JWST Image Processing Implementation Research

Technical Research Document

JWST Data Analysis Application

Date: January 26, 2026

Version: 1.0

This document presents comprehensive research into image processing algorithms and techniques for analyzing James Webb Space Telescope (JWST) data, including recommended implementations using Python scientific computing libraries.

1. Executive Summary

This research document outlines the recommended approach for implementing image processing capabilities in the JWST Data Analysis Application. The research covers industry-standard astronomical image processing techniques, leveraging established Python libraries used by the Space Telescope Science Institute (STScI) and the broader astronomical community.

Key Findings

- The official JWST pipeline (jwst package) provides calibration but not post-pipeline analysis
- Photutils is the recommended library for background estimation, source detection, and photometry
- Astropy provides essential FITS handling and NaN-safe convolution operations
- SciPy ndimage offers efficient filtering algorithms for noise reduction
- A modular pipeline architecture allows flexible algorithm composition

Recommended Implementation Priority

- 1. Background Estimation & Subtraction (foundation for all analysis)
- 2. Noise Reduction Filters (Gaussian, Median, Astropy convolution)
- 3. Image Enhancement (ZScale, Asinh stretch, histogram equalization)
- 4. Source Detection (point sources via DAOFIND, extended via segmentation)
- 5. Statistical Analysis (sigma-clipped statistics, robust estimators)

2. Current System Analysis

2.1 Existing Infrastructure

The JWST Data Analysis Application currently has a Python-based processing engine with FastAPI endpoints. The following components are already implemented and functional:

Implemented Components

Component	Status	Location
FITS File I/O	Complete	app/processing/utils.py
Array Normalization	Complete	app/processing/utils.py
Preview Generation	Basic	main.py (ZScale only)
MAST Integration	Complete	app/mast/
Chunked Downloads	Complete	app/mast/chunked_downloader.py
Processing Stubs	Placeholder	main.py

2.2 Available Dependencies

The processing engine currently includes the following Python packages:

Package	Version	Purpose
NumPy	1.26.2	Array operations and numerical computing
SciPy	1.11.4	Scientific computing, signal processing, filtering
Astropy	6.0.0	FITS handling, WCS, astronomical utilities
Matplotlib	3.8.2	Visualization and image rendering
Pillow	10.1.0	Image format conversion
Pandas	2.1.4	Data manipulation and analysis

2.3 Missing Dependencies

The following packages are recommended additions to enable full astronomical image processing:

Package	Version	Purpose
photutils	>=1.10.0	Background estimation, source detection, photometry
scikit-image	>=0.22.0	Additional image processing algorithms (optional)

3. Image Processing Algorithms

3.1 Background Estimation & Subtraction

Accurate background estimation is the foundation of astronomical image analysis. The background in JWST images consists of sky emission, detector bias, and scattered light. Removing this background is essential for accurate photometry and source detection.

Recommended Approach: 2D Background Estimation

The photutils Background2D class implements mesh-based background estimation with sigma clipping. This approach divides the image into boxes, estimates local background in each box using robust statistics, and interpolates to create a full-resolution background model.

Key parameters for Background2D:

- box_size: Size of boxes for local estimation (typically 50-100 pixels)
- filter_size: Median filter window to smooth local estimates (typically 3-5)
- sigma_clip: Sigma threshold for outlier rejection (typically 3-sigma)
- bkg_estimator: Statistical estimator (MedianBackground recommended)

Implementation Example

```
from photutils.background import Background2D, MedianBackground
from astropy.stats import SigmaClip

def estimate_background(data, box_size=50, filter_size=3):
    """
    Estimate 2D background using mesh-based approach.

    Parameters:
        data: 2D numpy array of image data
        box_size: Size of estimation boxes in pixels
        filter_size: Median filter window size

    Returns:
        background: 2D background model
        background_rms: 2D background noise model
    """
    sigma_clip = SigmaClip(sigma=3.0, maxiters=10)
    bkg_estimator = MedianBackground()

    bkg = Background2D(
        data,
        box_size=box_size,
        filter_size=filter_size,
        sigma_clip=sigma_clip,
        bkg_estimator=bkg_estimator
    )

    return bkg.background, bkg.background_rms
```

3.2 Noise Reduction Algorithms

JWST images contain various noise sources including readout noise, dark current, cosmic rays, and photon noise. Different filtering techniques are optimal for different noise characteristics.

Available Methods

Method	Best For	Library
Gaussian Filter	General smoothing, white noise	scipy.ndimage
Median Filter	Salt & pepper noise, cosmic rays	scipy.ndimage
Bilateral Filter	Edge-preserving smoothing	scipy.ndimage
Astropy Convolve	NaN-safe Gaussian smoothing	astropy.convolution
Sigma Clipping	Statistical outlier removal	astropy.stats

Critical Note: NaN Handling

JWST data frequently contains NaN values from bad pixels, cosmic ray hits, or masked regions. Standard scipy.ndimage filters propagate NaN values, making them unsuitable for direct use. The astropy.convolution module properly handles NaN values by ignoring them during convolution and optionally interpolating across them.

Implementation Example

```
from scipy.ndimage import gaussian_filter, median_filter
from astropy.convolution import convolve, Gaussian2DKernel

def reduce_noise(data, method='gaussian', kernel_size=3, sigma=1.0):
    """
    Apply noise reduction filter to image data.

    Parameters:
        data: 2D numpy array
        method: 'gaussian', 'median', or 'astropy_gaussian'
        kernel_size: Size of filter kernel
        sigma: Standard deviation for Gaussian filters

    Returns:
        Filtered image data
    """
    if method == 'gaussian':
        # Fast but propagates NaN
        return gaussian_filter(data, sigma=sigma)

    elif method == 'median':
        # Good for cosmic rays, propagates NaN
        return median_filter(data, size=kernel_size)

    elif method == 'astropy_gaussian':
        # Properly handles NaN values
        kernel = Gaussian2DKernel(x_stddev=sigma)
        return convolve(data, kernel, nan_treatment='interpolate')
```

3.3 Image Enhancement Techniques

Astronomical images have extreme dynamic ranges that require specialized visualization techniques. Standard linear scaling cannot display both faint nebulosity and bright stars simultaneously.

Scaling Methods

Method	Description	Use Case
ZScale	IRAF algorithm, samples pixels	General astronomical images
Asinh Stretch	Inverse hyperbolic sine	High dynamic range, galaxies
Log Stretch	Logarithmic scaling	Extended emission, nebulae
Sqrt Stretch	Square root scaling	Moderate dynamic range
Histogram Eq.	Equalize histogram	Maximize contrast
Power Stretch	Power law scaling	Customizable enhancement

Implementation Example

```
from astropy.visualization import (
    ZScaleInterval, AsinhStretch, LogStretch,
    SqrtStretch, HistEqStretch, ImageNormalize
)
import numpy as np

def enhance_image(data, method='zscale', **kwargs):
    """
    Apply contrast enhancement to astronomical image.

    Parameters:
        data: 2D numpy array
        method: Enhancement method name
        **kwargs: Method-specific parameters

    Returns:
        Enhanced image (0-1 range)
    """
    # Handle NaN values
    valid_data = data[~np.isnan(data)]

    if method == 'zscale':
        interval = ZScaleInterval(
            contrast=kwargs.get('contrast', 0.25)
        )
        vmin, vmax = interval.get_limits(valid_data)
        norm = ImageNormalize(vmin=vmin, vmax=vmax)
        return norm(data)

    elif method == 'asinh':
        stretch = AsinhStretch(a=kwargs.get('a', 0.1))
        normalized = (data - np.nanmin(data)) / np.nanptp(data)
        return stretch(normalized)

    elif method == 'log':
        stretch = LogStretch(a=kwargs.get('a', 1000))
        normalized = (data - np.nanmin(data)) / np.nanptp(data)
        return stretch(normalized)

    elif method == 'histogram_eq':
        stretch = HistEqStretch(valid_data)
        normalized = (data - np.nanmin(data)) / np.nanptp(data)
        return stretch(normalized)
```

3.4 Source Detection

Source detection identifies astronomical objects in images. Two primary approaches exist: point source detection optimized for stars, and segmentation-based detection for extended sources.

Point Source Detection (DAOFIND)

The DAOFIND algorithm (Stetson 1987) detects stellar sources by convolving the image with a Gaussian kernel matching the expected PSF and identifying local maxima above a threshold. This is implemented in photutils as DAOStarFinder.

Segmentation-Based Detection

For extended sources (galaxies, nebulae), image segmentation identifies connected regions of pixels above a threshold. The detect_sources function creates a segmentation map where each source is assigned a unique integer label.

Implementation Example

```
from photutils.detection import DAOStarFinder
from photutils.segmentation import detect_sources, deblend_sources

def detect_stars(data, background, fwhm=3.0, threshold_sigma=5.0):
    """
    Detect point sources using DAOFIND algorithm.

    Parameters:
        data: Background-subtracted image
        background: Background2D object
        fwhm: Expected FWHM of stars in pixels
        threshold_sigma: Detection threshold in sigma

    Returns:
        Table of detected sources with positions and fluxes
    """
    threshold = threshold_sigma * background.background_rms_median

    finder = DAOStarFinder(
        fwhm=fwhm,
        threshold=threshold,
        sharplo=0.2, sharphi=1.0,  # Shape constraints
        roundlo=-1.0, roundhi=1.0
    )

    sources = finder(data - background.background)
    return sources

def detect_extended(data, background, npixels=10, threshold_sigma=2.0):
    """
    Detect extended sources via segmentation.
    """
    threshold = background.background + (
        threshold_sigma * background.background_rms
    )

    # Initial detection
    segm = detect_sources(data, threshold, npixels=npixels)
    # Deblend overlapping sources
    segm_deblend = deblend_sources(
        data, segm, npixels=npixels,
        nlevels=32, contrast=0.001
    )

    return segm_deblend
```

3.5 Statistical Analysis

Robust statistical analysis is essential for characterizing astronomical data. Standard statistics (mean, standard deviation) are biased by outliers and source contamination. Astronomical analysis requires robust estimators that are resistant to these effects.

Robust Estimators

Estimator	Measures	Advantages
Sigma-clipped Mean	Central tendency	Removes outliers iteratively
Median	Central tendency	Robust to outliers, simple
Biweight Location	Central tendency	Most robust, handles asymmetry
MAD Std	Dispersion	Robust standard deviation estimate
Biweight Scale	Dispersion	Most robust dispersion estimate

Implementation Example

```
from astropy.stats import (
    sigma_clipped_stats, biweight_location,
    biweight_scale, mad_std
)
import numpy as np

def compute_statistics(data, mask=None, sigma=3.0):
    """
    Compute robust statistics for astronomical image.

    Parameters:
        data: 2D numpy array
        mask: Boolean mask (True = exclude)
        sigma: Sigma threshold for clipping

    Returns:
        Dictionary of statistical measures
    """
    # Apply mask if provided
    if mask is not None:
        valid_data = data[~mask & ~np.isnan(data)]
    else:
        valid_data = data[~np.isnan(data)]

    # Sigma-clipped statistics
    mean, median, std = sigma_clipped_stats(
        valid_data, sigma=sigma, maxiters=5
    )

    # Robust estimators
    bwloc = biweight_location(valid_data)
    bwscale = biweight_scale(valid_data)
    mad = mad_std(valid_data)

    return {
        'mean': float(mean),
        'median': float(median),
        'std': float(std),
        'biweight_location': float(bwloc),
        'biweight_scale': float(bwscale),
        'mad_std': float(mad),
        'min': float(np.nanmin(data)),
        'max': float(np.nanmax(data)),
        'n_pixels': int(valid_data.size),
        'n_nan': int(np.sum(np.isnan(data)))
    }
```

4. Proposed Architecture

4.1 Module Structure

The recommended architecture organizes processing capabilities into focused modules, each handling a specific aspect of image processing. This enables flexible composition of algorithms into processing pipelines.

```
processing-engine/app/processing/
    __init__.py           # Module exports
    utils.py              # FITS I/O (existing)
    background.py         # Background estimation & subtraction
    filters.py            # Noise reduction filters
    enhancement.py        # Contrast & scaling
    detection.py          # Source detection
    statistics.py         # Statistical analysis
    photometry.py         # Aperture/PSF photometry (future)
    pipeline.py           # Algorithm composition
```

4.2 Processing Pipeline Design

A typical JWST image processing pipeline follows this flow:

```
Input FITS File
       |
       v
+------------------+
| Load FITS Data   |  --> Header metadata extraction
+------------------+
       |
       v
+------------------+
| Background       |  --> Background model + RMS map
| Estimation       |
+------------------+
       |
       v
+------------------+
| Background       |  --> Science-ready image
| Subtraction      |
+------------------+
       |
       v
+------------------+
| Noise Reduction  |  --> Smoothed image (optional)
| (optional)       |
+------------------+
       |
       v
+------------------+
| Source Detection |  --> Source catalog (optional)
| (optional)       |
+------------------+
       |
       v
+------------------+
| Enhancement      |  --> Display-ready image
| for Display      |
+------------------+
       |
       v
+------------------+     +------------------+
| Save Processed   | --> | PNG Preview      |
| FITS             |     | Generation       |
+------------------+     +------------------+
```

4.3 API Endpoint Design

The processing engine exposes RESTful endpoints for each algorithm category. The existing /process endpoint can be extended with new algorithm types.

Endpoint	Method	Description
/process	POST	Execute processing algorithm
/algorithms	GET	List available algorithms
/preview/{id}	GET	Generate PNG preview
/statistics/{id}	GET	Compute image statistics
/background/{id}	POST	Estimate and subtract background
/detect/{id}	POST	Detect sources in image

4.4 Algorithm Registry

The /algorithms endpoint should return a comprehensive registry of available processing algorithms with their parameters:

```
{
  "algorithms": [
    {
      "name": "background_subtraction",
      "category": "preprocessing",
      "description": "Estimate and subtract 2D background",
      "parameters": {
        "box_size": {"type": "integer", "default": 50},
        "filter_size": {"type": "integer", "default": 3},
        "sigma_clip": {"type": "float", "default": 3.0}
      }
    },
    {
      "name": "noise_reduction",
      "category": "filtering",
      "description": "Apply noise reduction filter",
      "parameters": {
        "method": {"type": "string", "enum": ["gaussian",
                   "median", "astropy_gaussian"]},
        "kernel_size": {"type": "integer", "default": 3},
        "sigma": {"type": "float", "default": 1.0}
      }
    },
    {
      "name": "source_detection",
      "category": "analysis",
      "description": "Detect sources in image",
      "parameters": {
        "method": {"type": "string", "enum": ["daofind",
                   "segmentation"]},
        "threshold_sigma": {"type": "float", "default": 5.0},
        "fwhm": {"type": "float", "default": 3.0}
      }
    }
  ]
}
```

5. Implementation Roadmap

5.1 Phase 1: Foundation

Establish core infrastructure and dependencies.

- Add photutils to requirements.txt
- Create module structure (background.py, filters.py, etc.)
- Implement background estimation with Background2D
- Add comprehensive unit tests

5.2 Phase 2: Core Algorithms

Implement primary image processing algorithms.

- Noise reduction filters (Gaussian, Median, Astropy convolution)
- Image enhancement (ZScale, Asinh, Log, Histogram equalization)
- Statistical analysis (sigma-clipped stats, robust estimators)
- Update /algorithms endpoint with new capabilities

5.3 Phase 3: Source Detection

Add source detection capabilities.

- Point source detection with DAOStarFinder
- Extended source detection with segmentation
- Source catalog generation and storage
- Integration with frontend visualization

5.4 Phase 4: Advanced Features

Extend with advanced analysis capabilities.

- Pipeline composition (chain multiple algorithms)
- Batch processing support
- Aperture photometry
- PSF photometry (if needed)

6. References & Resources

6.1 Official Documentation

- JWST Pipeline: https://github.com/spacetelescope/jwst
- JWST User Documentation: https://jwst-docs.stsci.edu/
- STScI JDAT Notebooks: https://spacetelescope.github.io/jdat_notebooks/
- Photutils: https://photutils.readthedocs.io/
- Astropy: https://docs.astropy.org/

6.2 Key Libraries

Library	Documentation URL
photutils	https://photutils.readthedocs.io/en/stable/
astropy	https://docs.astropy.org/en/stable/
scipy.ndimage	https://docs.scipy.org/doc/scipy/reference/ndimage.html
numpy	https://numpy.org/doc/stable/

6.3 Scientific References

- Stetson, P.B. 1987, PASP, 99, 191 (DAOFIND algorithm)
- Bertin, E. & Arnouts, S. 1996, A&AS, 117, 393 (SExtractor)
- Bradley, L. et al. 2022, astropy/photutils (Photutils)

6.4 Community Resources

- STScI GitHub: https://github.com/spacetelescope
- Astropy GitHub: https://github.com/astropy
- JWST Help Desk: https://jwsthelp.stsci.edu/