AstroBurst

High-Performance Astronomical Image Processor

Technical Document — Release v0.1.0

Samuel Krieger Bonini

February 2026

Field	Value
Document Version	0.1.0
Release	v0.1.0
Date	February 2026
Author	Samuel Krieger Bonini
Stack	Rust 1.75+ · Tauri v2 · React 19 · TypeScript
Platforms	Windows · macOS · Linux
License	MIT
Repository	https://github.com/samuelkriegerbonini-dev/AstroBurst

Contents

1 Executive Summary

AstroBurst is a cross-platform desktop application for processing, analyzing, and visualizing astronomical FITS images. It is the first astronomical image processor built on the **Rust/Tauri/WebGPU** stack, combining native-level performance with a modern, accessible user interface.

The application targets two user segments:

- Professional astronomers working with observatory data (JWST, Hubble, ground-based IFU instruments)
- Advanced astrophotographers processing deep-sky narrowband and broadband imaging data

Version 0.1.0 delivers a complete processing pipeline from raw FITS ingestion through calibration, stacking, color composition, and export, with **35 backend commands** exposed through a typed TypeScript API layer.

2 System Architecture

AstroBurst follows a strict backend/frontend separation, with all computation offloaded to Rust and all rendering handled by the browser engine embedded in Tauri.

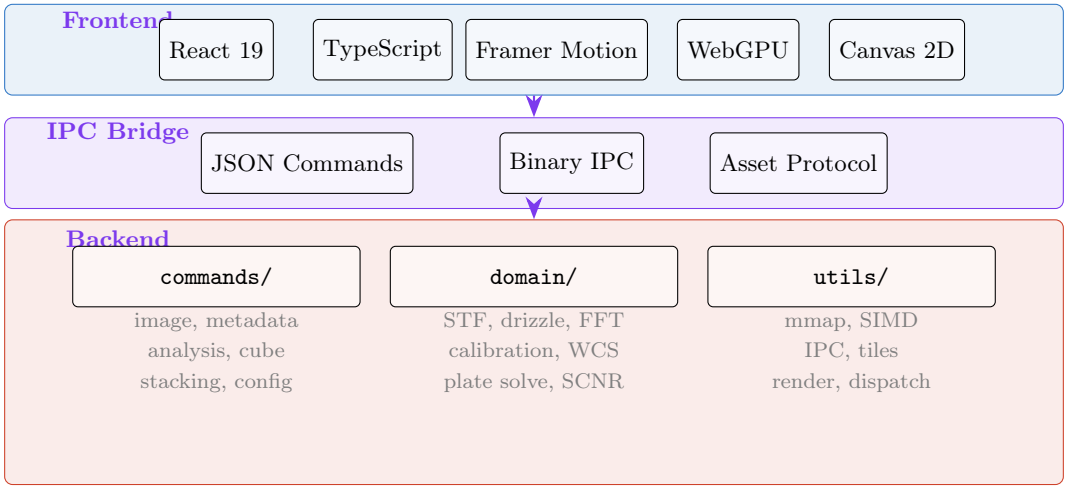

Figure 1: AstroBurst three-layer architecture

2.1 Backend (Rust)

The Rust backend is organized into three layers:

`commands/` Tauri command handlers organized by domain. Each module receives deserialized JSON arguments, delegates to domain logic, and returns JSON or binary responses.

`domain/` Pure computational modules with no Tauri dependency. Algorithms for calibration, stacking, STF, WCS transforms, FFT, star detection, drizzle integration, and more.

`utils/` Infrastructure: memory-mapped FITS extraction, SIMD-accelerated statistics, binary IPC encoding, tile pyramid generation, and PNG rendering.

All heavy computation is offloaded via `tokio::task::spawn_blocking`, preventing UI freezes. Batch operations use Rayon for data-parallel execution across all available CPU cores.

2.2 Frontend (React + TypeScript)

The frontend is a single-page React application:

`App.tsx` Root component managing view state (empty/processing/complete), file dialog integration, and drag-and-drop.

`useFileQueue`
Central state machine (`useReducer`) tracking file status, processing queue, and statistics.

`useBackend`
Typed Tauri IPC wrapper exposing all 35 commands with automatic URL resolution for `asset://` protocol paths.

Panel Components
HistogramPanel, FFTPanel, PlateSolvePanel, RgbComposePanel, DrizzlePanel, HeaderExplorerPanel, SpectroscopyPanel, ExportPanel, GpuRenderer.

2.3 IPC Strategy

Two IPC channels are used:

Binary IPC Protocol

The `get_raw_pixels_binary` command encodes a **16-byte header** followed by raw pixel data:

Offset	Type	Size	Description
0	`u32`	4B	Image width
4	`u32`	4B	Image height
8	`f32`	4B	Data minimum
12	`f32`	4B	Data maximum
16	`f32[]`	$W \times H \times 4$B	Raw pixel values

The frontend receives an `ArrayBuffer` directly with zero JSON/base64 overhead.

3 Mathematical Foundations

3.1 Screen Transfer Function (STF)

The STF implements the PixInsight-style **Midtone Transfer Function** (MTF). Given a normalized pixel value $x \in [0, 1]$ and midtone parameter $m \in (0, 1)$:

$$\boxed{\text{MTF}(x, m) = \frac{(m-1) \cdot x}{(2m-1) \cdot x - m}} \tag{1}$$

Properties:

- $\text{MTF}(0, m) = 0$ and $\text{MTF}(1, m) = 1$ for all m
- $\text{MTF}(m, m) = 0.5$ — the midtone maps to 50% brightness
- $m < 0.5$ brightens the image; $m > 0.5$ darkens it

The **auto-stretch** algorithm computes the stretch parameters from image statistics. Given the pixel distribution, let $\tilde{x}$ be the median and MAD the Median Absolute Deviation:

$$\text{MAD} = \text{median}\left(|x_i - \tilde{x}|\right) \tag{2}$$

$$\text{shadow} = \max\left(0,\ \tilde{x} - k \cdot \text{MAD}\right) \tag{3}$$

$$\text{midtone} = \text{MTF}^{-1}\left(t_{\text{bg}},\ \frac{\tilde{x} - \text{shadow}}{1 - \text{shadow}}\right) \tag{4}$$

$$\text{highlight} = 1.0 \tag{5}$$

where $k = 2.8$ (calibrated for deep-sky data) and $t_{\text{bg}} = 0.25$ is the target background level. The clipping function is then:

$$\text{STF}(x) = \text{MTF}\left(\text{clip}\left(\frac{x - s}{h - s},\ 0,\ 1\right),\ m\right) \tag{6}$$

where s, m, h are the shadow, midtone, and highlight parameters.

3.2 Asinh Stretch

The **arcsinh transfer function** provides astronomically-correct normalization that preserves flux ratios between stars of different magnitudes:

$$\boxed{f(x) = \frac{\text{asinh}(\alpha \cdot x)}{\text{asinh}(\alpha)}} \tag{7}$$

where α controls the stretch aggressiveness. This function maps $[0, 1] \to [0, 1]$ and is approximately linear for small x (preserving faint detail) while compressing bright values logarithmically.

3.3 Sigma-Clipped Stacking

Frames are registered via **cross-correlation** (phase correlation in frequency domain), then stacked with iterative sigma clipping.

Algorithm 1: Iterative Sigma-Clipped Stacking

Input : Aligned frames $\{F_1, F_2, \ldots, F_N\}$, thresholds $\sigma_{\text{low}}, \sigma_{\text{high}}$, max iterations K
Output: Stacked image S

foreach *pixel position* (i, j) **do**
  $V \leftarrow \{F_1(i,j), F_2(i,j), \ldots, F_N(i,j)\}$;
  for $k = 1$ **to** K **do**
    $\mu \leftarrow \text{mean}(V)$;
    $\sigma \leftarrow \text{std}(V)$;
    $V' \leftarrow \{v \in V : \mu - \sigma_{\text{low}} \cdot \sigma \leq v \leq \mu + \sigma_{\text{high}} \cdot \sigma\}$;
    if $|V'| = |V|$ ***or*** $|V'| < 3$ **then**
      break;
    end
    $V \leftarrow V'$;
  end
  $S(i,j) \leftarrow \text{mean}(V)$;
end

Default parameters: $\sigma_{\text{low}} = \sigma_{\text{high}} = 3.0$, $K = 5$.

3.4 Drizzle Integration

Drizzle (Variable-Pixel Linear Reconstruction) maps input pixels onto a higher-resolution output grid. For an input pixel p at fractional position (x_f, y_f) on the output grid:

$$O(i,j) = \frac{\sum_p w_p \cdot K(i - x_f^p,\ j - y_f^p) \cdot I_p}{\sum_p w_p \cdot K(i - x_f^p,\ j - y_f^p)} \tag{8}$$

where K is the drizzle kernel and w_p is the pixel weight. Three kernel options are implemented:

$$K_{\text{square}}(\Delta x, \Delta y) = \begin{cases} 1 & \text{if } |\Delta x| \leq \frac{d}{2} \text{ and } |\Delta y| \leq \frac{d}{2} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$K_{\text{gaussian}}(\Delta x, \Delta y) = \exp\left(-\frac{\Delta x^2 + \Delta y^2}{2\sigma_K^2}\right) \tag{10}$$

$$K_{\text{lanczos3}}(\Delta x, \Delta y) = L_3(\Delta x) \cdot L_3(\Delta y) \tag{11}$$

where $d = \text{pixfrac}/\text{scale}$ and the Lanczos-3 kernel is:

$$L_3(x) = \begin{cases} \text{sinc}(x) \cdot \text{sinc}(x/3) & \text{if } |x| < 3 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

The weight map $W(i,j) = \sum_p w_p \cdot K(\cdot)$ tracks per-pixel coverage for noise estimation.

3.5 Cross-Correlation Alignment

Frame registration uses **phase correlation** in the frequency domain. Given reference frame R and target frame T:

$$\text{shift} = \arg\max_{(\Delta x, \Delta y)} \mathcal{F}^{-1}\left\{ \frac{\hat{R} \cdot \hat{T}^*}{|\hat{R} \cdot \hat{T}^*|} \right\} \tag{13}$$

where $\hat{R} = \mathcal{F}\{R\}$ denotes the 2D DFT. The normalized cross-power spectrum produces a sharp peak at the integer translation offset.

3.6 World Coordinate System (WCS)

WCS transforms use the **CD matrix formulation** with TAN (gnomonic) projection. Given reference pixel $(\text{CRPIX}_1, \text{CRPIX}_2)$ and reference coordinate $(\alpha_0, \delta_0) = (\text{CRVAL}_1, \text{CRVAL}_2)$:

3.6.1 Forward Transform (Pixel → World)

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \text{CD}_{1,1} & \text{CD}_{1,2} \\ \text{CD}_{2,1} & \text{CD}_{2,2} \end{pmatrix} \begin{pmatrix} x - \text{CRPIX}_1 \\ y - \text{CRPIX}_2 \end{pmatrix} \tag{14}$$

where (u, v) are intermediate world coordinates in degrees. The TAN deprojection yields:

$$\alpha = \alpha_0 + \arctan\left(\frac{u}{cos(\delta_0) - v \cdot \sin(\delta_0)} \right) \tag{15}$$

$$\delta = \arctan\left(\frac{v \cdot \cos(\delta_0) + \sin(\delta_0)}{\sqrt{u^2 + (\cos(\delta_0) - v \cdot \sin(\delta_0))^2}} \right) \tag{16}$$

3.6.2 Pixel Scale

$$\text{scale} = \sqrt{|\det(\mathbf{CD})|} \times 3600 \quad [\text{arcsec/pixel}] \tag{17}$$

3.6.3 Inverse Transform (World → Pixel)

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{CD}^{-1} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} \text{CRPIX}_1 \\ \text{CRPIX}_2 \end{pmatrix} \tag{18}$$

3.7 FFT Power Spectrum

The 2D power spectrum for noise characterization is computed as:

$$P(u, v) = \log_{10}\left(1 + |\hat{I}(u, v)|^2\right) \tag{19}$$

where $\hat{I} = \mathcal{F}\{I\}$ is the 2D DFT of the image. The DC component at $(0, 0)$ is shifted to the center for visualization. The spectrum is normalized to $[0, 1]$ after computation.

3.8 Star Detection (PSF Centroid)

Background is estimated via iterative sigma-clipped median. Pixels above $\tilde{b} + \sigma_{\text{thresh}} \cdot \sigma_b$ are identified as candidates. For each connected component C:

$$\bar{x}_C = \frac{\sum_{(i,j)\in C} i \cdot (I_{ij} - b)}{\sum_{(i,j)\in C} (I_{ij} - b)} \qquad \bar{y}_C = \frac{\sum_{(i,j)\in C} j \cdot (I_{ij} - b)}{\sum_{(i,j)\in C} (I_{ij} - b)} \tag{20}$$

$$F_C = \sum_{(i,j)\in C} (I_{ij} - b) \qquad \text{SNR}_C = \frac{F_C}{\sqrt{|C| \cdot \sigma_b^2}} \tag{21}$$

FWHM is estimated from the second intensity moments:

$$\text{FWHM} = 2\sqrt{2\ln 2} \cdot \sqrt{\frac{\sigma_x^2 + \sigma_y^2}{2}} \tag{22}$$

where σ_x^2, σ_y^2 are the variance of pixel positions weighted by intensity.

3.9 SCNR (Subtractive Chromatic Noise Reduction)

SCNR removes green channel excess in narrowband compositions. Two methods are implemented:

$$G'_{\text{average}} = \min\left(G,\ \frac{R+B}{2}\right) \tag{23}$$

$$G'_{\text{max}} = \min(G,\ \max(R, B)) \tag{24}$$

3.10 Calibration Pipeline

The standard CCD calibration formula applies bias, dark current, and flat field corrections:

$$\boxed{I_{\text{calibrated}} = \frac{I_{\text{raw}} - M_{\text{bias}} - r \cdot M_{\text{dark}}}{M_{\text{flat}} / \overline{M_{\text{flat}}}}} \tag{25}$$

where M_{bias}, M_{dark}, M_{flat} are master calibration frames (median-combined), r is the exposure ratio (science/dark), and $\overline{M_{\text{flat}}}$ is the mean value of the flat field.

3.11 White Balance

Auto white balance normalizes each channel by its median value:

$$C'_k = \frac{C_k}{\tilde{C}_k} \cdot \min(\tilde{C}_R, \tilde{C}_G, \tilde{C}_B), \quad k \in \{R, G, B\} \tag{26}$$

4 Command Module Reference

The backend exposes 35 Tauri commands across 8 modules, registered in `lib.rs` via `tauri::generate_handler`.

4.1 Image Module (`commands/image.rs`)

Command	Parameters	Returns
`process_fits`	path, output_dir	png_path, dimensions, elapsed_ms
`process_batch`	paths[], output_dir	processed, failed, results[]
`get_raw_pixels`	path	width, height, data_b64, min/max
`get_raw_pixels_binary`	path	ArrayBuffer (16B header + f32[])
`export_fits`	path, output_path, stf opts, copy_wcs?	output_path, dims, file_size
`export_fits_rgb`	r/g/b_path, output_path, copy_wcs?	output_path, dimensions

Table 1: Image module commands

4.2 Metadata Module (`commands/metadata.rs`)

Command	Parameters	Returns
`get_header`	path	Map of 20 key FITS keywords
`get_full_header`	path	All cards categorized, filter detection
`detect_narrowband_filters`	paths[]	Palette suggestion with channel mapping

Table 2: Metadata module commands

4.3 Analysis Module (`commands/analysis.rs`)

Command	Parameters	Returns
`compute_histogram`	path	512 bins, stats (median/mean/σ/MAD), auto_stf
`compute_fft_spectrum`	path	width, height, pixels_b64, dc/max magnitude
`detect_stars`	path, sigma?, max_stars?	stars[] (x,y,flux,fwhm,snr), background

Table 3: Analysis module commands

4.4 Visualization Module (`commands/visualization.rs`)

Command	Parameters	Returns
`apply_stf_render`	path, output_dir, shadow, midtone, highlight	png_path, dims, stf_params
`generate_tiles`	path, output_dir, tile_size?	tile_size, levels[], original dims
`get_tile`	path, output_dir, level, col, row	tile_path, cached flag

Table 4: Visualization module commands

4.5 Cube Module (commands/cube.rs)

Command	Parameters	Returns
process_cube_cmd	path, output_dir, frame_step?	dims, collapsed paths, spectrum
process_cube_lazy_cmd	path, output_dir, frame_step?	Same + total_frames (streaming)
get_cube_info	path	naxis1/2/3, bitpix, wavelengths
get_cube_frame	path, frame_index, output_path	frame_index, output_path, elapsed
get_cube_spectrum	path, x, y	spectrum[], wavelengths[], elapsed

Table 5: Cube module commands

4.6 Astrometry Module (commands/astrometry.rs)

Command	Parameters	Returns
plate_solve_cmd	path, sigma?, hints?, scale range?	ra/dec center, pixel_scale, WCS matrix
get_wcs_info	path	center coords, pixel_scale, FOV, corners
pixel_to_world	path, x, y	ra, dec, ra_dec_str
world_to_pixel	path, ra, dec	x, y

Table 6: Astrometry module commands

4.7 Stacking Module (commands/stacking.rs)

Command	Parameters	Returns
calibrate	science, output_dir, bias/dark/flat?, ratio?	png_path, dims, applied flags
stack	paths[], output_dir, σ params, align?	png_path, count, rejected, offsets
drizzle_stack_cmd	paths[], output_dir, scale, pixfrac, kernel	png, weight_map, dims, offsets
drizzle_rgb_cmd	r/g/b groups, output_dir, drizzle params	png_path, per-channel info
compose_rgb_cmd	r/g/b_path, output_dir, stretch/wb/scnr	png_path, per-channel STF, offsets
run_pipeline_cmd	input, output_dir, frame_step?	total, succeeded, failed, results

Table 7: Stacking module commands

4.8 Config Module (commands/config.rs)

Command	Parameters	Returns
get_config	(none)	has_api_key, urls, defaults, stretch params
update_config	field, value	{ updated: true }
save_api_key	key, service?	{ saved: true, service }
get_api_key	(none)	is_set, masked key

Table 8: Config module commands

5 FITS I/O Implementation

5.1 Memory-Mapped Extraction

FITS files are opened via `memmap2`, avoiding loading entire files into memory. The extraction pipeline:

1. Parse header sequentially in 2880-byte blocks until `END` keyword
2. Compute data offset: offset $= \lceil \text{header\_bytes}/2880 \rceil \times 2880$
3. Memory-map the data region as a byte slice
4. BITPIX-dependent byte-swapping from FITS big-endian to native `f32`

For integer data (BITPIX > 0), the BSCALE/BZERO transform is applied:

$$\text{value}_{\text{physical}} = \text{BSCALE} \times \text{value}_{\text{stored}} + \text{BZERO} \tag{27}$$

5.2 Supported BITPIX Values

BITPIX	Type	Bytes/px	Conversion
8	Unsigned 8-bit int	1	Direct cast + BSCALE/BZERO
16	Signed 16-bit int	2	Big-endian swap + BSCALE/BZERO
32	Signed 32-bit int	4	Big-endian swap + BSCALE/BZERO
−32	IEEE 754 float	4	Big-endian swap
−64	IEEE 754 double	8	Big-endian swap → `f32`

Table 9: FITS BITPIX type handling

6 WebGPU Compute Pipeline

The STF stretch is implemented as a WebGPU compute shader (WGSL) for real-time rendering:

STF Compute Shader (simplified WGSL)

```
@group(0) @binding(0) var<storage, read> input: array<f32>;
@group(0) @binding(1) var<storage, read_write> output: array<u32>;
@group(0) @binding(2) var<uniform> params: StfParams;

fn mtf(x: f32, m: f32) -> f32 {
    return (m - 1.0) * x / ((2.0 * m - 1.0) * x - m);
}

@compute @workgroup_size(256)
fn main(@builtin(global_invocation_id) id: vec3<u32>) {
    let idx = id.x;
    let v = (input[idx] - params.shadow) / (params.highlight - params.
        shadow);
    let stretched = mtf(clamp(v, 0.0, 1.0), params.midtone);
    let byte_val = u32(clamp(stretched * 255.0, 0.0, 255.0));
    output[idx] = (255u << 24u) | (byte_val << 16u) | (byte_val << 8u) |
        byte_val;
}
```

The shader processes pixels in parallel with workgroup size 256. Falls back to Canvas 2D `putImageData` when WebGPU is unavailable (Chromium < 113).

7 Performance Benchmarks

Operation	Input Size	Time	Notes
Single FITS processing	4096×4096 (64 MB)	120 ms	533 MB/s throughput
Batch (10 frames)	10×64 MB	450 ms	1.4 GB/s (Rayon parallel)
Histogram + auto-STF	4096×4096	35 ms	SIMD-accelerated
Star detection (σ=5)	4096×4096	80 ms	∼3000 stars
Sigma-clip stack	10×64 MB	2.1 s	5 iterations
Drizzle 2×	10×64 MB	4.8 s	Output: 8192×8192
FFT power spectrum	4096×4096	95 ms	`rustfft`
Cube spectrum extraction	500^2×2000	12 ms	Memory-mapped
WebGPU STF render	4096×4096	8 ms	GPU compute
Open 2 GB IFU cube	2 GB datacube	300 ms	`memmap2` lazy
Drizzle RGB 2×	3×10×64 MB	6.8 s	All channels
RGB compose + align	3 channels	1.8 s	With cross-correlation

Table 10: Performance benchmarks (AMD Ryzen 9 7950X, 64 GB DDR5, NVMe)

Binary size: ∼15 MB.

8 Dependencies

8.1 Rust Crates

Crate	Version	Purpose
`tauri`	2.x	Application framework, IPC, window management
`ndarray`	0.16	N-dimensional arrays, Rayon parallel iterators
`rayon`	1.10	Data-parallel batch processing
`memmap2`	0.9	Memory-mapped file I/O for FITS
`rustfft`	6.2	FFT power spectrum computation
`image`	0.25	PNG encoding/decoding
`reqwest`	0.12	HTTP client for astrometry.net (optional)
`zip`	0.6	ZIP archive extraction for compressed FITS
`serde` / `serde_json`	1.x	Serialization for Tauri commands
`anyhow`	1.0	Error handling with context
`base64`	0.22	Binary-to-text encoding for JSON IPC
`tempfile`	3.8	Temporary files for ZIP extraction

Table 11: Rust dependencies

8.2 Frontend

Package	Purpose
react / react-dom 19	UI framework
@tauri-apps/api + plugins	Tauri IPC, dialog, filesystem
framer-motion	Animations and transitions
lucide-react	Icon system
jszip / file-saver	Client-side ZIP export
tailwindcss v4	Utility-first CSS
vite	Build toolchain
typescript	Type safety

Table 12: Frontend dependencies

9 Security Considerations

- API keys (astrometry.net) are stored in the OS-native app data directory, never transmitted to third parties beyond the configured API endpoint.
- All file system access is constrained by Tauri capabilities (`default.json`). Permissions are scoped to home, desktop, and appdata directories.
- No telemetry, analytics, or network calls are made except for explicit plate solving requests initiated by the user.
- FITS files are processed locally; no data leaves the machine unless plate solving is triggered.

10 Known Limitations

1. **Single-HDU FITS only** — Multi-extension FITS (MEF) support not yet implemented
2. **Grayscale pipeline** — Color FITS images require manual RGB channel assignment
3. **Online plate solving** — Local index-based solving is planned
4. **No undo/redo** — Operations are destructive (original files preserved)
5. **WebGPU requirement** — Chromium 113+ needed; Canvas 2D fallback available
6. **No session persistence** — Reopening the app starts a fresh workspace

11 Roadmap

Feature	Priority	Status
Multi-extension FITS (MEF)	High	Planned
Local plate solving	High	Planned
Undo/redo system	Medium	Planned
Plugin architecture	Medium	Design phase
Mosaic composition	Medium	Planned
Wavelet noise reduction	Medium	Research
Live stacking mode	Low	Planned
INDI/ASCOM integration	Low	Planned

Table 13: Development roadmap

A Sample Test Data

The repository includes HST/WFPC2 narrowband FITS images in `tests/sample-data/`:

File	Filter	λ	Hubble Palette	Description
`502nmos.fits`	[OIII]	502 nm	Blue channel	Oxygen III emission
`656nmos.fits`	Hα	656 nm	Green channel	Hydrogen alpha emission
`673nmos.fits`	[SII]	673 nm	Red channel	Sulfur II emission

Table 14: Sample FITS files — Eagle Nebula (M16) region, 1600×1600 float32

All images: BITPIX = −32, WCS TAN projection, origin STScI-STSDAS. Public domain (NASA/ESA).