tag{6.6}$$

The tangency condition is:

$$(\mathbf{r}^\top \mathbf{u}_i)^2 = \|\mathbf{r}\|^2 - R^2, \qquad i = 1, \ldots, N. \tag{6.7}$$

This implies all limb rays lie on a cone with axis $\mathbf{r}$, i.e. the dot product with the axis direction is (ideally) constant over all i (see Fig. 6.1).

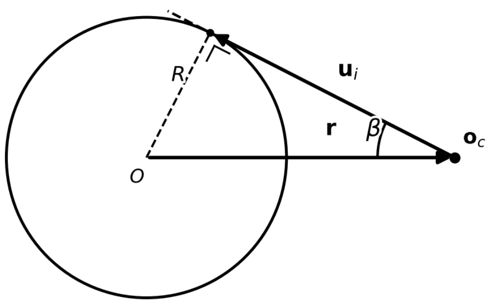


Figure 6.1: Spherical horizon tangent geometry

Cone-axis fit and range recovery

Define the unit position direction:

$$\mathbf{n} = \frac{\mathbf{r}}{\|\mathbf{r}\|}, \qquad \|\mathbf{n}\| = 1. \tag{6.8}$$

Using Eq. (6.7) and $\mathbf{r} = \rho\mathbf{n}$ with $\rho = \|\mathbf{r}\|$, one obtains:

$$(\rho\, \mathbf{n}^\top \mathbf{u}_i)^2 = \rho^2 - R^2 \;\Rightarrow\; (\mathbf{n}^\top \mathbf{u}_i)^2 = 1 - \left(\frac{R}{\rho}\right)^2. \tag{6.9}$$

Hence $\mathbf{n}^\top \mathbf{u}_i$ should be (up to sign) constant across limb rays. Because the observed limb is only a short arc, we estimate $\mathbf{n}$ and the constant c by least squares:

$$\min_{\|\mathbf{n}\|=1,\, c} \sum_{i=1}^{N} (\mathbf{n}^\top \mathbf{u}_i - c)^2. \tag{6.10}$$

Remark. Because the detected horizon typically covers only a short arc, the cone-axis estimation in Eq. (6.10) can become weakly constrained. In direction space, the limb rays $\{\mathbf{u}_i\}$ occupy only a small segment of the cone–sphere intersection, which limits the information available to determine the cone axis. As a result, the smallest-eigenvalue eigenvector of the covariance-like matrix $\mathbf{C}$ may become ill-conditioned [19] and sensitive to outliers in the limb points and small biases in the attitude solution.

For a given $\mathbf{n}$, the optimal c is $c = \frac{1}{N}\sum_i \mathbf{n}^\top \mathbf{u}_i$. Substituting this back shows Eq. (6.10) minimizes the variance of the projections $\mathbf{n}^\top \mathbf{u}_i$. Writing $\boldsymbol{\mu} = \frac{1}{N}\sum_i \mathbf{u}_i$ and the covariance-like matrix

$$\mathbf{C} = \sum_{i=1}^{N} (\mathbf{u}_i - \boldsymbol{\mu})(\mathbf{u}_i - \boldsymbol{\mu})^\top, \tag{6.11}$$

the minimizing $\mathbf{n}$ is the eigenvector of $\mathbf{C}$ associated with its smallest eigenvalue. This is equivalent to performing Principal Component Analysis (PCA) on the set of ray vectors and selecting the component of minimum variance [20, Ch. 3]. We then set

$$c = \frac{1}{N}\sum_{i=1}^{N} \mathbf{n}^\top \mathbf{u}_i. \tag{6.12}$$

For a valid configuration, $\mathbf{u}_i$ points approximately toward Jupiter, while $\mathbf{n}$ points from

Jupiter's center to the spacecraft; therefore c is expected to be negative. If the eigenvector sign yields $c > 0$, we flip $\mathbf{n} \leftarrow -\mathbf{n}$.

Finally, combining $c = \mathbf{n}^\top \mathbf{u}_i \approx -\cos\beta$ with the geometric relation $\sin\beta = R/\rho$ gives:

$$\rho = \frac{R}{\sqrt{1-c^2}}, \qquad \mathbf{r} = \rho\,\mathbf{n}. \tag{6.13}$$

Equation (6.13) yields the full Jupiter-centered position vector in inertial coordinates and follows directly from the cone geometry introduced in Fig. 6.1.

Optional ellipsoid refinement

If the spherical approximation is insufficient, Jupiter can be represented as an oblate spheroid (semi-major axis a, semi-minor axis b). In a Jupiter-centered frame aligned with the ellipsoid axes, the surface is described by

$$\mathbf{x}^\top \mathbf{Q}\,\mathbf{x} = 1, \qquad \mathbf{Q} = \mathrm{diag}\left(\frac{1}{a^2}, \frac{1}{a^2}, \frac{1}{b^2}\right). \tag{6.14}$$

Tangency of the line $\mathbf{x}(s) = \mathbf{r} + s\mathbf{u}$ to the quadric is again obtained by a zero-discriminant condition, yielding

$$(\mathbf{u}^\top \mathbf{Q}\mathbf{r})^2 = (\mathbf{u}^\top \mathbf{Q}\mathbf{u})\,(\mathbf{r}^\top \mathbf{Q}\mathbf{r} - 1). \tag{6.15}$$

A least-squares solve over Eq. (6.15) can be used to refine $\mathbf{r}$, using the spherical solution from Eq. (6.13) as initialization. In this work, the ellipsoidal model is employed only as a refinement step and not as a primary estimator.

6.1.2 Consistency checks

Tangency residuals

After estimating $\mathbf{r}$, the spherical tangency constraint is evaluated per limb ray using

$$e_i = (\mathbf{r}^\top \mathbf{u}_i)^2 - (\|\mathbf{r}\|^2 - R^2). \tag{6.16}$$

In the ideal noiseless model, $e_i = 0$. In practice, the distribution of $\{e_i\}$ provides an immediate diagnostic of:

- horizon detection quality (outliers from aurora, haze, or intensity gradient artifacts),
- residual camera modeling errors (e.g. imperfect distortion correction or intrinsics),
- attitude error from star mapping (rotation bias in $\mathbf{R}_{I \leftarrow C}$),
- physical limb offsets relative to the chosen reference radius R.

The spherical form of Eq. (6.16) is intentionally retained, as it yields a uniform and radius-independent consistency metric.

Cone projection dispersion

Because the cone model predicts $\mathbf{n}^\top \mathbf{u}_i \approx c$, the standard deviation

$$\sigma_c = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \left(\mathbf{n}^\top \mathbf{u}_i - c\right)^2} \tag{6.17}$$

should remain small for a consistent solution. This metric is independent of the assumed planetary radius and isolates purely angular consistency.

Inter-frame stability

The dataset contains image pairs separated by 500 ms. Applying the full pipeline to both frames of a pair should yield nearly identical position vectors $\mathbf{r}$, with deviations consistent with the spacecraft displacement over 0.5 s. Large discrepancies indicate that the detected horizon points are not sampling the same physical limb (e.g., contamination by the bright scattering boundary in Fig. 2.5), rather than variations in camera intrinsics, which are fixed by calibration.

6.1.3 Transforming to Jupiter-centric coordinates

The position vector $\mathbf{r}$ from Eq. (6.13) is expressed in a Jupiter-centered inertial frame aligned with J2000. For interpretation on the planet (latitude and longitude), or for comparison with body-fixed products, it is often desirable to express this vector in a Jupiter body-fixed frame.

Let $\mathbf{R}_{J\leftarrow I}(t)$ denote the rotation from inertial J2000 to the Jupiter-fixed frame at epoch t. The transformation is given by

$$\mathbf{r}_J(t) = \mathbf{R}_{J\leftarrow I}(t)\,\mathbf{r}_I. \tag{6.18}$$

This transformation is applied solely for interpretation and comparison; all estimation steps are performed in the inertial frame.

From $\mathbf{r}_J = [x\ y\ z]^\top$, the planetocentric longitude and latitude are obtained as

$$\lambda = \mathrm{atan2}(y, x), \qquad \varphi = \arctan 2\Big(z, \sqrt{x^2 + y^2}\Big), \tag{6.19}$$

with λ wrapped to $[0, 2\pi)$.

6.2 Atmospheric Investigation

6.2.1 Spatial resolution

The first step before any atmospheric analysis is to convert pixels into physical kilometers. This scaling factor, or spatial resolution S, is determined by the specific viewing geometry and intrinsic camera parameters. Using the principle of similar triangles (see sketch 6.2),

$$\frac{D}{f} = \frac{S}{p} \tag{6.20}$$

Where:

- D is the slant range to the limb provided by SPICE kernels ($60,621$ km for this observation).
- p is the width of a pixel (8.6 μm).
- f is the focal length ($20,006$ μm).

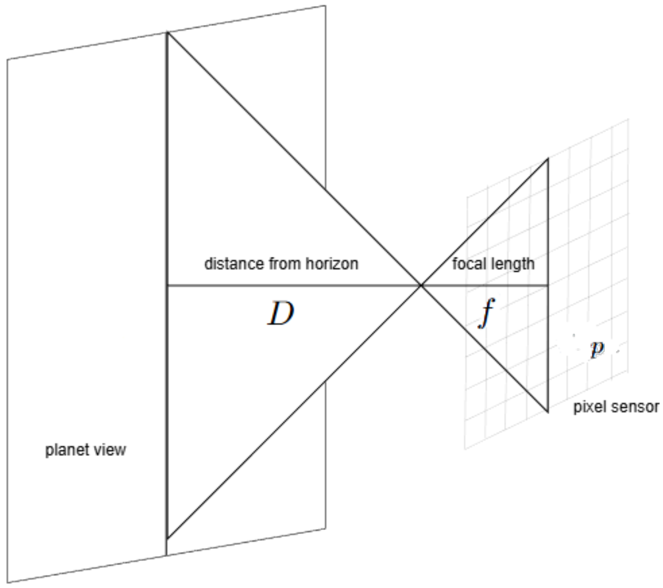


Figure 6.2: Schematic illustrating the projection of sensor pixel onto the target plane.

Substituting the value gives :

$$S = \frac{D \cdot p}{f} = \frac{60621\text{km} \cdot 8.6\mu\text{m}}{20006\mu\text{m}} = 26.06\text{km/px}$$

It is important to note that this value is valid strictly at the limb tangent point. For any features projected below the horizon (on the planetary disk), the distance D decreases slightly, resulting in a finer spatial resolution (smaller km/px value).

6.2.2 Validation of thermosphere detection

The atmospheric region extending above the limb is identified as the thermosphere, the upper boundary layer that separates Jupiter's envelope from the vacuum. This layer begins approximately 350 km above the 1-bar level [21]. Unlike the underlying troposphere, which is governed by solar heating and convection, the thermosphere is defined by a dramatic rise in temperature and intense interaction with the Jovian magnetosphere.

The defining characteristic of this layer is its immense extent, driven by temperatures that rise dramatically with altitude, ranging from 160 K to 900 K. This extreme thermal expansion is fueled by high-energy inputs: the layer absorbs solar ultraviolet radiation and is bombarded by precipitating charged particles from the magnetosphere. This particle precipitation is the primary driver of Jupiter's powerful auroral emissions. The detection of these auroral signatures in the data confirms that the observed limb profile corresponds to this active upper-atmospheric layer.

Scale Height Correlation

A critical parameter for characterizing this atmospheric layer is the scale height (H), which represents the vertical distance over which the atmospheric density decreases by a factor of e. In a hydrostatic atmosphere, it is defined as

$$H = \frac{kT}{mg} \tag{6.21}$$

where k is the Boltzmann constant, T is the kinetic temperature, m is the mean molecular mass, and g is the local gravitational acceleration. Due to very high temperatures, the atmosphere expands significantly, resulting in scale heights that are orders of magnitude larger than those found in the cloud decks below ($H \approx 27$ km at 1-bar level).

To analyze the intensity profile, radial line scans were performed on the cleaned data (cf. 6.3). As discussed in Chapter 4, the horizon is derived from SPICE kernels, eliminating approximation errors. These line scans are averaged and fit to an exponential curve (cf. 6.4) of the following form :

$$I(r) = I_0 e^{-r/H} \tag{6.22}$$

The fitting provides an estimated scale height H of 4.8 pixels, translating to $\sim$ 125 km using the spatial resolution derived earlier.

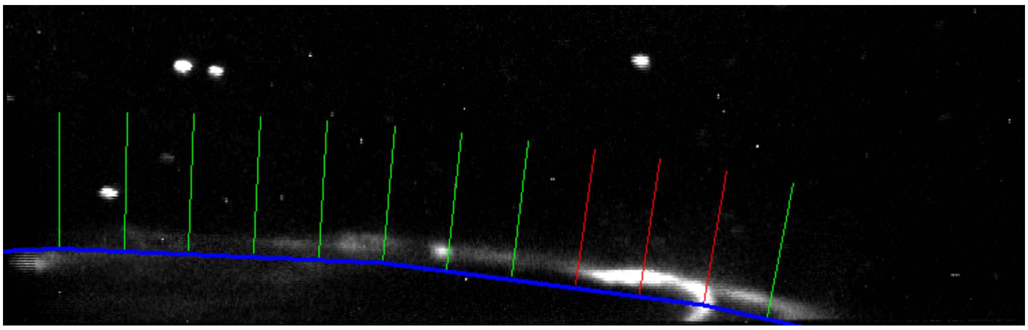


Figure 6.3: Radial line scans of Jovian atmosphere. Red lines are omitted because aurora interferes with the signal. Image gained x6.

This value correlates strongly with theoretical predictions, driven primarily by the thermal structure of the layer. As defined in equation 6.21, the scale height is directly proportional to the temperature ($H \propto T$). While the troposphere is cold (130 K), Yelle & Miller [21] describe the thermosphere as a region of extreme heating, where temperatures rise to between 800 K and 1000 K. Consequently, the scale height is effectively multiplied by a factor of roughly 5 to 7 relative to the lower atmosphere, where the scale height is 27 km. The derived value of 125 km is consistent with these high temperatures, confirming that the exponential decay detected in the imagery corresponds to the hot, expanded thermosphere.

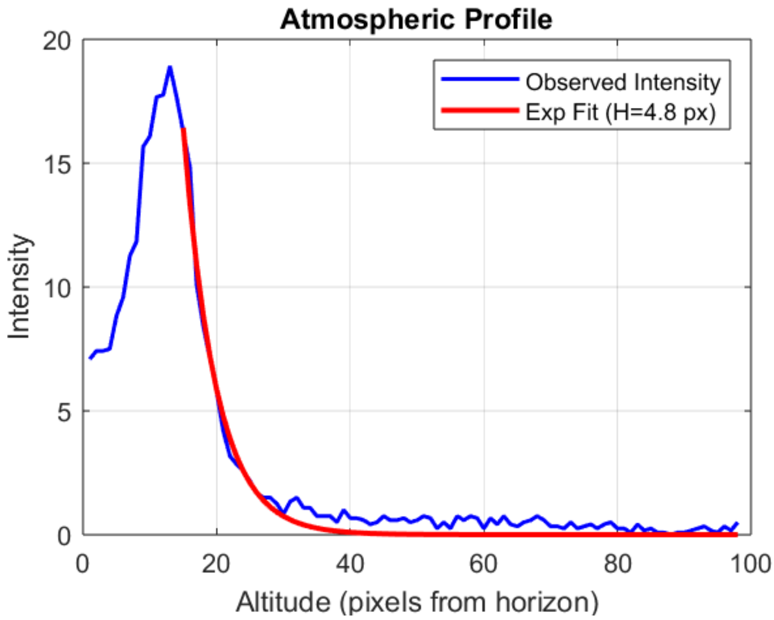


Figure 6.4: Averaged radial line scan starting from 1 bar horizon

Chapter 7

Discussion and Conclusions

7.1 Discussion

This study investigated whether the Juno ASC, originally designed as a navigation star tracker, can be repurposed into a scientifically useful imager for atmospheric inspection and as a supporting sensor for optical navigation. The results show that meaningful information can be extracted, but only when a dedicated processing chain (briefly summarized by Figure 7.1) designed for sensor bias, radiation contamination, and rotation-induced smear is applied.

7.1.1 What works best for this Juno dataset

Radiometric calibration and structured-bias suppression Dark-frame subtraction combined with optional column-bias removal removes sensor-induced background bias. In this dataset, the large fraction of deep-space pixels makes a column-median estimator effective for suppressing column-correlated residuals.

Radiation mitigation Radiation hits appear as compact, high-gradient impulses that must be removed without attenuating the star point-spread function or the faint limb gradient. Global smoothing reduces speckles but degrades both centroiding and edge localization. The gradient-based switching median filter provides the most suitable trade-off by selectively suppressing impulse artifacts while preserving broader structures. The effect of this step is illustrated by the reduction of radiation speckles between the intermediate frames in Fig. 7.1.

Robust star mapping for inertial attitude recovery Despite the limited number of detectable background stars and strong intensity variation across the frame, the star mapping pipeline succesfully recovers the camera-to-inertial rotation. A hybrid detection strategy increases the usable star set, while the robust matching stage resolves catalogue ambiguities by validating hypotheses against many additional detections (RANSAC inliers). After a consistent correspondence set is found, Wahba refinement directly provides the spacecraft orientation in inertial space.

Reconstruction through despinning and lens correction The 125 ms delay between odd and even readout fields corresponds to a measurable angular rotation, making raw frames inconsistent with a single pinhole camera model. Rotation-compensated despinning restores a coherent capture. While lens distortion is visually subtle, it becomes relevant when pixel-level errors are converted to physical distance at the limb.

Horizon estimation on short arcs Because the limb occupies only a short arc in a 16° field of view, standard algebraic least-squares circle fitting (Kåsa) becomes strongly biased. The Hyperaccurate fit is more stable for this regime and yields a radius estimate that is consistent with SPICE to within the reported residual level. This improves geometric extraction from the image, but does not remove the systematic mismatch that originates from the horizon definition.

7.1.2 Limitations and sources of error

Figure 7.1 provides an overview of the full processing pipeline: from the raw image (1) through despinning (2) and denoising (3) to an image suitable for feature extraction. The final outputs illustrate two parallel branches, star detection and catalogue-based attitude annotation (4) and horizon extraction via limb detection and geometric overlay (5).

Horizon detection compared to SPICE reference The dominant discrepancy in the horizon comparison is physical rather than numerical. The ASC detects a radiometric transition shaped by scattering and haze at altitudes above the 1-bar level used by SPICE. Consequently, horizon-based optical navigation inherits a systematic range bias unless an atmospheric offset model (or equivalent correction) is introduced.

Dataset scope Conclusions are based on a limited observation set, restricting validation across illumination conditions radiation variability.

Star detection and matching limitations Star mapping performance is constrained by the photon-limited regime and the presence of strong stray light gradients near the Jovian limb. Faint stars near the noise floor can be missed or fragmented into spurious detections, while radiation residuals can introduce false candidates. Although the robust solver mitigates these effects through inlier screening, frames with very low star counts or heavy contamination reduce attitude confidence and can increase sensitivity to centroiding and catalog matching errors.

Onboard infeasibility Full-frame calibration, filtering, and dense remapping exceed the μASC processing constraints at 4 Hz, confining the complete reconstruction to ground-based processing.

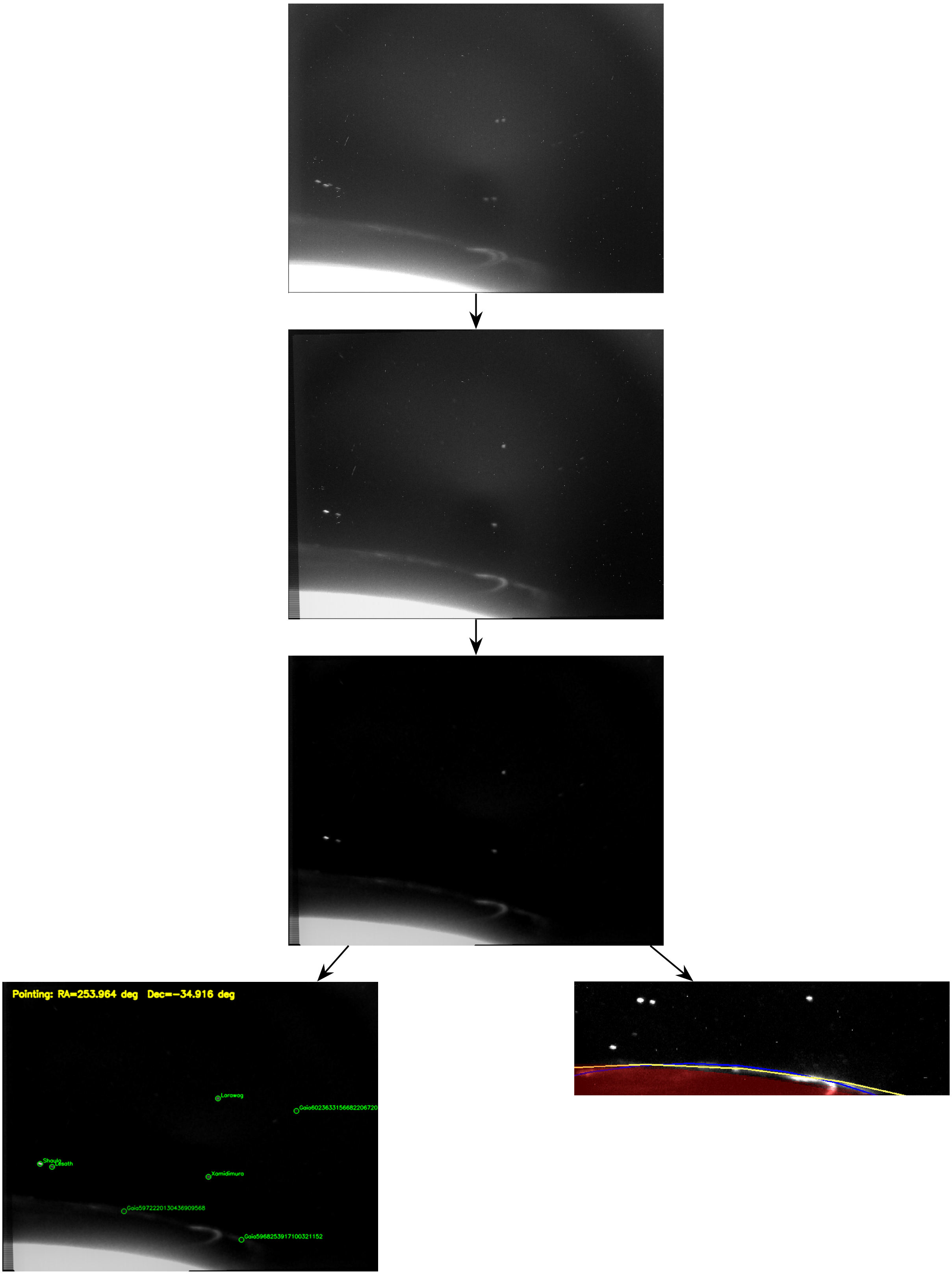


Figure 7.1: From raw image to star detection and horizon extraction.

7.2 Conclusions

7.2.1 Main findings

- A tailored image processing sequence can convert raw ASC frames into scientifically interpretable imagery.
- The gradient-based switching median filter preserves both images features more effectively than global smoothing in this dataset.
- Hyperaccurate circle fitting provides a good estimate of the horizon line, but optical navigation accuracy is inherently limited by the data itself without additional modeling.
- Robust star mapping (hybrid detection + RANSAC/Wahba refinement) yields a stable camera-to-inertial attitude estimate from sparse star fields.

7.2.2 Recommendations for future work

1. Parameterize the atmospheric offset. Introduce $R_{eff} = R + \Delta h$ or an intensity-based limb model so that the radiometric-to-geometric discrepancy becomes an estimable correction rather than a persistent bias.
2. Estimate the feasibility of removing heavy processing steps and still get a god accuracy on attitude determination for onboard attitude and, potentially, position determination.

Overall, the ASC images can be repurposed into a useful scientific and geometric dataset, but the limiting factors are dominated by the little amount of available data, sensor architecture, compute constraints (for the onboard attitude determination), and challenges in interpreting the observed limb.

Bibliography

[1] National Aeronautics and Space Administration. NASA's Juno Mission to Remain in Current Orbit at Jupiter, February 2017. News release (Release 17-018), Feb 17, 2017.

[2] NASA Science. Juno, 2025. Mission overview page (page last updated Sep 03, 2025).

[3] NASA Jet Propulsion Laboratory. Juno, 2025. JPL mission overview page.

[4] Space Instrumentation Group. Advanced Stellar Compass JUNO Software Interface Specification. Software Interface Specification JN-DTU-SP-3001, National Space Institute, Technical University of Denmark, Kongens Lyngby, Denmark, 2024. Issue 1.6.

[5] Matija Herceg, P. Jørgensen, J. Jørgensen, and J. Connerney. Thermoelastic response of the juno spacecraft's solar array/magnetometer boom and its applicability to improved magnetic field investigation. *Earth and Space Science*, 7, 11 2020.

[6] Troelz Denver, Julia Sushkova, John L. Jørgensen, Leonardo Ghizoni, Matija Herceg, Christina Toldbo, Mathias Benn, Peter S. Jørgensen, René Fléron, J. E. P. Connerney, Heidi N. Becker, and Scott J. Bolton. The Juno ASC as an energetic particle counter. *Space Science Reviews*, 220(8):86, 2024.

[7] Navigation and Ancillary Information Facility. Time Required Reading. `https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/time.html`. Accessed: 2025-12-10.

[8] Pieter G. van Dokkum. Cosmic-ray rejection by laplacian edge detection. *Publications of the Astronomical Society of the Pacific*, 113(789):1420, nov 2001.

[9] Olgierd Stankiewicz, Gauthier Lafruit, and Marek Domański. Chapter 1 - multiview video: Acquisition, processing, compression, and virtual view rendering. In Rama Chellappa and Sergios Theodoridis, editors, *Academic Press Library in Signal Processing, Volume 6*, pages 32–33. Academic Press, 2018.

[10] Carlo Tomasi. Vector representation of rotations. Duke University, Computer Science 527 Lecture Notes, 2013. Accessed: 2025-11-20.

[11] Vision Doctor. Optical errors - distortion, n.d.

[12] A. Al-Sharadqah and N. Chernov. Error analysis for circle fitting algorithms. *Electronic Journal of Statistics*, 3:886–911, 2009.

[13] European Space Agency. Hipparcos catalogues. `https://www.cosmos.esa.int/web/hipparcos/catalogues`, 1997. ESA SP-1200.

[14] European Space Agency. Gaia data release 3 (dr3) catalogue. `https://www.cosmos.esa.int/web/gaia/dr3`, 2022. Gaia DR3.

[15] Daniele Mortari, Malak A. Samaan, Christian Bruccoleri, and John L. Junkins. The "pyramid" star pattern recognition algorithm. *Navigation*, 51(3):171–183, 2004.

[16] Malcolm D Shuster and SD Oh. Three-axis attitude determination from vector observations. *Journal of Guidance and Control*, 4(1):70–77, 1981.

[17] Grace Wahba. A least squares estimate of satellite attitude. *SIAM review*, 7(3):409–409, 1965.

[18] James R Wertz. *Spacecraft attitude determination and control*, volume 73. Springer Science & Business Media, 1978.

[19] Aurelie Bellemans, Tim De Troyer, Mark Runacres, and Chris Lacor. *Basic Techniques for Computer Simulations.* Vrije Universiteit Brussel, 2023. Lecture notes, Faculty of Engineering.

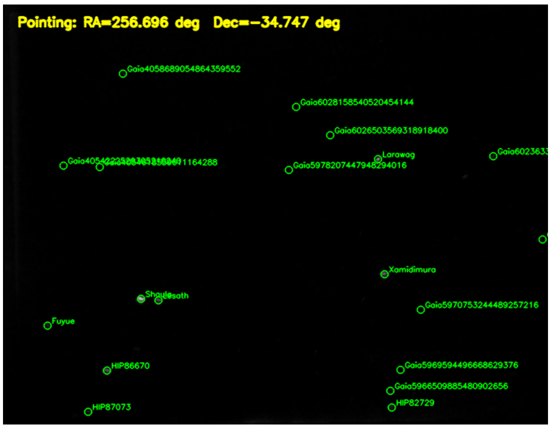
[20] Tue Herlau, Mikkel N. Schmidt, and Morten Mørup. *Introduction to Machine Learning and Data Mining*. Technical University of Denmark, 2023. Lecture notes, Fall 2023, version 1.0.

[21] Roger V. Yelle and Steve Miller. Jupiter's thermosphere and ionosphere. In Fran Bagenal, Timothy E. Dowling, and William B. McKinnon, editors, *Jupiter: The Planet, Satellites and Magnetosphere*, chapter 9, pages 185–218. Cambridge University Press, Cambridge, UK, 2004.

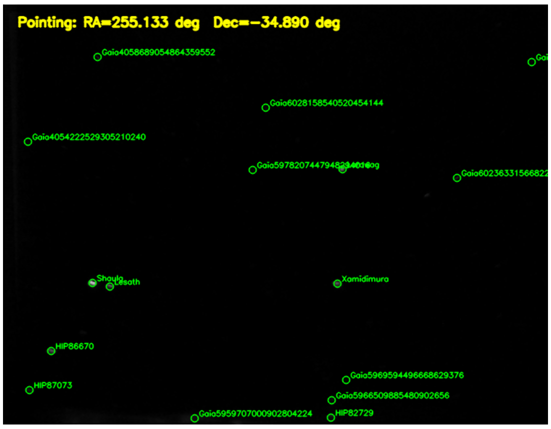
Appendix A

Additional Material

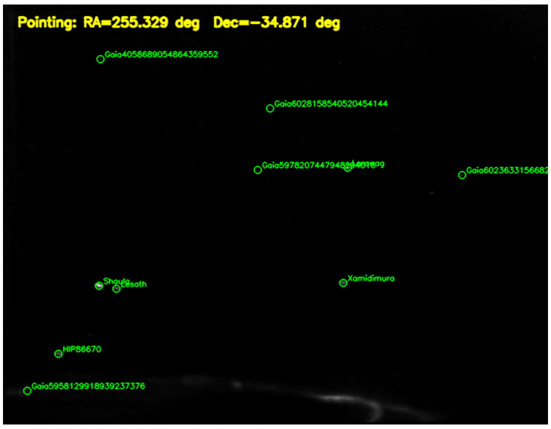
A.1 Star recognition output from whole dataset



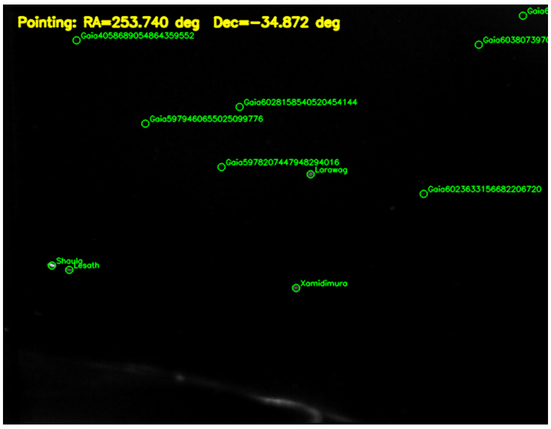
(a) Frame 1



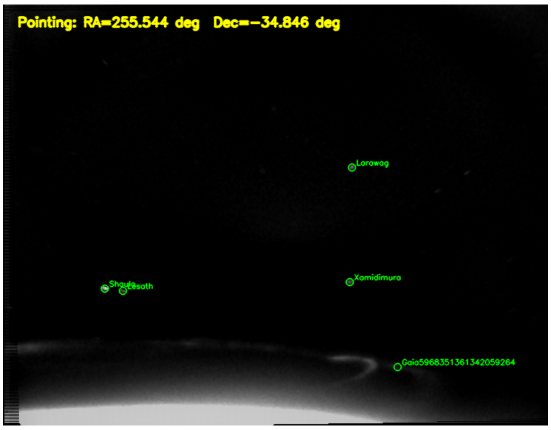
(b) Frame 2



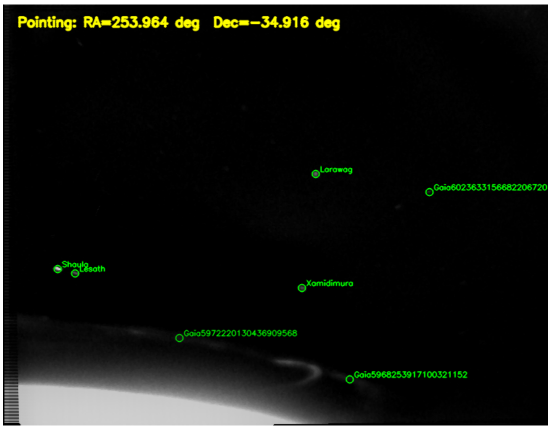
(c) Frame 3



(d) Frame 4



(e) Frame 5



(f) Frame 6

Figure A.1: Star-tracking results for six representative ASC frames.

A.2 Source Code

Note: The source code presented in this appendix was assembled, refactored, and cleaned with the assistance of AI-based tools to improve code structure, readability, and consistency across different scripts, while preserving the original algorithms and logic designed by the authors.

A.2.1 Main script

```
#!/usr/bin/env python3
"""
TotalCode: unified JUNO pipeline with all toggles + star tracker in
    one file.

Includes:
- Despin (per-pixel rotation)
- Lens distortion correction
- Denoising (dark frame, flatten, spike, median/Gaussian)
- Column-median glow removal
- Fourier notch filtering
- Edge tracking, circle fits, limb profiles
- Star tracker (catalog loading, detection, attitude, annotation)

Use the feature toggles below to enable/disable each stage.
"""

from __future__ import annotations

from dataclasses import dataclass, field
from pathlib import Path
from typing import Dict, Iterable, List, Optional, Sequence, Tuple

import cv2
import numpy as np
```

```
from PIL import Image
import star_tracker

# --- Feature toggles ---------------------------------------------------------

APPLY_DESPIN = True
APPLY_LENS = False
APPLY_DENOISE = True
APPLY_FFT_NOTCH = False
APPLY_EDGE_TRACKING = True
APPLY_CIRCLE_FITS = True
APPLY_LIMB_PROFILES = True
APPLY_RESIDUAL_GLOW = False  # column median subtract
APPLY_STAR_TRACKER = True
STAR_USE_DESPIN_MASK = False  # if True, pass the despin overlap mask into star tracker detection

# --- Paths -------------------------------------------------------------------

SCIENCE_DIR = Path("/Users/vojtadeconinck/Downloads/ImageAnalysis30300/JUNO␣Despin␣Images") / "ScienceCalibrated"
OUTPUT_DIR = Path("/Users/vojtadeconinck/Downloads/ImageAnalysis30300/JUNO␣Despin␣Images") / "Combined"
MASK_OUTPUT_DIR = OUTPUT_DIR / "Masks"
STAR_CATALOG_PATH = Path("/Users/vojtadeconinck/Downloads/ImageAnalysis30300/combined_stars.csv")

# --- Camera / geometry constants --------------------------------------------

```

```
OMEGA = np.array([-8.54302424e-05, -2.26443686e-04, -2.08965026e
    -01], dtype=np.float64)
OMEGA_FRAME = "spacecraft"  # "spacecraft" or "camera"
DELTA_T = 0.13  # seconds between odd and even fields

SC_TO_CHUD_MATRIX = np.array([
    [0.99995302, 0.00569167, 0.00784704],
    [0.00383956, -0.97582589, 0.21851563],
    [0.00890106, -0.21847523, -0.9758019],
])

EFL_MICRONS = 20_006.0
PIXEL_SIZE_X_MICRONS = 8.6
PIXEL_SIZE_Y_MICRONS = 8.3
PRINCIPAL_POINT = (383.0, 257.0)  # (cx, cy)
SENSOR_SHAPE = (580, 752)  # (H, W)

# Lens distortion
LENS_KAPPA = 3.3e-8

# --- Denoise settings
    --------------------------------------------------------


@dataclass
class FlattenConfig:
    mask_percentile: float = 95.0
    dilate_kernel: int = 11
    sigma: float = 40.0
    column_correction: bool = True
    column_mask_percentile: float = 99.0
    column_dilate: int = 5


```

```
@dataclass
class DenoiseConfig:
    apply_dark_frame: bool = False
    dark_frame_path: str | None = None
    dark_frame: np.ndarray | None = field(default=None, repr=False)

    apply_column_median: bool = False

    apply_flatten: bool = True
    flatten: FlattenConfig = field(default_factory=FlattenConfig)

    spike_mode: str = "gradient"  # "mean" or "gradient"
    spike_neighborhood: int = 1
    spike_threshold: float = 45.0
    gradient_threshold: float = 45.0

    median_kernel: int = 3
    gaussian_kernel: int = 0
    gaussian_sigma: float = 0.0


DENOISE_SETTINGS = DenoiseConfig(
    apply_dark_frame=False,
    dark_frame_path=str(Path("/Users/vojtadeconinck/Downloads/
        ImageAnalysis30300/BlackRefImages/master_black.png")),
    apply_column_median=False,
    apply_flatten=True,
    flatten=FlattenConfig(),
    spike_mode="gradient",
    spike_neighborhood=1,
    spike_threshold=45.0,
    gradient_threshold=45.0,
    median_kernel=3,
    gaussian_kernel=0,
```

```
    gaussian_sigma=0.0,
)

# --- Fourier notch default (legacy manual list) -----------------------------

MANUAL_NOTCHES = [
    (75, 0),
    (115, 0),
    (155, 0),
    (0, 40),
    (0, 80),
]
NOTCH_RADIUS = 25.0

# --- Basic helpers ------------------------------------------------------------


def _load_grayscale(path: Path) -> np.ndarray:
    arr = np.array(Image.open(path))
    if arr.ndim != 2:
        raise ValueError(f"{path} is not a single-channel image.")
    return arr


def _save_grayscale(data: np.ndarray, path: Path, dtype: np.dtype) -> None:
    info = np.iinfo(dtype)
    clipped = np.clip(np.rint(data), info.min, info.max).astype(dtype)
    path.parent.mkdir(parents=True, exist_ok=True)
    Image.fromarray(clipped).save(path)

```

```python

def _to_8bit(arr: np.ndarray, dtype: np.dtype) -> np.ndarray:
    info = np.iinfo(dtype)
    scaled = np.clip(arr, info.min, info.max) / float(info.max)
    scaled *= 255.0
    return np.clip(np.rint(scaled), 0, 255).astype(np.uint8)


# --- Lens correction ---------------------------------------------------------

def _build_lens_maps(kappa: float) -> tuple[np.ndarray, np.ndarray]:
    H, W = SENSOR_SHAPE
    cx, cy = PRINCIPAL_POINT
    pix_ratio = PIXEL_SIZE_Y_MICRONS / PIXEL_SIZE_X_MICRONS

    x = np.arange(W, dtype=np.float32) - cx
    y = (np.arange(H, dtype=np.float32)[:, None] - cy) * pix_ratio
    r2 = x[None, :] * x[None, :] + y * y
    S = 1.0 + kappa * r2

    map_x = (x[None, :] * S + cx).astype(np.float32)
    map_y = ((y * S) / pix_ratio + cy).astype(np.float32)
    return map_x, map_y


def _apply_lens_correction(
    img: np.ndarray,
    maps: tuple[np.ndarray, np.ndarray],
    *,
    interpolation: int = cv2.INTER_LINEAR,
    border_value: float = 0.0,
) -> np.ndarray:
```

```
    map_x, map_y = maps
    corrected = cv2.remap(
        img.astype(np.float32),
        map_x,
        map_y,
        interpolation=interpolation,
        borderMode=cv2.BORDER_CONSTANT,
        borderValue=border_value,
    )
    return corrected


# --- Denoising
    -----------------------------------------------------------------

def _as_float(gray: np.ndarray) -> np.ndarray:
    return gray.astype(np.float32, copy=False) if gray.dtype != np.
        float32 else gray


def _restore_dtype(clean: np.ndarray, dtype) -> np.ndarray:
    if np.issubdtype(dtype, np.floating):
        return clean.astype(dtype)
    info = np.iinfo(dtype)
    clipped = np.clip(np.rint(clean), info.min, info.max)
    return clipped.astype(dtype)


def _load_dark_frame(path: str) -> np.ndarray:
    arr = cv2.imread(path, cv2.IMREAD_UNCHANGED)
    if arr is None:
        raise FileNotFoundError(f"Dark frame not found: {path}")
    if arr.ndim != 2:
```

```
        raise ValueError(f"Dark frame must be single-channel: {path}")
    return arr


def subtract_dark_frame(gray: np.ndarray, dark: np.ndarray) -> np.ndarray:
    if gray.shape != dark.shape:
        raise ValueError(f"Dark frame shape {dark.shape} does not match image {gray.shape}")
    return _as_float(gray) - _as_float(dark)


def _build_mask(arr: np.ndarray, percentile: float, dilate_kernel: int) -> np.ndarray:
    thr = np.percentile(arr, percentile)
    mask = arr > thr
    k = max(1, int(dilate_kernel))
    if k > 1:
        kernel = np.ones((k, k), np.uint8)
        mask = cv2.dilate(mask.astype(np.uint8), kernel, iterations=1).astype(bool)
    return mask


def _column_correction(flat: np.ndarray, config: FlattenConfig) -> np.ndarray:
    work = flat.copy()
    mask = _build_mask(work, config.column_mask_percentile, config.column_dilate)
    H, W = work.shape
    for x in range(W):
        col = work[:, x]
        good = ~mask[:, x]
```

```python
        if not np.any(good):
            continue
        offset = np.median(col[good])
        work[:, x] -= offset
    return work


def column_median_subtract(gray: np.ndarray) -> np.ndarray:
    f = _as_float(gray)
    med = np.median(f, axis=0, keepdims=True)
    return f - med


def flatten_background(gray: np.ndarray, config: FlattenConfig | None = None) -> np.ndarray:
    if gray.ndim != 2:
        raise ValueError("flatten_background expects a single-channel image.")
    cfg = config or FlattenConfig()
    f = _as_float(gray)

    mask = _build_mask(f, cfg.mask_percentile, cfg.dilate_kernel)
    bg_level = np.median(f[~mask]) if np.any(~mask) else float(np.median(f))

    f_bgfit = f.copy()
    f_bgfit[mask] = bg_level

    bg = cv2.GaussianBlur(f_bgfit, (0, 0), sigmaX=cfg.sigma, sigmaY=cfg.sigma)
    f_flat = f - bg

    if cfg.column_correction:
        f_flat = _column_correction(f_flat, cfg)
```

```python
    return f_flat


def remove_radiation_spikes(gray: np.ndarray, neighborhood: int, threshold: float) -> np.ndarray:
    if gray.ndim != 2:
        raise ValueError("remove_radiation_spikes expects a single-channel image.")
    n = max(1, int(neighborhood))
    k = 2 * n + 1
    f = _as_float(gray)
    kernel = np.ones((k, k), dtype=np.float32)
    summed = cv2.filter2D(f, -1, kernel, borderType=cv2.BORDER_REFLECT)
    neighbor_sum = summed - f
    neighbor_count = k * k - 1
    neighbor_mean = neighbor_sum / float(neighbor_count)
    mask = (f - neighbor_mean) > threshold
    cleaned = f.copy()
    cleaned[mask] = neighbor_mean[mask]
    return cleaned


def remove_gradient_spikes(gray: np.ndarray, threshold: float) -> np.ndarray:
    if gray.ndim != 2:
        raise ValueError("remove_gradient_spikes expects a single-channel image.")
    f = _as_float(gray)
    out = f.copy()

    center = f[1:-1, 1:-1]
    left = f[1:-1, :-2]
    right = f[1:-1, 2:]
```

```
    up = f[:-2, 1:-1]
    down = f[2:, 1:-1]

    horiz_peak = (center - left > threshold) & (center - right > threshold)
    vert_peak = (center - up > threshold) & (center - down > threshold)
    mask = horiz_peak & vert_peak

    if np.any(mask):
        neighbors = np.stack([left, right, up, down], axis=0)
        med = np.median(neighbors, axis=0)
        out_view = out[1:-1, 1:-1]
        out_view[mask] = med[mask]
    return out


def clean_image(gray: np.ndarray, config: DenoiseConfig | None = None) -> np.ndarray:
    if gray.ndim != 2:
        raise ValueError("clean_image expects a single-channel image.")
    cfg = config or DenoiseConfig()
    work = _as_float(gray)

    if cfg.apply_dark_frame:
        dark = cfg.dark_frame
        if dark is None and cfg.dark_frame_path:
            dark = _load_dark_frame(cfg.dark_frame_path)
            cfg.dark_frame = dark
        if dark is not None:
            work = subtract_dark_frame(work, dark)

    if cfg.apply_column_median:
```

```
        work = column_median_subtract(work)

    if cfg.apply_flatten:
        work = flatten_background(work, cfg.flatten)

    if cfg.spike_mode == "gradient":
        work = remove_gradient_spikes(work, cfg.gradient_threshold)
    else:
        work = remove_radiation_spikes(work, cfg.spike_neighborhood, cfg.spike_threshold)

    if cfg.median_kernel and cfg.median_kernel >= 3:
        k = cfg.median_kernel if cfg.median_kernel % 2 == 1 else cfg.median_kernel + 1
        work = cv2.medianBlur(work.astype(np.float32), k)

    if cfg.gaussian_kernel and cfg.gaussian_kernel > 0:
        k = cfg.gaussian_kernel if cfg.gaussian_kernel % 2 == 1 else cfg.gaussian_kernel + 1
        work = cv2.GaussianBlur(work, (k, k), cfg.gaussian_sigma)

    return _restore_dtype(work, gray.dtype)


# --- Despin (per-pixel rotation) --------------------------------------------

def rodrigues(omega: np.ndarray, delta_t: float) -> np.ndarray:
    theta = float(np.linalg.norm(omega) * delta_t)
    if theta == 0.0:
        return np.eye(3, dtype=np.float64)
    k = omega / np.linalg.norm(omega)
    kx, ky, kz = k
    K = np.array(
```

```
        [
            [0.0, -kz, ky],
            [kz, 0.0, -kx],
            [-ky, kx, 0.0],
        ],
        dtype=np.float64,
    )
    sin_t = np.sin(theta)
    cos_t = np.cos(theta)
    return np.eye(3) + sin_t * K + (1.0 - cos_t) * (K @ K)


def _pixel_rays(x_full: np.ndarray, y_full: np.ndarray) -> np.ndarray:
    cx, cy = PRINCIPAL_POINT
    x_cam = (x_full - cx) * (PIXEL_SIZE_X_MICRONS / EFL_MICRONS)
    y_cam = (y_full - cy) * (PIXEL_SIZE_Y_MICRONS / EFL_MICRONS)
    rays = np.stack([x_cam, y_cam, np.ones_like(x_cam)], axis=-1)
    return rays


def _project_to_full_pixels(rays: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    cx, cy = PRINCIPAL_POINT
    x = rays[..., 0] / rays[..., 2]
    y = rays[..., 1] / rays[..., 2]
    u = x * (EFL_MICRONS / PIXEL_SIZE_X_MICRONS) + cx
    v = y * (EFL_MICRONS / PIXEL_SIZE_Y_MICRONS) + cy
    return u, v


def _warp_even_field_to_reference(
    even_img: np.ndarray,
    output_rows_full: np.ndarray,
```

```
    R_odd_to_even: np.ndarray,
) -> tuple[np.ndarray, np.ndarray]:
    H_full, W = SENSOR_SHAPE
    assert even_img.shape[0] == H_full // 2

    x_full = np.broadcast_to(np.arange(W, dtype=np.float64), (
        output_rows_full.size, W))
    y_full = np.broadcast_to(output_rows_full[:, None].astype(np.
        float64), (output_rows_full.size, W))

    rays_odd = _pixel_rays(x_full, y_full).reshape(-1, 3)
    rays_even = (R_odd_to_even @ rays_odd.T).T

    u_full, v_full = _project_to_full_pixels(rays_even)
    v_even_idx = (v_full - 1.0) * 0.5

    x = u_full.ravel()
    y = v_even_idx.ravel()
    H_even, W_even = even_img.shape

    valid = (x >= 0) & (x <= W_even - 1) & (y >= 0) & (y <= H_even
        - 1)

    x0 = np.clip(np.floor(x).astype(np.int64), 0, W_even - 1)
    y0 = np.clip(np.floor(y).astype(np.int64), 0, H_even - 1)
    x1 = np.clip(x0 + 1, 0, W_even - 1)
    y1 = np.clip(y0 + 1, 0, H_even - 1)

    wx = x - x0
    wy = y - y0

    Ia = even_img[y0, x0]
    Ib = even_img[y0, x1]
    Ic = even_img[y1, x0]
```

```
    Id = even_img[y1, x1]

    sampled = (
        (1 - wx) * (1 - wy) * Ia
        + wx * (1 - wy) * Ib
        + (1 - wx) * wy * Ic
        + wx * wy * Id
    )
    sampled[~valid] = 0.0
    valid_mask = valid.reshape(output_rows_full.size, W).astype(np.
        uint8)
    return sampled.reshape(output_rows_full.size, W).astype(np.
        float32), valid_mask


def recombine_with_rotation(img: np.ndarray, omega: np.ndarray,
    delta_t: float) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    H, W = img.shape
    if (H, W) != SENSOR_SHAPE:
        raise ValueError(f"Unexpected image shape {img.shape};
            expected {SENSOR_SHAPE}")

    odd = img[0::2, :]
    even = img[1::2, :]

    R_odd_to_even = rodrigues(omega, delta_t)

    odd_rows_full = np.arange(0, H, 2)
    even_rows_full = np.arange(1, H, 2)

    aligned_even_on_odd, _ = _warp_even_field_to_reference(even,
        odd_rows_full, R_odd_to_even)
    aligned_even_on_even, mask_on_even =
        _warp_even_field_to_reference(even, even_rows_full,
```

```
        R_odd_to_even)

    recombined = np.empty_like(img, dtype=np.float32)
    recombined[0::2, :] = odd
    recombined[1::2, :] = aligned_even_on_even

    mask_full = np.empty_like(img, dtype=np.uint8)
    mask_full[0::2, :] = 1
    mask_full[1::2, :] = mask_on_even
    return recombined, mask_full, R_odd_to_even


# --- FFT / notch filtering
    ---------------------------------------------------

def fft_shift(mag: np.ndarray) -> np.ndarray:
    return np.fft.fftshift(mag, axes=(0, 1))


def make_symmetric_notches(manual: Sequence[Tuple[int, int]], width
    : int, height: int) -> list[Tuple[int, int]]:
    out: list[Tuple[int, int]] = []
    for x, y in manual:
        out.append((x, y))
        out.append(((width - x) % width, (height - y) % height))
    return out


def apply_gaussian_notch_filter(complex_fft: np.ndarray,
    notch_centers: Sequence[Tuple[int, int]], radius: float) -> None
    :
    rows, cols = complex_fft.shape[:2]
    sigma = radius / 2.0
    two_sigma2 = 2.0 * sigma * sigma
```

```
    yy, xx = np.mgrid[0:rows, 0:cols]
    H = np.ones((rows, cols), dtype=np.float32)
    for cx, cy in notch_centers:
        d2 = (xx - cx) ** 2 + (yy - cy) ** 2
        notch = 1.0 - np.exp(-d2 / two_sigma2)
        H *= notch.astype(np.float32)
    complex_fft[..., 0] *= H
    complex_fft[..., 1] *= H


def fourier_notch_reconstruct(img: np.ndarray, manual_notches:
    Sequence[Tuple[int, int]], radius: float) -> np.ndarray:
    padded = cv2.copyMakeBorder(
        img,
        top=0,
        bottom=cv2.getOptimalDFTSize(img.shape[0]) - img.shape[0],
        left=0,
        right=cv2.getOptimalDFTSize(img.shape[1]) - img.shape[1],
        borderType=cv2.BORDER_CONSTANT,
        value=0,
    )
    planes = [padded.astype(np.float32), np.zeros_like(padded,
        dtype=np.float32)]
    complex_i = cv2.merge(planes)
    cv2.dft(complex_i, complex_i)
    complex_i = fft_shift(complex_i)

    h, w = complex_i.shape[:2]
    notches = make_symmetric_notches(manual_notches, w, h)
    apply_gaussian_notch_filter(complex_i, notches, radius)

    complex_i = fft_shift(complex_i)
    reconstructed = cv2.idft(complex_i, flags=cv2.DFT_REAL_OUTPUT |
        cv2.DFT_SCALE)
```

```
    return reconstructed.astype(np.float32)


# --- Edge tracking + circles
    ------------------------------------------------

def _column_gradients(gray: np.ndarray, mask: np.ndarray | None =
    None) -> np.ndarray:
    """Return |vertical gradient| per column; masked values set to
        0."""
    col_diff = np.abs(np.diff(gray.astype(np.int32), axis=0))
    if mask is not None:
        m = mask[1:, :] & mask[:-1, :]
        col_diff = np.where(m, col_diff, 0)
    return col_diff


def _find_strongest_edge_in_column(gray: np.ndarray, col: int) ->
    int:
    grad = np.abs(np.diff(gray[:, col].astype(np.int32)))
    if grad.size == 0:
        return -1
    idx = int(np.argmax(grad))
    return idx if grad[idx] > 0 else -1


def _compute_column_threshold(gray: np.ndarray, col: int, k: float
    = 1.0) -> int:
    grads = np.abs(np.diff(gray[:, col].astype(np.int32)))
    mean = grads.mean() if grads.size else 0.0
    std = grads.std() if grads.size else 0.0
    return int(round(mean + k * std))
```

```
def track_edge_andrey(gray: np.ndarray, start_col: int = 0, k: float = 3.0, searchrange: int = 3) -> list[Tuple[int, int]]:
    """
    Original C++-style tracker: per column strongest local gradient vs adaptive threshold,
    zero-order hold if below threshold.
    """
    edges: list[Tuple[int, int]] = []
    current_col = start_col
    current_row = _find_strongest_edge_in_column(gray, current_col)
    if current_row == -1:
        return edges
    edges.append((current_col, current_row))

    for current_col in range(start_col + 1, gray.shape[1]):
        best_row = current_row
        best_val = -1
        for offset in range(-searchrange, searchrange + 1):
            r = current_row + offset
            if r < 0 or r >= gray.shape[0] - 1:
                continue
            grad = abs(int(gray[r + 1, current_col]) - int(gray[r, current_col]))
            if grad > best_val:
                best_val = grad
                best_row = r
        thresh = _compute_column_threshold(gray, current_col, k)
        if best_val >= thresh:
            current_row = best_row
        edges.append((current_col, current_row))
        if current_row > gray.shape[0] - 20:
            return edges
    return edges

```

```

def fit_circle_ls(points: Sequence[Tuple[float, float]]) -> tuple[float, float, float] | None:
    if len(points) < 3:
        return None
    A = []
    Bv = []
    for x, y in points:
        A.append([x, y, 1.0])
        Bv.append(x * x + y * y)
    A = np.asarray(A, dtype=np.float64)
    Bv = np.asarray(Bv, dtype=np.float64)
    sol, _, _, _ = np.linalg.lstsq(A, Bv, rcond=None)
    a, b, c = sol
    cx = a / 2.0
    cy = b / 2.0
    r = np.sqrt(c + cx * cx + cy * cy)
    return cx, cy, r


def fit_circle_hyper(points: Sequence[Tuple[float, float]]) -> tuple[float, float, float] | None:
    n = len(points)
    if n < 3:
        return None
    pts = np.asarray(points, dtype=np.float64)
    mean = pts.mean(axis=0)
    x = pts[:, 0] - mean[0]
    y = pts[:, 1] - mean[1]
    z = x * x + y * y

    Mxx = np.mean(x * x)
    Myy = np.mean(y * y)
    Mxy = np.mean(x * y)
```

```
    Mxz = np.mean(x * z)
    Myz = np.mean(y * z)
    Mzz = np.mean(z * z)
    mean_z = np.mean(z)

    M = np.array(
        [
            [Mzz, Mxz, Myz, mean_z],
            [Mxz, Mxx, Mxy, 0.0],
            [Myz, Mxy, Myy, 0.0],
            [mean_z, 0.0, 0.0, 1.0],
        ]
    )

    N_inv = np.array(
        [
            [0.0, 0.0, 0.0, 0.5],
            [0.0, 1.0, 0.0, 0.0],
            [0.0, 0.0, 1.0, 0.0],
            [0.5, 0.0, 0.0, -2.0 * mean_z],
        ]
    )

    P = N_inv @ M
    eigvals, eigvecs = np.linalg.eig(P)
    mask_real = np.isclose(eigvals.imag, 0.0, atol=1e-10)
    eigvals = eigvals.real[mask_real]
    eigvecs = eigvecs[:, mask_real]

    pos = eigvals[eigvals > 1e-12]
    if pos.size == 0:
        return None
    idx = int(np.argmin(pos))
    v = eigvecs[:, np.where(eigvals > 1e-12)[0][idx]].real
```

```python
    A, B, C, D = v
    if abs(A) < 1e-7:
        return None
    det = B * B + C * C - 4 * A * D
    if det < 0:
        return None
    cx = -B / (2 * A)
    cy = -C / (2 * A)
    r = np.sqrt(det) / (2 * abs(A))
    return cx + mean[0], cy + mean[1], r


# --- Limb profiles ------------------------------------------------------------

def get_line_profile(img: np.ndarray, start: Tuple[float, float], end: Tuple[float, float]) -> list[int]:
    """
    Sample pixel values along the line from start to end (inclusive).
    Uses a simple integer-step interpolation (Bresenham-like) to avoid cv2.LineIterator.
    """
    H, W = img.shape
    x0, y0 = map(int, map(round, start))
    x1, y1 = map(int, map(round, end))

    dx = x1 - x0
    dy = y1 - y0
    steps = max(abs(dx), abs(dy))
    if steps == 0:
        if 0 <= x0 < W and 0 <= y0 < H:
            return [int(img[y0, x0])]
        return []
```

```
    xs = np.linspace(x0, x1, steps + 1)
    ys = np.linspace(y0, y1, steps + 1)
    vals: list[int] = []
    for x, y in zip(xs, ys):
        xi, yi = int(round(x)), int(round(y))
        if 0 <= xi < W and 0 <= yi < H:
            vals.append(int(img[yi, xi]))
    return vals


def analyze_limb_profiles(
    img: np.ndarray,
    center: Tuple[float, float],
    radius: float,
    scan_length: int = 160,
    margin: int = 20,
    step_deg: int = 1,
) -> list[dict]:
    H, W = img.shape
    cx, cy = center
    results: list[dict] = []
    for angle in range(0, 360, step_deg):
        rad = np.deg2rad(angle)
        r_inner = radius - margin
        r_outer = radius + (scan_length - margin)
        start = (cx + r_inner * np.cos(rad), cy + r_inner * np.sin(
           rad))
        end = (cx + r_outer * np.cos(rad), cy + r_outer * np.sin(
           rad))
        if not (0 <= start[0] < W and 0 <= start[1] < H and 0 <=
           end[0] < W and 0 <= end[1] < H):
            continue
        profile = get_line_profile(img, start, end)
```

```
        grads = [abs(profile[i + 1] - profile[i]) for i in range(
            len(profile) - 1)]
        if grads:
            max_grad = max(grads)
            max_pos = grads.index(max_grad)
        else:
            max_grad = 0
            max_pos = 0
        results.append(
            {
                "angle_deg": angle,
                "max_grad": max_grad,
                "max_grad_pos": max_pos,
                "profile_len": len(profile),
            }
        )
    return results


# --- Star tracker data structures
    ------------------------------------------
# --- Star tracker integration (delegated to star_tracker.py)
    ------------------

STAR_MAX_CATALOG_MAG = 8.0

def run_star_tracker_wrapper(
    image_path: str,
    catalog_path: str,
    max_catalog_mag: float = 8.0,
    output_path: str = "annotated.png",
    detection_mask: Optional[np.ndarray] = None,
):
```

```
    """Delegate to star_tracker.run_star_tracker for full functionality."""
    star_tracker.run_star_tracker(
        image_path=image_path,
        catalog_path=catalog_path,
        max_catalog_mag=max_catalog_mag,
        output_path=output_path,
        detection_mask=detection_mask,
    )


# --- main ----------------------------------------------------------------


def process_folder(
    science_dir: Path,
    output_dir: Path,
    omega: np.ndarray,
    delta_t_s: float,
    denoise_config: DenoiseConfig | None = None,
    lens_maps: tuple[np.ndarray, np.ndarray] | None = None,
) -> List[Path]:
    frames = sorted(science_dir.glob("IMD_*.png"))
    if not frames:
        print(f"No science frames found in {science_dir}")
        return []

    written: List[Path] = []
    for frame in frames:
        raw = _load_grayscale(frame)
        dtype = raw.dtype
        work = raw.astype(np.float32)

```

```
        if APPLY_DESPIN:
            recombined, mask_full, R = recombine_with_rotation(work, omega, delta_t_s)
            work = recombined
            work[mask_full == 0] = 0
        else:
            mask_full = np.ones_like(work, dtype=np.uint8)
            R = np.eye(3, dtype=np.float64)

        if APPLY_LENS and lens_maps is not None:
            work = _apply_lens_correction(work, lens_maps)
            mask_full = _apply_lens_correction(mask_full, lens_maps, interpolation=cv2.INTER_NEAREST)
            mask_full = (mask_full > 0.5).astype(np.uint8)
            work[mask_full == 0] = 0

        if APPLY_RESIDUAL_GLOW:
            work = column_median_subtract(work)

        if APPLY_FFT_NOTCH:
            work = fourier_notch_reconstruct(work, MANUAL_NOTCHES, NOTCH_RADIUS)

        if APPLY_DENOISE and denoise_config is not None:
            work = clean_image(work, denoise_config)

        # Analysis-only steps
        edges: list[Tuple[int, int]] = []
        circle_ls: Optional[Tuple[float, float, float]] = None
        circle_hyper: Optional[Tuple[float, float, float]] = None
        limb_stats: list[dict] = []

        if APPLY_EDGE_TRACKING:
            edge_img = work.astype(np.uint8)
```

```
            H_img = edge_img.shape[0]
            # Base ROI: lower portion + overlap mask
            roi_start = int(0.5 * H_img)
            edge_mask = np.ones_like(edge_img, dtype=bool)
            edge_mask[:roi_start, :] = False
            edge_mask[mask_full == 0] = False
            # Brightness gate to avoid tracking pure background
            valid_pixels = edge_img[edge_mask]
            if valid_pixels.size > 0:
                bright_thresh = np.percentile(valid_pixels, 50.0)
                bright_mask = edge_img >= bright_thresh
                edge_mask &= bright_mask
            edge_img_roi = edge_img.copy()
            edge_img_roi[~edge_mask] = 0
            # Estimate dominant gradient row to focus search band
            row_energy = np.sum(np.abs(np.diff(edge_img_roi.astype(np.int32), axis=0)), axis=1)
            if np.any(row_energy):
                peak_row = int(np.argmax(row_energy))
            else:
                peak_row = int(0.75 * H_img)
            band_half = 60
            band = (max(1, peak_row - band_half), min(H_img - 1, peak_row + band_half))
            edges = track_edge_andrey(edge_img_roi, start_col=10, k=3.0, searchrange=3)

        if APPLY_CIRCLE_FITS and len(edges) >= 8:
            circle_ls = fit_circle_ls(edges)
            circle_hyper = fit_circle_hyper(edges)

        if APPLY_LIMB_PROFILES and circle_ls:
            cx, cy, radius = circle_ls
```

```
            limb_stats = analyze_limb_profiles(work.astype(np.uint8), (cx, cy), radius)

        # Optional visualization of edge + fits
        if APPLY_EDGE_TRACKING:
            # Use the same dynamic range as the saved output (no extra normalization to avoid boosting noise)
            overlay_base = _to_8bit(work, dtype)
            overlay = cv2.cvtColor(overlay_base, cv2.COLOR_GRAY2BGR)
            # Red points: edge tracking samples
            for x, y in edges:
                cv2.circle(overlay, (int(x), int(y)), 1, (0, 0, 255), -1, lineType=cv2.LINE_AA)
            # Blue line: LS fit (Kasa)
            if circle_ls:
                cx_l, cy_l, r_l = circle_ls
                cv2.circle(overlay, (int(round(cx_l)), int(round(cy_l))), int(round(r_l)), (255, 0, 0), 2, lineType=cv2.LINE_AA)
            # Yellow line: Hyper fit
            if circle_hyper:
                cx_h, cy_h, r_h = circle_hyper
                cv2.circle(overlay, (int(round(cx_h)), int(round(cy_h))), int(round(r_h)), (0, 255, 255), 2, lineType=cv2.LINE_AA)
            # Write overlay alongside the main output
            overlay_path = output_dir / f"{frame.stem}_limb_overlay.png"
            cv2.imwrite(str(overlay_path), overlay)

        out_path = output_dir / frame.name
        _save_grayscale(work, out_path, dtype)
        if APPLY_DESPIN:
```

```
            mask_path = MASK_OUTPUT_DIR / frame.name
            _save_grayscale(mask_full, mask_path, np.uint8)

        if APPLY_STAR_TRACKER:
            try:
                annotated_path = out_path.with_name(f"{out_path.stem}_annotated.png")
                run_star_tracker_wrapper(
                    image_path=str(out_path),
                    catalog_path=str(STAR_CATALOG_PATH),
                    max_catalog_mag=STAR_MAX_CATALOG_MAG,
                    output_path=str(annotated_path),
                    detection_mask=mask_full if (APPLY_DESPIN and STAR_USE_DESPIN_MASK) else None,
                )
                print(f"  star tracker annotated -> {annotated_path}")
            except Exception as exc:
                print(f"  star tracker failed: {exc}")

        # Report
        tag = []
        if APPLY_DESPIN:
            theta_deg = np.linalg.norm(omega) * delta_t_s * 180.0 / np.pi
            tag.append(f"dtheta={theta_deg:.6f}deg")
        if APPLY_LENS:
            tag.append("lens")
        if APPLY_DENOISE:
            tag.append("denoise")
        if APPLY_FFT_NOTCH:
            tag.append("fft")
        if APPLY_RESIDUAL_GLOW:
            tag.append("colmed")
```

```
        summary = ",".join(tag) if tag else "raw"
        print(f"{frame.name} -> {out_path} [{summary}]")
        if circle_ls:
            cx, cy, r = circle_ls
            print(f"  circle LS: cx={cx:.2f}, cy={cy:.2f}, r={r:.2f}")
        if circle_hyper:
            cx, cy, r = circle_hyper
            print(f"  circle Hyper: cx={cx:.2f}, cy={cy:.2f}, r={r:.2f}")
        if limb_stats:
            best = max(limb_stats, key=lambda d: d["max_grad"])
            print(f"  limb max grad {best['max_grad']} at angle {best['angle_deg']} deg")
        written.append(out_path)
    return written


def main() -> None:
    output_dir = OUTPUT_DIR
    output_dir.mkdir(parents=True, exist_ok=True)
    denoise_cfg = DENOISE_SETTINGS if APPLY_DENOISE else None
    if denoise_cfg and denoise_cfg.apply_dark_frame and denoise_cfg.dark_frame is None and denoise_cfg.dark_frame_path:
        denoise_cfg.dark_frame = _load_grayscale(Path(denoise_cfg.dark_frame_path))

    omega = np.array(OMEGA, dtype=np.float64)
    if OMEGA_FRAME == "spacecraft":
        omega = np.array(SC_TO_CHUD_MATRIX @ omega, dtype=np.float64)

    lens_maps = _build_lens_maps(LENS_KAPPA) if APPLY_LENS else None
```

```

    process_folder(
        science_dir=SCIENCE_DIR,
        output_dir=output_dir,
        omega=omega,
        delta_t_s=DELTA_T,
        denoise_config=denoise_cfg,
        lens_maps=lens_maps,
    )


if __name__ == "__main__":
    main()
```

Listing A.1: Main script

A.2.2 Star tracker script

```
import numpy as np
import cv2
from dataclasses import dataclass
from typing import List, Tuple, Dict, Optional
import math
import re
import csv
from pathlib import Path
from denoising import DenoiseConfig, FlattenConfig, clean_image


# ------------------------------
# Data structures
# ------------------------------

@dataclass
class DetectedStar:
```

```
    x: float            # pixel x (column)
    y: float            # pixel y (row)
    flux: float         # integrated brightness
    ray_cam: Optional[np.ndarray] = None  # 3D unit vector in camera frame
    catalog_idx: Optional[int] = None     # catalog index after matching


@dataclass
class ImageInputConfig:
    """Configuration for loading the input image with optional rotation/resizing."""
    path: str
    rotate_deg: float = 0.0                      # CCW rotation for processing
    resize_to: Optional[Tuple[int, int]] = None  # (width, height) in px


@dataclass
class CatalogStar:
    name: str
    ra_deg: float
    dec_deg: float
    mag: float
    vec_inertial: np.ndarray


@dataclass
class ProcessingOptions:
    """Processing toggles for star detection."""
    background_subtraction: bool = False
    background_kernel: int = 51
```

```
    normalize: bool = False
    threshold_multiplier: float = 2.5
    morph_open: bool = False
    morph_kernel: int = 3
    morph_iterations: int = 1
    min_area: Optional[int] = 2
    max_area: Optional[int] = 20000
    max_stars: int = 200
    saturated_threshold: int = 225
    allow_oversized_saturated: bool = True


def _radec_to_vec(ra_deg: float, dec_deg: float) -> np.ndarray:
    ra = math.radians(ra_deg)
    dec = math.radians(dec_deg)
    x = math.cos(dec) * math.cos(ra)
    y = math.cos(dec) * math.sin(ra)
    z = math.sin(dec)
    return np.array([x, y, z], dtype=np.float64)

# ------------------------------
# 1. JUNO ASC (CHU-D) CONSTANTS
# ------------------------------
# Hardware values from the documentation
JUNO_HARDWARE_PARAMS = {
    "efl_um": 20006.0,      # Effective Focal Length
    "pixel_x_um": 8.6,      # Pixel size X (um)
    "pixel_y_um": 8.3,      # Pixel size Y (um)
    "res_x": 752.0,         # Native width
    "res_y": 580.0,         # Native height
    "cx_hardware": 383.0,   # Native optical center X
    "cy_hardware": 257.0,   # Native optical center Y
    "kappa": 3.3e-8         # Distortion coefficient (small but
        included)
```

```
}

# JUNO ASC (CHU-D) lens parameters
LENS_JUNO_ASC = {
    # derived focal lengths in pixels
    "fx": 20006.0 / 8.6,    # ~2326.28 px
    "fy": 20006.0 / 8.3,    # ~2410.36 px
    "cx": 383.0,
    "cy": 257.0,
    "kappa": 3.3e-8,        # single-term radial distortion
    "W": 752,
    "H": 580,
}

# ------------------------------
# Star-tracker runconfig (adjust for your frame/catalog)
# ------------------------------
STAR_IMAGE_PATH = Path("JUNO Despin Images") / "Despinned" / "Denoised"
STAR_IMAGE_FALLBACK = Path("JUNO Despin Images") / "Despinned" / "IMD_621530842.422424.png"
# Use a Gaia/HIP CSV as the primary catalog for labeling.
STAR_CATALOG_PATH = Path("/Users/vojtadeconinck/Downloads/ImageAnalysis30300/combined_stars.csv")
STAR_OUTPUT_PATH = Path("annotated.png")
STAR_MAX_CATALOG_MAG = 8.0
STAR_ROTATE_DEG = 0.0
STAR_RESIZE_TO: Optional[Tuple[int, int]] = None
STAR_FLIP_CODE: Optional[int] = None  # None (no flip), 0=vertical, 1=horizontal, -1=both
STAR_MIRROR_X_IN_RAYS = True        # mirror camera-x in the ray mapping instead of flipping the image
STAR_PROCESSING_MODE = "real"  # fallback preset
```

```
STAR_PROCESSING_OPTIONS: Optional[ProcessingOptions] =
    ProcessingOptions(
     background_subtraction=True,
     background_kernel=61,
     normalize=True,
     threshold_multiplier=1.4,
     morph_open=True,
     morph_kernel=3,
     morph_iterations=1,
     min_area=10,
     max_area=600,
     max_stars=400,
     saturated_threshold=220,
     allow_oversized_saturated=True,
)
STAR_CAMERA_OVERRIDE: Optional[Dict[str, float]] = None
STAR_MASK_BORDERS_PX = (20, 80, 20, 20)  # (top, bottom, left,
    right) mask away overlap-loss edges
# Debug output settings
STAR_SAVE_DEBUG_INTERMEDIATES = True
STAR_DEBUG_DIR = Path("JUNO Despin Images") / "star_tracker_debug"
# Precomputed valid-region masks (odd/even overlap) produced by
    juno_despin
STAR_VALID_MASK_DIR = Path("JUNO Despin Images") / "Despinned" / "
    Masks"
# Jupiter glare mask settings (optional)
STAR_SAVE_JUPITER_MASK = True
STAR_JUPITER_PERCENTILE = 90.0  # high percentile to isolate glare
    (lower to include halo)
STAR_JUPITER_MIN_AREA = 3000    # minimum glare component area;
    stars are << this area
STAR_JUPITER_MIN_Y_FRAC = 0.25  # only suppress blobs in lower part
     of image
STAR_JUPITER_DILATE = 35        # dilate glare mask to include halo
```

```
STAR_JUPITER_PAD_Y = 0          # no rectangular padding, rely on dilation
STAR_JUPITER_PAD_X = 0
STAR_DISABLE_JUPITER_FALLBACK = True  # skip row-mean fallback if no blob found
STAR_JUPITER_EXCLUDE_STAR_MAX_AREA = 400  # components <= this area are treated as stars, not glare
# Auto-build overlap mask from the input image instead of fixed borders (used if no precomputed mask)
STAR_AUTO_OVERLAP_MASK = True
STAR_AUTO_MASK_PERCENTILE = 70.0  # percentile to threshold non-zero region
STAR_AUTO_MASK_CLOSE_KERNEL = 9   # odd kernel for closing
STAR_AUTO_MASK_ERODE = 2          # pixels to erode after closing (set 0 to skip)

APPLY_DENOISING = False
STAR_DENOISE_SETTINGS = DenoiseConfig(
    apply_flatten=False,
    spike_neighborhood=1,
    spike_threshold=40.0,
    median_kernel=3,
    gaussian_kernel=0,
    gaussian_sigma=0.0,
)
STAR_ATTITUDE_MIN_INLIERS = 4
STAR_ATTITUDE_MAX_ERR_DEG = 0.2
STAR_ATTITUDE_MAX_ITER = 3000
# Name-lookup table to map HIP/Gaia IDs to readable labels.
STAR_NAME_LOOKUP_PATH = Path("gaia_to_names.csv")  # Gaia->HIP mapping source
STAR_GAIA_HIP_PATH = Path("2025-12-05-15-20-27-425171.csv")  # default Gaia->HIP crossmatch
```

```
# BSC TSV is only used as a positional fallback when the lookup is missing.
STAR_BSC_PATH = Path("bright_star_catalog.tsv")            # default Bright Star Catalog
STAR_IAU_PATH = Path("IAU-Catalog of Star Names (always up to date).csv")  # optional IAU names
STAR_HIP_TO_HD_PATH: Optional[Path] = None                  # set if you have a HIP->HD mapping CSV
STAR_GAIA_POS_PATH = Path("gaia_bright_stars.csv")          # optional Gaia positions to help name matching


def _rotate_image_keep_size(image: np.ndarray, angle_deg: float) -> np.ndarray:
    """Rotate around image center while preserving dimensions."""
    if abs(angle_deg) < 1e-6:
        return image
    h, w = image.shape[:2]
    M = cv2.getRotationMatrix2D((w / 2.0, h / 2.0), angle_deg, 1.0)
    return cv2.warpAffine(
        image,
        M,
        (w, h),
        flags=cv2.INTER_LINEAR,
        borderMode=cv2.BORDER_CONSTANT,
        borderValue=0,
    )

def _hms_to_deg(hms: str) -> float:
    """Convert 'HH:MM:SS.SSS' or compact 'HHMMSS.S' to RA in degrees."""
    hms = hms.strip()
    if ":" in hms:
        h, m, s = hms.split(":")
```

```
        h = float(h); m = float(m); s = float(s)
    else:
        # compact notation 'HHMMSS.S'
        if len(hms) < 6:
            raise ValueError(f"RA string too short: {hms!r}")
        h = float(hms[0:2])
        m = float(hms[2:4])
        s = float(hms[4:])
    return 15.0 * (h + m/60.0 + s/3600.0)


def _dms_to_deg(dms: str) -> float:
    """Convert '+DD:MM:SS.SS' or '+DDMMSS' style strings to Dec in
        degrees."""
    dms = dms.strip()
    if ":" in dms:
        sign = -1.0 if dms.startswith("-") else 1.0
        body = dms.lstrip("+-")
        d, m, s = body.split(":")
        d = float(d); m = float(m); s = float(s)
    else:
        sign = -1.0 if dms[0] == "-" else 1.0
        body = dms.lstrip("+-")
        if len(body) < 4:
            raise ValueError(f"Dec string too short: {dms!r}")
        d = float(body[0:2])
        m = float(body[2:4])
        s = float(body[4:]) if len(body) > 4 else 0.0
    return sign * (d + m/60.0 + s/3600.0)


def _to_uint8_debug(img: np.ndarray) -> np.ndarray:
    """Scale various arrays to uint8 for debug dumps."""
    if img.dtype == np.uint8:
```

```
        return img
    if img.dtype == np.bool_:
        return (img.astype(np.uint8) * 255)
    if np.issubdtype(img.dtype, np.integer):
        max_val = np.iinfo(img.dtype).max
        return np.clip(img.astype(np.float32) * (255.0 / max_val), 0, 255).astype(np.uint8)
    lo = float(np.percentile(img, 1.0))
    hi = float(np.percentile(img, 99.0))
    if hi <= lo:
        hi = lo + 1e-6
    norm = np.clip((img - lo) / (hi - lo), 0.0, 1.0)
    return np.clip(norm * 255.0, 0, 255).astype(np.uint8)


def _make_viz_binary(img_u8: np.ndarray) -> np.ndarray:
    """Create a more visible binary for sparse masks (dilate and invert)."""
    if img_u8.dtype != np.uint8:
        img_u8 = _to_uint8_debug(img_u8)
    if img_u8.max() == 0:
        return img_u8
    # If mostly 0/255, dilate to make sparse points visible and invert for contrast.
    unique_vals = np.unique(img_u8)
    if unique_vals.size <= 3:
        k = np.ones((3, 3), np.uint8)
        dil = cv2.dilate(img_u8, k, iterations=1)
        inv = 255 - dil
        return inv
    return img_u8


```

```python
def _make_overlay(mask_u8: np.ndarray, base_gray_u8: np.ndarray) -> np.ndarray:
    """Overlay binary mask as red contours on the grayscale base image."""
    if mask_u8.dtype != np.uint8:
        mask_u8 = _to_uint8_debug(mask_u8)
    if base_gray_u8.dtype != np.uint8:
        base_gray_u8 = _to_uint8_debug(base_gray_u8)
    color = cv2.cvtColor(base_gray_u8, cv2.COLOR_GRAY2BGR)
    if mask_u8.max() == 0:
        return color
    contours, _ = cv2.findContours(mask_u8, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    cv2.drawContours(color, contours, -1, (0, 0, 255), 1)
    return color


def _quadrilateral_mask_from_image(img_gray: np.ndarray,
                                   thresh_percentile: float = 0.5,
                                   close_kernel: int = 9,
                                   erode_px: int = 2) -> np.ndarray:
    """Build a convex/quad mask from the truly non-zero region of the image."""
    positive = img_gray[img_gray > 0]
    if positive.size == 0:
        return np.zeros_like(img_gray, dtype=np.uint8)
    thr_val = float(np.percentile(positive, thresh_percentile))
    thr_val = max(1.0, thr_val)
    mask = (img_gray > thr_val).astype(np.uint8)
    if close_kernel and close_kernel > 1:
        k = close_kernel if close_kernel % 2 == 1 else close_kernel + 1
        kernel = np.ones((k, k), np.uint8)
```

```
        mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel, iterations=1)
    if erode_px and erode_px > 0:
        k = max(1, int(erode_px))
        kernel = np.ones((k, k), np.uint8)
        mask = cv2.erode(mask, kernel, iterations=1)

    # Convex hull of non-zero pixels
    ys, xs = np.nonzero(mask)
    if ys.size == 0:
        return np.zeros_like(mask, dtype=np.uint8)
    pts = np.column_stack((xs, ys)).astype(np.int32)
    hull = cv2.convexHull(pts)

    # Try to approximate to 4 points if possible
    approx = cv2.approxPolyDP(hull, 0.01 * cv2.arcLength(hull, True), True)
    poly = approx if len(approx) >= 3 else hull
    quad_mask = np.zeros_like(mask, dtype=np.uint8)
    cv2.fillPoly(quad_mask, [poly], 1)
    return quad_mask


def _write_debug_images(prefix: str, images: Dict[str, np.ndarray], out_dir: Optional[Path] = None) -> None:
    """Write intermediate debug images (uint8 scaled) to STAR_DEBUG_DIR or a custom subfolder."""
    if not images:
        return
    out_dir = out_dir or STAR_DEBUG_DIR
    out_dir.mkdir(parents=True, exist_ok=True)
    base_gray = images.get("step_input_gray")
    base_gray_u8 = _to_uint8_debug(base_gray) if base_gray is not None else None
```

```
    for name, arr in images.items():
        try:
            out = _to_uint8_debug(arr)
        except Exception:
            continue
        cv2.imwrite(str(out_dir / f"{prefix}_{name}.png"), out)
        # Also write a visibility-enhanced version for sparse binaries
        viz = _make_viz_binary(out)
        if viz is not out:
            cv2.imwrite(str(out_dir / f"{prefix}_{name}_viz.png"), viz)
        # Overlay on base gray if available and this looks like a binary/sparse mask
        if base_gray_u8 is not None and out.max() > 0 and np.unique(out).size <= 10:
            overlay = _make_overlay(out, base_gray_u8)
            cv2.imwrite(str(out_dir / f"{prefix}_{name}_overlay.png"), overlay)


def _load_precomputed_mask(image_path: Path) -> Optional[np.ndarray]:
    """
    Load a precomputed valid-region mask (odd/even overlap) matching the image filename.
    Returns a uint8 mask with 1s where valid, or None if not found or unreadable.
    """
    mask_path = STAR_VALID_MASK_DIR / image_path.name
    if not mask_path.exists():
        return None
    mask = cv2.imread(str(mask_path), cv2.IMREAD_GRAYSCALE)
    if mask is None:
```

```
        return None
    return (mask > 0).astype(np.uint8)


def _compute_jupiter_mask(
    img_gray: np.ndarray,
    percentile: float = STAR_JUPITER_PERCENTILE,
    min_area: int = STAR_JUPITER_MIN_AREA,
    min_y_frac: float = STAR_JUPITER_MIN_Y_FRAC,
    dilate_px: int = STAR_JUPITER_DILATE,
    pad_y: int = STAR_JUPITER_PAD_Y,
    pad_x: int = STAR_JUPITER_PAD_X,
) -> Optional[np.ndarray]:
    """
    Detect bright Jupiter/glare blobs and return a uint8 mask (1 =
        glare).
    - Blur first to merge the limb/halo.
    - Threshold on a high percentile.
    - Keep large components low in the image (Jupiter region),
        union them, pad and dilate.
    Returns None if nothing is found.
    """
    # Light blur to merge halo structures
    img_blur = cv2.GaussianBlur(img_gray, (9, 9), 0)

    positive = img_blur[img_blur > 0]
    if positive.size == 0:
        return None
    thr = float(np.percentile(positive, percentile))
    if thr <= 0:
        return None
    bright = (img_blur >= thr).astype(np.uint8)
    num, labels, stats, centroids = cv2.
        connectedComponentsWithStats(bright, connectivity=8)
```

```
    H, W = img_gray.shape
    mask = np.zeros_like(img_gray, dtype=np.uint8)
    if num > 1:
        for idx in range(1, num):
            area = stats[idx, cv2.CC_STAT_AREA]
            if area < min_area:
                continue
            if area <= STAR_JUPITER_EXCLUDE_STAR_MAX_AREA:
                continue  # likely a star or small blob
            y_c = centroids[idx][1]
            if y_c < min_y_frac * H:
                continue
            # Use the actual component shape (not just its bounding box)
            mask = np.logical_or(mask, labels == idx).astype(np.uint8)

    # Fallback: if no blob matched, optionally try a row-mean band (disabled if flag set)
    if mask.max() == 0 and not STAR_DISABLE_JUPITER_FALLBACK:
        row_mean = img_blur.mean(axis=1)
        m = row_mean.mean()
        s = row_mean.std()
        cutoff = max(m + s, np.percentile(row_mean, 90))
        rows = np.where(row_mean >= cutoff)[0]
        if rows.size > 0:
            y0 = int(rows.min())
            y1 = int(rows.max())
            if y1 > H * 0.4:  # ensure it's towards the bottom
                mask[y0:y1 + 1, :] = 1

    if mask.max() == 0:
        return None

```

```
    # dilate to catch halo
    if dilate_px and dilate_px > 1:
        k = dilate_px if dilate_px % 2 == 1 else dilate_px + 1
        kernel = np.ones((k, k), np.uint8)
        mask = cv2.dilate(mask, kernel, iterations=1)
    return mask
def load_yale_bsc_ascii(path: str, max_mag: float = 6.5):
    """
    Parse the ASCII 'ybsc5' (Yale Bright Star Catalog 5) file.

    The parts[6] field looks like:
      '000001.1+444022000509.9+451345114.44-16.88'
       RA1950 Dec1950 RA2000 Dec2000   l      b

    Parsed sequentially instead of using hard-coded indices.
    """
    catalog = []
    with open(path, "r", encoding="ascii", errors="ignore") as f:
        for line in f:
            line = line.rstrip("\n")
            if not line.strip():
                continue

            # Skip headers/comment lines
            if line.startswith("sbsc5") or "BSC_number" in line or "BSC_num" in line:
                continue
            if not line.strip()[0].isdigit():
                continue

            parts = line.split()
            if len(parts) < 8:
                continue

```

```
            # Catalog number
            try:
                num = int(parts[0])
            except ValueError:
                continue

            ra_block = parts[6]
            mag_str = parts[7]

            # Minimum length check
            if len(ra_block) < 30:
                continue

            # Sequential parsing
            i = 0
            ra1950_str  = ra_block[i:i+7]; i += 7        # '000001.1'
            dec1950_str = ra_block[i:i+7]; i += 7        # '+444022'
            ra2000_str  = ra_block[i:i+7]; i += 7        # '000509.9'
            dec2000_str = ra_block[i:i+7]; i += 7        # '+451345'
            # Ignore remaining galactic l,b fields

            # Magnitude
            try:
                mag = float(mag_str)
            except ValueError:
                continue
            if mag > max_mag:
                continue

            # Convert RA/Dec to degrees
```

```python
            try:
                ra_deg  = _hms_to_deg(ra2000_str)
                dec_deg = _dms_to_deg(dec2000_str)
            except ValueError:
                # Skip malformed record
                continue

            vec = _radec_to_vec(ra_deg, dec_deg)

            name = f"BSC{num:04d}"
            catalog.append(
                CatalogStar(
                    name=name,
                    ra_deg=ra_deg,
                    dec_deg=dec_deg,
                    mag=mag,
                    vec_inertial=vec,
                )
            )

    if not catalog:
        raise RuntimeError("No stars parsed from ybsc5; the format may be unexpected.")
    catalog.sort(key=lambda c: c.mag)
    print(f"Loaded {len(catalog)} stars from {path}")
    return catalog

def load_gaia_csv(path: str, max_mag: float = 6.5) -> List[CatalogStar]:
    """
    Read Gaia/Hipparcos CSV and tolerate mixed-case column names.
    """
    catalog: List[CatalogStar] = []
    with open(path, "r", encoding="utf-8", newline="") as f:
```

```
        reader = csv.DictReader(f)

        # Force lowercase headers (ra, dec, mag) for robustness
        if reader.fieldnames:
            reader.fieldnames = [name.lower() for name in reader.
                fieldnames]

        for row in reader:
            try:
                ra_deg = float(row["ra"])
                dec_deg = float(row["dec"])
                mag = float(row["mag"])
            except (KeyError, ValueError):
                continue

            if mag > max_mag:
                continue

            name = row.get("source_id", "Unknown")
            vec = _radec_to_vec(ra_deg, dec_deg)

            catalog.append(
                CatalogStar(
                    name=str(name),
                    ra_deg=ra_deg,
                    dec_deg=dec_deg,
                    mag=mag,
                    vec_inertial=vec,
                )
            )

    if not catalog:
        raise RuntimeError("No stars parsed. Check the CSV file.")

```

```python
    catalog.sort(key=lambda c: c.mag)
    print(f"Loaded {len(catalog)} stars from {path}")
    return catalog


def _parse_hip_from_name(name: str) -> Optional[int]:
    """Extract HIP number from strings like 'HIP32349'."""
    m = re.match(r"hip\s*(\d+)", name.strip(), flags=re.IGNORECASE)
    if not m:
        return None
    try:
        return int(m.group(1))
    except ValueError:
        return None


def _parse_gaia_from_name(name: str) -> Optional[int]:
    """Extract Gaia source_id if the name is a pure digit string."""
    s = name.strip()
    if not s.isdigit():
        return None
    try:
        return int(s)
    except ValueError:
        return None


def load_gaia_to_hip_map(path: Path) -> Dict[int, int]:
    """Load Gaia->HIP mapping from gaia_to_names.csv (HIP column)."""
    if not path.exists():
        print(f"INFO: Gaia->HIP lookup file not found at {path}, skipping mapping.")
```

```python
        return {}
    gaia_to_hip: Dict[int, int] = {}
    with open(path, "r", encoding="utf-8", newline="") as f:
        reader = csv.DictReader(f)
        for row in reader:
            hip_raw = row.get("HIP") or row.get("hip")
            gaia_raw = row.get("gaia_id") or row.get("GAIA_ID") or row.get("gaia")
            if not gaia_raw or not hip_raw:
                continue
            try:
                gaia_val = int(float(gaia_raw))
                hip_val = int(float(hip_raw))
            except ValueError:
                continue
            gaia_to_hip[gaia_val] = hip_val
    print(f"Loaded {len(gaia_to_hip)} Gaia->HIP mappings from {path}")
    return gaia_to_hip



def load_iau_names(path: Path) -> Dict[int, str]:
    """
    Load HIP->IAU proper name map from the IAU catalog CSV if available.
    Prefer the Simbad spelling column when present.
    """
    if not path.exists():
        return {}
    hip_names: Dict[int, str] = {}
    with open(path, "r", encoding="utf-8", newline="") as f:
        reader = csv.DictReader(filter(lambda ln: ln.strip(), f))
        for row in reader:
```

```python
            # Strip simple HTML span wrappers and normalize keys
            cleaned = {}
            for k, v in row.items():
                if k is None:
                    continue
                key = re.sub(r"<[^>]+>", "", k).strip()
                cleaned[key.lower()] = v

            hip_raw = (
                cleaned.get("hip")
                or cleaned.get("hip id")
                or row.get("HIP")
                or row.get("hip")
                or row.get("Hip")
            )
            if not hip_raw:
                continue

            name = (
                cleaned.get("simbad spelling")
                or cleaned.get("proper names")
                or row.get("Name")
                or row.get("name")
            )
            if not name:
                continue

            try:
                hip_val = int(float(hip_raw))
            except ValueError:
                continue
            hip_names[hip_val] = name.strip()
    print(f"Loaded {len(hip_names)} IAU HIP names from {path}")
    return hip_names
```

```
def _parse_radec_from_bsc(row: Dict[str, str]) -> Optional[Tuple[float, float]]:
    ra = row.get("RAJ2000", "")
    dec = row.get("DEJ2000", "")
    try:
        h, m, s = [float(x) for x in ra.replace(":", " ").split()]
        ra_deg = 15.0 * (h + m / 60.0 + s / 3600.0)
        sign = -1.0 if dec.strip().startswith("-") else 1.0
        d, dm, ds = [float(x) for x in dec.replace(":", " ").split()]
        dec_deg = sign * (abs(d) + dm / 60.0 + ds / 3600.0)
        return ra_deg, dec_deg
    except Exception:
        return None


def load_bsc_for_positions(path: Path) -> Optional[List[Tuple[float, float, str]]]:
    """
    Light-weight BSC loader for positional fallback: returns list of (ra_deg, dec_deg, name).
    """
    if not path.exists():
        return None
    rows = []
    with open(path, "r", encoding="utf-8", newline="") as f:
        reader = csv.DictReader(filter(lambda ln: not ln.startswith("#") and ln.strip(), f), delimiter="|")
        for row in reader:
            name = (row.get("Name") or "").strip()
            if not name:
                continue
```

```
            radec = _parse_radec_from_bsc(row)
            if radec is None:
                continue
            ra_deg, dec_deg = radec
            rows.append((ra_deg, dec_deg, name))
    print(f"Loaded {len(rows)} BSC entries with positions for fallback naming")
    return rows


def apply_name_lookup(
    catalog: List[CatalogStar],
    gaia_to_hip_map: Optional[Dict[int, int]] = None,
    iau_name_map: Optional[Dict[int, str]] = None,
) -> None:
    """
    Replace catalog names with IAU proper names when possible; otherwise keep/format Gaia IDs.
      1) If the catalog name is a Gaia id and gaia_to_hip_map has a mapping, treat it as HIP for IAU lookup.
      2) For HIP ids: prefer IAU proper names.
      3) If no IAU name was assigned and we have a Gaia id, label as 'Gaia####'.
      4) Otherwise leave the existing name (e.g., HIP####).
    """
    gaia_to_hip_map = gaia_to_hip_map or {}
    replaced = 0
    for star in catalog:
        hip_val = _parse_hip_from_name(star.name)
        gaia_val = _parse_gaia_from_name(star.name)

        if gaia_val is not None and gaia_val in gaia_to_hip_map:
            hip_val = gaia_to_hip_map[gaia_val]

```

```
        assigned = False
        if hip_val is not None and iau_name_map:
            new_name = iau_name_map.get(hip_val)
            if new_name:
                if new_name != star.name:
                    replaced += 1
                star.name = new_name
                assigned = True

        if not assigned and gaia_val is not None:
            label = f"Gaia{gaia_val}"
            if label != star.name:
                replaced += 1
            star.name = label
    print(f"Applied name lookup to {replaced} catalog entries")


# ------------------------------
# Star detection
# ------------------------------
def detect_stars(image_gray: np.ndarray, options: ProcessingOptions
    , mask: Optional[np.ndarray] = None,
                 debug_images: Optional[Dict[str, np.ndarray]] =
                     None) -> List[DetectedStar]:
    """
    Hybrid detector:
    combines saturated-blob detection for very bright stars with
    background-subtracted detection for faint stars.
    """
    if image_gray.ndim != 2:
        raise ValueError("detect_stars expects a single-channel (
            grayscale) image.")

    mask_bool = None
```

```
    if mask is not None:
        if mask.shape != image_gray.shape:
            raise ValueError("Mask shape must match image.")
        mask_bool = mask.astype(bool)
        if debug_images is not None:
            debug_images["step_mask"] = mask_bool.astype(np.uint8)
                * 255
    if debug_images is not None:
        debug_images["step_input_gray"] = image_gray.copy()

    # ---------------------------------------------------------
    # Path 1: saturated stars on the raw image (e.g., Antares)
    # Anything above the saturation threshold in the original image
        counts as a star;
    # skip background subtraction to preserve bright cores.
    # ---------------------------------------------------------
    sat_thresh = int(np.clip(options.saturated_threshold, 0, 255))
    _, binary_saturated = cv2.threshold(image_gray, sat_thresh,
        255, cv2.THRESH_BINARY)
    if mask_bool is not None:
        binary_saturated = cv2.bitwise_and(binary_saturated,
            mask_bool.astype(np.uint8))

    # ---------------------------------------------------------
    # Path 2: faint-star detection using background subtraction
    # ---------------------------------------------------------
    work = image_gray.astype(np.float32)
    if mask_bool is not None:
        work = np.where(mask_bool, work, 0.0)
    if debug_images is not None:
        debug_images["step_masked_gray"] = work.copy()

    # Force background subtraction for the faint path unless
        explicitly disabled
```

```
    if options.background_subtraction:
        # Ensure the kernel is large enough (minimum 51) and odd
        k = max(options.background_kernel, 51)
        if k % 2 == 0:
            k += 1

        bg = cv2.medianBlur(image_gray, k)
        work = work - bg.astype(np.float32)
        if debug_images is not None:
            debug_images["step_background"] = bg
            debug_images["step_after_bg_sub"] = work.copy()

    if options.normalize:
        work -= work.min()
        max_val = work.max()
        if max_val > 0:
            work /= max_val
    if debug_images is not None:
        debug_images["step_after_norm"] = work.copy()

    # Threshold selection for faint detections
    mean, std = cv2.meanStdDev(work)
    mean_val = float(mean[0][0])
    std_val = float(std[0][0])
    thresh_val = mean_val + options.threshold_multiplier * std_val
    if debug_images is not None:
        debug_images["step_stats_mean_std"] = np.array([[mean_val,
            std_val]], dtype=np.float32)

    if options.normalize:
        _, binary_faint = cv2.threshold(work, thresh_val, 255, cv2.
            THRESH_BINARY)
    else:
```

```
        _, binary_faint = cv2.threshold(work, thresh_val, 255, cv2.THRESH_BINARY)

    binary_faint = binary_faint.astype(np.uint8)
    if mask_bool is not None:
        binary_faint = cv2.bitwise_and(binary_faint, mask_bool.astype(np.uint8))
    if debug_images is not None:
        debug_images["step_binary_faint"] = binary_faint.copy()

    # ------------------------------------------------------------
    # Step 3: merge saturated + faint detections (bitwise OR)
    # ------------------------------------------------------------
    final_binary = cv2.bitwise_or(binary_saturated, binary_faint)
    if mask_bool is not None:
        final_binary = cv2.bitwise_and(final_binary, mask_bool.astype(np.uint8))
    if debug_images is not None:
        debug_images["step_binary_saturated"] = binary_saturated.copy()
        debug_images["step_binary_final"] = final_binary.copy()

    # Clean up small noise
    if options.morph_open:
        kernel_size = max(1, options.morph_kernel)
        kernel = np.ones((kernel_size, kernel_size), np.uint8)
        iterations = max(1, options.morph_iterations)
        final_binary = cv2.morphologyEx(final_binary, cv2.MORPH_OPEN, kernel, iterations=iterations)
        if debug_images is not None:
            debug_images["step_binary_final_open"] = final_binary.copy()

    # ------------------------------------------------------------
```

```
    # Step 4: analyze blobs
    # -----------------------------------------------------------
    num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(final_binary)

    H, W = image_gray.shape
    detections: List[DetectedStar] = []

    # Safety: if max_area is unset, allow very large blobs for bright stars
    safe_max_area = options.max_area if options.max_area is not None else 100000

    for label in range(1, num_labels):
        x, y, w, h, area = stats[label]

        if options.min_area is not None and area < options.min_area:
            continue

        cx, cy = centroids[label]

        # Measure flux on the original image (not the background-subtracted one)
        r = int(math.sqrt(area)/2) + 2
        r = min(r, 40)  # cap radius

        x0 = max(int(cx) - r, 0)
        x1 = min(int(cx) + r + 1, W)
        y0 = max(int(cy) - r, 0)
        y1 = min(int(cy) + r + 1, H)

        patch = image_gray[y0:y1, x0:x1]
        if patch.size == 0:
```

```
            continue

        patch_max = int(patch.max()) if patch.size else 0
        is_saturated_blob = patch_max >= sat_thresh
        if area > safe_max_area and not (options.
            allow_oversized_saturated and is_saturated_blob):
            continue

        # Weighted centroid
        weights = patch.astype(float)
        s = weights.sum()

        if s <= 0:
            det = DetectedStar(x=cx, y=cy, flux=float(area))
        else:
            yy, xx = np.mgrid[y0:y1, x0:x1]
            cx_refined = (xx * weights).sum() / s
            cy_refined = (yy * weights).sum() / s
            det = DetectedStar(x=cx_refined, y=cy_refined, flux=s)

        detections.append(det)

    detections.sort(key=lambda d: d.flux, reverse=True)
    if options.max_stars and len(detections) > options.max_stars:
        detections = detections[:options.max_stars]

    print(f"DEBUG: Found {len(detections)} stars. Brightest flux: {
        detections[0].flux if detections else 0}")

    return detections
# ------------------------------
# Processing presets / toggles
# ------------------------------

```

```
def processing_preset(mode: str) -> ProcessingOptions:
    """Set default options for simulation (low noise) versus real
       data."""
    mode = (mode or "real").lower()
    if mode == "simulation":
        return ProcessingOptions(
            background_subtraction=False,
            background_kernel=31,
            normalize=True,
            threshold_multiplier=4.0,
            morph_open=False,
            min_area=1,
            max_area=800,
            max_stars=150,
        )
    # default: noisier real data
    return ProcessingOptions()


# ------------------------------
# Pixels -> unit vectors
# ------------------------------
def pixels_to_unit_vectors(detections: List[DetectedStar],
                           image_shape: Tuple[int, int],
                           hardware_params: Optional[Dict[str,
                              float]] = None) -> None:
    """
    Convert pixel detections into unit vectors using the Juno
       camera parameters,
    correcting for any resizing applied to the screenshot.
    hardware_params can override res_x/res_y/cx/cy/efl/pixel pitch.
    """
    H_img, W_img = image_shape

```

```
    params = dict(JUNO_HARDWARE_PARAMS)
    if hardware_params:
        params.update(hardware_params)

    # 1. Compute scale factor between screenshot and physical sensor
    # Assume the aspect ratio is roughly preserved.
    scale_x = W_img / params["res_x"]
    scale_y = H_img / params["res_y"]

    # Warn if aspect ratio is strongly distorted (stretched screenshot)
    if abs(scale_x - scale_y) > 0.05:
        print(f"WARNING:␣Aspect␣ratio␣looks␣off!␣X-scale={scale_x:.2f},␣Y-scale={scale_y:.2f}")

    # 2. Compute hardware fx/fy
    fx_hard = params["efl_um"] / params["pixel_x_um"] # ~2326 px
    fy_hard = params["efl_um"] / params["pixel_y_um"] # ~2410 px

    # 3. Scale to the screenshot dimensions
    fx = fx_hard * scale_x
    fy = fy_hard * scale_y
    cx = params["cx_hardware"] * scale_x
    cy = params["cy_hardware"] * scale_y
    kappa = params["kappa"]  # Distortion coefficient

    print(f"DEBUG:␣Juno␣hardware␣scale␣correction:␣{scale_x:.3f}x")
    print(f"␣␣␣␣␣␣␣Using␣fx={fx:.1f}␣(was␣{fx_hard:.1f}),␣cx={cx:.1f}␣(was␣{params['cx_hardware']})")

    for det in detections:
        # Pinhole model (with center correction and non-square pixels)
```

```
        # Image: x right, y down. Camera frame: y often up, z
            forward.

        # Step A: center and normalize with focal length
        x_norm = (det.x - cx) / fx
        y_norm = -(det.y - cy) / fy  # invert Y to map image to
            camera frame
        if STAR_MIRROR_X_IN_RAYS:
            x_norm *= -1.0  # mirror camera-x so geometry matches
                without flipping the image

        # Step B: distortion correction (inverse approximation for
            small kappa)
        r2 = x_norm**2 + y_norm**2
        if kappa != 0.0:
            factor = 1.0 - kappa * r2
            x_norm *= factor
            y_norm *= factor

        z = 1.0
        v = np.array([x_norm, y_norm, z], dtype=np.float64)
        v /= np.linalg.norm(v)

        det.ray_cam = v
# ------------------------------
# Catalog pair table
# ------------------------------

def build_catalog_pair_table(catalog: List[CatalogStar],
                             max_pairs: int = 5000,
                             angle_bin_deg: float = 0.01):
    """
    Compute pairwise angles between the brightest catalog stars.

```

```
    Returns:
        bins: dict bin_index -> list of (idx1, idx2, angle_rad)
        angle_bin_rad: bin size in radians
    """
    # choose N such that N(N-1)/2 ~ max_pairs
    N = min(len(catalog), int(0.5 * (1 + math.sqrt(1 + 8 * max_pairs))))
    N = max(3, N)
    angle_bin_rad = math.radians(angle_bin_deg)
    bins: Dict[int, List[Tuple[int, int, float]]] = {}
    for i in range(N - 1):
        vi = catalog[i].vec_inertial
        for j in range(i + 1, N):
            vj = catalog[j].vec_inertial
            cosang = float(np.clip(np.dot(vi, vj), -1.0, 1.0))
            ang = math.acos(cosang)
            b = int(ang / angle_bin_rad)
            bins.setdefault(b, []).append((i, j, ang))
    return bins, angle_bin_rad


# ------------------------------
# TRIAD: snelle attitude uit 2 vectorparen
# ------------------------------

def triad_attitude(v_cam: np.ndarray, w_cam: np.ndarray,
                   v_inertial: np.ndarray, w_inertial: np.ndarray) -> np.ndarray:
    """TRIAD: return R (3x3) mapping camera -> inertial frame."""
    # Camera triad
    v1 = v_cam / np.linalg.norm(v_cam)
    w1 = w_cam / np.linalg.norm(w_cam)
    t1_cam = v1
    t2_cam = np.cross(v1, w1)
```

```
    n2 = np.linalg.norm(t2_cam)
    if n2 < 1e-6:
        raise ValueError("TRIAD: camera vectors are nearly colinear.")
    t2_cam /= n2
    t3_cam = np.cross(t1_cam, t2_cam)
    C_cam = np.column_stack((t1_cam, t2_cam, t3_cam))

    # Inertial triad
    v2 = v_inertial / np.linalg.norm(v_inertial)
    w2 = w_inertial / np.linalg.norm(w_inertial)
    t1_in = v2
    t2_in = np.cross(v2, w2)
    n2 = np.linalg.norm(t2_in)
    if n2 < 1e-6:
        raise ValueError("TRIAD: inertial vectors are nearly colinear.")
    t2_in /= n2
    t3_in = np.cross(t1_in, t2_in)
    C_in = np.column_stack((t1_in, t2_in, t3_in))

    # R: camera -> inertiaal
    R = C_in @ C_cam.T
    return R


# ------------------------------
# Wahba via Davenport Q
# ------------------------------

def wahba_davenport_q(camera_vecs: np.ndarray,
                      inertial_vecs: np.ndarray,
                      weights: Optional[np.ndarray] = None) -> np.ndarray:
```

```
    """
    Solve Wahba's problem using Davenport's Q method.

    camera_vecs: (N,3)
    inertial_vecs: (N,3)
    """
    if weights is None:
        weights = np.ones(camera_vecs.shape[0], dtype=np.float64)
    W = np.diag(weights)
    B = camera_vecs.T @ W @ inertial_vecs
    S = B + B.T
    sigma = np.trace(B)
    Z = np.array([
        B[1, 2] - B[2, 1],
        B[2, 0] - B[0, 2],
        B[0, 1] - B[1, 0],
    ], dtype=np.float64)

    K = np.zeros((4, 4), dtype=np.float64)
    K[:3, :3] = S - sigma * np.eye(3)
    K[:3, 3] = Z
    K[3, :3] = Z
    K[3, 3] = sigma

    eigvals, eigvecs = np.linalg.eigh(K)
    q = eigvecs[:, np.argmax(eigvals)]  # largest eigenvalue
    q_vec = q[:3]
    q0 = q[3]

    # Normalize quaternion
    qx, qy, qz = q_vec
    norm_q = math.sqrt(q0*q0 + qx*qx + qy*qy + qz*qz)
    if norm_q == 0:
        raise RuntimeError("Zero quaternion in Wahba solution.")
```

```
    q0 /= norm_q
    qx /= norm_q
    qy /= norm_q
    qz /= norm_q

    # Quaternion -> rotatiematrix
    R = np.array([
        [1 - 2*(qy**2 + qz**2),     2*(qx*qy - qz*q0),         2*(qx*qz + qy*q0)],
        [2*(qx*qy + qz*q0),         1 - 2*(qx**2 + qz**2),     2*(qy*qz - qx*q0)],
        [2*(qx*qz - qy*q0),         2*(qy*qz + qx*q0),         1 - 2*(qx**2 + qy**2)],
    ], dtype=np.float64)

    return R


# ------------------------------
# RANSAC + pair-matching attitude
# ------------------------------

def attitude_from_star_pairs_ransac(
    detections: List[DetectedStar],
    catalog: List[CatalogStar],
    pair_table,
    angle_bin_rad: float,
    max_iterations: int = 500,
    max_inlier_err_deg: float = 0.2,
    min_inliers: int = 8,
):
    """
    Estimate attitude with RANSAC:
    - compare angles between detected star pairs to catalog pairs
```

```
    - build hypotheses via TRIAD
    - score on inlier count
    - refine best solution with Davenport Q
    """
    bins = pair_table
    cam_vecs = np.array([d.ray_cam for d in detections])
    if cam_vecs.shape[0] < 2:
        raise RuntimeError("Need at least 2 detections for attitude.")

    cat_vecs = np.array([c.vec_inertial for c in catalog])
    max_inlier_err_rad = math.radians(max_inlier_err_deg)
    cos_max_err = math.cos(max_inlier_err_rad)

    best_inliers: List[Tuple[int, int]] = []
    best_R = None

    # Precompute all detection pairs
    M = len(detections)
    det_pairs = []
    for i in range(M - 1):
        for j in range(i + 1, M):
            vi = cam_vecs[i]
            vj = cam_vecs[j]
            cosang = float(np.clip(np.dot(vi, vj), -1.0, 1.0))
            ang = math.acos(cosang)
            det_pairs.append((i, j, ang))

    if not det_pairs:
        raise RuntimeError("No detection pairs found.")

    import random
    for _ in range(max_iterations):
        # Pick a random detection pair
```

```
        i_det, j_det, ang_det = random.choice(det_pairs)
        bin_idx = int(ang_det / angle_bin_rad)
        candidate_pairs = []
        for b in (bin_idx - 1, bin_idx, bin_idx + 1):
            if b in bins:
                candidate_pairs.extend(bins[b])
        if not candidate_pairs:
            continue
        i_cat, j_cat, _ = random.choice(candidate_pairs)

        # Try both mappings (i->i_cat, j->j_cat) and swapped
        for mapping in [((i_det, i_cat), (j_det, j_cat)),
                        ((i_det, j_cat), (j_det, i_cat))]:
            (i1, c1), (i2, c2) = mapping
            v_cam1 = cam_vecs[i1]
            v_cam2 = cam_vecs[i2]
            v_cat1 = cat_vecs[c1]
            v_cat2 = cat_vecs[c2]
            try:
                R = triad_attitude(v_cam1, v_cam2, v_cat1, v_cat2)
            except ValueError:
                continue

            # Score hypothesis
            inliers: List[Tuple[int, int]] = []
            for k, v_cam in enumerate(cam_vecs):
                v_in = R @ v_cam
                dots = cat_vecs @ v_in
                idx = int(np.argmax(dots))
                if dots[idx] >= cos_max_err:
                    inliers.append((k, idx))

            if len(inliers) > len(best_inliers):
                best_inliers = inliers
```

```
                best_R = R

    if best_R is None or len(best_inliers) < min_inliers:
        raise RuntimeError("RANSAC could not find a consistent attitude.")

    # Refine with Wahba on all inliers
    cam_list = np.array([detections[i].ray_cam for (i, _) in best_inliers])
    in_list = np.array([catalog[j].vec_inertial for (_, j) in best_inliers])
    R_refined = wahba_davenport_q(cam_list, in_list)

    return R_refined, best_inliers


def debug_residuals(R: np.ndarray,
                    detections: List[DetectedStar],
                    catalog: List[CatalogStar],
                    inliers: List[Tuple[int, int]]) -> None:
    """Print median/max angular residuals in degrees for matched stars."""
    import numpy as np  # local to keep global imports light
    errs = []
    for det_idx, cat_idx in inliers:
        v_cam = detections[det_idx].ray_cam
        v_cat = catalog[cat_idx].vec_inertial
        v_pred = R @ v_cam
        cosang = float(np.clip(np.dot(v_pred, v_cat), -1.0, 1.0))
        err = math.degrees(math.acos(cosang))
        errs.append(err)
    if not errs:
        print("Residuals: none (no inliers).")
        return
```

```
    print(f"Residuals: median={np.median(errs):.3f} deg, max={max(
        errs):.3f} deg, N={len(errs)}")


def match_all_detections(R: np.ndarray,
                         detections: List[DetectedStar],
                         catalog: List[CatalogStar],
                         max_err_deg: float = 0.3) -> List[Tuple[
                             int, int]]:
    """Match remaining detections to nearest catalog star within
        angular threshold."""
    import numpy as np
    if not detections:
        return []
    cat_vecs = np.array([c.vec_inertial for c in catalog])
    max_err_rad = math.radians(max_err_deg)
    cos_min = math.cos(max_err_rad)

    extra_matches = []
    for i, det in enumerate(detections):
        if det.catalog_idx is not None:
            continue
        v_cam = det.ray_cam
        v_in = R @ v_cam
        dots = cat_vecs @ v_in
        idx = int(np.argmax(dots))
        if dots[idx] >= cos_min:
            det.catalog_idx = idx
            extra_matches.append((i, idx))
    return extra_matches


# ------------------------------
# Boresight RA/Dec
```

```
# ------------------------------

def rotation_to_boresight_radec(R: np.ndarray) -> Tuple[float,
    float]:
    """Return RA/Dec of the camera z-axis (boresight)."""
    boresight_cam = np.array([0.0, 0.0, 1.0], dtype=np.float64)
    v = R @ boresight_cam
    x, y, z = v
    dec = math.degrees(math.asin(z))
    ra = math.degrees(math.atan2(y, x))
    if ra < 0:
        ra += 360.0
    return ra, dec


# ------------------------------
# Image annotation
# ------------------------------

def annotate_image(
    image_bgr: np.ndarray,
    detections: List[DetectedStar],
    catalog: List[CatalogStar],
    matches: List[Tuple[int, int]],
    boresight_radec: Tuple[float, float],
) -> np.ndarray:
    """Draw circles and labels for matched stars."""
    out = image_bgr.copy()
    for det_idx, cat_idx in matches:
        det = detections[det_idx]
        star = catalog[cat_idx]
        center = (int(round(det.x)), int(round(det.y)))
        cv2.circle(out, center, 5, (0, 255, 0), 1, lineType=cv2.
            LINE_AA)
```

```
        label = star.name
        cv2.putText(out,
                    label,
                    (center[0] + 6, center[1] - 3),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    0.35,
                    (0, 255, 0),
                    1,
                    lineType=cv2.LINE_AA)

    ra_deg, dec_deg = boresight_radec
    txt = f"Pointing: RA={ra_deg:7.3f} deg  Dec={dec_deg:6.3f} deg"
    cv2.putText(out,
                txt,
                (20, 30),
                cv2.FONT_HERSHEY_SIMPLEX,
                0.6,
                (0, 255, 255),
                2,
                lineType=cv2.LINE_AA)
    return out


def annotate_detections_only(
    image_bgr: np.ndarray,
    detections: List[DetectedStar],
    header: str = "ATTITUDE FAILED",
) -> np.ndarray:
    """Overlay raw detections (no catalog matches) so failures are inspectable."""
    out = image_bgr.copy()
    for i, det in enumerate(detections):
        center = (int(round(det.x)), int(round(det.y)))
```

```
        cv2.circle(out, center, 5, (0, 255, 0), 1, lineType=cv2.
            LINE_AA)
        cv2.putText(
            out,
            f"{i+1}",
            (center[0] + 6, center[1] - 3),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.35,
            (0, 255, 0),
            1,
            lineType=cv2.LINE_AA,
        )
    if header:
        cv2.putText(
            out,
            header,
            (20, 30),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.6,
            (0, 0, 255),
            2,
            lineType=cv2.LINE_AA,
        )
    return out


# ------------------------------
# High-level pipeline
# ------------------------------

def run_star_tracker(
    image_path: str,
    catalog_path: str,
    max_catalog_mag: float = 8.0,
```

```
    output_path: str = "annotated.png",
    processing_options: Optional[ProcessingOptions] = None,
    image_config: Optional[ImageInputConfig] = None,
    camera_params_override: Optional[Dict[str, float]] = None,
    denoise_config: Optional[DenoiseConfig] = None,
    detection_mask: Optional[np.ndarray] = None,
):
    """
    Full pipeline: image -> attitude -> annotation.
    """
    img_stem = Path(image_path).stem
    debug_dir = STAR_DEBUG_DIR / img_stem if
        STAR_SAVE_DEBUG_INTERMEDIATES else None

    # Load image + optional resize/rotate
    cfg = image_config or ImageInputConfig(path=image_path)
    img_bgr = cv2.imread(cfg.path, cv2.IMREAD_COLOR)
    if img_bgr is None:
        raise RuntimeError(f"Could␣not␣read␣image␣{cfg.path}")
    if STAR_FLIP_CODE is not None:
        img_bgr = cv2.flip(img_bgr, STAR_FLIP_CODE)
        print(f"DEBUG:␣Input␣image␣flipped␣code={STAR_FLIP_CODE}")
    if cfg.resize_to:
        w, h = cfg.resize_to
        img_bgr = cv2.resize(img_bgr, (w, h), interpolation=cv2.
            INTER_LINEAR)
        print(f"DEBUG:␣Input␣image␣resized␣to␣{w}x{h}")
    if abs(cfg.rotate_deg) > 1e-6:
        img_bgr = _rotate_image_keep_size(img_bgr, cfg.rotate_deg)
        print(f"DEBUG:␣Input␣image␣rotated␣{cfg.rotate_deg:.2f}␣deg
            ␣CCW")
    img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
    if denoise_config is not None:
        img_gray = clean_image(img_gray, denoise_config)
```

```
    if detection_mask is not None:
        if detection_mask.shape != img_gray.shape:
            raise ValueError("Detection mask shape must match image.")
        detection_mask = detection_mask.astype(bool)
    else:
        detection_mask = None

    # Catalog (Yale ASCII or Gaia CSV)
    if catalog_path.lower().endswith(".csv"):
        catalog = load_gaia_csv(catalog_path, max_mag=max_catalog_mag)
    else:
        catalog = load_yale_bsc_ascii(catalog_path, max_mag=max_catalog_mag)

    # Name replacement: prefer IAU names (via HIP), otherwise Gaia IDs
    gaia_to_hip = load_gaia_to_hip_map(STAR_NAME_LOOKUP_PATH) if STAR_NAME_LOOKUP_PATH.exists() else {}
    iau_name_map = load_iau_names(STAR_IAU_PATH)
    apply_name_lookup(catalog, gaia_to_hip_map=gaia_to_hip, iau_name_map=iau_name_map)

    if processing_options is None:
        processing_options = ProcessingOptions()
    print(f"Processing options: {processing_options}")

    # Star detection
    debug_images = {} if STAR_SAVE_DEBUG_INTERMEDIATES else None
    detections = detect_stars(img_gray, options=processing_options, mask=detection_mask, debug_images=debug_images)
    if debug_images is not None:
```

```
        _write_debug_images(f"{img_stem}_detect_main", debug_images, out_dir=debug_dir)
    print(f"Detected {len(detections)} star candidates")

    # Retry with more sensitive preset if detections are too few
    if len(detections) < STAR_ATTITUDE_MIN_INLIERS:
        print(f"INFO: only {len(detections)} detections; retrying with a more sensitive preset")
        fallback_opts = ProcessingOptions(
            background_subtraction=True,
            background_kernel=max(processing_options.background_kernel, 71),
            normalize=True,
            threshold_multiplier=1.25,
            morph_open=False,
            morph_kernel=processing_options.morph_kernel,
            morph_iterations=processing_options.morph_iterations,
            min_area=3,
            max_area=processing_options.max_area,
            max_stars=400,
            saturated_threshold=min(processing_options.saturated_threshold, 215),
            allow_oversized_saturated=True,
        )
        debug_images_fb = {} if STAR_SAVE_DEBUG_INTERMEDIATES else None
        detections = detect_stars(img_gray, options=fallback_opts, mask=detection_mask, debug_images=debug_images_fb)
        if debug_images_fb is not None:
            _write_debug_images(f"{img_stem}_detect_fallback", debug_images_fb, out_dir=debug_dir)
        print(f"Detected {len(detections)} star candidates after fallback")
    if len(detections) < 2:
```

```
        print("WARN: Insufficient detections (<2) even after fallback; writing detections-only overlay.")
        annotated_fail = annotate_detections_only(img_bgr, detections, header="ATTITUDE FAILED: too few detections")
        cv2.imwrite(output_path, annotated_fail)
        return

    # Pixels -> 3D rays
    pixels_to_unit_vectors(detections, img_gray.shape, hardware_params=camera_params_override)

    # Catalog pairs
    pair_table, angle_bin_rad = build_catalog_pair_table(catalog)

    # Attitude via RANSAC + TRIAD + Davenport
    min_inliers = max(2, min(STAR_ATTITUDE_MIN_INLIERS, len(detections)))
    try:
        R, inliers = attitude_from_star_pairs_ransac(
            detections, catalog, pair_table, angle_bin_rad,
            max_iterations=STAR_ATTITUDE_MAX_ITER,
            max_inlier_err_deg=STAR_ATTITUDE_MAX_ERR_DEG,
            min_inliers=min_inliers,
        )
    except Exception as exc:
        print(f"ATTITUDE FAILED: {exc}")
        annotated_fail = annotate_detections_only(img_bgr, detections, header="ATTITUDE FAILED")
        cv2.imwrite(output_path, annotated_fail)
        return
    print(f"Found attitude with {len(inliers)} inliers")
    debug_residuals(R, detections, catalog, inliers)
```

```
    extra_matches = match_all_detections(R, detections, catalog, max_err_deg=max(STAR_ATTITUDE_MAX_ERR_DEG, 0.3))
    all_matches = inliers + extra_matches

    # Attach matches to detections
    for det_idx, cat_idx in all_matches:
        detections[det_idx].catalog_idx = cat_idx

    # Boresight RA/Dec
    ra_deg, dec_deg = rotation_to_boresight_radec(R)
    print(f"Boresight pointing: RA={ra_deg:.3f} deg  Dec={dec_deg:.3f} deg")

    # Annotation
    annotated = annotate_image(img_bgr, detections, catalog, all_matches,
                               (ra_deg, dec_deg))
    cv2.imwrite(output_path, annotated)
    print(f"Annotated image written to {output_path}")


def main() -> None:
    processing_opts = STAR_PROCESSING_OPTIONS or processing_preset(STAR_PROCESSING_MODE)
    base_path = STAR_IMAGE_PATH
    image_paths: List[Path] = []
    if base_path.is_dir():
        image_paths = sorted(p for p in base_path.glob("IMD_*.png") if p.is_file())
    elif base_path.exists():
        image_paths = [base_path]
    elif STAR_IMAGE_FALLBACK.exists():
        image_paths = [STAR_IMAGE_FALLBACK]
    if not image_paths:
```

```
            raise RuntimeError(
                f"No␣input␣images␣found.␣Tried␣directory/file␣{
                    STAR_IMAGE_PATH}␣and␣fallback␣{STAR_IMAGE_FALLBACK}"
            )

    print(f"Processing␣{len(image_paths)}␣image(s)")
    for image_path in image_paths:
        print(f"->␣{image_path}")
        image_cfg = ImageInputConfig(
            path=str(image_path),
            rotate_deg=STAR_ROTATE_DEG,
            resize_to=STAR_RESIZE_TO,
        )
        # Build mask that keeps the overlap region and removes
            borders
        mask = _load_precomputed_mask(image_path)
        gray_for_mask = None
        if mask is None:
            if STAR_AUTO_OVERLAP_MASK:
                gray_for_mask = cv2.imread(str(image_path), cv2.
                    IMREAD_GRAYSCALE)
                if gray_for_mask is None:
                    raise RuntimeError(f"Could␣not␣read␣image␣for␣
                        mask:␣{image_path}")
                mask = _quadrilateral_mask_from_image(
                    gray_for_mask,
                    thresh_percentile=STAR_AUTO_MASK_PERCENTILE,
                    close_kernel=STAR_AUTO_MASK_CLOSE_KERNEL,
                    erode_px=STAR_AUTO_MASK_ERODE,
                )
                if STAR_SAVE_DEBUG_INTERMEDIATES:
                    _write_debug_images("mask_build", {"auto_mask":
                        mask, "input_for_mask": gray_for_mask})
            elif STAR_MASK_BORDERS_PX:
```

```
                top, bottom, left, right = STAR_MASK_BORDERS_PX
                gray_for_mask = cv2.imread(str(image_path), cv2.IMREAD_GRAYSCALE)
                if gray_for_mask is None:
                    raise RuntimeError(f"Could not read image for mask: {image_path}")
                H, W = gray_for_mask.shape
                mask = np.zeros((H, W), dtype=np.uint8)
                mask[top : H - bottom, left : W - right] = 1

        # Optional Jupiter/glare mask; combine and save for preview
        if STAR_SAVE_JUPITER_MASK:
            if gray_for_mask is None:
                gray_for_mask = cv2.imread(str(image_path), cv2.IMREAD_GRAYSCALE)
            if gray_for_mask is not None:
                glare_mask = _compute_jupiter_mask(gray_for_mask)
                # Save glare mask for preview (even if None -> save all zeros)
                STAR_DEBUG_DIR.mkdir(parents=True, exist_ok=True)
                glare_path = STAR_DEBUG_DIR / f"{image_path.stem}_jupiter_mask.png"
                if glare_mask is None:
                    blank = np.zeros_like(gray_for_mask, dtype=np.uint8)
                    cv2.imwrite(str(glare_path), blank)
                else:
                    cv2.imwrite(str(glare_path), glare_mask * 255)
                    # Apply to detection mask (AND with inverse)
                    if mask is None:
                        mask = np.ones_like(glare_mask, dtype=np.uint8)
                    mask = (mask.astype(np.uint8) & (1 - glare_mask).astype(np.uint8)).astype(np.uint8)
```

```
                        if STAR_SAVE_DEBUG_INTERMEDIATES:
                            overlay = _make_overlay(glare_mask * 255, _to_uint8_debug(gray_for_mask))
                            cv2.imwrite(str(STAR_DEBUG_DIR / f"{image_path.stem}_jupiter_mask_overlay.png"), overlay)

        output_path = image_path.with_name(f"{image_path.stem}_annotated.png")
        run_star_tracker(
            image_path=str(image_path),
            catalog_path=str(STAR_CATALOG_PATH),
            max_catalog_mag=STAR_MAX_CATALOG_MAG,
            output_path=str(output_path),
            processing_options=processing_opts,
            image_config=image_cfg,
            camera_params_override=STAR_CAMERA_OVERRIDE,
            denoise_config=STAR_DENOISE_SETTINGS if APPLY_DENOISING else None,
            detection_mask=mask,
        )


if __name__ == "__main__":
    main()
```

Listing A.2: Star tracker script