Nancy Grace Roman Space Telescope (Roman)
Technical Report

<table>
<thead>
<tr>
<th>Title: Roman SOC Science Validation Report for SOC Release 2 DMS</th>
<th>Doc #: Roman-STScI-000619, SE-01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date: March 6, 2024</td>
<td>Date:</td>
</tr>
<tr>
<td>Rev: -</td>
<td>Rev: -</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Authors: Desjardins, T. D., Betti, S., Cosentino, R. G., Hoffmann, S. L., Otor, O. J., Sánchez, J., and Sharma, S.</th>
<th>Phone: (667) 218-6457</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release Date: April 10, 2024</td>
<td>Release Date:</td>
</tr>
</tbody>
</table>

Operated by the Association of Universities for Research in Astronomy, Inc., for the National Aeronautics and Space Administration under Contract #80GSFC19C0054

Check with the SOCCER Database at: [https://soccer.stsci.edu](https://soccer.stsci.edu)
To verify that this is the current version.
# Table of Contents

1. Executive Summary ........................................................................................................... - 3 -
2. Introduction .......................................................................................................................... - 3 -
3. Scope of Testing .................................................................................................................. - 4 -
4. Requirements Tested ........................................................................................................... - 5 -
   4.1 SOC-581: Mosaic Astrometric Frame ........................................................................... - 5 -
      4.1.1 Test Data Description ............................................................................................ - 6 -
      4.1.2 Testing Strategy .................................................................................................... - 6 -
      4.1.3 Results .................................................................................................................. - 7 -
   4.2 SOC-582: Mosaic Information ...................................................................................... - 11 -
      4.2.1 Test Data Description ............................................................................................ - 11 -
      4.2.2 Testing Strategy .................................................................................................... - 12 -
      4.2.3 Results .................................................................................................................. - 12 -
   4.3 SOC-666 and SOC-669: WFI Level 2 Science Data Products ...................................... - 14 -
      4.3.1 Test Data Description ............................................................................................ - 14 -
      4.3.2 Testing Strategy .................................................................................................... - 15 -
      4.3.3 Results .................................................................................................................. - 17 -
5. Conclusions ....................................................................................................................... - 20 -
6. Bibliography ....................................................................................................................... - 21 -
1 Executive Summary

This report contains the scientific validation of the following Science Operations Center (SOC) Data Management Subsystem (DMS) Level 4 systems engineering requirements for SOC Release 2. All requirements tested in this report have successfully passed scientific validation.

<table>
<thead>
<tr>
<th>SOC Level 4 Requirement ID</th>
<th>DMS Level 5 Requirement ID(s)</th>
<th>Level 4 Requirement</th>
<th>Validation Result [Pass/Fail]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOC-581</td>
<td>DMS-342</td>
<td>In Wide Field Imaging Mode, the DMS shall generate mosaic images using coordinates tied to the astrometric frame defined by the International Celestial Reference Frame (ICRF).</td>
<td>Pass</td>
</tr>
<tr>
<td>SOC-582</td>
<td>DMS-343, DMS-344, DMS-345</td>
<td>In Wide Field Imaging Mode, the DMS shall generate mosaic images that include the following information: (a) Data quality and uncertainty information (b) Total exposure time (c) Metadata used in the mosaic generation process.</td>
<td>Pass</td>
</tr>
<tr>
<td>SOC-666</td>
<td>DMS-280, DMS-281</td>
<td>The DMS shall generate WFI Level 2 science data products in Wide Field Imaging Mode.</td>
<td>Pass</td>
</tr>
<tr>
<td>SOC-669</td>
<td>DMS-278, DMS-279</td>
<td>The DMS shall generate WFI Level 2 science data products in Wide Field Spectroscopy Mode.</td>
<td>Pass</td>
</tr>
</tbody>
</table>

2 Introduction

In February 2024, the Roman Telescope Branch (RTB) in the Instruments Division at the Space Telescope Science Institute (STScI) performed scientific validation of Element Level requirements levied on the Roman Science Operations Center (SOC) included in Release 2. SOC Release 2 included development work that was delivered for testing starting in SOC Build 23Q1_B8 up to and including SOC Build 24Q1_B12 (September 15, 2022 through December 15, 2023). The scientific validation includes two subsystems at the SOC: the Data Management Subsystem (DMS) and the Planning and Scheduling Subsystem (PSS). This report concerns the DMS science validation, while a companion report (Schultz, et al. 2024) addresses the validation of the PSS. The validation of the DMS requirements in SOC Release 1 may be found in (Desjardins, et al. 2023).

Two of the DMS requirements validated in Release 2 are related to the creation of Level 2 science data products, i.e., data that have been corrected for detector-related calibration effects, have been cleaned of jumps (cosmic-rays), and have been converted from ramp cubes to rate images. The remaining two requirements concern Level 3 science data products in the WFI imaging mode. Level 3 products are created after Level 2 products have been resampled onto a
new coordinate frame and, optionally, combine multiple Level 2 products into a larger mosaic image.

Science validation by the RTB differs from the verification process performed by Integration and Test in that the validation checks that specific inputs produce scientifically correct outputs that meet the user’s expectations. Both verification and validation are important and necessary steps in assessing the successful development of the Roman DMS and documenting that development with the Roman Project.

To perform the validation, requirements were assigned to testers who developed a testing strategy, determined success criteria, and created code to evaluate the tests on the DMS software released at the end of SOC Build 24Q1_B12. All validation code was run by the testers on the same virtual machine. Use of a common virtual machine and computing environment settings ensured that all tests were performed using identical environments.

In the Release 2 scientific validation, there was one pair of requirements (SOC-666 and 669) that were functionally the same with respect to the scientific validation; therefore, we group these requirements into a combined pair for testing purposes. This yields a total of three groups of validation tests that are described in the sections below.

In addition to the following sections that describe the overall scope of testing, testing strategies, test data, and test results, we also include the following appendices at the end of the document: Appendix A describes the virtual machine used to conduct the testing, as well as the software environment and any additional configuration information for the tests; Appendix B contains a manifest of test artifacts; and Appendices C through G reproduce the code used for the validation tests.

3 Scope of Testing

As previously stated, the scientific validation performed by the RTB checked that for a given input value, the correct outputs were generated, or that the correct algorithm was applied to the inputs. The RTB validation was performed against SOC Level 4 requirements that were assigned to the DMS. Note that not all Level 4 requirements in Release 2 required scientific validation, and that those included in this report are a subset of the DMS requirements that were identified for scientific validation by one or more of Roman SOC Systems Engineering, Integration and Test, the DMS, and/or the RTB. Table 3-1 contains a list of the Level 4 requirements validated in this report, the requirement text, and the corresponding romanocal step if applicable.

<table>
<thead>
<tr>
<th>SOC Level 4 Requirement ID</th>
<th>Level 4 Requirement</th>
<th>romanocal Step</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOC-581</td>
<td>In Wide Field Imaging Mode, the DMS shall generate mosaic images using coordinates tied to the astrometric frame defined by the International Celestial Reference Frame (ICRF).</td>
<td>ResampleStep</td>
<td></td>
</tr>
</tbody>
</table>

Released via SOCCER Database at: https://soccer.stsci.edu

- 4 -
### SOC Level 4 Requirement ID

<table>
<thead>
<tr>
<th>SOC Level 4 Requirement ID</th>
<th>Level 4 Requirement</th>
<th>romanca1 Step</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOC-582</td>
<td>In Wide Field Imaging Mode, the DMS shall generate mosaic images that include the following information: (a) Data quality and uncertainty information (b) Total exposure time (c) Metadata used in the mosaic generation process.</td>
<td>ResampleStep</td>
<td></td>
</tr>
<tr>
<td>SOC-666</td>
<td>The DMS shall generate WFI Level 2 science data products in Wide Field Imaging Mode.</td>
<td>DQInitStep, SaturationStep, LinearityStep, RefPixStep, DarkCurrentStep, RampFitStep</td>
<td>Validated with the same method as SOC-669.</td>
</tr>
<tr>
<td>SOC-669</td>
<td>The DMS shall generate WFI Level 2 science data products in Wide Field Spectroscopy Mode.</td>
<td>DQInitStep, SaturationStep, LinearityStep, RefPixStep, DarkCurrentStep, RampFitStep</td>
<td>Validated with the same method as SOC-666.</td>
</tr>
</tbody>
</table>

### 4 Requirements Tested

#### 4.1 SOC-581: Mosaic Astrometric Frame

For this requirement, the Level 3 (L3) mosaic products created by high-level data processing must have a WCS frame defined by the International Celestial Reference Frame (ICRF). While processing data from Level 1 (L1) to Level 2 (L2), exposure-level processing aligns images with the Gaia DR3 astrometric frame, which itself is aligned with ICRF3 (i.e., the third realization of the ICRF) to within ~0.01 mas RMS (Gaia Collaboration, et al. 2021). Note that future pipeline versions will likely adopt the Gaia DR4 astrometric frame as a reference when it becomes available prior to Roman operations. Alignment of L2 data products with respect to Gaia astrometry is described by requirement SOC-1031, which has not yet been validated and, at the time of writing, is not planned for delivery until SOC build 27Q2_B25 (SOC Release 5). Requirement SOC-1031 states:

In Wide Field Imaging Mode, the DMS shall produce Level 2 science data products consistent with the Gaia astrometric reference frame.

To validate SOC-581, we assume that the WCS information of L2 products has already been matched to the Gaia reference frame. By confirming that the positions of sources match between the input L2 and output L3 products (i.e., that source position is preserved by the
romancal.resample code), we satisfy requirement SOC-581.

### 4.1.1 Test Data Description

For this test, we use images of two types of sources: 1) single-pixel sources; and 2) simulated point sources using Roman model point spread functions (PSFs). In both cases, the images contain a regularly-spaced grid of sources with known input positions and source fluences. We simulate only a single detector (WFI01) in our test.

The single-pixel source image contains bright pixels spaced 100 pixels apart in both the X and Y directions of the L2 rate image. The bright pixels have fluxes of 5000 electrons per second per pixel. The image is free of cosmic rays, and the background level is negligible. The romancal.resample code requires an association file as input with more than one input image, and thus we use an association file with the same image repeated twice.

The images containing simulated Roman point sources use a grid of sources defined in right ascension (\( \alpha \)) and declination (\( \delta \)) to be approximately 250 pixels apart in both directions when projected onto the WFI (assuming pixel scale of 0.11 arcseconds per pixel). We choose a larger grid size compared to the single-pixel sources to minimize overlap between point spread functions (PSFs). The input catalog is centered at \(( \alpha, \delta ) = (80.0, 0.0)\) degrees. We simulate L2 products containing this grid of point sources by “pointing” the WFI at the center of this catalog, and then dither using a four-point gap-filling dither pattern, which is described in Table 4-1. For the test, the WFI was rotated on the simulated sky by 60 degrees such that the y-axis of the WFI focal plane is aligned with celestial North. The simulated point sources each contain an integrated fluence of 5000 electrons per second.

In all simulated L2 images, we use the romancal.assign_wcs.AssignWcsStep of the WFI exposure pipeline to generate a gwcs.wcs.WCS object that includes the distortion model provided by the Goddard Space Flight Center Optics Lab. This distortion model is stored in the Calibration Reference Data System (CRDS) distortion reference files in roman_0053.pmap on the Roman CRDS Ops server.

<table>
<thead>
<tr>
<th>Step</th>
<th>X Offset (arcseconds)</th>
<th>Y Offset (arcseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.55</td>
<td>103.00</td>
</tr>
<tr>
<td>3</td>
<td>24.00</td>
<td>103.55</td>
</tr>
<tr>
<td>4</td>
<td>24.55</td>
<td>0.55</td>
</tr>
</tbody>
</table>

### 4.1.2 Testing Strategy

First, we create L3 products from the input L2 images using romancal.resample. We use the default resample settings except we set the weight_time parameter to “exptime” rather than inverse variance map (IVM) as we have made no effort to properly populate all the variance information in the simulated input files. We note that this should have no impact on the test results, and in fact using IVM weighting may negatively affect our ability to measure source centroids without careful handling of variances in the simulated data files. The default parameters for romancal.resample are such that the output mosaics created in this test are sampled at the native pixel scale.
To test if the astrometry between an output L3 mosaic product and its input L2 products match, we measure the source positions in the output L3 and compare them against the catalog of input sources positions used to simulate the L2 products. We perform this procedure on both the single-pixel source and point source L3 images. For the single-pixel source image, we detected the sources using a simple threshold detection algorithm via `photutils.detection.DAOStarFinder`, which is based on the DAOPHOT algorithm (Stetson 1987). The positions reported by this algorithm are determined via centroiding assuming a 2-D Gaussian distribution of signal. For the images containing simulated point sources, this assumption breaks down due to the non-Gaussian nature of the Roman PSF, and therefore we use PSF-fitting to better measure the source positions in the point source image. As the PSF is resampled onto the L3 WCS frame, we use `photutils.psf` to construct an “empirical” PSF (ePSF) based on the point sources in the L3 image (`extract_stars` and `EPSFBuilder`) to perform the measurements (PSFPhotometry). The resampled ePSF is determined from 240 sources spread across a sensor chip assembly (SCA), and thus represents a typical PSF at any given location but may not perfectly match a given source due to the spatial variation in the PSFs used to create the input L2 images. We also expect that the ePSF will have limited accuracy due to basing it on resampled PSFs.

We then compare the source positions measured in the L3 products with the known positions of the sources from the catalogs used to create the L2 input products. In the case of the single-pixel sources, this is accomplished by taking the (X, Y) positions of the pixels and transforming them to sky coordinates using the `gwcs` object in the image metadata. For the images with simulated point sources, we use the input source catalog as truth. Matching catalogs is performed by converting ($\alpha$, $\delta$) positions to `astropy.coordinates.SkyCoord` objects and using the `match_to_catalog_sky` method, which returns a 2-D angular distance between matched pairs of sources.

4.1.3 Results

4.1.3.1 Single-Pixel Source Test

We present the results of the source positions in the L3 output products matched to the input catalog for the single-pixel sources in Figure 1 and Figure 2. Figure 1 displays the 2-D angular distance in units of degrees between the source in the L3 image and the best match from the input source catalog as a function of the X and Y positions from the input source catalog (i.e., in the L2 frame). We find no correlation between the 2-D angular separation and the source position.

In Figure 2, we plot the 2-D angular separation as a function of the measured peak value in the L3 image. A clear correlation between the 2-D angular separation and peak pixel flux is visible, which we attribute to pixel phase. During the drizzling process, signal from the input L2 pixels is apportioned onto the output L3 frame, and may be split into one or more output pixels. Pixels in the L3 image with peak values of 5000 electrons per second per pixel have not had any signal split into adjacent pixels, and thus have the most reliable position measurements, while our confidence in the position measured with `photutils.detection.DAOStarFinder` decreases with decreasing peak pixel value (i.e., as the input single-pixel source flux is more...
Figure 1: 2-D angular separation between measured L3 product positions and input L2 positions for single-pixel sources as a function of L2 X coordinate (left) and Y coordinate (right).

Figure 2: 2-D angular separation between L3 product positions and L2 input positions as a function of peak source value in electrons per second per pixel. The shape of the distribution indicates that deviations from expected positions are due to pixel phase.

divided in the output L3 frame). If we confine our sample to the brightest 2% of sources in the L3 image (as measured by the peak pixel value), we find that the L2 and L3 positions match within a median 2-D angular separation of 2.87 milli-arcseconds.
4.1.3.2 Point Source Test

If pixel phase is indeed responsible for the behavior seen in the single-pixel test above, then using simulated point sources and dithered observations should mitigate the effect. We explore this hypothesis in this section. Figure 3 shows both the L3 mosaic and the residual image after subtracting the reconstructed ePSF from 240 point sources in the L3 image (all detected sources excluding several sources too near the edge of the image). The reconstructed ePSF performs reasonably well at subtracting the sources from the image, though imperfections are present and vary across the image. We attribute these imperfections to the spatially varying shape of the PSF across the simulated input L2 images.

Figure 4 shows the 2-D angular separation between the output L3 image and input source catalog as a function of source brightness as measured from the integration of the fitted ePSF out to a radius of 15 pixels (~1.65 arcseconds) in the L3 frame. In this case, we do not see a strong correlation between these parameters, though there is some evidence of a general trend such that brighter measured sources have smaller 2-D angular separations. Recall that the input sources all have fluxes of 5000 electrons per second, so large deviations from the mean measured brightness may be regarded as indicative of goodness of fit. We compute the weighted average in the 2-D angular separation to be 4.65 ± 3.47 milli-arcseconds, where the uncertainty is the 1-sigma standard deviation of the 2-D angular separations. The typical 2-D measurement error in the source positions is 0.57 milli-arcseconds. We remind the reader that we have used a single ePSF that does not accurately account for spatial variations in the input PSFs, and the ePSF is constructed from resampled point sources, which also likely affects the accuracy.

To test our ability to accurately measure the source positions in the L3 images, we apply our PSF fitting procedure to one of the input L2 images used to create the L3 mosaic. The 2-D angular separations as a function of source flux are shown in Figure 5. Compared to

![Figure 3](https://soccer.stsci.edu)
Figure 4: 2-D angular distance between the L3 product measured source positions and the catalog used to simulate the input L2 products as a function of the integrated flux contained within 15 pixels of the fitted ePSF. A weak correlation is seen between 2-D distance and ePSF flux. The median 2-D separation is $4.65 \pm 3.47$ milli-arcseconds with a typical measurement error of 0.57 milli-arcseconds.

Figure 5: 2-D angular distance between the L2 product measured source positions and the catalog used to simulate the L2 product as a function of the integrated flux contained within 15 pixels of the fitted ePSF. The separations are evenly distributed over the approximate range 0 – 4 mas.
Figure 4, the 2-D separations are typically smaller, though they are not negligible. This suggests two possibilities, one or both of which may be contributing to the offsets we observe in the L3 products:

1. The difference between the truth and measured positions in the L2 images may be real and due to how the images were simulated.
2. The difference between the truth and measured positions in the L2 images are the result of the accuracy of our test procedure, which ignores the spatially varying nature of the PSF in the simulated data.

The bulk of the offsets observed in the L3 mosaic are accounted for by the offsets we measure in the L2 image; therefore, we argue that these are consistent with each other.

For the interested reader, we note that the flux measurements based on our ePSF fitting systematically underestimate the simulated source flux of 5000 electrons per second in both the L2 and L3 products. The drizzling algorithm, which is used by roman_cal.resample, is known to conserve flux. Because the underestimate is present in both the L2 and L3 measurements, we attribute this effect to a combination of imperfections in the fitted ePSF to the data and the radius of 15 pixels (~1.65 arcseconds) used for the flux measurement.

4.1.3.3 Summary

Combining the analyses described above, we are confident that the source positions in the L3 products generated from input L2 images with known source positions are the same, and that the small separations (~few milli-arcseconds) between the measured source positions in the L3 products and the truth positions are the result of our ability to accurately measure those positions in the resampled images using our test procedure. Thus, on the assumption that the L2 images are registered to the Gaia DR3 astrometric frame, the L3 products are also based on a realization of the ICRF, and we determine that SOC-581 has passed scientific validation.

4.2 SOC-582: Mosaic Information

SOC-582 indicates that L3 mosaic products created by the high-level pipeline should include the following:

1. Data quality and uncertainty information,
2. Total exposure time, and
3. Metadata used in the mosaic generation process.

Mosaic products from the Hubble (HST) and James Webb Space Telescope (JWST) missions have not previously contained data quality arrays in the same way that detector-level calibrated products (e.g., L2 WFI products) do. Instead, pixels that are poor quality for scientific analysis are removed during the image combination process, while pixels with data quality flags that are considered acceptable, which may include some effects, are included in the mosaic product. We have interpreted the above requirement to be satisfied by including information about which input images contributed to each pixel in the final output mosaic.

4.2.1 Test Data Description

For these tests, we use the single-pixel source images described in Section 4.1.1. These images contain 1600 bright pixels evenly distributed in a regular grid across the WFI01 detector, with each of these pixels assigned a value of 5000 electrons per second per pixel. The background level is negligible, and no cosmic rays are included in the data. The image was
copied to a new file and the data quality array altered to include flags for undesirable data: \([y, x] = [3400:3600, 200:400]\) were given a flag of 1 ("do not use") and \([y, x] = [3700:3900, 500:700]\) were given a flag of 1024 ("dead").

4.2.2 Testing Strategy

We combine the two images using `romancal.resample` and examine the output L3 product. We do this twice, the first with the default parameters (except weight_type), and a second time with the output pixel scale and pixfrac parameters of the output mosaic set to 0.5. In both cases, the weight_type is set to “exptime.”

The output product is examined for the following:

1. The product contains an error array.
2. The product contains a context image, and the context image correctly identifies the images that contribute to the output pixels.
3. The combined exposure time of the product is correctly recorded.
4. The association file name is correctly recorded.
5. The files listed in the association file are correctly recorded.
6. The pixel scale ratio is correctly recorded.
7. The pixel fraction is correctly recorded.
8. The weight type is correctly recorded.
9. The coordinate reference frame name in both the L2 inputs and the L3 output match.

4.2.3 Results

First, we describe the results of our examination of the L3 product metadata. We calculate the exposure time by multiplying the number of resampled files by the effective exposure time of the L2 simulated images, and this value matches the exposure time in the L3 mosaic product. The name of the association file is correctly recorded in the L3 product metadata. The pixel scale, pixel fraction, and weight type used by `romancal.resample` are all correctly recorded. Finally, the coordinate reference frame name is recorded and matches the L2 input products.

4.2.3.1 Discrepancies

During testing, we found several discrepancies with the `romancal.resample` code, and we enumerate them below in Table 4-2. In each case, we list the Jira ticket that has been filed to track the work to correct these issues.

<table>
<thead>
<tr>
<th>Description</th>
<th>Recommendation</th>
<th>Jira Ticket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log information from the resample code, which is printed to the standard output while running, is not recorded in the output L3 product.</td>
<td>Record the log information that is printed to the standard output in the L3 product metadata.</td>
<td>RSOCDR-121</td>
</tr>
<tr>
<td>Description</td>
<td>Recommendation</td>
<td>Jira Ticket</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>The resample code does not provide the capability for users to select data quality flags to either include or exclude in the output mosaic.</td>
<td>In the resample code, add an optional parameter for data quality flags that resample should include in the output product. This optional parameter should override the default masking behavior.</td>
<td>RSOCDR-122</td>
</tr>
<tr>
<td>The list of default data quality flags allowed to be included in the output mosaic appears to include 1024 “dead pixel.” At present, it is unclear how this decision about default data quality flags is made in the code.</td>
<td>Review the list of data quality flags and identify which should be considered acceptable for default data processing, then update the code appropriately. Perhaps store the default data quality flags included in L3 products in an explicit location (e.g., a parameter file).</td>
<td>RSOCDR-123</td>
</tr>
<tr>
<td>The names of the input L2 images used to create the mosaic were not correctly recorded in the metadata. Instead, the value of the L2 product metadata value meta.filename was recorded, which in our test did not match the file names on disk, which was altered using shell commands. In this case, the values of meta.filename were the original archive-style file names, which use the WFI observation identifier.</td>
<td>Save both the observation identifiers and the names of the files on disk of the L2 input products used to create the L3 product.</td>
<td>RSOCDR-124</td>
</tr>
<tr>
<td>The L3 products contain weight maps in which every pixel is set to a value of zero.</td>
<td>Correctly populate the L3 weight map array.</td>
<td>RSOCDR-133</td>
</tr>
</tbody>
</table>

### 4.2.3.2 Summary

We find that the L3 products created by roman.cal contain the set of information necessary to satisfy requirement SOC-582: the L3 context image representing the data quality information is correctly populated; the total exposure time value in the metadata matches expectations based on the input L2 products; and the L3 products contain information about the input data and code parameters used to generate the L3 mosaics. We will defer our analysis of the scientific correctness of the uncertainty array until delivery of SOC-667 (the Level 3 pipeline) in SOC Release 3 as this is a very complex analysis that is contingent upon the entirety of the Level 3 pipeline being functional. During our validation of SOC-582, we encountered some unexpected behaviors for which we have filed five DRs. None of these DRs prevent us from passing SOC-582 as they are either enhancements or they are filed against SOC-667 in
Release 3. Based on the above summary, we determine that SOC-582 has passed scientific validation.

4.3 SOC-666 and SOC-669: WFI Level 2 Science Data Products

Requirements SOC-666 and SOC-669 encompass the processing of WFI science data products from L1 raw, uncalibrated ramps to L2 detector-level calibrated rate images. We note that, in SOC Release 1, several requirements related to L1 to L2 processing were previously validated and discussion concerning these may be found in Desjardins, et al. (2023), namely: the presence of information that describes the geometric distortion (SOC-587); the inclusion of absolute photometric calibration information (SOC-588 and SOC-589); compensation for the spatially variable response of the WFI detectors, i.e., the flat field correction (SOC-591); and the proper selection of calibration reference files for both the WFI imaging and spectroscopic modes (SOC-303 and SOC-628). In the validation of SOC-666 and SOC-669, we focus on the remaining steps in the calibration of WFI data from L1 to L2 that have been determined as of the end of SOC Release 2 in December 2023. These steps are, in order or pipeline processing:

1. Application of a bad pixel mask (DQInitStep).
2. Flagging of pixels that are saturated in the ramp cube (SaturationStep).
3. Application of the Improved Roman Reference Correction (IRRC) to mitigate the effect of 1/f noise imparted by the readout electronics (RefPixStep).
4. Correction for the effect of classic non-linearity (LinearityStep).
5. Subtraction of the dark current signal (DarkCurrentStep).
6. Detection of jumps and fitting up the ramp (RampFitStep).

The romanCal step names are provided above in parentheses. As previously stated in Section 4.1, the alignment of WFI L2 products with the Gaia astrometric frame is described by requirement SOC-1031, which is not due for delivery until SOC Release 5; therefore, we neither consider it as part of SOC-666 and SOC-669 nor do we validate SOC-1031 in this report. While the performance of the WFI imaging and spectroscopic modes may be different, the algorithms for the steps listed above that are used to calibrate both modes are the same, and thus we treat these two requirements as functionally identical for the purpose of scientific validation.

We emphasize to the reader that these tests concern the algorithms as implemented in December 2023. Neither the tests nor the algorithms reflect any changes that may have been made after December 2023, and they do not include any knowledge of the results of the analysis and characterization of data from the first WFI thermal vacuum test campaign (TVAC1), which was carried out in the third quarter of calendar year 2023. However, as discussed below when describing the test data, we do use reference pixel correction weights and dark exposures from TVAC1 to test the implementation of the IRRC algorithm. This is necessary as no WFI image simulators at the time of writing are capable of simulating the 1/f noise and corresponding reference pixel values in WFI images for a known set of correction weights.

4.3.1 Test Data Description

We separate the tests and their input data into three categories: 1) data needed to test steps 1 – 5 enumerated above (i.e., DQInitStep up to and including DarkCurrentStep; hereafter referred to as exposure-level pipeline (ELP)); 2) jump detection; and 3) ramp fitting.

4.3.1.1 ELP Test Data

We use WFIsim to create simple L1 images that contain instrumental effects, but no
external sources. We assign one image to the WFI imaging mode (WIM) and the other to the WFI spectroscopic mode (WSM) to validate both SOC-666 and 669. While running the test code, several pixels in these images will be intentionally altered to assess the correctness of the applied *romancal* algorithms.

4.3.1.2 Jump Detection Test Data

We use WFIsim to create simulated images with a realistic distribution of cosmic rays (Bellini, et al. 2022). The images are designed to have a ramp of 250 data numbers (DN) per second per pixel and read noise of 7.5 DN. We use two MA tables for this experiment: 1) a table similar to that used in the Design Reference Mission (DRM) for the Galactic Bulge Time-Domain Survey (GBTDS), formerly known as the exoplanet microlensing survey; and 2) a 150-second high-dynamic-range MA table with fewer reads per resultant at the beginning of the table to accommodate bright sources, and more reads per resultant at the end to obtain sufficient signal-to-noise for faint sources. Following the nomenclature used in Casertano (2022), we refer to these tables as the ML and HDR-150 tables, respectively. Definitions for these MA tables may be found in Table 1 of Casertano (2022). For each MA table, we create three realizations of images with cosmic rays, which allows us to test the consistency of the algorithm. For each MA table, we also simulated a set of images that includes an additional single-read last resultant (SRLR), which has been suggested by Sharma & Casertano (submitted) to help mitigate cosmic rays in the final resultant from being undetected.

We note that our test data are biased towards the worst cases for jump detection. Notably, the count rate per pixel in the simulated images is similar to that of stars rather than thermal or astronomical backgrounds, which makes detection of the cosmic rays particularly difficult. As previously stated, the count rate per pixel in our simulated images is 250 DN per second per pixel, while according to Bellini et al. (2022) simulated cosmic rays deposit a randomized value between 100 and 1000 electrons per µm. In our validation plan for Release 3, we intend to revisit the jump detection step and test it in further detail as part of the validation of requirements SOC-1064 and SOC-1066, which pertain to uneven sampling up the ramp.

4.3.1.3 Ramp Fitting Test Data

Our test code is capable of generating simple ramps that do not include detector effects, e.g., classic non-linearity. These simple ramps can be saved on-the-fly as L1 ASDF files that are compatible with *romancal* for testing. We also generate a simulated image of a star field using WFIsim, which contains an independent implementation of the Casertano (2022) ramp fitting algorithm. To test only the ramp fitting algorithm, we used dummy calibration reference files that remove detector effects like classic non-linearity and dark current. These dummy reference files are used both in the creation of the WFIsim images as well as in *romancal*.

4.3.2 Testing Strategy

4.3.2.1 ELP Testing Strategy

For each *romancal* step we test, we examine the output of the step for the correct calibration reference file selection from the Calibration Reference Data System (CRDS) and the correct application to the data of the information contained with the reference file. We briefly describe the expected behavior that we examine for each step here:

1. **DQInitStep**: Convert the WFIScienceRaw data model to a Ramp model and update
the 2-D pixeldq array with the values from the mask reference file.

2. **SaturationStep:** For each resultant in the ramp that consists of multiple, combined reads, use the saturation reference file to determine the number of counts at the mean time of the resultant for which a pixel would have saturated in the last read of the resultant. In the case of a single read resultant, simply compare the pixel values to the value in the saturation reference file. In both cases, if any pixels are greater than saturation value, then update the 3-D groupdq array at the detector location and resultant number with the data quality flag 2 “SATURATED.”

3. **RefPixStep:** Use the IRRC algorithm to apply the weights from the refpix reference file to the three reference streams: the four left column, four right column, and amplifier 33 pixels. More information about the IRRC algorithm is contained within a document in preparation by Rauscher et al.

4. **LinearityStep:** Apply the coefficients from the linearity reference file to the ramp. For each resultant, the output should be equal to a polynomial of order N – 1, where N is the length of the third axis of the data array in the linearity reference file, and the polynomial coefficients are the values of the data array. Pixels that do not have a satisfactory linearity correction should be marked in the 2-D pixeldq array of the ramp model.

5. **DarkCurrentStep:** For each resultant in the ramp model, subtract the corresponding resultant from the dark reference file. Also update the 2-D pixeldq array with any data quality flags from the dark reference file (e.g., hot pixel, warm pixel, etc.).

### 4.3.2.2 Jump Detection Testing Strategy

We examine the performance of the jump detection algorithm implemented in *romancal* by processing simulated ramps with known cosmic ray properties. These properties include the rate of cosmic rays in a WFI exposure, the distribution of signal induced by a cosmic ray incident on a WFI detector, and the location of cosmic rays in both pixel coordinates and time within a ramp. We create both simple tests with small numbers of pixels and complex tests with images simulated by WFIsim, which contain a constant background level and cosmic rays. In the following analysis, we will focus on the results from WFIsim simulated images.

To evaluate the performance of the jump detection algorithm, we compute the false positive (FPR) and false negative rates (FNR) as defined in *Sharma & Casertano* (submitted):

\[
FPR = \frac{FP}{FP + TN}
\]

and

\[
FNR = \frac{FN}{FN + TP}
\]

In these equations, FPR and FNR are the false positive and false negative rates, respectively. The false positive variable FP is the number of pixels falsely identified as a jump by the algorithm when a jump is not present. The false negative variable FN is the number of pixels known to contain a jump and that are not identified by the algorithm. The true negative variable TN is the number of pixels that do not contain jumps and are correctly identified by the algorithm. Finally, the true positive variable TP is the number of pixels that contain jumps and are correctly identified by the algorithm.

We compare the FPR and FNR values in our test data with expected values from the
authors of the algorithm that were determined by empirical testing. We note that the absolute performance of the algorithm is highly dependent on several factors: the sampling of the MA table; the count rate of the pixels; and the level of signal induced by a jump. We do not comment on the absolute performance as the final set of MA tables for the mission have not yet been determined, we simply examine the results of the version of the jump detection algorithm implemented in romancal and compare them to what is expected by the algorithm creators.

Note that we do not attempt to test the algorithm for the detection of the effect known as “snowballs” that have been observed in on-orbit JWST observations. The algorithm has not been enhanced with this capability as of the time of writing, nor do we have access to a simulator capable of injecting snowball events at this time. Tests that validate the ability to detect snowballs will be included in a future report.

4.3.2.3 Ramp Fitting Testing Strategy

To validate that the ramp fitting algorithm is working correctly in the presence of different unevenly spaced MA tables, we first focus on the simple case where we generate a constant slope. We then examine: 1) that the L2 ramp matches the results from our own implementation of the Casertano (2022) ramp fitting method; and 2) that the L2 ramp matches the ground-truth expected slope within a tolerance that depends on roundoff errors, noise levels, and the input MA table. These mock images are generated using very simple calibration reference products, i.e., zero dark current, no data quality flags, perfect classic linearity, and a tunable constant read noise. We override the romancal steps with these calibration reference files, which are generated as part of the test code. This ensures that our test results are not biased by any other calibration effects.

We next use the simulated L1 image products created by WFI sim described in Section 4.3.1.3. We process these images using romancal to obtain the L2 output products making sure that we use appropriate calibration reference files equivalent to those used by WFI sim to create the simulated L1 files. Finally, we compare the romancal-processed L2 slopes with those reported by WFI sim.

4.3.3 Results

4.3.3.1 ELP Results

Quantitative comparisons are included in the test code, which may be reviewed. For all of the DQInitStep, SaturationStep, LinearityStep, and DarkCurrentStep, we find that the romancal steps produce results identical to manual application of the correction algorithms.

Regarding the RefPixStep, we provide a comparison of a 55 read WFI TVAC1 dark image before and after the application of the IRRC algorithm as implemented in romancal. Figure 6 shows, for WFI02, the simple slope images before and after the 1/f noise model was subtracted, as well as a panel showing the 1/f noise model itself. The simple slope is computed by subtracting the next to last read from the last read in the ramp and dividing by the frame time, which for TVAC1 is 3.04 seconds. The weights used in the correction were computed separately as part of an analysis and characterization of TVAC data, which we do not discuss in this report. Visually, the 1/f noise is noticeably reduced in the corrected image.
Figure 6: Qualitative evaluation of the IRRC implementation in ROMANCAL using a 55 read dark exposure obtained from WFI TVAC1. The detector displayed is WFI02. The pixel values are simple slopes obtained by subtracting last read by the next to last read and dividing by the frame time (3.04 seconds). The middle panel is the difference between the simple slope images with and without the IRRC algorithm applied.

4.3.3.2 Jump Detection Results

We provide a summary of the FPR and FNR values determined from our test data in Error! Reference source not found.. We find that, while the FNRs are high, they are consistent with expectations given the relatively high-count rate in the image (Sharma and Casertano submitted). These values are similar to what one may expect for the detection of a cosmic ray that overlaps an astronomical source. Such an event produces a much smaller bias between resultants compared to if the jump occurred in a pixel with a lower count rate. The FPR values are low and within expectations. During validation of SOC-1064 and SOC-1066 in SOC Release 3, which are related to uneven sampling up the ramp, we intend to re-validate the jump detection step with additional test cases that better sample the parameter space.

Table 4-3: Metrics for Jump Detection in WFIsim Simulated Ramps

<table>
<thead>
<tr>
<th>MA Table</th>
<th>Test Iteration</th>
<th>SRLR FNR (%)</th>
<th>SRLR FPR (%)</th>
<th>No SRLR FNR (%)</th>
<th>No SRLR FPR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>1</td>
<td>62.95</td>
<td>0.22</td>
<td>57.54</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>62.80</td>
<td>0.22</td>
<td>56.46</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>63.28</td>
<td>0.22</td>
<td>57.18</td>
<td>0.18</td>
</tr>
<tr>
<td>HDR-150</td>
<td>1</td>
<td>68.36</td>
<td>1.67</td>
<td>64.45</td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>68.34</td>
<td>1.67</td>
<td>64.02</td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>68.53</td>
<td>1.67</td>
<td>64.45</td>
<td>0.48</td>
</tr>
</tbody>
</table>
4.3.3.3  Ramp Fitting Results

For the simple test using constant-slope ramps, both the romancal output and our implementation of the Casertano (2022) algorithm produce identical results within tolerances for all the cases that we test.

In the case of the more complex test using a simulated astronomical scene, we first make a visual inspection of the slope images to ensure qualitative agreement (Figure 7). The main differences between these two slope images are due to masking differences and not the result of the ramp fitting algorithm.

We next compare the pixel values in the slope images. A histogram of the pixel ratios is shown in Figure 8. The vast majority of pixels agree perfectly with each other, and the remainder show only small deviations (~5 – 10%) that we attribute to the aforementioned masking and saturation flagging effects.

4.3.3.4  Summary

We find that each of the romancal steps tested have performed as expected based on the descriptions in Section 4.3.2. The RampFitStep, which combines both jump detection and ramp fitting, produces results that are expected based on the inputs. The false negative rate appears quite high, but is a reflection of the input test data and the results are in agreement with the report describing the algorithm. The ramp fit result matches expectations compared with simulated data except for minor deviations that we attribute to differences in data quality flagging between the simulator and romancal. Based on these results, we determine that SOC-666 and SOC-669 have passed scientific validation.

Figure 7: A comparison of the slope images created by romancal (left) and WFIsim (right). Small differences are observed, which we attribute to difference masking choices. Also note that the cores of several stars in the romancal image are marked as saturated.
Figure 8: A histogram of the ratio the slope images created by romancal and WFIsim. Note that the y-axis scale is millions of pixels. While most pixels are in near-perfect agreement, we attribute the small differences to differences in masking and saturation flagging.

5 Conclusions

We have designed and run scientific validation tests on four SOC Level 4 requirements for the Roman DMS as part of SOC Release 2. The requirement numbers and the result of their validation tests are shown below in Table 5-1. All requirements that have undergone scientific validation in Release 2 have received a pass.

Table 5-1 Summary of Validation Results

<table>
<thead>
<tr>
<th>SOC Level 4 Requirement ID</th>
<th>Level 4 Requirement</th>
<th>Validation Result [Pass/Fail]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOC-581</td>
<td>In Wide Field Imaging Mode, the DMS shall generate mosaic images using coordinates tied to the astrometric frame defined by the International Celestial Reference Frame (ICRF).</td>
<td>Pass</td>
</tr>
<tr>
<td>SOC-582</td>
<td>In Wide Field Imaging Mode, the DMS shall generate mosaic images that include the following information: (a) Data quality and uncertainty information (b) Total exposure time (c) Metadata used in the mosaic generation process.</td>
<td>Pass</td>
</tr>
<tr>
<td>SOC-666</td>
<td>The DMS shall generate WFI Level 2 science data products in Wide Field Imaging Mode.</td>
<td>Pass</td>
</tr>
<tr>
<td>SOC-669</td>
<td>The DMS shall generate WFI Level 2 science data products in Wide Field Spectroscopy Mode.</td>
<td>Pass</td>
</tr>
</tbody>
</table>
6 Bibliography


Appendix A. System and Computing Environment Specifications

The SOC DMS Release 2 science validation was performed on a Linux virtual machine in the STScI Flexible Data Center. The machine (dlrrtb1.stsci.edu) uses 18 Intel® Xeon® Gold 6254 CPUs with clock speeds of 3.10 GHz, and 64 GB of DDR-4 2933 MHz RAM. The operating system during testing was Red Hat Enterprise Linux (RHEL) 8.9 with Linux kernel 4.18.0-513.18.1.el8_9.x86_64 (release date February 28, 2024).

Conda version 24.3.0 was used to control the computing environment for the science validation. An environment YAML file (reproduced below) was created and shared with the testers to make sure that all versions of software dependencies were the same as well as the CRDS context that controls the calibration reference file versioning. A copy of the YAML file is also included with the validation test artifacts.

The roman_cal package and its dependencies compose the implementation of the Roman science calibration pipeline. The version of roman_cal tested in this report is version 0.13.0 (https://github.com/spacetelescope/roman_cal/releases/tag/0.13.0), and this release was delivered to support functionality through SOC/DMS Build 24Q1_B12.

name: release2
cannels:
  - conda-forge
dependencies:
  - _libgcc_mutex=0.1=conda_forge
  - _openmp_mutex=4.5=2_gnu
  - asttokens=2.4.1=pyhd8ed1ab_0
  - bzip2=1.0.8=hd590300_5
  - ca-certificates=2024.2.2=hbcca054_0
  - decorator=5.1.1=pyhd8ed1ab_0
  - exceptiongroup=1.2.0=pyhd8ed1ab_2
  - executing=2.0.1=pyhd8ed1ab_0
  - ipython=8.22.2=pyh707e725_0
  - jedi=0.19.1=pyhd8ed1ab_0
  - ld_impl_linux-64=2.40=h41732ed_0
  - libexpat=2.6.2=h59595ed_0
  - libffi=3.4.2=h7f98852_5
  - libgcc-ng=13.2.0=h807b86a_5
  - libgomp=13.2.0=h807b86a_5
  - libsnl=2.0.1=hd590300_0
  - libsqlite=3.45.2=h2797004_0
  - libuuid=2.38.1=h0b41bf4_0
  - libxcrypt=4.4.36=hd590300_1
- libzlib=1.2.13=hd590300_5
- matplotlib-inline=0.1.6=pyhd8ed1ab_0
- ncurses=6.4.20240210=h59595ed_0
- openssl=3.2.1=hd590300_1
- pexpect=4.9.0=pyhd8ed1lab_0
- pickleshare=0.7.5=py_1003
- pip=24.0=pyhd8ed1lab_0
- prompt-toolkit=3.0.42=pyha770c72_0
- ptyprocess=0.7.0=pyhd3deb0d_0
- pure_eval=0.2.2=pyhd8ed1ab_0
- pygments=2.17.2=pyhd8ed1lab_0
- python=3.11.8=hab00c5b_0_cpython
- readline=8.2=h8228510_1
- setuptools=69.2.0=pyhd8ed1lab_0
- six=1.16.0=pyh6c4a22f_0
- stack_data=0.6.2=pyhd8ed1ab_0
- tk=8.6.13=noxft_h4845f30_101
- traitlets=5.14.2=pyhd8ed1lab_0
- typing_extensions=4.10.0=pyha770c72_0
- wcwidth=0.2.13=pyhd8ed1lab_0
- wheel=0.43.0=pyhd8ed1lab_1
- xz=5.2.6=h166bdaf_0
- pip:
  - alabaster==0.7.16
  - asdf==3.1.0
  - asdf-astropy==0.6.0
  - asdf-coordinates-schemas==0.3.0
  - asdf-standard==1.1.1
  - asdf-transform-schemas==0.5.0
  - asdf-unit-schemas==0.2.0
  - asdf-wcs-schemas==0.4.0
  - astropy==6.0.1
  - astropy-iers-data==0.2024.4.1.0.33.14
  - astroquery==0.4.7
  - attrs==23.2.0
  - babel==2.14.0
  - beautifulsoup4==4.12.3
  - certifi==2024.2.2
- cffi==1.16.0
- charset-normalizer==3.3.2
- contourpy==1.2.1
- crds==11.17.19
- cryptography==42.0.5
- cyclery==0.12.1
- docutils==0.20.1
- drizzle==1.15.1
- et-xmlfile==1.1.0
- filelock==3.13.3
- fonttools==4.50.0
- gwcs==0.21.0
- html5lib==1.1
- idna==3.6
- imagesize==1.4.1
- importlib-metadata==7.1.0
- jaraco-classes==3.4.0
- jaraco-context==4.3.0
- jaraco-functools==4.0.0
- jeepney==0.8.0
- jinja2==3.1.3
- jmespath==1.0.1
- jsonschema==4.21.1
- jsonschema-specifications==2023.12.1
- keyring==25.1.0
- kiwisolver==1.4.5
- lxml==5.2.1
- markupsafe==2.1.5
- matplotlib==3.8.3
- more-itertools==10.2.0
- numpy==1.26.4
- numpydoc==1.7.0
- opencv-python-headless==4.9.0.80
- openpyxl==3.1.2
- packaging==24.0
- pandas==2.2.1
- parsley==1.3
- photutils==1.11.0
- pillow==10.3.0
variables:
  CRDS_PATH: ./crds_cache
  CRDS_SERVER_URL: https://roman-crds.stsci.edu/
  CRDS_CONTEXT: roman_0053.pmap
Appendix B. Testing Artifact Manifest

Notes:
1. Plot files (e.g., PNG, PDF, etc.) are not listed here as artifacts.
2. The data files for soc_666_669_jump.py must be in subdirectories with names as indicated below. The code parses the directory names for information necessary in the tests.
3. The data input data files for the ramp fitting tests are optional, but are included here for completeness. If no input files are given, the test code generates ramps. If an input L1 file is given, the corresponding L2 file is required for comparison, and they are included here in the test artifacts.

SOC-581
- Python code files
  - soc_581.py
- Input data files
  - r0000901001001001001_01101_0001_WFI01_cal.asdf
  - r0000901001001001001_01101_0002_WFI01_cal.asdf
  - r0000901001001001001_01101_0003_WFI01_cal.asdf
  - r0000901001001001001_01101_0004_WFI01_cal.asdf
  - grid_wcs.asdf
  - grid_wcs_dq.asdf
  - pixel_grid_cat.txt
  - ra80_dec0_5000_pos0.xy
- Output data files
  - 581_wcs_pixel_asn.json
  - 581_wcs_pixel_highlevelpipeline.asdf
  - 581_wcs_romanpsf_asn.json
  - 581_wcs_romanpsf_highlevelpipeline.asdf

SOC-582
- Python code files
  - soc_582.py
- Input data files
  - grid_wcs.asdf
- Output data files
  - 582_default_test_asn.json
  - 582_default_test_highlevelpipeline.asdf
  - 582_scale_ratio_05_test_asn.json
  - 582_scale_ratio_05_test_highlevelpipeline.asdf

SOC-666 & SOC-669
- Python code files
Input data files
- ELPP testing
  - r066601001001001001_01101_0001_WFI01_uncal.asdf
  - r066691001001001001_01101_0001_WFI01_uncal.asdf
- Jump testing
  - hdr150_nosingle/r0000101001001001001_01101_0001_crmask.fits
  - hdr150_nosingle/r0000101001001001001_01101_0001_WFI01_uncal.asdf
  - hdr150_nosingle/r0000101001001001001_01101_0002_crmask.fits
  - hdr150_nosingle/r0000101001001001001_01101_0002_WFI01_uncal.asdf
  - hdr150_nosingle/r0000201001001001001_01101_0001_crmask.fits
  - hdr150_nosingle/r0000201001001001001_01101_0001_WFI01_uncal.asdf
  - hdr150_nosingle/r0000201001001001001_01101_0002_crmask.fits
  - hdr150_nosingle/r0000201001001001001_01101_0002_WFI01_uncal.asdf
  - hdr150_nosingle/r0000201001001001001_01101_0003_crmask.fits
  - hdr150_nosingle/r0000201001001001001_01101_0003_WFI01_uncal.asdf
  - ml_nosingle/r0000301001001001001_01101_0001_crmask.fits
  - ml_nosingle/r0000301001001001001_01101_0001_WFI01_uncal.asdf
  - ml_nosingle/r0000301001001001001_01101_0002_crmask.fits
  - ml_nosingle/r0000301001001001001_01101_0002_WFI01_uncal.asdf
  - ml_nosingle/r0000301001001001001_01101_0003_crmask.fits
  - ml_nosingle/r0000301001001001001_01101_0003_WFI01_uncal.asdf
  - ml_single/r0000401001001001001_01101_0001_crmask.fits
  - ml_single/r0000401001001001001_01101_0001_WFI01_uncal.asdf
  - ml_single/r0000401001001001001_01101_0002_crmask.fits
  - ml_single/r0000401001001001001_01101_0002_WFI01_uncal.asdf
  - ml_single/r0000401001001001001_01101_0003_crmask.fits
  - ml_single/r0000401001001001001_01101_0003_WFI01_uncal.asdf
- Ramp testing
  - r0000101001001001001_01101_0001_WFI01_cal.asdf
  - r0000101001001001001_01101_0001_WFI02_cal.asdf
  - r0000101001001001001_01101_0001_WFI03_cal.asdf
  - r0000101001001001001_01101_0001_WFI01_uncal.asdf
  - r0000101001001001001_01101_0001_WFI02_uncal.asdf
  - r0000101001001001001_01101_0001_WFI03_uncal.asdf
Appendix C. SOC-581 Validation Code

""
Validation code for SOC Requirement 581.
""
import os
import pdb
import shutil
import warnings
import glob
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import astropy.table import Table
import astropy.units as u
import astropy.coordinates import SkyCoord
import astropy.stats import sigma_clipped_stats
import astropy.nddata import NDData
import astropy.visualization import simple_norm

import roman_datamodels as rdm
import romancal.resample.resample_utils as resample_utils
import romancal.associations.asn_from_list as asn_from_list
from romancal.pipeline.highlevel_pipeline import HighLevelPipeline

from photutils.detection import DAOStarFinder
from photutils.psf import extract_stars, EPSFBuilder, PSFPhotometry

def prRed(s):
    ""
    Simple function to print messages to stdout in red.

    Inputs
    ------
    s (str): String to print.

    Returns
    ------
    None
    ""
    print(f"\033[91m {s}\033[00m")
def prepare_data(l3name, l2list, weight_type='exptime',
                  pixel_scale_ratio='1.0', pixfrac='1.0', overwrite=False):
    ""
    Creates an association file and then runs romancal on the input
    files to generate the L3 data from
    the initial L2 data.
    ""

    Inputs
    ------
    l3name (str): the name of the association and L3 files that will
    be created.

    l2list (list of str): the names of the input L2 files that will be
    processed by the pipeline.

    weight_type (str; default = 'exptime'): The options are 'none',
    'ivm', and 'exptime'. The resample pipeline default
    is 'ivm' but 'exptime' is more useful to evaluate the weights
    and exposure times of romancal.

    pixel_scale_ratio (str; default = '1.0'): Ratio of input to output
    pixel scale used by the high level pipeline
    resample step. The default is 1.0.

    pixfrac (str; default = '1.0'): the pixel fraction that the high
    level pipeline resample pipeline will use
    during processing. The default is 1.0.

    overwrite (bool; default = False): Flag to overwrite the L3 data
    files if they already exist. Default is False.
    A value of True will delete the existing files. If the files
    exist and overwrite is False, a warning will
    appear and the data processing will be skipped.

    Returns
    ------
    pipeline_output (str): the output asdf file that the high level
    pipeline creates during processing.
    ""

    process = True
    association = l3name+'_asn.json'
    pipeline_output = l3name+'_highlevelpipeline.asdf'
    # check if L3 output files already exist
if os.path.exists(association):
    if overwrite:
        os.remove(association)
        os.remove(pipeline_output)
    else:
        warnings.warn(f"Skipping data processing. File {l3name} already exists and overwrite == {overwrite}!")
        process = False

if process:
    print(f"Running the highlevel_pipeline on {association} to prepare data for analysis...")
    # Create required association file.
    product_name = l3name
    n_elements = len(l2list)
    if n_elements == 0:
        warnings.warn(f"No input L2 files were provided!"
    if n_elements == 1:
        # Repeating the same file because romancal is not set up for single file processing
        shutil.copy(l2list[0], 'copy_'+l2list[0])
        l2list.append('copy_'+l2list[0])
        l2_list_in_str = ' '.join(l2list)
        os.system('asn_from_list --product-name=' + product_name + ' ' + l2_list_in_str + ' -o ' + association)

        # run the highlevel_pipeline
        os.system('strun --disable-crds-steppar roman_hlp ' + association + ' --output_file=' + product_name + '
                   --steps.resample.pixfrac=' + pixfrac + ' --steps.resample.pixel_scale_ratio=' + pixel_scale_ratio + ' --steps.resample.weight_type=' + weight_type)
        # originally wanted to run this programmatically instead of with the os command line
        # but it wouldn't work. See below.
        # args = ['--disable-crds-steppar',
        #         'roman_hlp',
        #         association]
        # highlevel_result = HighLevelPipeline()
        # highlevel_result.from_cmdline(args)
# highlevel_result.write_asdf(path=product_name+'.asdf')

return pipeline_output

def convert_l2_xy_to_l3(df, l2wcs, l3wcs):
    """

    Parameters
    ----------
    df (pandas dataframe): a pandas dataframe that contains three columns; x coords in the L2 frame ('x'), y coords in the L2 frame ('y'), and the associated flux value of the artificial source ('flux')

    Returns
    ------
    converted_df (pandas dataframe): a pandas dataframe with the same structure as df but the x and y coords are transformed into the L3 coordinate system
    """

    # set up conversion function
    conversion_function = resample_utils.reproject(l2wcs, l3wcs)

    # convert the xy columns from the pandas dataframe
    converted_xy = conversion_function(df.x, df.y)

    # create a new dataframe and populate it with the converted values
    converted_df = pd.DataFrame.copy(df)
    converted_df['x'] = converted_xy[0]
    converted_df['y'] = converted_xy[1]

    return converted_df

def source_detection(data, fwhm=1.0, std_multiplier=5.0):
    """
    Finds sources with photutils in a science image.

    Inputs
    ------
    data (roman_datamodels object): data of a Roman image as opened by roman_datamodels
    """
fwhm (float, default = 1.0): the full width at half max of the expected sources. The default is 1.0

std_multiplier (float, default = 5.0): the value used to specify the number of standard deviations above the background in order to be classified as a detection. The default is 5.0

Returns
--------
detections_df (pandas dataframe): a pandas dataframe that contains three columns; x coords ('xcentroid'), y coords ('ycentroid'), and the peak pixel value detected ('peak')

median = np.median(data.data.value)
std = np.std(data.data.value)
detect_func = DAOStarFinder(threshold=(std*std_multiplier), fwhm=fwhm)
detections = detect_func(data.data.value)
detections_df = detections.to_pandas()
detections_df = detections_df.drop(['id', 'sharpness', 'roundness1', 'roundness2', 'npix', 'sky', 'flux', 'mag'], axis=1)

return detections_df

def psf_photometry(psf_sources, image, extract_star_size, pixel_fit_box=(49, 49),
                   fwhm = 2.5, threshold = 10., aperture_radius = 15.):
    """
    Function that performs PSF photometry using photutils

    Inputs
    ----
    psf_sources (list): a list of sources that can be used to create a PSF model

    image (roman_datamodel): the image upon which the PSF photometry will be performed

    extract_star_size (int): the size of the cut out around the
sources being modelled for the PSF

    pixel_fit_box (list of odd integers): the size of the fitting shape used by the PSF photometry. The default is (49, 49).

    fwhm (float): the full width half max of the Roman PSF. The default is 2.5.

    threshold (float): the threshold above which a source is detected. The default is 10.

Returns
------
phot_results (astropy table): the output table containing the PSF photometry results

```
# photutils needs astropy table format
psf_sources_tab = Table.from_pandas(psf_sources)

# remove the background of the image to measure the PSF
mean_val, median_val, std_val = sigma_clipped_stats(image.data, sigma=2.0)
image_nobkg = image.data - median_val

data = image_nobkg

# Create the PSF model
epsf_builder = EPSFBuilder(oversampling=3, maxiters=10, progress_bar=False)
epsf, fitted_stars = epsf_builder(extracted_sources)

# Perform the PSF photometry
finder = DAOStarFinder(threshold, fwhm)
phot_results = PSFPhotometry(epsf, fit_shape=pixel_fit_box, finder=finder,
    aperture_radius=aperture_radius)

# Make a residual image
residual_image = psfphot.make_residual_image(image.data)
```
def main(l2data, l3data, source_filename, fwhm, threshold):
    """
    Code to do the main single pixel validation test for SOC 581. See
    science validation report for test description.

    Inputs
    ------
    l2data (roman_datamodels object): data of L2 image as opened by
    roman_datamodels

    l3data (roman_datamodels object): data of L3 image as prepared by
    prepare_data()

    source_filename (str): name of the file that contains the
    information about the artificial sources

    fwhm (float): the full width half max used in the source detection

    threshold (float): the threshold amount over the background
    measurement used to define a detection

    Returns
    -------
    None
    """

    # Open artificial source file
    l2sources = pd.read_csv(source_filename, sep=",", header=None,
    names=["x", "y", "flux"])

    # Convert L2 X&Y coordinates into the L3 coordinate frame
    l3sources = convert_l2_xy_to_l3(l2sources, l2data.meta.wcs,
    l3data.meta.wcs)

    # Detect sources in L3 output data
    l3detections = source_detection(l3data, fwhm=fwhm,
    std_multiplier=threshold)

    # Convert X&Y positions to sky coords
```python
l2sky = l2data.meta.wcs(l2sources['x'], l2sources['y'])
l3sky = l3data.meta.wcs(l3detections['xcentroid'], l3detections['ycentroid'])

# Match the two sky coordinate catalogs
l2catalog = SkyCoord(ra=l2sky[0] * u.degree, dec=l2sky[1] * u.degree)
l3catalog = SkyCoord(ra=l3sky[0] * u.degree, dec=l3sky[1] * u.degree)
idx, d2d, d3d = l3catalog.match_to_catalog_sky(l2catalog)

# Determine if the sky coords are good based on the match
# Evaluate the statistics for the distance between the L2 and L3 sources

d2d_median = np.median(d2d.degree)
d2d_std = np.std(d2d.degree)
print(f"Evaluate the distance between the L2 and L3 sources:")
print(f"The median of the 2-D distance in degrees is: {d2d_median}")
print(f"The standard deviation in degrees is: {d2d_std}")

# Add plots to evaluate any offset
d2dvsx = 'SOC_581_distance_vs_xposition_figure.png'
d2dvsy = 'SOC_581_distance_vs_yposition_figure.png'
d2dvspeak = 'SOC_581_distance_vs_peak_figure.png'
print(f"Review plots to evaluate any discrepancies: {d2dvsx}, {d2dvsy}, and {d2dvspeak}")

# D2D vs X
plt.scatter(l3detections['xcentroid'], d2d.degree, s=2.5)
plt.title('Comparison of L2 and L3 WCS Positions')
plt.xlabel('X Positions in Pixels')
plt.ylabel('2-D Distance Apart in Degrees')
plt.savefig(d2dvsx, bbox_inches='tight')
plt.close()

# D2D vs Y
plt.scatter(l3detections['ycentroid'], d2d.degree, s=2.5)
plt.title('Comparison of L2 and L3 WCS Positions')
plt.xlabel('Y Positions in Pixels')
plt.ylabel('2-D Distance Apart in Degrees')
plt.savefig(d2dvsy, bbox_inches='tight')
plt.close()
```

Released via SOCCER Database at: https://soccer.stsci.edu
# D2D vs Peak
plt.scatter(l3detections['peak'], d2d.degree, s=2.5)
plt.title('Comparison of L2 and L3 WCS Positions')
plt.xlabel('Peak Source Value (electron/sec/pixel)')
plt.ylabel('2-D Distance Apart in Degrees')
plt.savefig(d2dvspeak, bbox_inches='tight')
plt.close()

# Evaluate only brightest 2% peak of the sources
l3detections['d2d'] = d2d.degree
len_big_peak = int(0.02*len(l3detections))
big_peak_l3detections = l3detections.nlargest(len_big_peak, 'peak')
big_peak_median = big_peak_l3detections['d2d'].median()
big_peak_median = (big_peak_median*u.degree).to(u.marcsec)
print(f"Evaluate only the sources with the highest peak values (biggest 2%):")
requirement = 5.*u.marcsec
if big_peak_median <= requirement:
    print(f"PASSED: The median distance of the brightest sources (big_peak_median) is within the requirement (requirement).")
else:
   prRed(f"ISSUE: The median distance of the brightest sources (big_peak_median) is NOT within the requirement (requirement)!")

return

def reset_directory(test_name):
    """
    Code that resets the main directory and puts all test products and figures in a subdirectory.
    Inputs
    -----  
    test_name (str): the name of the test being run that defines the L3 product names and the new subdirectory
    Returns
    """

None

```

os.mkdir(test_name)
os.rename(test_name+'_asn.json',
    test_name+'/'+test_name+'_asn.json')
os.rename(test_name+'_highlevelpipeline.asdf',
    test_name+'/'+test_name+'_highlevelpipeline.asdf')
files = os.listdir()
for f in files:
    if f.startswith('SOC_581'):
        os.rename(f, test_name+'/'+f)

return

def simulated_star_analysis(l2data, l3_mosaic_name, source_list_name):
    """
    Code that checks the wcs results with simulated Roman data.
    
    Inputs
    ------
    l2data (roman_datamodel): the l2data based on the
    
    l3_mosaic_name (str): the name of the L3 mosaic product made
during the prepare_data() step
    
    source_list_name (str): the name of the file containing the L2
    artificial sources
    
    Returns
    ------
    None
    """

    # Open the L3 mosaic product
    l3data = rdm.open(l3_mosaic_name)

    # Read in the source list
    all_l2_artificial_sources = pd.read_csv(source_list_name, sep=" ",
            header=None,
            names=["fov_x", "fov_y", 'mag', 'x', 'y', 'ra',
            'dec']
```
# Select only the WFI01 sources

```
wfi01 = (all_l2_artificial_sources['fov_x'] > 180) &
        (all_l2_artificial_sources['fov_x'] < 4160) &
        (all_l2_artificial_sources['fov_y'] > -819) &
        (all_l2_artificial_sources['fov_y'] < 3161)

l2_artificial_sources = all_l2_artificial_sources[wfi01]
```

# Convert WFISim's (1,1) positions to photutil's (0,0) positions

```
12_artificial_sources['x'] = 12_artificial_sources['x'] - 1.0
12_artificial_sources['y'] = 12_artificial_sources['y'] - 1.0
```

# Convert the L2 sources into the L3 coordinate frame

```
l3_converted_sources = convert_l2_xy_to_l3(l2_artificial_sources, l2data.meta.wcs, l3data.meta.wcs)
```

# Run PSF Photometry on the L3 mosaic product

```
psf_phot_results, residual = psf_photometry(l3_converted_sources, l3data, extract_star_size=95,
                                          aperture_radius=15.)
```

# Make residual image figure to evaluate the PSF fitting

```
residual_figure = 'SOC_581_PSF_fitting_residual.png'

norm = simple_norm(residual.data, 'log', percent=99.0)
plt.imshow(residual.data, norm=norm, origin='lower', cmap='viridis')
plt.title("Residual Image from PSF Photometry on L3 Mosaic Product")
plt.colorbar()
plt.savefig(residual_figure)
plt.close()
```

# Match the L2 sources and the L3 PSF photometry

```
l2sky = l2data.meta.wcs(l2_artificial_sources['x'],
                        l2_artificial_sources['y'])
l3sky = l3data.meta.wcs(psf_phot_results['x_fit'],
                        psf_phot_results['y_fit'])

l2catalog = SkyCoord(ra=l2sky[0], dec=l2sky[1], unit='degree')
l3catalog = SkyCoord(ra=l3sky[0], dec=l3sky[1], unit='degree')

idx, d2d, d3d = l2catalog.match_to_catalog_sky(l3catalog)
```

# Plot the discrepancy between L2 and L3 positions

```
psf_comparison_figure = 'SOC_581_L2_L3_PSF_comparison.png'
quad_err = np.sqrt(psf_phot_results['x_err'][idx] ** 2. +
psf_phot_results['y_err'][idx] ** 2.) * 0.11 * u.arcsec
plt.errorbar(psf_phot_results['flux_fit'][idx], d2d.to(u.marcsec),
yerr=quad_err.to(u.marcsec), fmt='.'
plt.title('PSF Comparison of L2 and L3 positions')
plt.xlabel('PSF Flux (electron/sec)')
plt.ylabel('2-D Distance Apart (mas)')
plt.savefig(psf_comparison_figure)
plt.close()

# Calculate the statistics and evaluate the final result

d2d_weighted_mean = np.average(d2d.to(u.marcsec), weights=1.0 /
quad_err.to(u.marcsec) ** 2.)
d2d_std = np.std(d2d.to(u.marcsec))
requirement = 5.*u.marcsec
if d2d_weighted_mean <= requirement:
    print(f"PASSED: The weighted average distance of the PSF
measurements is \{d2d_weighted_mean\} +/- \{d2d_std\} "
    f"which is within the requirement \{requirement\}.")
else:
    prRed(f"ISSUE: The weighted average distance of the PSF
measurements is \{d2d_weighted_mean\} +/- \{d2d_std\} "
    f"which is NOT within the requirement \{requirement\}!")

return

# main code to run test
if __name__ == "__main__":

    # SINGLE PIXEL SIMULATED DATA TEST
    # Open L2 product
l2asdf = rdm.open('grid_wcs.asdf')

    # Define artificial source list
source_list = "pixel_grid_cat.txt"

    # Make L3 product
name = '581_wcs_pixel'
l2files = ['grid_wcs.asdf', 'grid_wcs_dq.asdf']
l3name = prepare_data(name, l2files)

    # Open L3 product
l3asdf = rdm.open(l3name)
# Set source detection values
full_width_half_max = 1.0
sigma_threshold = 20.0

# Run analysis on single pixel simulated data
main(l2asdf, l3asdf, source_list, full_width_half_max, sigma_threshold)

# Put all output products in a subdirectory
reset_directory(name)

# SIMULATED WFI PSF DATA TEST
# Open L2 product
l2simfile = rdm.open('r0000901001001001001_01101_0001_WFI01_cal.asdf')

# Define artificial L2 sources list
artificial_star_source_list = "ra80_dec0_5000_pos0.xy"

# Make L3 product
name = '581_wcs_romanpsf'
12files = ['r0000901001001001001_01101_0001_WFI01_cal.asdf',
          'r0000901001001001001_01101_0002_WFI01_cal.asdf',
          'r0000901001001001001_01101_0003_WFI01_cal.asdf',
          'r0000901001001001001_01101_0004_WFI01_cal.asdf']
l3name = prepare_data(name, l2files)

# Run PSF analysis on artificial Roman sources to verify single pixel results
simulated_star_analysis(l2simfile, l3name, artificial_star_source_list)

# Put all output products in a subdirectory
reset_directory(name)
Appendix D. SOC-582 Validation Code

""
Validation code for SOC Requirement 582.
""

import os
import shutil
import warnings
import glob
import matplotlib.pyplot as plt
import numpy as np
from math import isclose
import json
import pdb

import roman_datamodels as rdm
import roman_cal.associations.asn_from_list as asn_from_list
from roman_cal.pipeline.highlevel_pipeline import HighLevelPipeline

def prRed(s):
    ""
    Simple function to print messages to stdout in red.

    Inputs
    ------
    s (str): String to print.

    Returns
    ------
    None
    ""
    print(f"\033[91m{s}\033[00m")

def prepare_data(l3name, weight_type='exptime',
                pixel_scale_ratio='1.0', pixfrac='1.0', overwrite=False):
    ""
    Creates an association file and then runs romancal on the
    input file to generate the L3 data from

Released via SOCCER Database at: https://soccer.stsci.edu

D-1
the initial L2 data.

Inputs
------

l3name (str): the name of the association and L3 files that will be created.

weight_type (str; default = 'exptime'): The options are 'none', 'ivm', and 'exptime'. The resample pipeline default is 'ivm' but 'exptime' is more useful to evaluate the weights and exposure times of romancal.

pixel_scale_ratio (str; default = '1.0'): Ratio of input to output pixel scale used by the high level pipeline resample step. The default is 1.0.

pixfrac (str; default = '1.0'): the pixel fraction that the high level pipeline resample pipeline will use during processing. The default is 1.0.

overwrite (bool; default = False): Flag to overwrite the L3 data files if they already exist. Default is False.

A value of True will delete the existing files. If the files exist and overwrite is False, a warning will appear and the data processing will be skipped.

Returns
------

pipeline_output (str): the output asdf file that the high level pipeline creates during processing.

""

process = True
association = l3name+'_asn.json'
pipeline_output = l3name+'_highlevelpipeline.asdf'
# check if L3 output files already exist
if os.path.exists(association):
    if overwrite:
        os.remove(association)
        os.remove(pipeline_output)
else:
    prRed('Skipping data processing. Output file already exists and overwrite == False!')
    process = False

if process:
    print(f"Running the highlevel_pipeline on {association} to prepare data for analysis...")
    # Find all asdf files in the directory
    files = glob.glob('*.asdf')
    # Create required association file.
    product_name = l3name
    n_elements = len(files)
    if n_elements == 0:
        warnings.warn(f"No ASDF files were found in the current directory!")
    if n_elements == 1:
        # Repeating the same file because romancal is not set up for single file processing
        shutil.copy(files[0], 'copy_' + files[0])
        os.system(f'asn_from_list --product-name={product_name} *.asdf -o {association}')
    # run the highlevel_pipeline
    prRed('ISSUE: No way to define "good" DQ flags that need to be kept when running romancal.')
    os.system(f'strun --disable-crds-steppar roman_hlp
        +association+ --output_file='+product_name+
        ' --steps.resample.pixfrac='+pixfrac+' --
        steps.resample.pixel_scale_ratio='+pixel_scale_ratio+
        ' --steps.resample.weight_type='+weight_type)
    # originally wanted to run this programmatically instead of with the os command line
    # but it wouldn't work. See below.
    # args = ['--disable-crds-steppar',
    #         'roman_hlp',
    #         association]
    # highlevel_result = HighLevelPipeline()
def examine_resample_data(l2name, l3name, association, l3_weight, l3_pixel_scale_ratio, l3_pixfrac):
    
    Examines resultant data from prepare_data() function. The input data must have gone through the Resample step.

    Inputs
    ------
    l2name (str): Name of the input L2 file
    l3name (str): Name of the resultant L3 file
    association (str): Name of the association file used for L3 processing
    l3_weight (str): The weight type used in L3 processing
    l3_pixel_scale_ratio (str): The pixel scale ratio used in L3 processing
    l3_pixfrac (str): The pixel fraction used in L3 processing

    Returns
    ------
    None
    ""

    l2data = rdm.open(l2name)
    l3data = rdm.open(l3name)

    # SOC 582 A
    if hasattr(l3data, 'err'):
        print(f"PASSED: The processed data {l3name} contains an error array as expected."")
    else:
        print(f"ISSUE: No error attribute found in the processed data: {l3name}.")

    soc582A_figure = 'SOC_582A_error_figure.png'
    print(f"Creating PNG image of error array for visual"
inspection. See {soc582A_figure}"

    med = np.median(l3data.err.value)
    std = np.std(l3data.err.value)
    plt.imshow(l3data.err.value, origin='lower', cmap='viridis',
    vmin=med - 1 * std, vmax=med + 1 * std)
    plt.title('Error Information of L3 Output Data from High
    Level Pipeline')
    plt.colorbar()
    plt.savefig(soc582A_figure, bbox_inches='tight')
    plt.close()

# SOC 582 B
number_of_resampled_files =
float(len(l3data.meta.resample.members))
if
isclose(l2data.meta.exposure.effective_exposure_time*number_of_resampled_files,
        l3data.meta.exposure.exposure_time,
        rel_tol=0.00001):
    print(f"PASSED: L3 metadata product exposure time
matches L2 within 0.001%...")
else:
    prRed('ISSUE: L3 metadata product exposure time does not
match L2!')

# Does the context/wht image provide the fractional exposure
time of each pixel?
if hasattr(l3data, 'context'):
    print(f"PASSED: The processed data {l3name} contains a
context array as expected.")
else:
    prRed('ISSUE: No context attribute found in the
processed data: {l3name}.')

    soc582B_figure = 'SOC_582B_context_figure.png'
    print(f"Creating PNG image of context array for visual
inspection. See {soc582B_figure}"
    plt.imshow(l3data.context[0], origin='lower',
    cmap='viridis')
    plt.title('Context Information of L3 Output Data from High
    Level Pipeline')
plt.colorbar()
plt.savefig(soc582B_figure, bbox_inches='tight')
plt.close()

# SOC 582 C
# Record of association file/id and list of Level 2 images drizzled together
if l3data.meta.asn.table_name == association:
    print(f"PASSED: The correct association filename was saved in the L3 metadata."
else:
    prRed('ISSUE: L3 metadata association name was not recorded correctly!')

with open(association) as info:
    asn = json.load(info)
    if any(asn['products'][0]['members'][0]['expname'] in x for x in l3data.meta.resample.members):
        print(f"PASSED: Input image filenames from association JSON are recorded in the L3 metadata."
    else:
        prRed('ISSUE: L2 input image filenames from the association JSON are NOT recorded in the L3 metadata."

# Metadata lists the pixel scale used by resample
if l3data.meta.resample.pixel_scale_ratio:
    print(f"PASSED: The pixel scale ratio value is included in the L3 metadata."
else:
    prRed('ISSUE: Pixel scale ratio used in processing was not recorded in L3 metadata!"

# Metadata lists the CORRECT pixel scale used by resample
if l3data.meta.resample.pixel_scale_ratio == float(l3_pixel_scale_ratio):
    print(f"PASSED: The recorded pixel scale ratio matches the value selected for L3 processing."
else:
    prRed('ISSUE: The pixel scale ratio in the L3 metadata DOES NOT match the value selected for L3 processing!')
# Metadata lists the pixel fraction used by resample
if l3data.meta.resample.pixfrac:
    print(f"PASSED: The pixel fraction is included in the L3 metadata.")
else:
    prRed('ISSUE: Pixel fraction value used in processing was not recorded in L3 metadata!')

# Metadata lists the CORRECT pixel fraction used by resample
if l3data.meta.resample.pixfrac == float(l3_pixfrac):
    print(f"PASSED: The recorded pixel fraction matches the value selected for L3 processing.")
else:
    prRed('ISSUE: The pixel fraction in the l3 metadata DOES NOT match the value selected for L3 processing!')

# Metadata lists the weight type used by resample
if l3data.meta.resample.weight_type:
    print(f"PASSED: The resample weight type is included in the L3 metadata.")
else:
    prRed('ISSUE: Weight type used in processing was not recorded in L3 metadata!')

# Metadata lists the CORRECT weight type used by resample
if l3data.meta.resample.weight_type == l3_weight:
    print(f"PASSED: The recorded weight type matches the value selected for L3 processing.")
else:
    prRed('ISSUE: The weight type in the l3 metadata DOES NOT match the value selected for L3 processing!')

# Record of the output log from resample - no option to ask for a saved output file in resample call
prRed('ISSUE: Cannot find log output or information from the resample run in the metadata!')

# Metadata contains the original background level of the individual images combined by resample
prRed('ISSUE: Cannot find list of original background levels of individual images before resampled!')
# Metadata records what DQ flags were accepted/rejected during the resample step.

# No option to provide list of "good" DQ flags when calling resample.
prRed('ISSUE: Resample metadata does not contain the list of DQ flags kept/rejected.\\)

# Metadata records the WCS solution name? And what catalog was used to derive it?
if l2data.meta.coordinates.reference_frame == l3data.meta.coordinates.reference_frame:
    print(f"PASSED: L3 metadata coordinate reference frame name matches L2 input image.\\")
else:
    prRed(f"ISSUE: L3 metadata coordinate reference frame name does not match input L2!\\")

return

def main(l2name, l3name, l3association, l3_pipeline_weight_type, l3_pipeline_pixel_scale, l3_pipeline_pixfrac):
    """
    Code to do the validation test for SOC 582. See science validation report for test description.
    
    Inputs
    ------
    l2name (str): name of L2 data
    l3name (str): name of L3 data as prepared by prepare_data()
    l3association (str): name of association used by prepare_data()
    l3_pipeline_weight_type (str) = weight type specified in L3 processing
    l3_pipeline_pixel_scale (str) = pixel scale ratio specified in L3 processing
    l3_pipeline_pixfrac (str) = pixel fraction specified in L3 processing
    
    Returns
    """
None

```
print(f"\nWorking on resampled data {l3name}..."

    examine_resample_data(l2name, l3name, l3association,
    l3_pipeline_weight_type,
                        l3_pipeline_pixel_scale,
    l3_pipeline_pixfrac)

    return

def reset_directory(test_name):
    
    Code that resets the main directory and puts all test
    products and figures in a subdirectory.

    Inputs
    -----
    test_name (str): the name of the test being run that defines
    the L3 product names and the
    new subdirectory

    Returns
    -----
    None

    os.mkdir(test_name)
    os.rename(l3asn, test_name+'/'+l3asn)
    os.rename(test_name+'_highlevelpipeline.asdf',
                 test_name+'/'+test_name+'_highlevelpipeline.asdf')
    files = os.listdir()
    for f in files:
        if f.startswith('SOC_582'):
            os.rename(f, test_name+'/'+f)

    return
```
# main code to run test
if __name__ == "__main__":

    # Verify with single pixel sources in L2 data
    name = '582_default_test'
    l2asdf = 'grid_wcs.asdf'

    # Use default resample values
    selected_weight_type = 'exptime'
    selected_pixel_scale_ratio = '1.0'
    selected_pixfrac = '1.0'
    l3asdf, l3asn = prepare_data(name,
                              weight_type=selected_weight_type,
                              pixel_scale_ratio=selected_pixel_scale_ratio,
                              pixfrac=selected_pixfrac)

    main(l2asdf, l3asdf, l3asn, selected_weight_type,
         selected_pixel_scale_ratio, selected_pixfrac)

    # Move the new files into a subdirectory
    reset_directory(name)

    # Repeat with different resample values
    name = '582_scale_ratio_05_test'
    l2asdf = 'grid_wcs.asdf'

    # Use default resample values
    selected_weight_type = 'exptime'
    selected_pixel_scale_ratio = '0.5'
    selected_pixfrac = '0.5'
    l3asdf, l3asn = prepare_data(name,
                              weight_type=selected_weight_type,
                              pixel_scale_ratio=selected_pixel_scale_ratio,
                              pixfrac=selected_pixfrac)

    main(l2asdf, l3asdf, l3asn, selected_weight_type,
         selected_pixel_scale_ratio, selected_pixfrac)

    # Move the new files into a subdirectory
    reset_directory(name)
Appendix E. SOC-666 and SOC-669 ELP Validation Code

"""
Validation code for SOC Requirements 666 and 669.
Excludes jump detection and ramp fitting, which are
in separate Python files.
"""

import os
import asdf
import glob
import shutil
import numpy as np
import astropy.units as u
import roman_datamodels as rdm
from crds.client import api
from crds import getrecommendations
from romancal.lib import dqflags

from romancal.dq_init.dq_init_step import DQInitStep
from romancal.saturation.saturation_step import SaturationStep
from romancal.linearity.linearity_step import LinearityStep
from romancal.dark_current.dark_current_step import DarkCurrentStep
from romancal.refpix import RefPixStep

from roman_datamodels.maker_utils._ref_files import mk_mask

import matplotlib.pyplot as plt

data_dir = './'

def prRed(s):
    """
    Simple function to print messages to stdout in red.

    Inputs
    ------
    s (str): String to print.
    """

     Released via SOCCER Database at: https://soccer.stsci.edu

E-1
Returns
-------
None

""
print(f"\033[91m{s}\033[00m")

```python
def get_powers_of_2(x):
    ""
    Function to decompose a number into its powers of 2.
    
    Inputs
    ------
    x (int): Number to decompose.
    
    Returns
    -------
    powers (list): List of the exponents of 2 that make up x.
    ""
    powers = []
    i = 0
    while x:
        if x & 1:
            powers.append(i)
            x >>= 1
        i += 1
    return powers
```

def soc_666_data_prep(soc_666_uncal):
    ""
    Function to prepare SOC-666 data by getting CRDS reference files for the test input file, downloading and copying them to accessible directory for tests.
    
    Inputs
    ------
    soc_666_uncal (obj): The uncalibrated asdf file test object.
Returns
-------
soc_666_ref_files (list): List of CRDS reference files names for test file.

""
# Construct the selector information for CRDS.
selectors = {'ROMAN.META.INSTRUMENT.NAME': 'WFI',
             'ROMAN.META.INSTRUMENT.DETECTOR': soc_666_uncal.meta.instrument.detector,
             'ROMAN.META.INSTRUMENT.OPTICAL_ELEMENT': soc_666_uncal.meta.instrument.optical_element,
             'ROMAN.META.EXPOSURE.START_TIME': soc_666_uncal.meta.exposure.start_time.isot,
             'ROMAN.META.EXPOSURE.MA_TABLE_NUMBER': soc_666_uncal.meta.exposure.ma_table_number,
             'ROMAN.META.EXPOSURE.TYPE': soc_666_uncal.meta.exposure.type}
reftypes = ['mask', 'saturation', 'refpix', 'linearity', 'dark']
context = "roman_0056.pmap"
soc_666_crds_files = getrecommendations(selectors, reftypes=reftypes, observatory='roman', context=context)
soc_666_ref_files = [filename for filename in soc_666_crds_files.values()]
results = api.dump_references(context, soc_666_ref_files)
# dump_references() downloads files to /home/rcosenti/tmp_files/references/roman/wfi/
# copying to data_dir if they aren't already there
for file_name, source_path in results.items():
    destination_path = os.path.join(data_dir, file_name)
    if not os.path.exists(destination_path):
        shutil.copy(source_path, destination_path)
        print(f'File "{file_name}" copied to "{data_dir}"')
    else:
        print(f'File "{file_name}" already exists in "{data_dir}"')

return soc_666_ref_files
def soc_669_data_prep(soc_669_uncal):
"""
Function to prepare SOC-669 data by getting CRDS reference files for the test input file, downloading and copying them to accessible directory for tests.

Inputs
------
soc_669_uncal (obj): The uncalibrated asdf file test object.

Returns
------

```
# Construct the selector information for CRDS.
selectors = {'ROMAN.META.INSTRUMENT.NAME': 'WFI',
             'ROMAN.META.INSTRUMENT.DETECTOR': soc_669_uncal.meta.instrument.detector,
             'ROMAN.META.INSTRUMENT.OPTICAL_ELEMENT': soc_669_uncal.meta.instrument.optical_element,
             'ROMAN.META.EXPOSURE.START_TIME': soc_669_uncal.meta.exposure.start_time.isot,
             'ROMAN.META.EXPOSURE.MA_TABLE_NUMBER': soc_669_uncal.meta.exposure.ma_table_number,
             'ROMAN.META.EXPOSURE.TYPE': soc_669_uncal.meta.exposure.type}
reftypes = ['mask', 'saturation', 'refpix', 'linearity', 'dark']
context = "roman_0056.pmap"
soc_669_crds_files = getrecommendations(selectors, reftypes=reftypes, observatory='roman', context=context)
soc_669_ref_files = [filename for filename in soc_669_crds_files.values()]
results = api.dump_references(context, soc_669_ref_files)
# dump_references() downloads files to /home/rcosenti/tmp_files/references/roman/wfi/
# copying to data_dir if they aren't already there
for file_name, source_path in results.items():
    destination_path = os.path.join(data_dir, file_name)
    if not os.path.exists(destination_path):
        shutil.copy(source_path, destination_path)
```
print(f'File "{file_name}" copied to "{data_dir}"')
else:
    print(f'File "{file_name}" already exists in "{data_dir}"."

return soc_669_ref_files

def make_new_mask(orig_mask):
    
    Function to make new mask reference file that will place all
dq flag bits
    along the pixels [100,100], [101,101], ...[131,131] so each
    bit is assigned
    at the pixel 100 + bit value.

    Inputs
    ------
    orig_mask (obj): The CRDS reference file mask object needed
to get meta data
to have same matching criteria.

    Returns
    ------
    fl_name (str): Filename generated for new CRDS mask and
    passed into DQInitstep
    with override option and tested for use.

    
    fl_name = 'roman_mask_soc666_SM1.asdf'
outfile = data_dir + fl_name
print('Making new mask reference file ', fl_name)
mask_image = np.zeros((4096, 4096), dtype=np.uint32)
for i in range(0, 31):
    mask_image[i + 100, i + 100] = 2 ** i

mask = mk_mask(dq=mask_image, filepath=outfile)
os.chmod(outfile, 0o666)

return fl_name
def test_DQstep(data_uncal, orig_mask_ref_fl):
    
    Function for tests of SOC-666 and SOC-669 for the DQInit step.

    Inputs
    ------
    data_uncal (obj): The uncalibrated asdf file test object.
    orig_mask_ref_fl (str): Reference file name from CRDS.

    Returns
    ------
    tests (list): List of boolean True or False statements for each test result.

    tests = []
    result = DQInitStep.call(data_uncal)
    # Test that correct reference file from CRDS was used in DQ step on uncal object
    test_ref_match = result.meta.ref_file.mask[7:] == orig_mask_ref_fl
    tests.append(test_ref_match)
    orig_mask = rdm.open(data_dir + orig_mask_ref_fl)
    # Find original mask values should be for GOOD, DO_NOT_USE, and REFERENCE_PIXEL
    orig_mask_values = np.unique(orig_mask.dq)
    dqflag_defs = dqflags.pixel
    
    # Test the original mask for the 3 expected values
    tests.append(np.isin(dqflag_defs['GOOD'], orig_mask_values))
    tests.append(np.isin(dqflag_defs['DO_NOT_USE'], orig_mask_values))
    tests.append(np.isin(dqflag_defs['REFERENCE_PIXEL'], orig_mask_values))

    # Test the result of the DQinit step with the original mask to find flagged bits:
    # DO_NOT_USE, GW_AFFECTED_DATA, and REFERENCE_PIXEL by decomposing values into integer
# powers of 2.
# This approach ignores GOOD pixels because 0 is not a power of 2 for any integer.
all_result_DQstep_mask_values = np.unique(result.pixeldq)
bits = []
for i in range(0, len(all_result_DQstep_mask_values)):
    tmp = get_powers_of_2(all_result_DQstep_mask_values[i])
bits.append(tmp)  # These are all bits
bits = bits[1:]  # Removing empty set for GOOD no power of 2 equals zero
# Flatten all bits and make a unique list of found values.
flat_bits = [item for sublist in bits for item in sublist if isinstance(sublist, list)]
unique_values = 2**np.unique(flat_bits)
tests.append(np.isin(dqflag_defs['DO_NOT_USE'], unique_values))
tests.append(np.isin(dqflag_defs['GW_AFFECTED_DATA'], unique_values))
tests.append(np.isin(dqflag_defs['REFERENCE_PIXEL'], unique_values))

# Now use new mask file to run DQstep on data and check that each
# bit is found in pixels 100,100....131,131
new_mask_ref_fl = make_new_mask(orig_mask)
DQresult = DQInitStep.call(data_uncal,
    override_mask=data_dir + new_mask_ref_fl,
    save_results=False)  # output_file = data_dir + "soc_666_DQresult",
# Test new mask file used in DQinit step.
# New mask file name is not from CRDS - test whole string to match.
test_new_ref_match = DQresult.meta.ref_file.mask == new_mask_ref_fl
# print(test_new_ref_match)
tests.append(test_new_ref_match)

# Test new values in mask from range of 0-30
for i in range(0, 31):
    bit = get_powers_of_2(DQresult.pixeldq[i + 100, i +
test_new_bit_mask = bit[0] == i
# print(test_new_bit_mask)
tests.append(test_new_bit_mask)

# Close asdf objects
result.close()
DQresult.close()
return tests

def test_SATstep(data_uncal, saturation_ref_fl):
    
    Function for tests of SOC-666 and SOC-669 for the Saturation step.

    Inputs
    ------
    data_uncal (obj): The uncalibrated asdf file test object.
    saturation_ref_fl (str): Reference file name from CRDS.

    Returns
    ------
    tests (list): List of boolean True or False statements for each test result.

    tests = []
    result = DQInitStep.call(data_uncal)
    result = SaturationStep.call(result)
    # Test that correct reference file from CRDS was used in Saturation step on uncal object
    test_ref_match = result.meta.ref_file.saturation[7:] == saturation_ref_fl
    tests.append(test_ref_match)

    data_SATuncal= data_uncal.copy()
    data_SATuncal.data[0, 200:205, 200] = [50, 54999, 54999.9, 54999.99, 60000]* u.DN
    data_SATuncal.data[2, 200:205, 201] = [100, 54999, 54999.99, 55000.00, 65500.0] * u.DN
    data_SATuncal.data[-1, 200:205, 202] = [1000, 54999, 55000,
55000.01, np.finfo(np.float16).max] * u.DN

data_SATuncal.data[:, 203, 203] = 55000.1 * u.DN

sat_ref = rdm.open(data_dir + saturation_ref_fl)
sat_limit = np.unique(sat_ref.data)
res1_sat_flags = data_SATuncal.data[0, 200:205, 200].value >= sat_limit.value
res3_sat_flags = data_SATuncal.data[2, 200:205, 201].value >= sat_limit.value
res5_sat_flags = data_SATuncal.data[-1, 200:205, 202].value >= sat_limit.value
res_all_sat_flags = data_SATuncal.data[:, 203, 203].value >= sat_limit.value

result = DQInitStep.call(data_SATuncal)
SATresult = SaturationStep.call(result)

dqflag_defs = dqflags.pixel
res1_SATresult_flagged = SATresult.groupdq[0, 200:205, 200] == dqflag_defs['SATURATED']
tests.append(np.array_equal(res1_SATresult_flagged, res1_sat_flags))
res3_SATresult_flagged = SATresult.groupdq[2, 200:205, 201] == dqflag_defs['SATURATED']
tests.append(np.array_equal(res3_SATresult_flagged, res3_sat_flags))
res5_SATresult_flagged = SATresult.groupdq[-1, 200:205, 202] == dqflag_defs['SATURATED']
tests.append(np.array_equal(res5_SATresult_flagged, res5_sat_flags))

# do not do for now - Tyler discussion 2/20/24
# resall_SATresult_flagged = SATresult.groupdq[0, 200:205, 200] == dqflag_defs['SATURATED']
# tests.append(np.array_equal(resall_SATresult_flagged, res_all_sat_flags))

result.close()
SATresult.close()
return tests
def test_LINstep(data_uncal, linearity_ref_fl):
    
    """
    Function for tests of SOC-666 and SOC-669 for the Linearity step.
    
    Inputs
    ------
    data_uncal (obj): The uncalibrated asdf file test object.
    linearity_ref_fl (str): Reference file name from CRDS.
    
    Returns
    ------
    tests (list): List of boolean True or False statements for each test result.
    """

    #data_uncal = rdm.open(data_dir + fl_WIM)
    #linearity_ref_fl = soc_666_ref_files[3]

    tests = []
    result = DQInitStep.call(data_uncal)
    input_2_LIN = SaturationStep.call(result)
    input_values = np.uint16([100, 1000, 5000, 10000, 50000])
    input_2_LIN.data[0, 400, 400:405] = input_values * u.DN
    LINresult = LinearityStep.call(input_2_LIN)  # why is this updating the data model in the input
    #print(input_2_LIN.data[0, 400, 400:405])
    #print(LINresult.data[0, 400, 400:405])

    # Test that correct reference file from CRDS was used in Linearity step on uncal object
    test_ref_match = LINresult.meta.ref_file.linearity[7:] == linearity_ref_fl
    tests.append(test_ref_match)
    lin_ref = rdm.open(data_dir + linearity_ref_fl)
    for i in range(0, 5):
        corrected_value = np.polyval(lin_ref.coeffs[:-1, 400, 400+i], input_values[i])
        #print(corrected_value, LINresult.data[0, 400, 400+i].value)
ratio_2_test = LINresult.data[0, 400, 400+i].value / corrected_value == 1
    tests.append(ratio_2_test)

LINresult.close()
input_2_LIN.close()
lin_ref.close()
return tests

def test_DARKstep(data_uncal, dark_ref_fl):
    
    Function for tests of SOC-666 and SOC-669 for the Dark Current Subtraction step.

    Inputs
    ------
    data_uncal (obj): The uncalibrated asdf file test object.
    dark_ref_fl (str): Reference file name from CRDS.

    Returns
    ------
    tests (list): List of boolean True or False statements for each test result.

    # data_uncal = rdm.open(data_dir + fl_WIM)
    # dark_ref_fl = soc_666_ref_files[4]

    tests = []
    result = DQInitStep.call(data_uncal)
    result = SaturationStep.call(result)
    result = LinearityStep.call(result)
    input_2_DARK = result.data.copy()
    DARKresult = DarkCurrentStep.call(result)
    # DARKresult.data == result.data these shouldn't be equal in my opinion or an option to not be

    # Test that correct reference file from CRDS was used in Dark step on uncal object
    test_ref_match = DARKresult.meta.ref_file.dark[7:] ==
dark_ref_f1
tests.append(test_ref_match)
dark_ref = rdm.open(data_dir + dark_ref_f1)
# Open reference file and then subtract it from the
input_2_DARK and test that it is equal to the
# output from the Dark Current Step.
tests.append(input_2_DARK - dark_ref.data ==
DARKresult.data)

DARKresult.close()
dark_ref.close()
return tests

def test_REFPIXstep(save_plot=False,
plotfile='./Uncal_RefPixStep_diff.png'):
    ""
    Function to use the weights calculated from a TVAC 1 data
set that are saved in
    a reference pixel reference file in asdf format that is used
in the RefPix step
    in romanCal. The function generates a plot file that shows
    the difference between
    the input data into the RefPix step and the output. The plot
generally shows
    horizontal banding residuals indicative of 1/f noise in the
detector data.

    Inputs
    ------
    save_plot (bool): Boolean to save plot file; default = False.
    plotfile (str): Plot filename written to current directory.

    Function Artefacts
    ------
    uncal_tvac_data (file): File in asdf format from TVAC1.
    saturation_ref_fl (str): Reference file name from CRDS.

    NOTES:
    1 - Reference pixel reference files are actively in
development and are
not accessible from CRDS as of this time.

2 - Demonstration of general methodology is in place as shown here while
the final implementation of the algorithm and romancal as well as reference
file details are actively being explored.

Output
-------

Figure (png): Plot showing the difference of one frame from an uncalibrated
data set and the output from the reference pixel correction showing 1/f noise.

```python
uncal_tvac_file =
'/grp/roman/TEST_DATA/IRRC/irrc_test_WFI06_uncal.asdf'
uncal_tvac = rdd.open(uncal_tvac_file)

refpix_ref_file =
'/grp/roman/TEST_DATA/IRRC/irrc_test_WFI06_weights.asdf'
result = DQInitStep.call(uncal_tvac, save_results=False)

# Extract a single frame from the tvac exposure to compare the output from
# the RefPix step to the input
uncal_frame = np.copy(result.data[3,:,:].value)

result = RefPixStep.call(result,
result,
override_refpix=refpix_ref_file,
remove_offset=True,
remove_trends=True,
cosine_interpolate=True,
fft_interpolate=True,
)

if save_plot is True:
plt.figure()
plt.imshow(uncal_frame - result.data[3,:,:].value,
```
def main(run=True):
    """
    Code to do the validation test. See science validation report for test description.
    Eliminating files from repo for tracking.
    """
    # Select WIM file from data
    fl_WIM = 'r0066601001001001001_01101_0001_WFI01_uncal.asdf'
    soc_666_uncal = rdm.open(data_dir + fl_WIM)
    soc_666_ref_files = soc_666_data_prep(soc_666_uncal)
    print(soc_666_ref_files)

    soc_666_tests = []

    DQstep_test_results = test_DQstep(soc_666_uncal, soc_666_ref_files[0])
    print(DQstep_test_results)
    soc_666_tests.append(DQstep_test_results)
    #soc_666_tests.append(False)

    SATURATIONstep_test_results = test_SATstep(soc_666_uncal, soc_666_ref_files[1])
    print(SATURATIONstep_test_results)
    soc_666_tests.append(SATURATIONstep_test_results)
    #soc_666_tests.append(False)

    LINEARITYstep_test_results = test_LINstep(soc_666_uncal, soc_666_ref_files[3])
    print(LINEARITYstep_test_results)
soc_666_tests.append(LINEARITYstep_test_results)  
#soc_666_tests.append(False)

DARKstep_test_results = test_DARKstep(soc_666_uncal,  
soc_666_ref_files[4])
print(DARKstep_test_results)
soc_666_tests.append(DARKstep_test_results)

#soc_666_tests.append(False)
print(soc_666_tests)
if False in soc_666_tests:
    prRed(' tTEST FAILED!')
else:
    print(' tTEST PASSED!')

# Select WSM file from data
fl_WIM = 'r00669010010001001_01101_0001_WFI01_uncal.asdf'
soc_669_uncal = rdm.open(data_dir + fl_WIM)
soc_669_ref_files = soc_669_data_prep(soc_669_uncal)
print(soc_669_ref_files)

soc_669_tests = []

DQstep_test_results = test_DQstep(soc_669_uncal,  
soc_669_ref_files[0])
print(DQstep_test_results)
soc_669_tests.append(DQstep_test_results)  
#soc_669_tests.append(False)

SATURATIONstep_test_results = test_SATstep(soc_669_uncal,  
soc_669_ref_files[1])
print(SATURATIONstep_test_results)
soc_669_tests.append(SATURATIONstep_test_results)  
#soc_669_tests.append(False)

LINEARITYstep_test_results = test_LINstep(soc_666_uncal,  
soc_669_ref_files[3])
print(LINEARITYstep_test_results)
soc_669_tests.append(LINEARITYstep_test_results)
#soc_669_tests.append(False)

DARKstep_test_results = test_DARKstep(soc_669_uncal,
soc_669_ref_files[4])
print(DARKstep_test_results)
soc_669_tests.append(DARKstep_test_results)

print(soc_669_tests)
if False in soc_669_tests:
    prRed('
TEST FAILED!')
else:
    print('
TEST PASSED!')

if __name__ == '__main__':
    main(True)
Appendix F.  SOC-666 and SOC-669 Jump Detection Code

from astropy.time import Time
from collections import defaultdict
from romancal.dq_init import DQInitStep
from romancal.saturation import SaturationStep
from romancal.refpix import RefPixStep
from romancal.linearity import LinearityStep
from romancal.dark_current import DarkCurrentStep
from romancal.jump import JumpStep
from romancal.lib import dqflags
from roman_datamodels.maker_utils import (mk_level1_science_raw,
                                          mk_refpix,
                                          mk_ref_dark_meta,
                                          mk_ref_common,
                                          mk_ref_readnoise_meta,
                                          mk_ref_units_dn_meta,
                                          save_node)
import astropy.units as u
import numpy as np
import roman_datamodels.stnode as rds

from astropy.io import fits
import asdf
from pathlib import Path

DT_IMG = 3.04 # seconds, per Casertano (2022) pg. 7
DT_SPEC = 4 # seconds, per Casertano (2022) pg. 7
SEED = 1

def prRed(s):
    """
    Simple function to print messages to stdout in red.
    """
    Inputs
    ------
    s (str): String to print.

    Released via SOCCER Database at: https://soccer.stsci.edu

F-1
Returns
-------
None
"

print(f"\033[91m {s}\033[00m")

def create_ma_table(name, borders, pattern_dict=None, in_place=True):
    ""
    Save information about a readout pattern to a dictionary.

    name (str): The intended key to access the pattern in pattern_dict.
    borders (list): The first read in each resultant. (This function is written
        assuming that no reads are skipped.)
    pattern_dict (defaultdict, optional): A pre-existing default dictionary of
        readout patterns to which a new entry will be added. A new dict is
        generated from scratch and returned if this argument is None. Keys are
        'borders' (explained earlier) and 'sequence', a list of arrays where
        each array corresponds with a resultant and contains its matching read
        numbers.
    in_place (bool, optional): Modified image is not returned if True (default)
        or is returned if False.
    ""

    if pattern_dict is None:
        in_place = False
        pattern_dict = defaultdict(dict)

    pattern_dict[name]['borders'] = borders
    pattern_dict[name]['sequence'] = [np.arange(borders[i], borders[i + 1])
                                      for i in
                                      range(len(borders) - 1)]
if not in_place:
    return pattern_dict

def create_image(shape, dt, count_rate, patt, read_noise=1e-16,
                 incl_read_noise=False,
                 incl_poisson_noise=False, seed=SEED):
    
    Simulate a resultant-averaged image of ramp fluxes.

    shape (int): The number of pixels that make up the image's x
    and y dimensions.
    dt (float): The amount of time that elapses during a read.
    See DT_IMG and
    DT_SPEC at top of file for canonical Roman values.
    count_rate (float): The amount of scientific signal added to
    the image,
    measured in counts per second.
    patt (list): A list of arrays indicating a readout pattern.  
    Each array
    corresponds with a resultant and contains its matching 
    read numbers.
    read_noise (float, optional): The amount of read noise to
    add to the ramps.
    incl_read_noise (bool, optional): Whether or not to add read
    noise to add
    to the ramps.
    incl_poisson_noise (bool, optional): Whether or not to add 
    Poisson noise to
    add to the ramps.
    seed (int, optional): The random seed used to calculate
    noise.  (No effect
    if both incl_*_noise arguments are False.)
    
    last_read = patt[-1][-1]

    # Record flux at every read
    counts_by_time = dt * count_rate * np.linspace(1, last_read,
        last_read)
    img_counts = np.dstack(counts_by_time).T *
    np.ones((last_read, shape, shape))
# If requested, add noise to each read
if incl_read_noise:
    rng = np.random.default_rng(seed)
    img_counts += np.cumsum(read_noise
                          * rng.normal(size=img_counts.shape), axis=0)
if incl_poisson_noise:
    rng = np.random.default_rng(seed)
    img_counts += np.cumsum(rng.poisson(count_rate * dt,
                          size=img_counts.shape), axis=0)

# Average fluxes by resultant according to readout pattern
img = np.array([img_counts[res - 1].mean(axis=0) for res in
                          patt])
# (subtract 1 because pattern is 1-indexed and t is zero-indexed)

# img[7:, 19:22, 19:22] += 5e1 # proto-simulation of a jump

return img

def simulate_jump_explicit(img, n_res, x_peak_pix, y_peak_pix,
                           peak_jump=250):
    ""
    Apply a jump to a 3x3 area of a specific resultant and all
    the ones
    following it in a pre-existing image. Jumps are applied in a
    3x3 area around
    the specified central pixel. Returns the modified image
    alongside the index
    of the jump's central pixel.
    
    img (array): A 3-D array of resultant-averaged ramp flux
    values.
    n_res (int): The (zero-indexed) index of the resultant where
    the cosmic ray
        will be applied first.
    *_peak_pix (int): The pixel in the * direction that receives
    the most flux
from the simulated cosmic ray. Its neighbors in a 3x3
grid receive half
of the peak signal.
peak_jump (float): The amount of signal, in counts, added to
the center
pixel. Default value of 250 is a typical peak pixel
amplitude per
pg. 4 of Sharma+ (2023).

""
# Apply the jump to a 3x3 area around the chosen pixel from
the chosen
# resultant onward, then add extra signal to differentiate
the peak pixel
img[n_res:, x_peak_pix-1:x_peak_pix+2, y_peak_pix-
1:y_peak_pix+2] += peak_jump / 2
img[n_res:, x_peak_pix, y_peak_pix] += peak_jump / 2

return img, np.array([n_res, x_peak_pix, y_peak_pix])

def simulate_jump_random(img, dt, patt_info, peak_jump=250,
seed=SEED):

""
Randomly add jumps to a pre-existing image based on cosmic
ray probability
from Sharma+ (2023).
(4.03e-6 jumps per pixels per second comes from pg. 4 of
Sharma+ (2023))
Jumps can occur in any 3x3 area that doesn't overlap a
reference pixel, with
the outer pixels receiving half the flux as the central
pixel.
Jumps can occur during any read, so those that occur in the
middle of a
resultant are weighted appropriately. Returns the modified
image alongside
lists of the read-based and resultant-based indices where
jumps were applied.

img (array): A 3-D array of resultant-averaged ramp flux
values.

dt (float): The amount of time that elapses during a read.
See DT_IMG and
patt_info (dict): An inner dictionary from the ma_tables dict with keys 'borders' and 'sequence'. See create_ma_table() docstring for more on its contents.

peak_jump (float): The amount of signal, in counts, added to the center pixel. Default value of 250 is a typical peak pixel amplitude per pg. 4 of Sharma+ (2023).

seed (int, optional): The random seed used to select jump locations.

```
last_read = patt_info['sequence'][-2][-1]  
# (-2 index disallows last resultant, where jumps won't be detected)

# Add a probabilistically realistic number of cosmic rays to img based on its resolution and number of reads (not resultants)
prob_jump = 4.03e-6 * dt
# if dt = 3.04 seconds, multiply that in to get probability of 1.23e-5 per pixel
n_jumps = np.ceil(img.size * prob_jump).astype(int)
# 2.56 occurrences with 51 reads (HDR-150_1) and a 64x64 image
# 10.24 occurrences with 51 reads (HDR-150_1) and a 128x128 image

# Randomly select read/pixel combination(s) to which jumps will be applied
img_all_reads_shape = (last_read,) + (img.shape[1:]):
reads_of_jumps = []
for i in range(n_jumps):
    # Divide pixels into n_jumps groups and select one from each bin
    first_elig_pixel = int(i / n_jumps * np.prod(img_all_reads_shape))
    reads_of_jumps.append((last_read+i,) + (img.shape[1:]))
```
last_elig_pixel = int((i + 1) / n_jumps * np.prod(img_all_reads_shape))

# Skip this group of pixels if all are part of the zeroth read
if last_elig_pixel < (img_all_reads_shape[1] * img_all_reads_shape[2]):
    continue

# Randomly select a pixel from the bin, then convert it to a 3-D index
# suitable for slicing an image containing all reads while True:
    try:
        read_ind = np.array(np.unravel_index(rng.integers(first_elig_pixel,
                                     last_elig_pixel),
                                     img_all_reads_shape))

        # Disallow jumps in zeroth read or in 4 border reference pixels
        # (since jumps are being simulated over a 3x3 area, an extra 1
        # keeps neighbor pixels out of reference area, too)
        if any((read_ind[0] == 0) | (read_ind[1:] < 4 + 1) | (read_ind[1:] >= img.shape[1] - 4 - 1)):
            raise IndexError
        else:
            read_ind[0] += 1 # bump up since MA patterns are 1-indexed
            break
    except IndexError:
        continue
reads_of_jumps.append(read_ind)

# Convert read-based indices to resultant space then add
signals from cosmic rays to img, accounting for partial effects from mid-resultant jumps

jump_inds = []

for all_reads_ind in reads_of_jumps:
    read = all_reads_ind[0]
    x_ind = all_reads_ind[1]
    y_ind = all_reads_ind[2]

    # Find the chosen read's matching resultant, then calculate a weight
    # for the jump signal based on when in the resultant it occurred
    res_bin = np.searchsorted(patt_info['borders'], read, side='right') - 1
    # (modified side arg corrects sorting in single-read resultants)
    res_length = patt_info['sequence'][res_bin].size
    jump_weight = (res_length - np.searchsorted(patt_info['sequence'][res_bin], read)) / res_length

    # Add (potentially) partially weighted jump to that resultant
    img[res_bin, x_ind-1:x_ind+2, y_ind-1:y_ind+2] += jump_weight * (peak_jump / 2)
    img[res_bin, x_ind, y_ind] += jump_weight * (peak_jump / 2)

    # Add fully weighted jump to all resultants afterward
    img[res_bin+1:, x_ind-1:x_ind+2, y_ind-1:y_ind+2] += peak_jump / 2
    img[res_bin+1:, x_ind, y_ind] += peak_jump / 2

    # Save all altered indices in format that can slice an array
    jump_inds.append((res_bin, np.s_[x_ind-1:x_ind+2], np.s_[y_ind-1:y_ind+2]))

return img, reads_of_jumps, jump_inds
def mock_L1_image(img, dt, patt, img_path='mock_synth.asdf'):
    ""
    Create a simulated a Level 1 ASDF file according to the
    characteristics
    of a previously generated image.

    img (array): A 3-D array of resultant-averaged ramp flux
    values.
    dt (float): The amount of time that elapses during a read.
    See DT_IMG and
    DT_SPEC at top of file for canonical Roman values.
    patt (list): A list of arrays indicating a readout pattern.
    Each array
    corresponds with a resultant and contains its matching
    read numbers.
    img_path (str, optional): The path to which the image will
    be saved.
    ""
    # Generate mock L1 file
    mock_synth = mk_level1_science_raw(shape=img.shape)

    t_init = Time('2023-12-15 16:35:11')
    last_read = patt[-1][-1]

    # Fill appropriate fields in the exposure datamodel
    according to img's properties
    mock_synth['meta']['exposure']['start_time'] = t_init
    mock_synth['meta']['exposure']['mid_time'] = t_init + .5 *
    (dt * last_read) * u.s
    mock_synth['meta']['exposure']['end_time'] = t_init + dt *
    last_read * u.s
    # MJD and TDB variants don't seem to be needed
    mock_synth['meta']['exposure']['ngroups'] = len(patt) #
    number of resultants
    mock_synth['meta']['exposure']['nframes'] = last_read
    mock_synth['meta']['exposure']['data_problem'] = False
    mock_synth['meta']['exposure']['sca_number'] = 1
    mock_synth['meta']['exposure']['gain_factor'] = 1
    mock_synth['meta']['exposure']['integration_time'] = dt *
    last_read
mock_synth['meta']['exposure']['elapsed_exposure_time'] = dt * last_read
    # skipping frame_divisor since some patterns have uneven resultants
mock_synth['meta']['exposure']['groupgap'] = 0  # no dropped reads in chosen patterns
mock_synth['meta']['exposure']['frame_time'] = dt
    # skipping group_time
mock_synth['meta']['exposure']['exposure_time'] = dt * last_read
mock_synth['meta']['exposure']['effective_exposure_time'] = dt * last_read
mock_synth['meta']['exposure']['duration'] = dt * last_read
mock_synth['meta']['exposure']['ma_table_name'] = 'DMS_R2_validation'
    # skipping ma_table_number since these patterns are custom
    # skipping level0_compressed blank for now
mock_synth['meta']['exposure']['read_pattern'] = [res.tolist() for res in patt]

    # Potentially avoid weird logs in DQI step by manually assigning guidestar rows
if img.shape[1:] != (4096, 4096):
    mock_synth['meta']['guidestar']['gw_window_xstart'] = 4
mock_synth['meta']['guidestar']['gw_window_ystart'] = 4
mock_synth['meta']['guidestar']['gw_window_xstop'] = 6
mock_synth['meta']['guidestar']['gw_window_ystop'] = 6
mock_synth['meta']['guidestar']['gw_window_xsize'] = 2
mock_synth['meta']['guidestar']['gw_window_ysize'] = 2
else:
    mock_synth['meta']['guidestar']['gw_window_xstart'] = 363
mock_synth['meta']['guidestar']['gw_window_ystart'] = 1784
mock_synth['meta']['guidestar']['gw_window_xstop'] = 379
mock_synth['meta']['guidestar']['gw_window_ystop'] = 1800
mock_synth['meta']['guidestar']['gw_window_xsize'] = 16
mock_synth['meta']['guidestar']['gw_window_ysize'] = 16

    # Add image data to the file; save it locally
mock_synth['data'] = (img * u.DN).astype(np.uint16)
mock_synth['amp33'] = (np.zeros(img.shape[:1] + (128,)) *
u.DN).astype(np.uint16)
# amp33's 128 channels are hard-coded into refpix step

save_node(mock_synth, filepath=img_path)

def mock_L2.refs(img, read_noise,
    mask_path='ref_mock_mask.asdf',
    saturation_path='ref_mock_sat.asdf',
    refpix_path='ref_mock_refpix.asdf',
    linearity_path='ref_mock_lin.asdf',
    dark_path='ref_mock_dark.asdf',
    gain_path='ref_mock_gain.asdf',
    readnoise_path='ref_mock_readnoise.asdf'):
    ""
    Save versions of L2 reference files that can get an image
    through romancal's
    pre-jump detection steps with no modifications to its data
    or *dq arrays.
    
    *img (array): A 3-D array of resultant-averaged ramp values.
    *read_noise (float): The amount of read noise to add to the
    ramps.
    *_path (str, optional): The path to which each mock
    reference file will be saved.
    ""

    # Mask reference file - included in DQI step
    mask_ref = rds.MaskRef()
    mask_ref['meta'] = mk_ref_common("MASK")
    mask_ref['dq'] = np.zeros(img.shape[1:], dtype=np.uint32)
    save_node(mask_ref, filepath=mask_path)

    # Saturation reference file - included in saturation step
    sat_ref = rds.SaturationRef()
    sat_ref['meta'] = mk_ref_common("SATURATION")
    sat_ref['dq'] = np.zeros(img.shape[1:], dtype=np.uint32)
    sat_ref['data'] = 1e6 * np.ones(img.shape[1:],
        dtype=np.float32) * u.DN
    save_node(sat_ref, filepath=saturation_path)
save_node(sat_ref, filepath=saturation_path)

# Refpix reference file - included in refpix step
# (currently skipping this step)
# refpix_ref = mk_refpix()
# refpix_ref['gamma'] = np.ones((286721,1), dtype=np.complex128) # Left column correction coefficients (detector values, 2-dim)
# refpix_ref['zeta'] = np.ones((286721,1), dtype=np.complex128) # Right column correction coefficients (amp33 values, 2-dim)
# refpix_ref['alpha'] = np.ones((286721,1), dtype=np.complex128) # Reference output correction coefficients (2-dim)
# save_node(refpix_ref, filepath=refpix_path)

# Linearity reference file - included in linearity step
lin_ref = rds.LinearityRef()
lin_ref['meta'] = mk_ref_units_dn_meta("LINEARITY")
lin_ref['dq'] = np.zeros(img.shape[1:], dtype=np.uint32)
lin_ref['coeffs'] = np.stack((np.zeros(img.shape[1:]), np.ones(img.shape[1:])), dtype=np.float32)
# (attempting 0 + 1*F correction with no other coeffs after to keep values the same)
save_node(lin_ref, filepath=linearity_path)

# Dark reference file - included in dark subtraction step
dark_ref = rds.DarkRef()
dark_ref['meta'] = mk_ref_dark_meta()
dark_ref['data'] = np.zeros(img.shape, dtype=np.float32) * u.DN # zeroed out for convenience
dark_ref['dq'] = np.zeros(img.shape[1:], dtype=np.uint32)
dark_ref['dark_slope'] = np.zeros(img.shape[1:], dtype=np.float32) * u.DN / u.s
dark_ref['dark_slope_error'] = np.zeros(img.shape[1:], dtype=np.float32) * u.DN / u.s
save_node(dark_ref, filepath=dark_path)

# Gain reference file - included in jump detection step
gain_ref = rds.GainRef()
gain_ref['meta'] = mk_ref_common("GAIN")
gain_ref['data'] = (np.ones(img.shape[1:]) * u.electron / u.DN).astype(np.float32)
save_node(gain_ref, filepath=gain_path)

# Readnoise reference file - includd in jump detection step
readnoise_ref = rds.ReadnoiseRef()
readnoise_ref['meta'] = mk_ref_readnoise_meta()
readnoise_ref['data'] = (np.full(img.shape[1:], read_noise) * u.DN).astype(np.float32)
save_node(readnoise_ref, filepath=readnoise_path)

def detect_jumps_custom(img, dt, count_rate, patt, read_noise=1e-16):
    
    Perform a standalone jump detection analysis on an image following
    procedures from Casertano (2022) and Sharma+ (2023) for ramp fitting and
    jump detection (respectively) with uneven resultants.

    Also see romancal documentation on handling uneven resultants in ramp fitting
    and jump detection

    img (array): A 3-D array of resultant-averaged ramp values. See DT_IMG and
    DT_SPEC at top of file for canonical Roman values.
    dt (float): The amount of time that elapses during a read.
    count_rate (float): The amount of scientific signal added to the image,
    measured in counts per second.
    patt (list): A list of arrays indicating a readout pattern. Each array
    corresponds with a resultant and contains its matching
read numbers.

    read_noise (float, optional): The amount of read noise to add to the ramps.

    """
    # I. Ramp fitting
    # Average read times to get mean resultant time and the midpoint of the
    # first and last mean resultant times
    t_res_means = np.array([dt * (res.mean() - 1) for res in patt])
    # (subtract 1 because pattern is 1-indexed and t is zero-indexed)
    t_res_mid = (t_res_means[0] + t_res_means[-1]) / 2

    # Save the number of reads contained in each resultant
    res_lengths = np.array([res.size for res in patt])

    # Calculate overall signal-to-noise ratio based on readnoise and the pixel
    # that saw the highest increase in flux during the exposure
    s_max = img[-1].max() - img[0].min() # shouldn't need to index, but helps with speed on full-size images?
    snr = s_max / np.sqrt(read_noise**2 + s_max)

    # Generate weight for every resultant pixels based on exponent P (which is
    # in turn based on the signal-to-noise ratio).
    P_exps = np.array([0, .4, 1, 3, 6, 10])
    S_vals = np.array([0, 5, 10, 20, 50, 100])
    P = P_exps[np.searchsorted(S_vals, snr) - 1]

    weights = ((1 + P) * res_lengths) / (1 + P * res_lengths) * np.abs(t_res_means - t_res_mid)**P

    # Define auxiliary quantities to simplify the ramp slope calculation
    F0 = np.sum(weights)
    F1 = np.sum(weights * t_res_means)
    F2 = np.sum(weights * t_res_means**2)
    denominator = F2 * F0 - F1**2
# Use them and the weights to generate a coefficient for each resultant;
# estimate ramp slope by multiplying each resultant by its coefficient
K_coeffs = (F0 * t_res_means - F1) * weights / denominator
F_slopes = np.sum(np.dstack(K_coeffs).T *
img.astype(np.uint16), axis=0)

# II. Jump detection
# Find the excess slope between a) the scaled difference between each
# pixel's flux from one resultant to the next (or next to the next), and
# b) each pixel's expected ramp slope, as calculated earlier
single_diff_xs_slope = ((img[1:] - img[:-1])
/ np.dstack(t_res_means[1:] -
t_res_means[:-1])).T

   - F_slopes)

double_diff_xs_slope = ((img[2:] - img[:-2])
/ np.dstack(t_res_means[2:] -
t_res_means[:-2])).T

   - F_slopes)

# Get components of jump statistic. These include...
### ..."tau_i", the variance-based time per resultant

t_res_var_based = dt * np.array([np.sum([(2 * (res.size - k)
- 1) * (t_ik - 1)
# subtract 1 from t_ik because patt	# is 1-indexed and
times are zero-indexed
for k, t_ik in
enumerate(res)]) / res.size**2
for i, res in
enumerate(patt)])
# [Casertano (2022), pg. 17: "In general, tau_i < t-bar_i if
N_i > 1, as[,]
# in the variance expression[,] earlier times appear more
# often."]
```python
### ...the variances of the single and double differences
### (where possible)
var_single_diff_res = np.array([read_noise * (1 / patt[j].size + 1 / patt[j - 1].size)
                                 + F_slopes * (t_res_var_based[j] + t_res_var_based[j - 1]
                                 - 2 * min(t_res_means[j], t_res_means[j - 1]))
                                 for j in range(1, len(patt))])

var_double_diff_res = np.array([read_noise * (1 / patt[j].size + 1 / patt[j - 2].size)
                                 + F_slopes * (t_res_var_based[j] + t_res_var_based[j - 2]
                                 - 2 * min(t_res_means[j], t_res_means[j - 2]))
                                 for j in range(2, len(patt))])

### ...and the variances of the excess single and double-difference slopes
var_single_diff_slope = (var_single_diff_res / np.dstack((t_res_means[1:] - t_res_means[:-1])**2).T
                         + F_slopes / (t_res_means[-2] - t_res_means[0]))
var_double_diff_slope = (var_double_diff_res / np.dstack((t_res_means[2:] - t_res_means[:-2])**2).T
                         + F_slopes / (t_res_means[-2] - t_res_means[0]))

# Save a single and double-difference statistic for each possible pixel
stat_single_diffs = single_diff_xs_slope / np.sqrt(var_single_diff_slope)
stat_double_diffs = double_diff_xs_slope / np.sqrt(var_double_diff_slope)

# Finally, condense the statistics into a 3-D array where each pixel (except
# those in the final resultant) is assigned the higher of
```
the two
    stat_jumps = np.max((stat_single_diffs,
        # concatenate an extra frame of low
    values to
        # stat_double_diffs to match
    stat_single_diffs' shape
        # while only values from the latter are
    kept
        np.vstack((stat_double_diffs,
    img.shape[1:], -1e5)),
    axis=0)
    # (can there be a mixture of pixels with values from single
    and double in the same resultant??)

    # Also calculate empirical threshold for jumps based on the
    ramp-fitted slope
    s_thresh_slope = 5.5 - 1/3 * np.log10(F_slopes)
    # (romancal docs say threshold is "5.5 sigma when the count
    rate is 1
    # electron per second and 4.5 sigma when the count rate is
    1000 electrons
    # per second")

    return stat_jumps, s_thresh_slope

def detect_jumps_og(rejection_threshold=5,
    img_path='mock_synth.asdf',
    mask_path='ref_mock_mask.asdf',
    saturation_path='ref_mock_sat.asdf',
    refpix_path='ref_mock_refpix.asdf',
    linearity_path='ref_mock_lin.asdf',
    dark_path='ref_mock_dark.asdf',
    gain_path='ref_mock_gain.asdf',
    readnoise_path='ref_mock_readnoise.asdf',
    max_jump_to_flag_neighbors=1000,
    fast_forward=False):
    """
    Run the romancal pipeline through the jump detection step,
    using mock
    reference files to pass through earlier steps without
adjusting the science data or *dq arrays. Returns the resulting datamodel's GROUPDQ array.

rejection_threshold (float, optional): The sigma level above which a pixel will be marked as experiencing a jump when there are 3, 4, or >4 resultantts. Passed directly to romancal's JumpStep.

*_path (str, optional): The paths from which to fetch the mock image and reference files.

max_jump_to_flag_neighbors (float, optional): Jumps with sigma above this level will not trigger flagging of their neighbors.

fast_forward (bool, optional): If True, skip directly from DQI to jump detection. Applies to WFI images specially simulated for this test.

""
# Run prerequisite pipeline steps
datamodel = DQInitStep.call(img_path, override_mask=mask_path)

if not fast_forward:
    datamodel = SaturationStep.call(datamodel, override_saturation=saturation_path)
    # datamodel = RefPixStep.call(datamodel, override_refpix=refpix_path)
    datamodel = LinearityStep.call(datamodel, override_linearity=linearity_path)
    datamodel = DarkCurrentStep.call(datamodel, override_dark=dark_path)

    # Carry out jump detection, making sure to set the sigma rejection threshold arguments to the same value found in the out-of-pipeline algorithm
    datamodel = JumpStep.call(datamodel, override_gain=gain_path, override_readnoise=readnoise_path, use_ramp_jump_detection=False,
# jump detection is saved for RampFitStep unless above is False

max_jump_to_flag_neighbors=max_jump_to_flag_neighbors,
    # above is set high so neighbors are always flagged if deemed necessary

three_group_rejection_threshold=rejection_threshold,

four_group_rejection_threshold=rejection_threshold,

rejection_threshold=rejection_threshold

)

return datamodel.to_flat_dict()['roman.groupdq']


def calc_class_rates(p_tot, tp, p_found, tot_pix):
    ""
    Calculate false negative and false positive rates of cosmic ray detection for a given algorithm.

    p_tot (int): The total number of positive events (jumps) across all pixels in all resultants.
    tp (int): The number of "true positives," or pixels correctly identified as having been impacted by a cosmic ray.
    p_found (int): The total number of pixels the algorithm marked as having been impacted by a cosmic ray, whether or not the classification was correct.
    tot_pix (int): The total number of pixels across all resultants.
    (Would be n_resultants * 4096 * 4096 for a typical WFI image.)
    ""
    fn = p_tot - tp
    fp = p_found - tp
    tn = tot_pix - tp - fn - fp
fn_rate = fn / (fn + tp)
fp_rate = fp / (fp + tn)
return fn_rate, fp_rate

def main(from_files=True, custom_trials=3):
    
    Test jump detection in roman_cal against a custom algorithm using a variety of MA tables, readnoise levels, and fluxes.

    from_files (bool, optional): If True, read in WFI images simulated for this.
    If False, generate custom images on the fly for each readout pattern.
    mock_trials (int, optional): The number of mocked images to test per 
    readout pattern if from_files=False. (No effect if from_files=True.)
    The default value, 3, matches the number of images read in per readout 
    pattern when from_files=True.

    # Save readout patterns of interest from Casertano (2022):
    # - ML (uneven resultants)
    # - HDR-150 (uneven resultants)
    # - variations of ML and HDR-150 with a single-read resultant added to the end
    #   [Sharma+ (2023) concludes that this helps with jump detection]
    # - EVEN 5-10 (even resultants)
    # - EVEN 6-16 (even resultants)
    ma_tables = create_ma_table('ml_nosingle', [1, 2, 4, 7, 11, 15, 20])
    create_ma_table('ml_single', ma_tables['ml_nosingle']['borders'] + [21], ma_tables)
    create_ma_table('hdr150_nosingle', [1, 2, 3, 5, 7, 11, 19, 27, 35, 43, 51], ma_tables)
    create_ma_table('hdr150_single', ma_tables['hdr150_nosingle']['borders'] + [52], ma_tables)
    # create_ma_table('E510', list(range(1, 51 + 1, 5)))
ma_tables)
    # create_ma_table('E616', list(range(1, 97 + 1, 6)),
    ma_tables)

    # If reading from files, create a dict containing a dict for each readout
    # pattern. Each pattern's inner dict consists of lists of image paths, image
    # data, and cosmic ray mask data for each pre-existing simulation
    if from_files:
        sim_files = defaultdict(lambda: defaultdict(list))

        for patt in ma_tables.keys():
            patt_dir = Path='./' / patt

            for wfi_path in
                sorted(patt_dir.glob('*_WFI01_uncal.asdf')):
                sim_files[patt]['paths'].append(wfi_path)
                wfi_data = asdf.open(wfi_path)

                sim_files[patt]['data'].append(wfi_data['roman']['data'])
                wfi_data.close()

            for mask_path in
                sorted(patt_dir.glob('*_crmask.fits')):
                jump_locs = fits.open(mask_path)

                sim_files[patt]['jumps'].append(jump_locs[0].data)
                jump_locs.close()

        # Record signal and noise values from simulated files for consistency
        count_rate = 500 # signal added to ramps, in counts per second
        read_noise = 7.5 # counts/pixel

        # Fetch/create images, mock up reference files, run jump detection
        # algorithm(s), and return false positive and false negative rates
jump_code = dqflags.group["JUMP_DET"]
results_dict = defaultdict(lambda: defaultdict(list))

for pattern in ma_tables.keys():
    if from_files: # pre-simulated images case
        sim_dict = sim_files[pattern]

        for i in range(len(sim_dict['data'])):
            img = sim_dict['data'][i]
            img_path = sim_dict['paths'][i]

            mock_L2_refs(img, read_noise)

            # Run abbrev. version of romancal pipeline (DQI + jump detection)
            stat_threshold = 5.5 - 1/3 * np.log10(count_rate)
            romancal_groupdq =
            detect_jumps_og(img_path=img_path,
            rejection_threshold=stat_threshold,
            fast_forward=True)

            # Calculate romancal's false -/+ classification
            rates
            mask = sim_dict['jumps'][i][::, ::, :4096] != 0
            p_tot = mask.sum()

            tp_romancal = (romancal_groupdq[mask] ==
            jump_code).sum()
            p_romancal = (romancal_groupdq ==
            jump_code).sum()
            fn_rate_romancal, fp_rate_romancal =
            calc_class_rates(p_tot, tp_romancal,
            p_romancal, img.size)

            results_dict[pattern][\'romancal
            FNRS\'].append(fn_rate_romancal)
            results_dict[pattern][\'romancal
FPRs'.append(fp_rate_romancal)

else:  # custom images case
    pattern_dict = ma_tables[{'sequence'}]
    for i in range(trials):
        shape = 1028  # x and y dimensions of mock image.
        # (smaller because full-size 4096x4096 image is
        # slower than simmed images for some reason...)
        img = create_image(shape, DT_IMG, count_rate,
                           pattern_dict['sequence'],
                           read_noise=read_noise,
                           incl_read_noise=True,
                           # incl_poisson_noise=True,
                           seed=seed + i)
        img, read_jump_inds, res_jump inds =
        simulate_jump_random(img, DT_IMG,
                           pattern_dict,
                           seed=seed + i)
        img_path = f"mock_synth_{pattern}.asdf"

        mock_L1_image(img, DT_IMG,
                      pattern_dict['sequence'], img_path)
        mock_L2_refs(img, read_noise)

        # Run custom jump detection algorithm first
        stat_jumps, stat_thresholds =
        detect_jumps_custom(img, DT_IMG, count_rate,
                           pattern_dict['sequence'],
                           read_noise)

        # Run romancal pipeline next, using the mean(?)
        # of the empirical
        # statistical threshold value above as
        JumpStep's cutoff
        romancal_groupdq =
        detect_jumps_og(rejection_threshold=stat_thresholds.mean(),
# Calculate each algorithm's false -/+ classification rates

```python
p_tot = 3**2 * len(res_jump_inds)  # still assuming 3x3 jump area

tp_custom = np.sum([stat_jumps[res_jump_inds[i]] > stat_thresholds[res_jump_inds[i][1:]]
                       for i in range(len(res_jump_inds))])

p_custom = (stat_jumps > stat_thresholds).sum()

fn_rate_custom, fp_rate_custom = calc_class_rates(p_tot, tp_custom, p_custom, img.size)

results_dict[pattern]['custom FNRs'].append(fn_rate_custom)
results_dict[pattern]['custom FPRs'].append(fp_rate_custom)
```

```python
tp_romancal = np.sum([romancal_groupdq[res_jump_inds[i]] == jump_code
                       for i in range(len(res_jump_inds))])

p_romancal = (romancal_groupdq == jump_code).sum()

fn_rate_romancal, fp_rate_romancal = calc_class_rates(p_tot, tp_romancal, p_romancal, img.size)

results_dict[pattern]['romancal FNRs'].append(fn_rate_romancal)
results_dict[pattern]['romancal FPRs'].append(fp_rate_romancal)
```

# Print test results

```python
for py, lab_dict in results_dict.items():
    print(py.upper())
    for lab, rates in lab_dict.items():
        print(rates)
```
print(f"{lab:>13}: ", end='')
    [print(f"{r:7.03%}"), end=', ' if not j == len(rates) - 1 else '
')
    for j, r in enumerate(rates)]
print('
')

if __name__ == "__main__":
    main()
import numpy as np
from roman_datamodels.maker_utils import mk_level1_science_raw
from roman_datamodels.maker_utils import save_node,
mk_ref_common
from roman_datamodels.maker_utils import mk_ref_units_dn_meta,
mk_ref_dark_meta
from roman_datamodels.maker_utils import mk_ref_readnoise_meta
import astropy.time
import astropy.units as u
import datetime
import roman_datamodels.stnode as rds
from romancal.dq_init import DQInitStep
from romancal.saturation import SaturationStep
from romancal.linearity import LinearityStep
from romancal.dark_current import DarkCurrentStep
from romancal.ramp_fitting import RampFitStep
import asdf
from astropy.io import fits

t0 = datetime.datetime.now()

def prRed(s):
    """
    Simple function to print messages to stdout in red.
    Inputs
    ------
s (str): String to print.
    Returns
    ------
    None
    """
    print(f"\033[91m {s}\033[00m")

def create_asdf_img(img_data, read_pattern, nmax, delta_t=3.04,
    filepath='test_img_uncal.asdf'):
Wrapper of `mk_level1_science_raw` to generate a dummy ASDF file with the requested data (img_data) and read pattern (read_pattern).

Inputs
------

img_data (numpy.ndarray): Input image array
read_pattern (list): List of lists containing the input read pattern
nmax (int): Total number of reads.
delta_t (float): Elapsed time between reads in seconds. Default: 3.04s.
filepath (str): Filename or path to output L1 ASDF following the WfiScienceRaw datamodel containing the image in `img_data`.

Returns
-------
None

dummy_data = mk_level1_science_raw(shape=img_data.shape)  # Create dummy WfiScienceRaw and modify
   dummy_data['meta']['exposure']['ngroups'] = img_data.shape[0]
dummy_data['meta']['exposure']['nframes'] = nmax
dummy_data['meta']['instrument']['detector'] = 'WFI06'
dummy_data['meta']['exposure']['read_pattern'] = read_pattern
dummy_data['meta']['exposure']['start_time'] = astropy.time.Time(t0)
dummy_data['meta']['exposure']['end_time'] = astropy.time.Time(t0) + delta_t*u.s*nmax  # This is always the same
dummy_data['meta']['exposure']['mid_time'] = astropy.time.Time(t0) + delta_t*0.5*u.s*nmax  # This is always the same
dummy_data['meta']['exposure']['exposure_time'] = delta_t*nmax
dummy_data['meta']['exposure']['effective_exposure_time'] =
delta_t*nmax
dummy_data['meta']['exposure']['frame_time'] = delta_t
dummy_data['meta']['exposure']['frame_divisor'] = 1  # This
is not used anyway
dummy_data['meta']['exposure']['duration'] = delta_t*nmax
dummy_data['meta']['exposure']['elapsed_exposure_time'] =
delta_t*nmax
dummy_data['meta']['exposure']['integration_time'] =
delta_t*nmax
dummy_data['meta']['exposure']['ma_table_number'] = 998
dummy_data['meta']['exposure']['ma_table_name'] = 'TEST_R2'
dummy_data['meta']['exposure']['gain_factor'] = 1
dummy_data['data'] = u.Quantity(img_data.astype(np.uint16),
unit=u.DN, dtype=np.uint16)
save_node(dummy_data, filepath=filepath)

def create_dummy_refs(img_data, readnoise=1e-16):
    ""
    Routines to generate dummy reference files.
    Inputs
    ------
    img_data (numpy.ndarray): Input image array
    readnoise (float): Read noise level (in counts/s)
    Returns
    ------
    None
    ""
    dark_ref = np.zeros_like(img_data)
gain_ref = np.ones(img_data.shape[1:])
lin_ref = np.zeros((7, img_data.shape[1],
img_data.shape[2])))
    lin_ref[-2, :, :] = 1.0
    lin_ref = lin_ref[:-1, :, :]
sat_ref = np.ones(img_data.shape[1:])*1000000
    readnoise_ref = readnoise*np.ones(img_data.shape[1:])  # It
can't be zero

    # Use their data models

Released via SOCCER Database at: https://soccer.stsci.edu

G-3
maskref = rds.MaskRef()
maskref["meta"] = mk_ref_common("MASK",)
maskref["dq"] = np.zeros(img_data.shape[1:],
dtype=np.uint32)
save_node(maskref, filepath='mask_dummy.asdf')

# Saturation -- set to 1,000,000 DN threshold, i.e., we
# never flag
saturationref = rds.SaturationRef()
saturationref["meta"] = mk_ref_common("SATURATION")
saturationref["dq"] = np.zeros(sat_ref.shape,
dtype=np.uint32)
saturationref["data"] = u.Quantity(sat_ref, u.DN,
dtype=np.float32)
save_node(saturationref, filepath='saturation_dummy.asdf')

# Linearity -- set to totally linear with coeff 1
linearityref = rds.LinearityRef()
linearityref["meta"] = mk_ref_units_dn_meta("LINEARITY")
linearityref["dq"] = np.zeros(img_data.shape[1:],
dtype=np.uint32)
linearityref["coeffs"] = lin_ref.astype(np.float32)
save_node(linearityref, filepath='linearity_dummy.asdf')

# Inverse linearity -- since it is 1 it should be the same
# as linearity
invlinearityref = rds.InverselinearityRef()
invlinearityref["meta"] =
mk_ref_units_dn_meta("INVERSELINEARITY")
invlinearityref["dq"] = np.zeros(img_data.shape[1:],
dtype=np.uint32)
invlinearityref["coeffs"] = lin_ref.astype(np.float32)
save_node(invlinearityref, filepath='inverselinearity_dummy.asdf')

# Dark -- set to zero
darkref = rds.DarkRef()
darkref["meta"] = mk_ref_dark_meta()

darkref["data"] = u.Quantity(dark_ref, u.DN,
dtype=np.float32)
darkref["dq"] = np.zeros(img_data.shape[1:],
dtype=np.uint32)
darkref["dark_slope"] =
    u.Quantity(np.zeros(img_data.shape[1:], dtype=np.float32), u.DN / u.s, dtype=np.float32)
darkref["dark_slope_error"] =
    u.Quantity(np.zeros(img_data.shape[1:], dtype=np.float32), u.DN / u.s, dtype=np.float32)
save_node(darkref, filepath='dark_dummy.asdf')

# Gain -- set to ones
gainref = rds.GainRef()
gainref["meta"] = mk_ref_common("GAIN")
gainref["data"] = u.Quantity(gain_ref.astype(np.float32),
    u.electron / u.DN, dtype=np.float32)
save_node(gainref, filepath='gain_dummy.asdf')

# Read noise -- set to readnoise level
readnoiseref = rds.ReadnoiseRef()
readnoiseref["meta"] = mk_ref_readnoise_meta()
readnoiseref["data"] = u.Quantity(readnoise_ref, u.DN,
    dtype=np.float32)
save_node(readnoiseref, filepath='readnoise_dummy.asdf')

def process_ramp(filepath='test_img_uncal.asdf',
saturation_path='saturation_dummy.asdf',
    linearity_path='linearity_dummy.asdf',
    dark_path='dark_dummy.asdf',
    gain_path='gain_dummy.asdf',
    readnoise_path='readnoise_dummy.asdf',
    mask_path='mask_dummy.asdf'):
    ""
    Routine to process L1 image up to the ramp-fitting step.

    Inputs
    ------
    filepath (str): Path to input L1 ASDF image.
    saturation_path (str): Path to saturation reference file to
        override roancal`s default.
    linearity_path (str): Path to linearity reference file to
        override romancal`s default.
dark_path (str): Path to dark reference file to override romancal`s default.
gain_path (str): Path to gain reference file to override romancal`s default.
readnoise_path (str): Path to readnoise reference file to override romancal`s default.
mask_path (str): Path to mask reference file to override romancal`s default.

Returns
-------

result (dict) / None: If `output_path = None` a dictionary with the ramp is returned. Otherwise, a file is generated and `None` is returned

```
result = DQInitStep.call(filepath, override_mask=mask_path)
result = SaturationStep.call(result, override_saturation=saturation_path)
result = LinearityStep.call(result, override_linearity=linearity_path)
result = DarkCurrentStep.call(result, override_dark=dark_path)
result = RampFitStep.call(result, override_gain=gain_path, override_readnoise=readnoise_path)
return result.to_flat_dict()
```

def cas22_fit(img_data, read_pattern, nmax, readnoise=1e-16, delta_t=3.04):
    """
    Routine wrapping up the Casertano 22 ramp-fitting method.
    """
    Inputs
    ------
    img_data (numpy.ndarray): Input image array.
    read_pattern (list): Input read pattern.
    nmax (int): Maximum number of reads.
    readnoise (float): Input read noise (in counts/s).
    delta_t (float): Time between frames (in seconds).
Returns
-------
fi (numpy.ndarray): Array with fitted ramps.

```python
t_all = delta_t*np.arange(1, nmax+1)
bar_ti = np.zeros(len(read_pattern))
ni = np.zeros(len(read_pattern), dtype=np.int32)
for i, reads in enumerate(read_pattern):
    reads = np.array(reads) - 1
    bar_ti[i] = np.average(t_all[reads], axis=0)
    ni[i] = len(reads)
bar_tmid = 0.5*(bar_ti[0]+bar_ti[-1])
s = np.max(img_data.astype(np.uint16))/np.sqrt(readnoise**2+np.max(img_data.astype(np.uint16)))
s_th = np.array([0, 5, 10, 20, 50, 100, np.inf])
p_list = np.array([0, 0.4, 1, 3, 6, 10])
p = p_list[np.digitize(s, s_th) - 1]
wi = (1 + p) * ni / (1 + p * ni) * np.abs(bar_ti - bar_tmid)**p
f0 = np.sum(wi)
f1 = np.sum(wi*bar_ti)
f2 = np.sum(wi*bar_ti**2)
d = f2*f0 - f1**2
ki = (f0*bar_ti - f1)*wi/d
fi = np.sum(ki[:, None, None]*img_data.astype(np.uint16), axis=0)
return fi
```

```python
def generate_test(nx, ny, count_rate=1., nreads_orig=55, delta_t=3.04, read_patterns=None, readnoise=1e-16, add_readnoise=False, add_poisnoise=False, verbose=True):
    
    Function to generate noiseless test images with size (nx x ny) and nreads.
    It performs 2 checks: Check 1 compares the value of `count_rate` with the output
    L2 image after processing. Check 2 comapres the output L2 with the result from
```
the fitting using `cas22_fit`

**Inputs**

```
nx (int): Input image size in the X axis.
ny (int): Input image size in the Y axis.
count_rate (float): Target count rate for the synthetic image (in counts/s).
nreads_orig (int): Number of reads of synthetic image before any averaging.
delta_t (float): Time between reads (in seconds).
read_patterns (list of lists): Read pattern(s) requested for synthetic images.
add_readnoise (bool): If `True` it adds read noise.
add_poisnoise (bool): If `True` it adds Poisson noise to the image.
verbose (bool): If `True` it increases the verbosity.
```

**Returns**

None

```python
max_counts = nreads_orig*delta_t*count_rate

if max_counts >= 2**16:
    raise ValueError('The maximum number of counts of this exposure saturate uint16. 
    Please lower the value one or more of the following variables: 
    `count_rate`, `delta_t`, `nreads`
)

t_step = delta_t*np.arange(1, nreads_orig+1)

if not add_poisnoise:
    test_img = count_rate*t_step[:, None, None]*np.ones((ny, nx))  # Image with all reads
else:
    test_img = np.cumsum(np.random.poisson(lam=count_rate*delta_t, ...```
if add_readnoise:
    test_img = test_img.astype(np.float64) + \n
np.cumsum(readnoise*np.random.normal(size=test_img.shape),
axis=0)  # Cumulative read (Gaussian) process

for read_pattern in read_patterns:
    nmax = max(map(lambda x: x[-1], read_pattern))
    nf = np.mean([len(reads) for reads in read_pattern])
    test_img2 = np.zeros((len(read_pattern), ny, nx))
    for i, reads in enumerate(read_pattern):
        reads = np.array(reads) - 1
        test_img2[i, :, :] = np.average(test_img[reads, :, :], axis=0)

    create_asdf_img(test_img2, read_pattern, nmax,
    delta_t=delta_t) # Create test files
    create_dummy_refs(test_img2, readnoise=readnoise) # Create reference files
    result_dict = process_ramp() # Process the ramp
crosscheck = cas22_fit(test_img2, read_pattern, nmax,
    readnoise=readnoise, delta_t=delta_t)
    # uint16 round off error
    nres = len(read_pattern)
    tol_here = 1./nmax * np.sqrt(nf/(nmax-nf))  # This approximation seems to be working well

    if add_readnoise:
        tol_here += readnoise / np.sqrt(nres)  # Increase tolerance if readnoise is added

    if add_poisnoise:
        tol_here += 3*np.sqrt(count_rate*delta_t/nres)  # Adding 3-sigma so we succeed >99% of the time

    # Check 1 -- compare with our version of the Casertano et al. 2022 fit
    check1 = np.allclose(crosscheck[4:-4, 4:-4],
result_dict['roman.data'].value)

if check1:
    print('Passed check 1!')
else:
    prRed('Failed check 1!')

mask = np.where(np.abs(result_dict['roman.data'].value - crosscheck[4:-4, 4:-4]) > tol_here)

print(result_dict['roman.data'].value[mask] - crosscheck[4:-4, 4:-4][mask])
print(result_dict['roman.dq'][mask])

# Check 2 -- compare with ground truth
check2 = np.allclose(result_dict['roman.data'].value, count_rate*np.ones_like(result_dict['roman.data'].value), atol=tol_here)

if check2:
    print('Passed check 2!')
else:
    prRed('Failed check 2!')

mask = np.where(np.abs(result_dict['roman.data'].value - count_rate) > tol_here)

print(result_dict['roman.data'].value[mask] - count_rate)
print(result_dict['roman.dq'][mask])

def create_wfisim_refs(SCA, undo_lin_gain=False):
    """
    Routine to create reference files from WFISim to romancal so everything is consistent in the testing
    """

dark_slope = fits.open(f'/grp/roman/rcosenti/WFIsim/WFIsim.refs/DARK/DARK_SCA{SCA:02d}.fits')[0].data    # This is the dark_slope in e-/s
gain_ref = fits.open(f'/grp/roman/rcosenti/WFIsim/WFIsim.refs/GAIN/GAIN_SCA{SCA:02d}.fits')[0].data
read_pattern = [[1], [2, 3], [5, 6, 7], [9, 10, 11, 12], [13]]
dark_all = np.zeros((13, dark_slope.shape[0],
dark_slope.shape[1]))
dark_all = dark_slope/gain_ref[None, :, :]*np.arange(1,
14)[None, None, None]*3.04
dark_ref = np.zeros((len(read_pattern), dark_all.shape[1],
dark_all.shape[2]))
for i, reads in enumerate(read_pattern):
    reads = np.array(reads) - 1
dark_ref[i, :, :] = np.average(dark_all[reads, :, :],
axis=0)
lin_ref =
fits.open(f'/grp/roman/rcosenti/WFIsim/WFIsim_refs/CNL/CNL_SCA{SCA:02d}.fits')[0].data
sat_ref =
fits.open(f'/grp/roman/rcosenti/WFIsim/WFIsim_refs/SAT/SAT_SCA{SCA:02d}.fits')[0].data/gain_ref

# Trying to take out the gain -- inverse linearity from e- to e- in WFIsim?
if undo_lin_gain:
    lin_ref[0, :, :] /= gain_ref
    lin_ref[2, :, :] *= gain_ref
    lin_ref[3, :, :] *= gain_ref**2
    lin_ref[4, :, :] *= gain_ref**3
    lin_ref[5, :, :] *= gain_ref**4
    lin_ref[6, :, :] *= gain_ref**5

# Because the linearity file is so "linear", this does not
make any difference -- not sure if this should be applied
# in any case due to the way WFIsim works...

readnoise_ref =
fits.open(f'/grp/roman/rcosenti/WFIsim/WFIsim_refs/RN/RN_SCA{SCA:02d}.fits')[0].data
readnoise_ref /= gain_ref # to get it in DN/s

# Use their data models
maskref = rds.MaskRef()
maskref["meta"] = mk_ref_common("MASK",)
maskref["dq"] = np.zeros(dark_slope.shape, dtype=np.uint32)
save_node(maskref, filepath='mask_dummy.asdf')
saturationref = rds.SaturationRef()
saturationref["meta"] = mk_ref_common("SATURATION")
saturationref["dq"] = np.zeros(sat_ref.shape, dtype=np.uint32)
saturationref["data"] = u.Quantity(sat_ref, u.DN, dtype=np.float32)
save_node(saturationref, filepath='saturation_dummy.asdf')

linearityref = rds.LinearityRef()
linearityref["meta"] = mk_ref_units_dn_meta("LINEARITY")
linearityref["dq"] = np.zeros(lin_ref.shape[1:], dtype=np.uint32)
linearityref["coeffs"] = lin_ref.astype(np.float32)
save_node(linearityref, filepath='linearity_dummy.asdf')

invlinearityref = rds.InverselinearityRef()
invlinearityref["meta"] = mk_ref_units_dn_meta("INVERSELINEARITY")
invlinearityref["dq"] = np.zeros(lin_ref.shape[1:], dtype=np.uint32)
invlinearityref["coeffs"] = lin_ref.astype(np.float32)
save_node(invlinearityref, filepath='inverselinearity_dummy.asdf')

darkref = rds.DarkRef()
darkref["meta"] = mk_ref_dark_meta()
darkref["data"] = u.Quantity(dark_ref, u.DN, dtype=np.float32)
darkref["dq"] = np.zeros(dark_slope.shape, dtype=np.uint32)
darkref["dark_slope"] = u.Quantity(dark_slope/gain_ref, u.DN / u.s, dtype=np.float32)
darkref["dark_slope_error"] = u.Quantity(np.zeros(dark_slope.shape, dtype=np.float32), u.DN / u.s, dtype=np.float32)
save_node(darkref, filepath='dark_dummy.asdf')

# Gain -- set to ones
gainref = rds.GainRef()
gainref["meta"] = mk_ref_common("GAIN")
```python
gainref['data'] = u.Quantity(gain_ref.astype(np.float32), u.electron / u.DN, dtype=np.float32)
save_node(gainref, filepath='gain_dummy.asdf')

# Read noise -- set to readnoise level
readnoiseref = rds.ReadnoiseRef()
readnoiseref['meta'] = mk_ref_readnoise_meta()
readnoiseref['data'] = u.Quantity(readnoise_ref, u.DN, dtype=np.float32)
save_node(readnoiseref, filepath='readnoise_dummy.asdf')

def main(filepath_L1=None, filepath_L2_sim=None):
    
    Main function for testing. If no filepath is passed along it generates its own data.

    Inputs
    ------

    filepath_L1 (str or None): If a path to a L1 file is passed, it processes it and compares it to a pre-generated L2.
    filepath_L2_sim (str or None): Path to pre-generated L2 to compare against the input L1.

    Returns
    ------

    None

    if filepath_L1 is None:
        # If there's no input file, we generate our own data.
        # Let's start with these MA tables
        ma_tab = [[1], [2, 3, 4], [5, 6], [7, 8, 9, 10], [11, 13, 14],
                   [15, 17, 18], [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32],
                   [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47],
                   [48, 49, 50, 51, 52, 53, 54], [55]]
        ma_tab2 = [[1], [3], [5, 6], [33], [44], [46, 47, 48, 49], [54]]
```

Released via SOCCER Database at: https://soccer.stsci.edu
ma_tabs = [np.arange(1, 55).reshape((54//k, k)).tolist() for k in [1, 2, 3, 6, 9, 27]]
ma_tabs.append(ma_tab)
ma_tabs.append(ma_tab2)
# And these input parameters for the images
nx = 64  # x-size -- 64 by default
ny = nx  # y-size
count_rate = 1.  # target count-rate (in counts/s/px)
nreads_orig = 55  # Total number of reads -- it needs to match with max(ma_tabs)
dt = 3.04  # Time between frames (in seconds)
# Generate tests

for rn in [1e-16, 1e-3, 1]:  # Check different read-noise levels.
    generate_test(nx, ny, count_rate=count_rate, nreads_orig=nreads_orig, delta_t=dt,
                   read_patterns=ma_tabs, readnoise=rn, add_readnoise=False, add_poisnoise=False, verbose=True)
else:
    if filepath_L2_sim is None:
        raise ValueError('`filepath_L2_sim` needs to be not `None` if a L1 file is passed to `main`. \n                        Please use the appropriate L1 and L2 files to compare against each other.')
    else:
        SCA = int(filepath_L1.split('WFI')[-1][:2])
        create_wfisim.refs(SCA, undo_lin_gain=False)
        _test_img = process_ramp(filepath=filepath_L1)['roman.data'].value
        _input_img = asdf.open(filepath_L2_sim)[0]['roman'][:, 'data'].value
        tol = 0.1  # This number should depend on noise levels
        ratio = _test_img / _input_img - 1.
        px_in_tol = np.count_nonzero(np.abs(ratio) < tol)
        frac_in_tol = px_in_tol / len(ratio)
        if frac_in_tol >= 0.9:
            print('Passed')
else:
    prRed(f'Comparison failed: \{frac_in_tol*100\}
percent of the pixels \\
is within \{tol\}"
    
if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--filepath-L1', help='path to L1 images', required=False)
    parser.add_argument('--filepath-L2-sim', help='path to L2 WFIsim-generated images', required=False)
    o = parser.parse_args()
    main(filepath_L1=o.filepath_L1, filepath_L2_sim=o.filepath_L2_sim)