SPACE TELESCOPE SCIENCE INSTITUTE
Operated for NASA by AURA

Technical Instrument Report STIS 2014-XX

Characterization of the STIS NUV MAMA Vignetting by Optical Element

Kenneth Hart[1], K. Azalee Bostroem[1]

[1]Space Telescope Science Institute, Baltimore, MD

18 June 2014

ABSTRACT

This document describes the process used to generate a new bad pixel table for the STIS NUV MAMA that includes the vignetted corners. This bad pixel table differs from previous tables as it defines bad pixels by optical element in the new OPT_ELEM column.

Contents

1. Introduction

The Space Telescope Imaging Spectrograph (STIS) is one of the two spectrographs on-board the Hubble Space Telescope (HST). STIS was installed on HST in 1997 and recorded data until an electronics failure in 2004. It was repaired during Servicing Mission 4 in May 2009. STIS has three detectors which cover a wide range of visible and ultra-violet wavelengths. One of these detectors, the near-ultraviolet multi-anode microchannel array (NUV MAMA) detector provides imaging and spectroscopic observations between 1600 and 3100 Å.

Corners of NUV observations are vignetted by an obstruction in the light path. The severity of the vignetting is dependent on the optical element used in the observation. Prior to this analysis, the vignetting was marked in the data quality array (extension 3) of the NUV pixel to pixel flat field which does not depened on optical element. This leads to incorrect data quality flags for every optical element except that used to make the flat field (G230M).

To better mark the bad pixels in the vignetted corners, a new optical element column (OPT_ELEM) has been added to the bad pixel table. This allows for bad pixels to be marked differently for each optical element. A pipeline program was written to find the corners for each optical element and add them to the original bad pixel table[1]. This bad pixel table correctly marks the pixels in the vignetted corners of each optical element with the data quality flag 64.

The following describes the programs used to find the corners, write the corner information to a database, and use that database to create a new bad pixel table. Also included is an explanation of the supporting programs, the analysis of central wavelength dependence, and future work.

[1] Throughout the entire document, the bad pixel table used by CALSTIS prior to this analysis (found at /grp/hst/cdbs/oref/hcm14407o_bpx.fits) is refered to as the original bad pixel table.

2. Observations

There are 3 types of NUV MAMA observations which illuminate the entire detector and can therefore be used to map the vignetted regions. The echelle gratings fill the detector with multiple spectral orders, long-slit first order grating observations of a source which fills the slit also span the detector, and some imaging observations with the MIRNUV optical element illuminate the detector. Figure 1 shows an example of the detector illuminated with an echelle grating (left) and a first order grating (right).

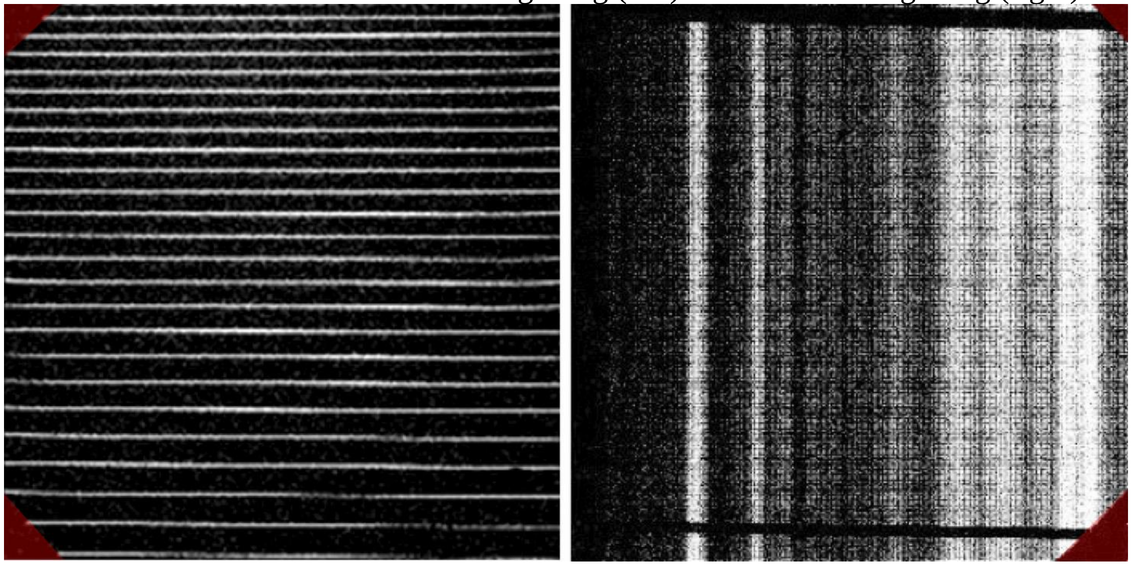

Figure 1 A comparison of the images produced by the different NUV MAMA gratings. The image on the left is from the E230M echelle grating and the image on the right is from the G230L first order grating. The vignetted regions are highlighted in red.

3. Pipeline Structure

The ***pipeline.py*** script is the main program that generates the NUV MAMA bad pixel table in two major steps. First data is analyzed and a fit to the vignetted region for each observation is recorded in a text file referred to as the database file. Then the database file is used to create a new bad pixel table.

3.1 Creating the Database

Information in the database file is found using the *process_archive* function in the ***pipeline.py*** module. Images from the Mikulski Archive for Space Telescopes (MAST) (archive.stsci.edu) are downloaded into an archive folder. From these files, *process_archive* retrieves header information, determines the optical element, and finds the best fit equation for the vignetting of each image. It then creates a block of data which is appended to the end of the database file using *addtofile*. Once this process is complete, the file is moved from the archive folder to the processed folder so that previously processed images are not reprocessed. The percentage of files processed is also displayed and the total processing time printed at the end. See **Figure 2** for the flow chart of the *process_archive* function.

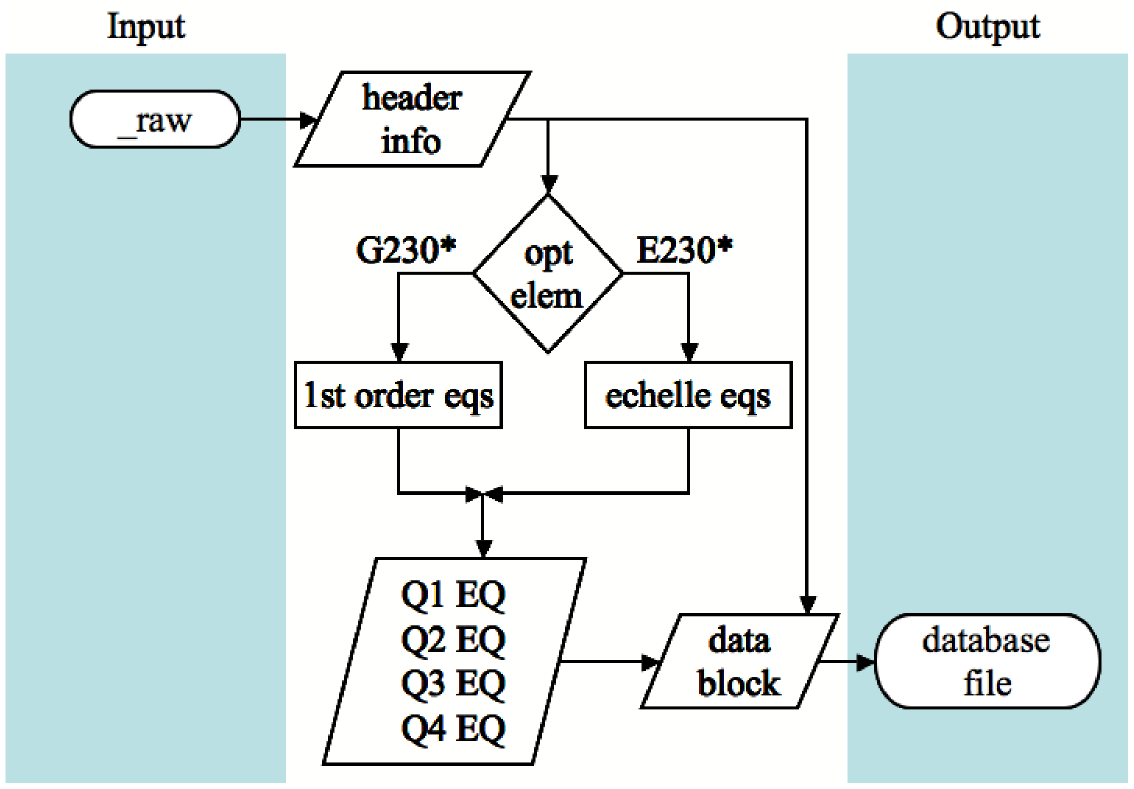

Figure 2 Flow chart for the *process_archive* function in ***pipeline.py***. The file's header information is used to determine which program will be used to find the corners. The equations that fit the edge of the corners, along with relevant header information, are saved to the database file.

3.2 Creating the NUV MAMA Bad Pixel Table

The *generate_tables* function sorts the information in the database file by optical element and creates a bad pixel table for each optical element using the *opt_elem_bpx* function found in the ***table_creation.py*** module (see Section 6.4). The corner regions for each optical element are appended to the original bad pixel table using the *NUV_MAMA_bpx* function (see Section 6.5). **Figure 3** shows how the combined bad pixel table is created from the database file by the *generate_tables* function.

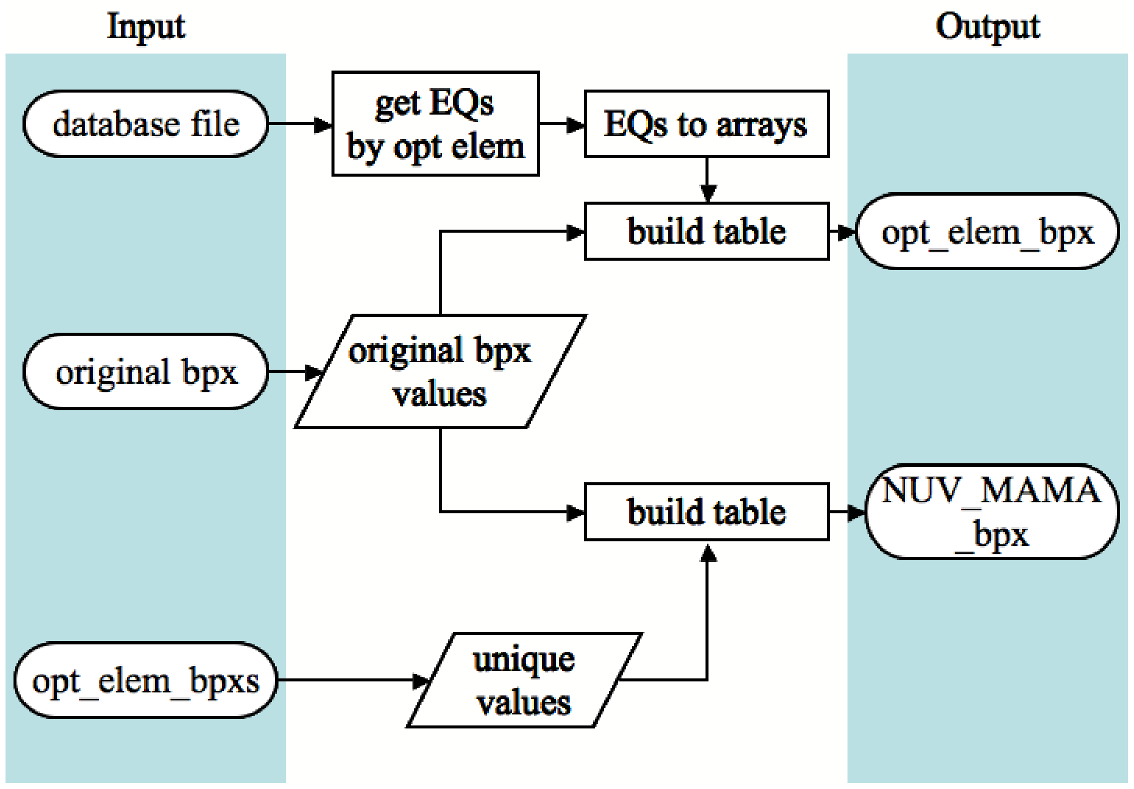

Figure 3 Flow chart for the *generate_tables* function in the ***pipeline.py*** module. The function first creates the optical element bad pixel tables using the equations found in the database file and the values in the original bad pixel table. The final NUV MAMA bad pixel table is created by combining the optical element bad pixel tables and the original NUV MAMA bad pixel table.

4. Deriving the Corner Locations

The vignetting affects only the some corners of the NUV. Which corners and the degress of vignetting is dependent on the grating used. Each vignetted region is defined by a diagonal line forming a right triangle with the corner of the image.

4.1 Echelle Modes

The echelles module, ***echelle_points.py***, contains two functions that are used to determine the vignetted pixels in the corners of data taken using the E230H and E230M gratings.

The first function, *vertical_points*, determines the vertical location of the spectral orders and returns arrays of their lower and upper y pixel bounds (each order is about 10 pixels thick). The corner lies in the outermost 150 rows x 150 columns. The location of the orders is found by comparing the mean image count rate to the median count rate of each row in a box defined near each corner outside of the vignetted region. The boxes are defined in the first and last 200 rows of an image and from columns 150 – 200 on the left and columns 1848-1898 on the right. A 200 row by 1 column mask is created where each row in the mask is marked as True if the median value of that row is above the mean of the entire image and False otherwise. For an explanation of why the image mean is used instead of median, see Appendix A.

The function then looks through the mask two rows at a time, ignoring the first and last 10 rows of the image to exclude edge effects. If the first value is false (value is below the mean) and the second is true (value is above the mean), then a 1 is inserted in the *i*th position of an the order location array, a 200 element 1D array initialized with zeros, where *i* is the position of the second value. This marks the beginning of an order. If the opposite occurs, i.e. the value drops from above the mean to below it and the values aren't oscillating between true and false, then a 2 is placed to mark the end of the order. If the values are oscillating (a true occurs in the next 3 pixels) then nothing happens and the value in the zero array remains zero. This ensures that if there are changes occuring within the order, they are ignored

Finally, *vertical_points* checks to make sure there are the same number of order beginnings as order endings so that orders that run off the edge of the detector are excluded. An image is considered too faint to be used if there are less than two orders found. In this case, all values in the order location array are reset to 0 so that the rest of the program runs knowing that there is no visible vignetting in this corner. If there are more order beginnings than order endings, then the last beginning is eliminated, and if there are more order endings than order beginnings, the first ending is eliminated. The program also checks that the endpoints of an order are within 15 pixels of each other and eliminates points appropriately so that an order is not defined to be greater than 15 pixels wide.

The next function, *order_points*, creates a line one pixel wide which represents the center of each order. This is done by finding the midpoint between the beginning and ending pixel of each order for each column. These center lines are considered to be the line on which the orders lie on. To find where the vignetting begins, the function *order_points* scans each order in the dispersion (x) direction and determines the outermost location on that line where the value is greater than the mean value of the image. The x and y location of that outer pixel is recorded. If no value is discovered, then a zero value is recorded for the x location. The function then eliminates values that are within 15 pixels of the edge of the image, again to aviod confusing the rapid loss in sensitivity with vignetting for the non-vignetted orders.

Once *order_points* finds the x and y positions of the ends of the orders it converts these points to low resolution pixels by dividing by two; these values are stored in lists which are returned to the module ***findline.py***. This module linearly fits the x and y vignetting location values for each corner, marking the edge of the vignetted area. The slope and intercept of this fit for each quadrant[2] are stored in the database file.

In the E230H images, very faint vignetting occurs in the bottom right corner. This corner is only visible in high signal to noise data or in the rare occasion where an order falls very close to the bottom edge of the detector. Since it is not visible in every spectrogram, this corner was marked manually using the ***manual_update.py*** module (see Section 7.4). Visible vignetting is marked by hand from images taken at several

[2] When the slope and intercept are found for the corners, the origin of the image is at index [0,0] in the bottom left corner for all quadrants. When referring to the corners in each quadrant, the upper right corner is called quadrant 1, upper left is quadrant 2, etc. All numbers are defined relative to the origin, the quadrant terminology is used to more clearly indicate corner locations to the reader.

different central wavelengths. Table C1 lists the dataset used to generate the vignetted edge of the E230H bottom right corner.

4.2 First Order Modes

The module ***first_order_points.py*** contains functions for finding the points along the edge of the vignetted corners for the G230M and G230L gratings. The first function, *g_points*, takes the data and the corner quadrant and bins the data from high to low resolution pixels using *imshrink2* (see Section 7.5). It then calls *quadrant_points* to find the x and y coordinates of the points that lie on the edge of the vignetting.

The function quadrant_points finds the average count rate of the unvignetted region near each quadrant by finding the median value of each column in regions defined by rows 150-200 and rows 824 – 874 and columns 1-200 and 824 – 1024. For each column in each corner the edge of the vignetting is identified as the first pixel where the pixel count rate is greater than the average count rate of the unvignetted region found previously. To avoid columns with low contrast between the data and the vignetted region, columns where the average count rate of the unvignetted region is less than 5 for the bottom corners and 3 for the top corners are not used. Additionally, columns for which the median count rate of the last 50 pixels is less than 1 are not used as there is not enough contrast.

g_points checks that there are more than 5 points returned by *quadrant_points*, then passes the x and y positions as two lists to the *line* function in the module ***findline.py***[3]. This module takes the x and y positions and fits a line, recording the slope and intercept in the database file.

There are some images with a horizontal shadow from the occulting bar across the image (see **Figure 4)**. In these images, some points inside the stripe are spuriously marked as vignetted, biasing the line towards the center of the image.

To eliminate these values, *quadrant_points* calls the function *find_stripe*. The *find_stripe* function looks for a horizontal stripe in the image that intersects the vignetted corners. The function takes the data and the quadrant number of the corner as input and returns the upper and lower bounds of the stripe. These values are determined by defining a region 200 rows long by 50 columns wide at either the top or bottom of the image, 150 pixels from the vertical edges. The mean across each row is taken, so the region becomes a 1D array with 200 elements. This array is binned down to one fourth of its original length to smooth small scale brightness variations. The greatest decrease in brightness indicates the bottom of the stripe and the greatest increase in brightness indicates the top of the stripe. The stripe edge values are scaled by 4 to preserve the original position of the stripe, then the bar is widened by 5 pixels to ensure all of the pixels within the stripe are eliminated. If there is no stripe or the contrast between the stripe and the illiminated part of the detector is too low (no pixels > 0.125

[3] The module **findline.py** can be used to call to any user supplied function that returns a slope and intercept and add the necessary information to the database file. This step was put into place so that another user of this pipeline could write their own program for finding corners and use the rest of the pipeline to make the bad pixel table.

in the whole image), then zero (in the no stripe case) and -1 (in the low contrast case) is returned for the bounds of the stripe.

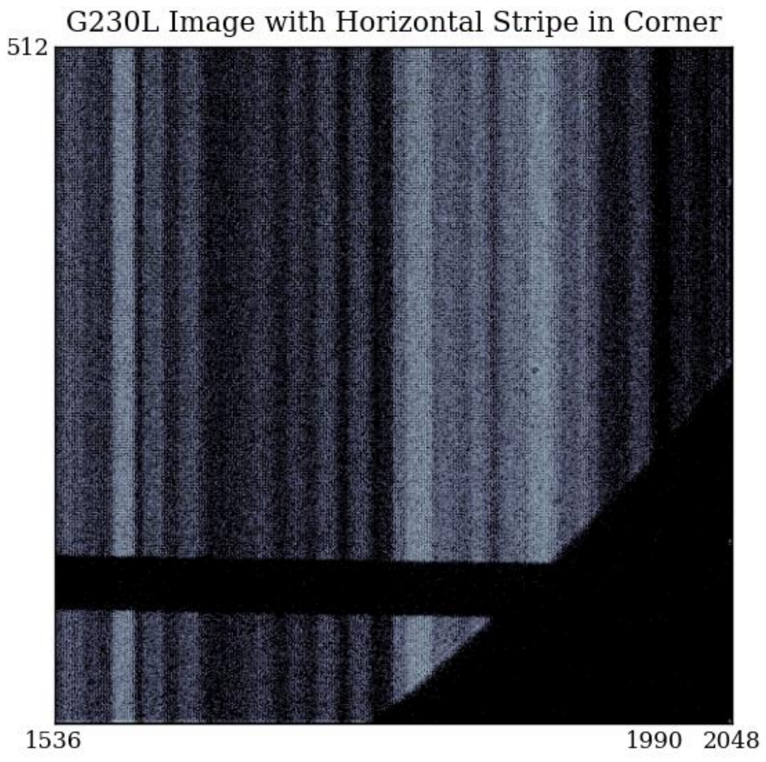

Figure 4 Bottom right corner showing a horizontal stripe. This image was taken from dataset o67w07csq.

The output of *find_stripe* is used by *quadrant_points* to eliminate points that have y values within the limits of the occulting bar shadow. The vertical striping, seen in Figure 4 near column 1990, can flag data points that are not on the edge of the vignetted region. These points are eliminated before the final set of parameters for the line of best fit were output in the following manner. If the R^2 value of the regression to the vignetted region is less than 0.98, the point with the greatest residual value is eliminated from the set and the regression was recalculated. This process is repeated until the R^2 condition is satisfied.

4.3 Mirror

The MIRNUV optical element displays some vignetting. The vignetted corners of the MIRNUV setting are marked manually using ***manual_update.py***. See Table D1 for a list of the datasets used.

5. Database File

5.1 Structure

The database file is a text file that contains information gathered about each image. The first entry in the file is the number of blocks of information contained in the file, which is equal to the number of images that have been processed. Each block of information contains two major parts: basic header information from the image and the coefficients used to define the vignetted region for each quadrant. The header information stored for each block is the file name, optical element, central wavelength, and exposure time. For quadrants that do not show vignetting, the coefficients are stored as 0. See Table 1 for an example block of information found in the database file.

Line	Information
o42f06vdq_raw.fits	file name
G230M	optical element
2898	central wavelength (angstroms)
1200	exposure time (seconds)
-0.8571	quadrant 1 slope
1830.9940	quadrant 1 y-intercept
0.9405	quadrant 2 slope
915.1531	quadrant 2 y-intercept
-1.0185	quadrant 3 slope
41.9379	quadrant 3 y-intercept
0	quadrant 4 slope
0	quadrant 4 y-intercept

Table 1 This is an example block of information that is stored to the database file. The first column contains the example block and the second column explains what that information represents. This example observation shows vignetting in quadrants 1-3.

5.2 Populating the Database

The module ***database_management.py*** contains functions that handle data stored in the database file. The most important function in this module is *addtofile which writes each data block to the database file.* In addition to being used for the initial ingest of data, this function can also be used to add new images to the database. This is done by passing *addtofile* an image name. The header information to be stored in the database file is extracted from the raw file and the exposure time is checked to ensure that it is greater than 0. The function then reads the database file and checks that the image has not been previously analyzed. *addtofile* then finds the slopes and intercepts for each quadrant using the *line* function found in the module ***findline.py*** (as described in Section 4). The slopes and intercepts are combined with the header information to create a database block. This block is written to the database file and the number of blocks is increased by one.

Another important function contained in the module ***database_management.py*** is called *get_info*. This function takes several optional parameters: file name, optical element, central wavelength, and exposure time to get blocks of information from the database file. This is useful for creating a bad pixel table for a single optical element, and displaying the corners for a specific subset of data (e.g. central wavelength). It was used in the analysis of the central wavelength dependence of the vignetting.

The final support function is *update_entry* in the ***manual_update.py*** module. This function takes a file name and new information about that file as input and updates the database file with the new information (e.g. update_entry(name, opt_elem, cenwave, exptime, [m_1, m_2, m_3, m_4], [b_1, b_2, b_3, b_4])). *Update_entry* is contained within ***manual_update.py*** because it is used most often when marking the corners by hand. See Section 4.1 for more details on how ***manual_update.py*** was used to mark the lower right corner of the E230H images, and see Section 7.5 for more details about the ***manual_update.py*** module.

6. Bad Pixel Table Creation Programs

The table programs contained in the ***table_creation.py*** module read information from the database file, convert the corner line equations into arrays of bad pixels, and write these arrays to a binary FITS table. Bad pixel tables are first generated for each optical element, then these tables are combined into a single bad pixel table for the entire detector. While the bad pixels in the original bad pixel table are repeated in each optical element bad pixel table, they are only listed once in the combined table.

Once the columns are added to the file, the headers are updated with the date the table was generated, the description of who created the file, and a history of how the file was created. These keywords need to be updated everytime a bad pixel table is made. A file called history.txt is also created which provides a summary of how the corners were found and stored, and how the table was created. The text in this files is used to populate the history keyword of the primary header of the bad pixel table.

6.1 Deriving the Bad Pixel Table

The function *NUV_MAMA_bpx* passes the final column arrays for the detector bad pixel table to *build_table* (see Section 6.2). Through a series of function calls described below, *NUV_MAMA_bpx* gets the bad pixel table data from the original bad pixel table and creates an optical element (OPT_ELEM) column with the value “ANY”.

The function then loops over each optical element for which a bad pixel table exists (created by *opt_elem_bpx* (see Section 6.1.1)) and creates an array for each column. The arrays are then appended to the corresponding arrays from the original bad pixel table, including the bad pixels which are in both the original bad pixel table and the optical element bad pixel table only once.

6.1.1 *opt_elem_bpx*

The function, *opt_elem_bpx*, takes an optical element as input and creates a bad pixel table for it. It calls *table_arrays* (see Section 6.1.2) to get the bad pixel arrays (pix1, pix2, length, axis, value, opt_elem) for a given optical element. These arrays are passed to *build_table*, which writes a FITS binary table file for the optical element.

6.1.2 *table_arrays*

The *table_arrays* function takes an optical element as input. It first gets the data in the original bad pixel table creating an OPT_ELEM column with the entries set to “ANY”. Next, the function creates bad pixel table column arrays (pix1, pix2, length, axis, value, opt_elem) for each quadrant using *EQs_to_arrays* (see Section 6.1.3). These arrays are specific to the input optical element. Each array from *EQs_to_array* is appended to the corresponding array from the original bad pixel table. The output of *table_arrays*, a list of the arrays representing the columns of the bad pixel table, is passed to the function *opt_elem_bpx*.

6.1.3 *EQs_to_arrays*

The *EQs_to_arrays* function selects information from the database file that pertains to a given optical element and quadrant. This information, a list of slopes and intercepts, is then used to create a list of locations where the lines intersect the edges of the image. The innermost intersections are then used to create the final equation for the corner line. Using the absolute extrema ensures that no bad pixel lies outside vignetted region marked in the image. These extreme intercepts were visually inspected using the ***xandy.py*** module and corrected if necessary (see Section 7.3).

Next the slope and intercept of a corner are converted into the list of bad pixels that can be understood by CALSTIS. Each vignetted row in a corner of the detector is represented by a row in the bad pixel table. The bad pixel table defines a starting x and y pixel and the horizontal length (axis 1) of the bad pixel region. The following equations show how x, y, and length are calculated for the ith row in the upper left corner.

$$x_o = x_i = 1$$
$$y_i = y_0 + i$$
$$len = x_0 + \mathrm{floor}(i \cdot \Delta x)$$

Where x_0 and y_0 are the pixel location of the beginning of the lowest row in the corner and $\Delta x = 1/m$ where m is the slope of the of the vignetting line. The length is rounded down to ensure that if a fraction of a pixel lies on the line, then that entire pixel is considered bad. This technique is applied to the other corners taking into account coordinate transformations. There are also some safeguards from errors that might crash CALSTIS. For example, if there’s a length value that is zero it is changed to 1.

All of the data quality flags in the value column for the vignetted corners are set to 64, which had previously been an unassigned data quality flag. The bad pixel table arrays are then returned to the function *table_arrays*. The order of the arrays output by this function does not reflect their order in the bad pixel table.

6.2 Writing the Bad Pixel Table

The *build_table* function is a simple function that uses arrays of bad pixels to creates a binary FITS table. If a file of the same name exists, it is overwritten. The formatting parameters were copied from the original bad pixel and the parameters for the OPT_ELEM column are found in Section 5.4.1 of the PyFITS Handbook (see Table 2 for the column parameters).

Column	Format	Unit	Null	Disp
PIX1	1I	pixel	-32767	I4
PIX2	1I	pixel	-32767	I4
LENGTH	1I	pixel	-32767	I4
AXIS	1I	-	-32767	I1
VALUE	1I	-	-32767	I5
OPT_ELEM	5A	-	n/a	I5

Table 2 Parameters for each column in the bad pixel table.

7. Support & Test Programs

7.1 Display Corner Data

The ***display_corner_data.py*** module is used to visualize the information stored in the database file to determine if the vignetting depends on central wavelength. Information for a single, user specified, optical element is read from the database file. The script then creates arrays of the non-zero slopes and intercepts for each quadrant and each central wavelength for the input optical element. By default, the median slope and intercept for a given central wavelength and quadrant is found, however, there is a commented section which, if uncommented, allows the user to see the lines for all datasets. The *batch_plot* function takes two or three arrays as input and outputs a plot of the input data. For example, if the inputs are (ms,bs,cs) where ms is a slope array, bs is a y-intercept array, and cs is a central wavelength array for a specific optical element, then the function will plot the lines given by the slopes and intercepts, and color the lines based on central wavelength. If the inputs are (ms,cs) or (bs, cs) then the function will plot the relationship between slope and central wavelength or y-intercept and central wavelength, respectively.

The coloring of the lines on the vignetted image are scaled so that the central wavelength range spans the color wheel. The function *cenwave_to_rgb* scales the central wavelength to be on a scale from 0 to 300 degrees of a 360 degrees color wheel. See Figure 5 for an illustration of how *cenwave_to_rgb* interprets central wavelength as a hue of the color wheel.

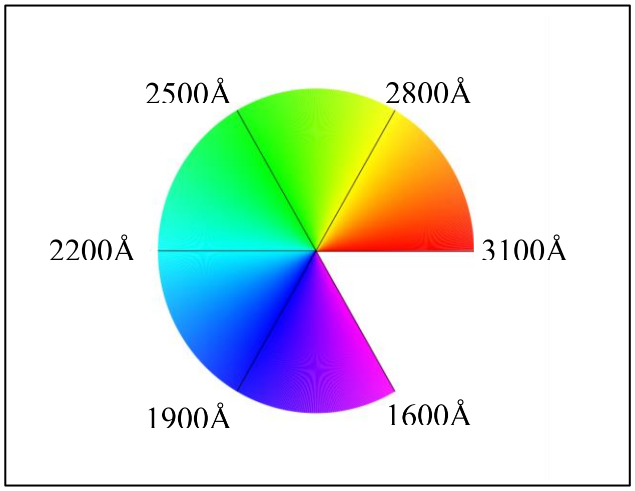

Figure 5 This color wheel shows how a central wavelength is given a color based on the spectral range of the STIS NUV MAMA.

Coloring the lines based on the central wavelength allows for an initial qualitative analysis of central wavelength dependency. The script produces two additional figures to aid in this analysis. The first figure is a plot for each quadrant of slope vs central wavelength for every dataset used. In each plot, the medain value for all datasets of a central wavelength is plotted in black. A similar figure is created that shows the relationship between y-intercept and central wavelength.

7.2 Display Bad Pixel Table

Another helpful program that supports the creation of an accurate bad pixel table is ***display_bpx.py***. This program displays graphs of the locations of the vignetted corners as marked in the original flat field file as well as in the new bad pixel table. This enables a direct comparison of the previous vignetting model and the model described in this paper. The data quality flags in the previous flat field are used to generate the visual of the previous vignetting model. The pixels that were appended to the bad pixel table are used to generate the visual of the new vignetting model. These models are superimposed on aggregate images of each kind of grating, primarily to verify that the new model accurately represents the vignetting pattern for each optical element.

The aggregate images are made by taking 10 images at a time and finding the mean for each pixel and writing the result to a file. Once this process is done, 10 files are read at a time, their means calculated, then stored to another file. This process is repeated until there is only one file left. This mean image smoothes over some of the small differences in the location of the vignetted edge for each dataset making it easier to see the vignetting location, especially using the flag colormap in matplotlib.

7.3 X and Y

The ***xandy.py*** program allows the user to correct line equations that inaccurately mark the vignetted corner. If left uncorrected, these incorrect equations can be selected by *EQs_to_arrays* as the extreme x and y intercept values to create the final line for each optical element (see discussion in Section 5.5). To make sure the final values used by

EQs_to_arrays are accurate based on the images from which they are taken, the lines for the extreme x and y intercepts are displayed on top of the images from which they are taken. The program asks the user to validate the lines, and if they invalid the program calls the *update_entry* function in ***manual_update.py*** (see Section 7.4) to update the database file with the correct values. This display is performed for each corner, and can be done multiple times until all of the lines are validated. It should be run whenever a bad pixel table is created.

7.4 Manual Update

The ***manual_update.py*** script allows the users to change the outliers for a given optical element, central wavelength, and quadrant. Outliers are defined as corners whose slope or intercept is outside the interquartile range. These outliers were first seen using the program ***display_corner_data.py***.

For each corner in an image with an outlier slope or intercept, the user manually redefines the corner by clicking on the points that define the edge of the vignetting using the ***pylab*** function *ginput*. [4] A linear regression is performed on the manually selected points and the database is updated with the new values for slope and intercept in that quadrant.

7.5 Tools

The ***tools.py*** module contains scripts that are used by a number of different programs.

a*ppend4* takes in four arrays as input and returns an array that has appended the last three arrays to the first one.

fits_info takes a fits file name and extension number as input and returns either the image or header for the extension. The default for this function is the data in extension one.

imshrink takes in a 2048x2048 hi-res image and shrinks it to 1024x1024 lo-res image. It does this by taking each row and grouping pixels in sets of two and finding their median. This creates a 1024x2048 image, and the process is repeated for each column. This is not the same method as LORSCORR in CALSTIS, however, this function is only used for display purposes and so will not affect the bad pixel locations.

imshrink2 runs through the same process as *imshrink*, however it finds the sum instead of the median. This is the same method as LORSCORR and is used in finding the vignetting for the first order grating images.

8. Wavelength Dependence Analysis

[4] The pylab.ginput function takes an integer, n, number of clicks as input and returns an n by 2 array of (x,y) coordinates.

The data were examined for a dependence on central wavelength as detailed in Section 7.2. While variation in slope and intercept exists within a given central wavelength, no trend was observed.

9. Future Work

9.1 First Order Curved Corners

In the first-order gratings, many of the corners appeared to be curved so the linear regression included good data within the marked corner. The function *polyfit* in the ***numpy*** module could be used to fit these corners with a polynomial. If *polyfit* is used, the structure of the database file needs to be changed to include the other coefficients and the method for generating the tables also needs to be modified.

9.2 Default Monthly Offset Position for Central Wavelength Dependence

It is possible that some of the scatter within a central wavelength may be eliminated in the echelle data by using only data taken after August 2002 (when the monthly offsetting was turned off). With reduced scatter, a central wavelength dependence may be visible.

References

Dressel, L., et al. 2007, “STIS Data Handbook”, Version 5.0, (Baltimore: STScI)
Hack, W. and Greenfield, P. 2010, “The PyFITS Handbook”, (Baltimore: STScI)

Appendix A

Mean vs Median
The mean pixel value was used as the brightness threshold for the echelles because many of the pixels (between the orders) were dark so the median yielded a low value, often between 0 and 1. **Figure 6** shows the distribution of pixel values compared to frequency for an E230H image.

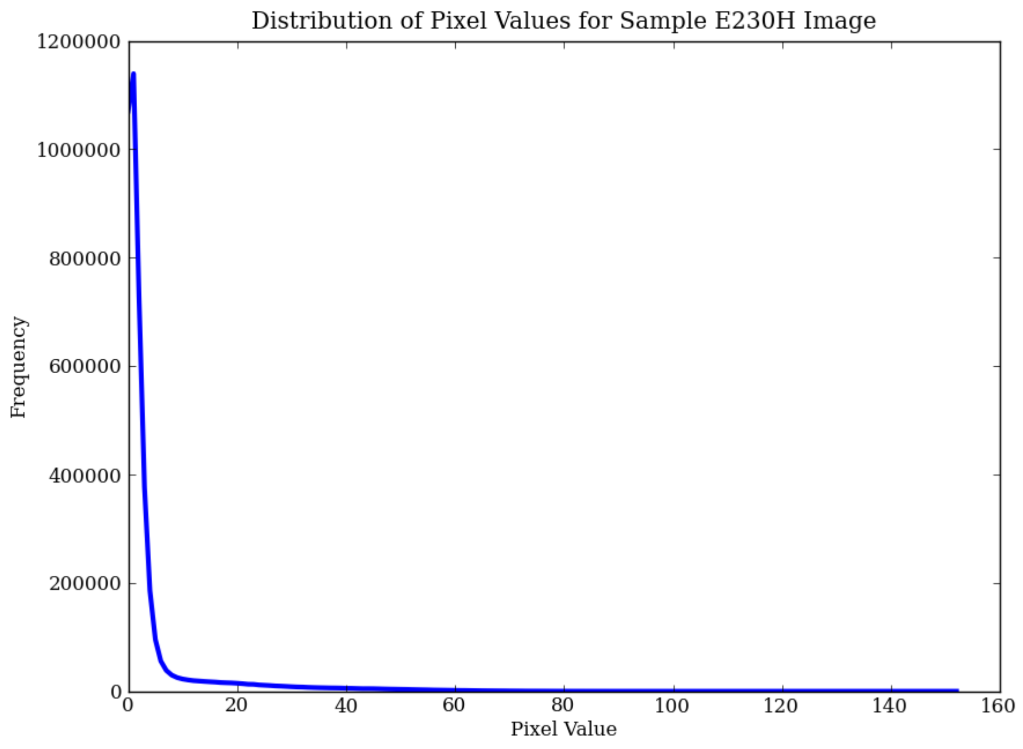

Figure 6 Graph of frequency of pixel values for a sample E230H image.

From this graph the indicator value should be about 10 where the curve flattens. The median for this image was a value of 1, while the mean was a value of 4.31. While the mean is not as good an indicator as 10, it is significantly better than the median and proved sufficient for this project.

Appendix B

Alphabetical List of Modules and Functions

database_management.py
- *add_to_seen_file*
- *addtofile*
- *get_info*
- *readfile*

display_bpx.py
- *determine_subplot*
- *display_graphs*
- *prep_files*
- *run_calstis*

display_corner_data.py
- *batchplot*
- *cenwave_to_rgb*
- *mbplot*

echelle_points.py
- *order_points*
- *vertical_endpoints*

findline.py
- *line*

first_order_points.py
- *distance_from_linreg*
- *g_points*
- *quadrant_points*

manual_update.py
- *update_entry*

pipeline.py
- *generate_tables*
- *process_archive*

table_creation.py
- *build_table*
- *EQs_to_arrays*
- *NUV_MAMA_bpx*
- *opt_elem_bpx*
- *table_arrays*

tools.py
- *append4*
- *fits_info*
- *imshrink*
- *imshrink2*

xandy.py

Appendix C

E230H Q4 File Names

Faint vignetting in corner Q4 of the E230H grating is only detected in some files. Table C1 lists the file names of the data used to characterize the vignetting in this corner.

Dataset Name	Central Wavelength	Exposure Time
o45930020	2263	1617
o45931010	2263	1618
o45932010	2263	1618
o45933010	2263	1618
o45934010	2263	1618
o4ao12010	2513	1200
o4ao12020	2762	211
o4ao12030	2762	1079
o4dd05020	2013	2040
o4dd05030	1763	2799
o4dd05040	2263	1200
o4dd17020	2013	2040
o4dd17030	1763	2198
o4dd17050	2263	1200
o4g001020	2013	80
o4g002020_	2013	80
o4o001040	2513	1296
o4o001050	2762	1296
o4qx01040	1763	2220
o4qx02040	1763	4080
o4qx04040	1763	4020
o4w250010	2063	900
o53p02020	2013	2000
o54302020	1763	1466
o54304020	1763	1506
o54359020	1763	1296
o56l02040	1963	1152
o56l04040	1963	1152
o62l02010	2762	1820
o62l02020	2762	2895
o66p03010	1813	1979
o66p03020	1813	8820
o66p06010	2463	1979
o66p06020	2463	8820
o6bg01030	1913	445
o6e608010	2513	1956
o6fg01010	2013	550
o6lt02020	2013	430
o6lv05010	2013	572
o6lv05020	2013	1080
o6lv05030	2013	1260
o6lv05040	2013	1272
o6lv05050	2013	1200

o6lv05060	2013	624
o6lv05080	2013	1410
o6lv05090	2013	1410
o6lv050a0	2013	1388
o8ma81060	2513	3000
o8ma81070	2762	3000
o8ma81090	2563	3200
o8ma91030	2513	2600
o8ma91050	3012	2610
o8ma93030	2513	1600
o8ma93050	3012	1600

Table C1: Table of datasets from which the bottom right (Q4) corner of the E230H grating were derived manually.

Appendix D:

MIRNUV File Names

The files in Table D1 demonstrated clear vignetting and were used to mark the corners of the MIRNUV vignetting.

Dataset	Dataset cont.	Dataset cont.
o3zka6bhq	o5bq17010	o6i101ohq
o43n02x2q	o5bq20010	o69g02gxq
o43n02x4q	o5bq49010	o69g02gyq
o43n02x6q	o5dc01bxq	o69g02h4q
o43n02x8q	o5ec01snq	o6a351ewq
o43n02xaq	o5ec01sqq	o6bz06woq
o43n02xcq	o5ec01ssq	o6cr01wbq
o43n02xeq	o5in01ssq	o6cr01x3q
o43n02xgq	o5in01stq	o6cr03liq
o43n02xiq	o5in01svq	o6cr03lkq
o46h01ccq	o5in01sxq	o6i101o6q
o46h01ceq	o5in01szq	o6i101o7q
o46h01cgq	o5in01t3q	o6i101obq
o46h01ciq	o5in01t6q	o6i101odq
o46h01ckq	o5in02cgq	o6i101okq
o46h01cmq	o5in02chq	o6i102g9q
o46h02tbq	o5in02cjq	o6i102geq
o46h02tdq	o5in02clq	o6i102giq
o46h02tfq	o5in02cnq	o8h901vfq
o46h02thq	o5in02crq	o8h901vgq
o46h02tjq	o5in02cuq	o8h901vmq
o46h03kbs	o60q03i2q	o8h901voq
o46h03kcq	o60q03i4q	o8h901vqq
o46h03keq	o60q03i6q	o8h901vuq
o46h03kgq	o60q03i9q	o8h901vwq
o46h03kiq	o60q03iaq	o8os03jfq
o46h03kkq	o60q03icq	o8os03jiq
o46h04f0q	o60q03ieq	o8os03jrq
o46h04f1q	o60q03igq	o8q006fgq
o46h04f3q	o60q53y5q	o8q006fjq
o46h04f5q	o60q53y6q	o8q006flq
o49y01tsq	o60q53y8q	o8q006fqq
o4ia1sgeq	o60q53yaq	o8rt02i5q
o4iaa6jfq	o60q53ydq	o8vw01duq
o4j946xkq	o60q53ygq	o8vw01dvq
o4rl01vhq	o60q53yiq	o8vw01dxq
o4xi01wtq	o60q53ykq	o8vw01dzq
o4xi01x1q	o66410mtq	o8vw01e1q
o4xi01x8q	o66420fqq	o8vw01e5q
o4xi02efq	o66y14vuq	obav01v9q
o4xi02enq	o69g01awq	obav01vaq
o4xi02euq	o69g01axq	obav01vcq
o4xi02f1q	o69g01azq	obav01vkq
o53c61keq	o69g01b3q	obav01vmq
o56w01jhq	o69g01b7q	
o5bq16010	o69g01baq	

Table D1 A list of the datasets used to define the MIRNUV vignetting