Skimmed 2D Images

Jeren Suzuki

Last Edited August 27, 2013

Contents

I

1 Introduction

An easier and faster way to crop our three suns in a single image is to find the brightest sun-like shape in our image, crop around it, set the area to zero, then find the next brightest sun-like shape. If we use sort to get a "master array" of positions and values, we can zero-out the parts of the image that are sun-like on the same array multiple times. The result is a fast and efficient cropping method. We also use the term "skimmed" to signify that we skim off the top 1% of the pixels so that any extremely bright outliers will not be incorporated into our image. Actually, that's a complete lie. We only used this "skimming" for the old way of finding a threshold which we have proven to be replaceable by our more robust method. Keeping the carton of skim milk though.

2 1D Plot of a 2D image

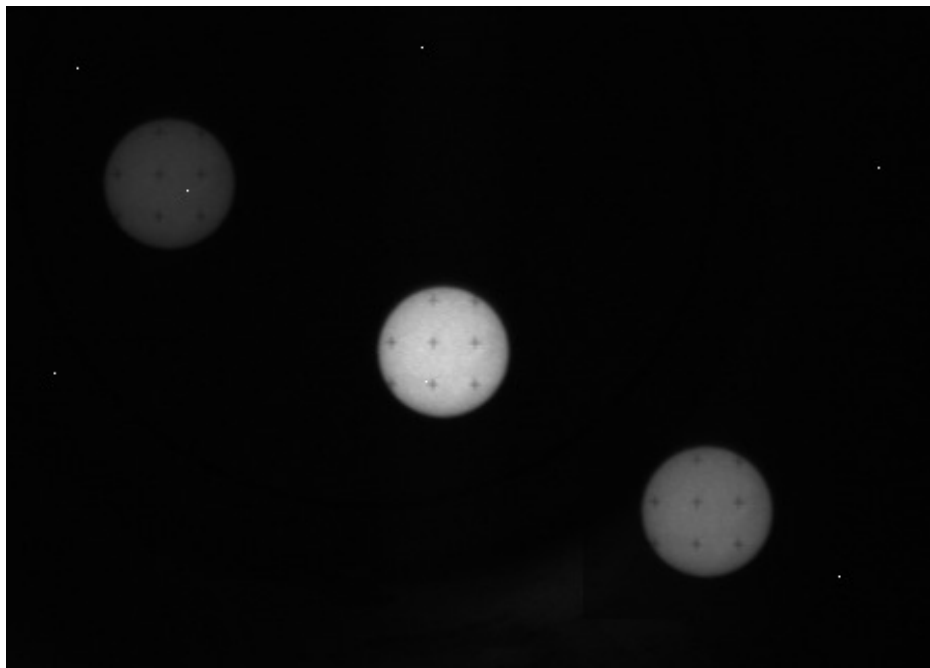We plot a 2D image as a 1D spectrum to identify the shapes in our 2D image.

Figure 1 The raw 2D image we started out with. There are 7 pixels in this image equal to 255, the max brightness for a byte array. These pixels simulate abnormally high values in our image as a result of bad pixels, gamma rays, etc.

Starting from Figure 1, we plotted the lowest 99% of the pixels when ordered by brightness to eliminate the abnormally high pixels to get Figure 2.

We see three distinct humps, indicative of our three suns in the 2D image. Now, to find the boundaries where one sun ends and the other begins, we look at the derivative of Figure 2. However, simply taking the derivative does not result in a usable result so we must smooth our data first. I use both `smooth()` and `ts_smooth()` in Figure 3.

It turns out that `ts_smooth` takes an incredibly long time to run when the order of the autoregressive model is greater than 10. As such, I chose an order of 3 so that it didn't take too long. Even then, running a simple `smooth()` filter results in some pretty good plots. It's important to note that the derivative must be smoothed before I can take another another derivative; also, it doesn't seem to matter if I use `ts_smooth` or `smooth()` when taking the second smooth.
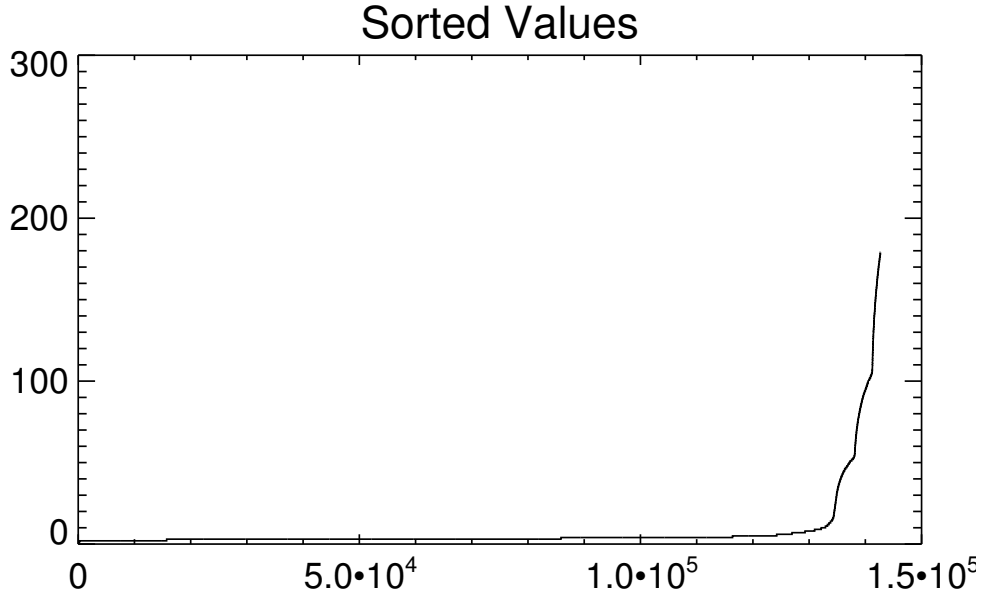
Figure 2 Lowest 99% of sorted 2D image.

3 Drawbacks

The biggest problem with this method is that the pixel values overlap somewhat with different regions. Say the brightest region, region 1, is in the center of our image. Region 2 is at a brightness 50% of the main sun. Say we now arrange the 2D plot as a 1D array and see two bumps, one for each solar region. Now, region 2 is intrinsically dimmer than region 1, which means there are going to be pixels in region 2 that will be comparative to the dark limb pixels of region 1. Since our 1D array (now that it's sorted, especially) has no spatial information, we cannot tell two pixels apart that have the same value but are in different regions.

Another obvious drawback is the necessity of a `smooth` filter. Not only does this require parameterization, it blurs out our data set. Instead of applying a filter, we can attempt to use a histogram plot to isolate the shape of our sorted main array. Figure 4 shows the structure of our three suns.

Bring your attention to the double peak shape around the 50 tick on the x-axis. This is probably the result of the aforementioned inability to extract spatial data from a sorted array. The low-pixel values from one region are being binned into the same bin as the low-pixel values from another region.

4 This Plot Says it All

I wanted to see how many pixels were being counted in the wrong region, henceforth called "leakage-pixels". It turns out that it's not easy to see but it *is* obvious that there are a fair number of them. Depending on how we handle our cropping however, we may not have to worry about it. For example, after we find the center of the brightest region, we set the cropped region to zero so that the middle row thing in the top two plots of Figure 5 is blanked out horizontally. If that's the case, then when we look vertically in the second brightest region, we shouldn't have to worry about counting any pixels from the first region (since it's cropped out).

I also wanted to see what it would look like when the suns are lined up in a single axis: see Figure 6

2

deriv(ts_smooth)

deriv(smooth)

300
200
100
0
−100

25% sun   50% sun   Main sun

1.30•10^5 1.35•10^5 1.40•10^5 1.45•10

300
200
100
0
−100

25% sun   50% sun   Main sun

1.30•10^5 1.35•10^5 1.40•10^5 1.45•10

deriv(ts_smooth(deriv(ts_smooth)))

deriv(ts_smooth(deriv(smooth)))

300
200
100
0
−100

25% sun   50% sun   Main sun

1.30•10^5 1.35•10^5 1.40•10^5 1.45•10

300
200
100
0
−100

25% sun   50% sun   Main sun

1.30•10^5 1.35•10^5 1.40•10^5 1.45•10

deriv(smooth(deriv(ts_smooth)))

deriv(smooth(deriv(smooth)))

300
200
100
0
−100

25% sun   50% sun   Main sun

1.30•10^5 1.35•10^5 1.40•10^5 1.45•10

300
200
100
0
−100

25% sun   50% sun   Main sun

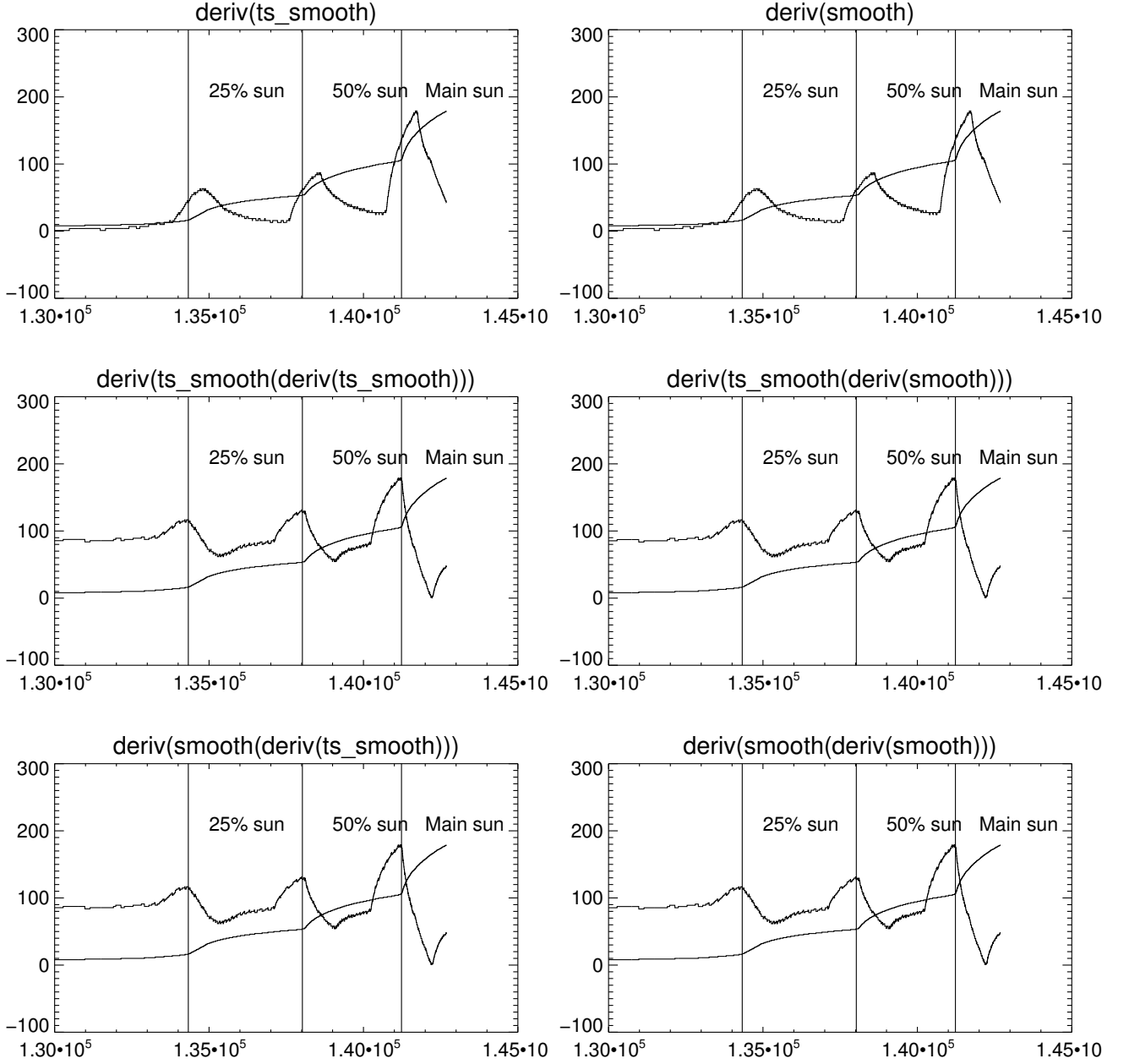1.30•10^5 1.35•10^5 1.40•10^5 1.45•10

Figure 3 The vertical lines correspond to eyeballed boundaries of the sorted array. A large part of the left half of the array is cropped out to emphasize the shape of the humps and peaks. The derivative has been scaled to within the max/min of the starting master array. The width of the smoothing filter is 1000 wide.

5 Some Numbers

Using the above methods, we find peaks of the second derivative of the smoothed 2D→1D array. Using the peak positions, we return the value of the sorted array at that peak position and set that to be a threshold. The result is a set of three thresholds based on the shape of the 1D image. We keep our code identical and only change the threshold value. Table 1 shows that the thresholds we find are good! There is one small note however, the exactly position of the peak is not used, but rather the position +.1% of the total number of pixels in the 1D array. The problem was that if we did simply return the sorted
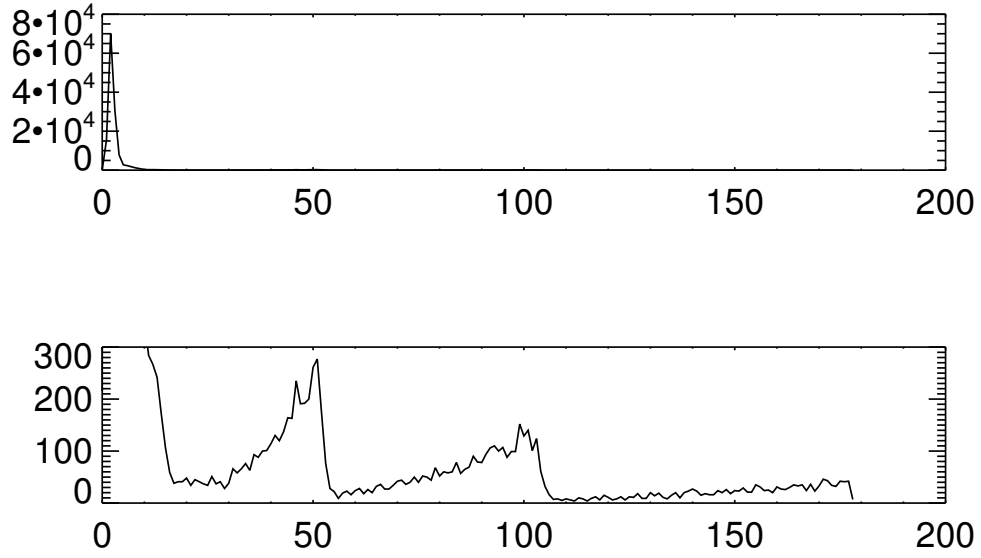
Figure 4 The top is the histogram of the sorted array. The bottom is the same plot but zoomed to a y-range of [0:300].

array value as a threshold, we'd get cropping errors. Adding a little bit to the threshold value is safe, however, since it reduces the number of pixels in our centering mask. The less pixels there are, the higher on the dome-shape we lie, which means the more circular our mask is. A more circular mask means a more accurate center position to use for our limb-fitting program.

6 Taking this Method Further

Once we have the sorted array we can try to do cool things with it. For example, we can attempt to find the quick centers of the three suns. We can do this by partitioning the sorted array into three parts, 1 for each sun. We average the positions of all pixels above a certain threshold and then zero-out a box that is 120x120 pixels wide centered at the recently-found center. We repeat the same process, computing the average position of all pixels above a certain threshold. We don't have to worry about the darker limb pixels of the main sun because we have set all the pixels around the center to 0.

Table 1.   Center Positions Using Different Thresholds

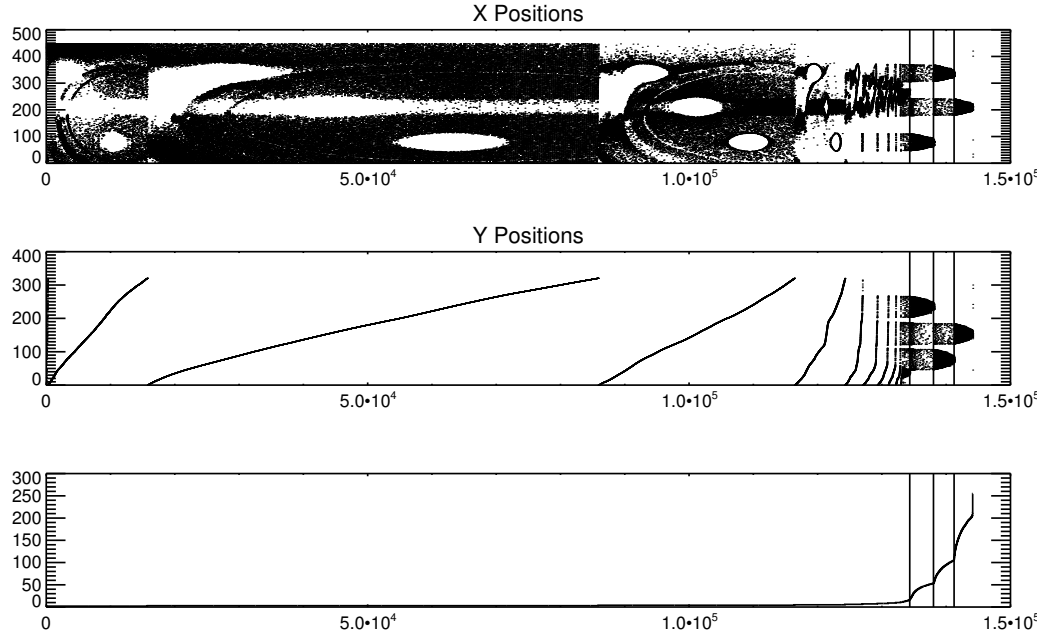| Sun Brightness (%)	X Position	Y Position	Threshold
			Old
100	210.50238	154.27054	116.35
50	337.80600	76.894958	76.5
25	78.683426	235.11536	35.8
			New
100	210.48686	154.25601	123
50	337.91956	77.234901	62
25	78.887283	234.69048	20

Figure 5 This plot reveals that when sorted, the regions bounded by the vertical lines (which are artificially places, not the result of any computation) include pixels from neighboring regions. For high-value pixels, i.e., the region bounded by the rightmost vertical line and the last element of the sorted array, the x and y positions are completely (no vertical dots) isolated from other regions. When we look at the middle region, we see that low-value pixels from the brightest region(most likely dark limb pixels) are being sorted into the middle region of the sorted array. The worst case scenario is the left-most region which has pixels from both the first and second regions. This approach, however, is without the inclusion of cropping and then setting the cropped area to zero.

6.1 Without Zeroing Out the Array

There is another method of finding the centers, this one without the need to set the parts of the sorted array to zero. The benefit is that if we need to use the sorted array for something else, we won't waste time having to redefine it. The downside is that it is a factor of 2 slower. Instead of setting parts of the array to 0, I use `where()` statements to simply avoid looking at those areas. Figure 8 compares the relative accuracy of the rough centering methods.

7 Robustness of Finding Thresholds

We've established that we can use this method to find the approximate centers of the main and auxiliary suns, but perhaps the code is not robust enough to properly threshold our image in all scenarios. To test this, we have scaled our image to different levels and see how the program deals with a significantly brighter/dimmer image. Tables 3 and 4 reveal that using the `max()` method provides a more robust calculation of the threshold. We could apply some dynamic changes to the threshold value, but it may be easier to just use `max()` instead.

Also, it turns out that eliminating the top 1% of the image when sorting it clips a bit too much of the top off. Instead, we must resort to clipping the top .1% instead.

Table 5 shows that regardless of how scaled our image is, the `max()` method to find peaks of the `deriv(smooth(deriv(smoothedarray)))` array return consistent results.

Table 2. Center Positions Using Different Methods

Type	Sun Brightness (%)	X Position	Y Position
Old	100	210.174	153.800
"	50	337.717	76.7694
"	25	83.4095	232.106
Zeroing Out	100	211.276	153.170
"	50	336.498	72.692
"	25	78.4613	237.583
Using where() statements	100	211.276	153.170
"	50	338.652	83.962
"	25	81.698	229.154

Table 3. Thresholds of Brighter/Dimmer Images Using Threshold on deriv(smooth(deriv(sorted)))

% Scaling of 0-255	100% Peak	50% Peak	25% Peak	100% Peak Position	50% Peak Position	25% Peak Position
10	12.0079	6.10	0	141225	138114	-1
20	24.4961	12.4488	3.81496	141225	138114	134328
30	36.5039	18.5512	0	141225	138114	-1
40	48.9921	24.8976	7.62992	141225	138114	134328
50	61	31	9	141225	138114	134328
60	73.4882	37.3465	0	141225	138114	-1
70	85.4961	43.4488	13.3150	141225	138114	134328
80	97.9843	49.7953	15.2598	141225	138114	134328
90	109.992	55.8976	17.1299	141225	138114	134328
100	122.48	62.2441	19.0748	141225	138114	134328

Table 4. Thresholds of Brighter/Dimmer Images Using max() of on deriv(smooth(deriv(sorted)))

% Scaling of 0-255	100% Peak	50% Peak	25% Peak	100% Peak Position	50% Peak Position	25% Peak Position
10	10.4331	5.31496	1.47638	141241	138114	134328
20	21.2835	10.8425	3.01181	141242	138114	134328
30	31.4173	16.1575	4.48819	141229	138114	134328
40	42.5669	21.6850	6.02362	141242	138114	134328
50	53	27	7.5	141234	138114	134328
60	63.8504	32.5276	9.03543	141242	138114	134328
70	73.5827	37.8425	10.5118	141229	138114	134327
80	85.1339	43.3701	12.0472	141242	138114	134328
90	94.6654	48.6850	13.5236	141229	138114	134328
100	106.417	54.2126	15.0591	141243	138114	134328

Table 5. Center Positions Using max() of on deriv(smooth(deriv(sorted)))

| % Scaling of 0-255	100% X	100% Y	50% X	50% Y	25% X	25% Y
10	210.7676	153.4890	336.5794	72.6291	78.4616	237.5705
20	210.7702	153.5003	336.5794	72.6291	78.4616	237.5705
30	211.2508	153.2018	336.5870	72.6267	78.4608	237.5807
40	210.7702	153.5003	336.5794	72.6291	78.4616	237.5705
50	211.0959	153.2774	336.5890	72.6232	78.4608	237.5807
60	210.7702	153.5003	336.5794	72.6291	78.4608	237.5807
70	211.2508	153.2018	336.5870	72.6267	78.4608	237.5807
80	210.7702	153.5003	336.5794	72.6291	78.4616	237.5705
90	211.2508	153.2018	336.5870	72.6267	78.4608	237.5807
100	210.7678	153.5117	336.5794	72.6291	78.4608	237.5807
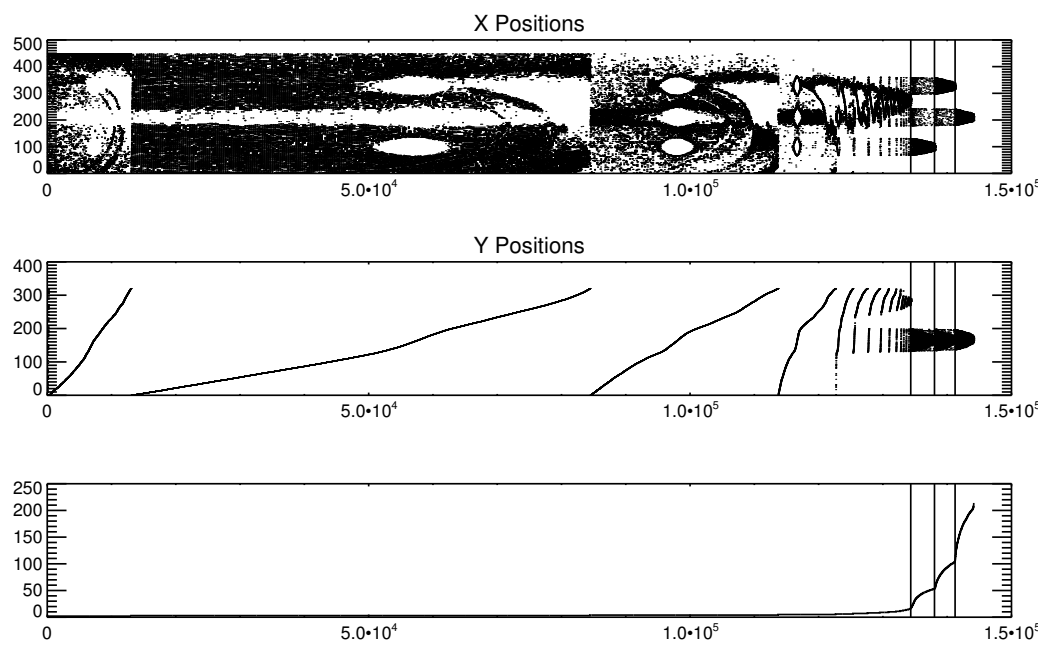
Figure 6 The suns are lined up vertically

7
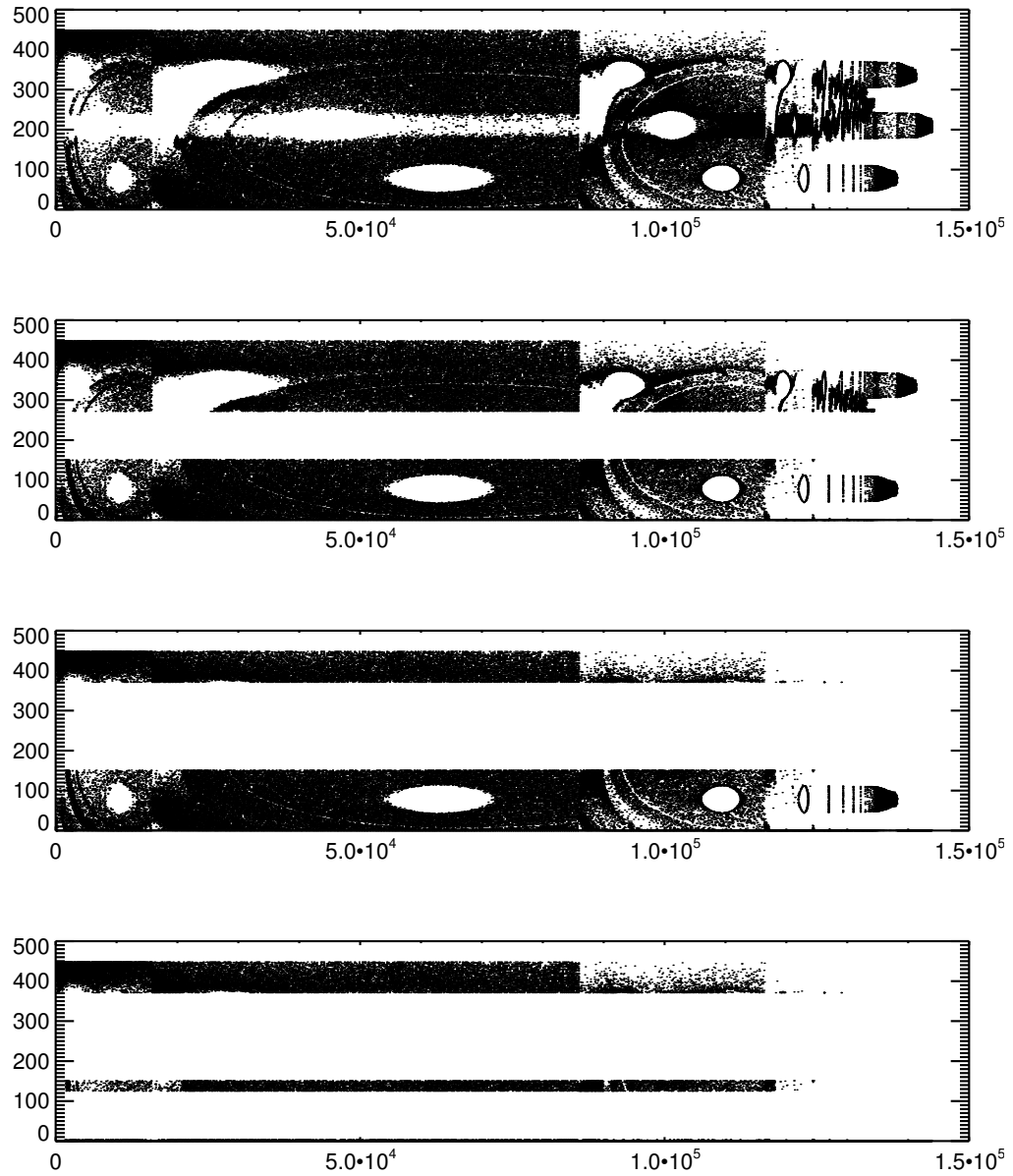
Figure 7 The top row are the x positions plotted against the sorted pixel values. In the second row we find the center of the brightest region and set the x (and y) positions in a 60-wide window to 0. In the third region we find the pixel positions of the second brightest region and repeat the process of zeroing out neighboring pixels. In the third row, only the dimmest region remains, free from the dim pixels of the first two regions.
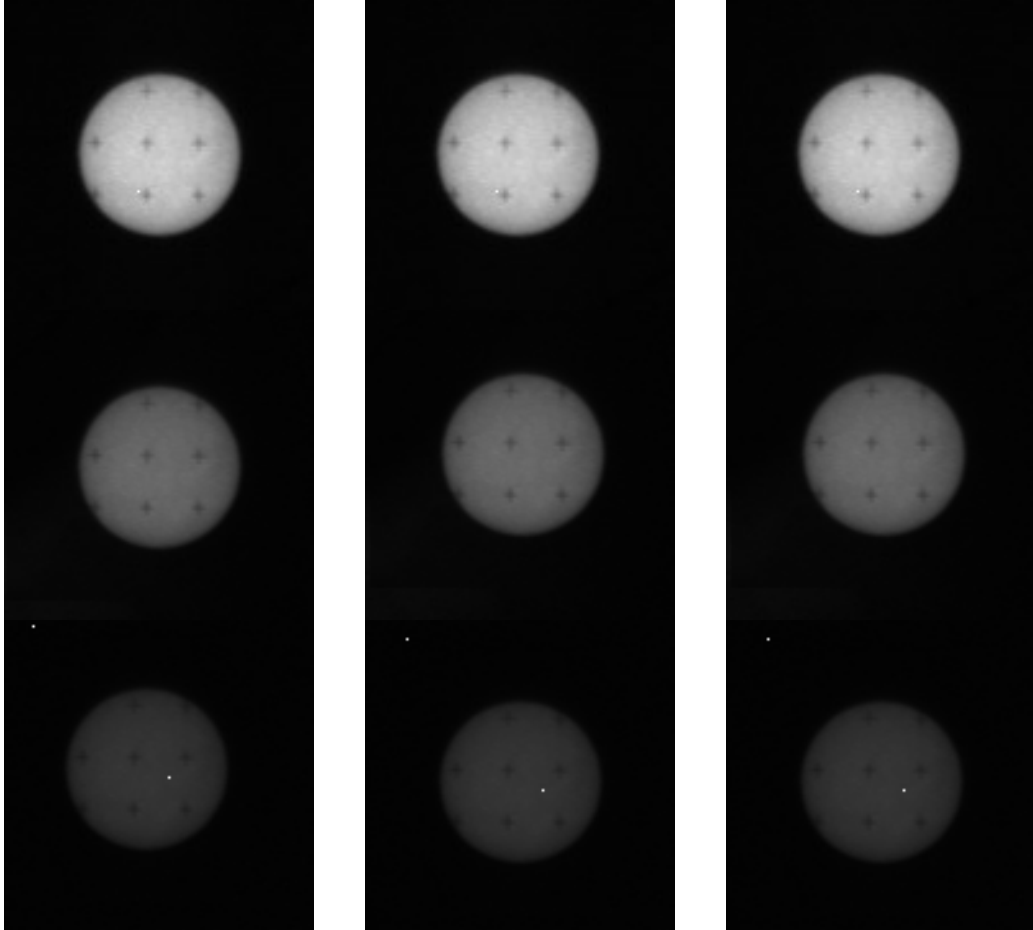
Figure 8 The left column is our present cropping method, the middle column is the method using `where()` statements, and the right column uses the zero-out method.