Skimmed 2D Images

Jeren Suzuki

Last Edited April 24, 2013

Contents

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

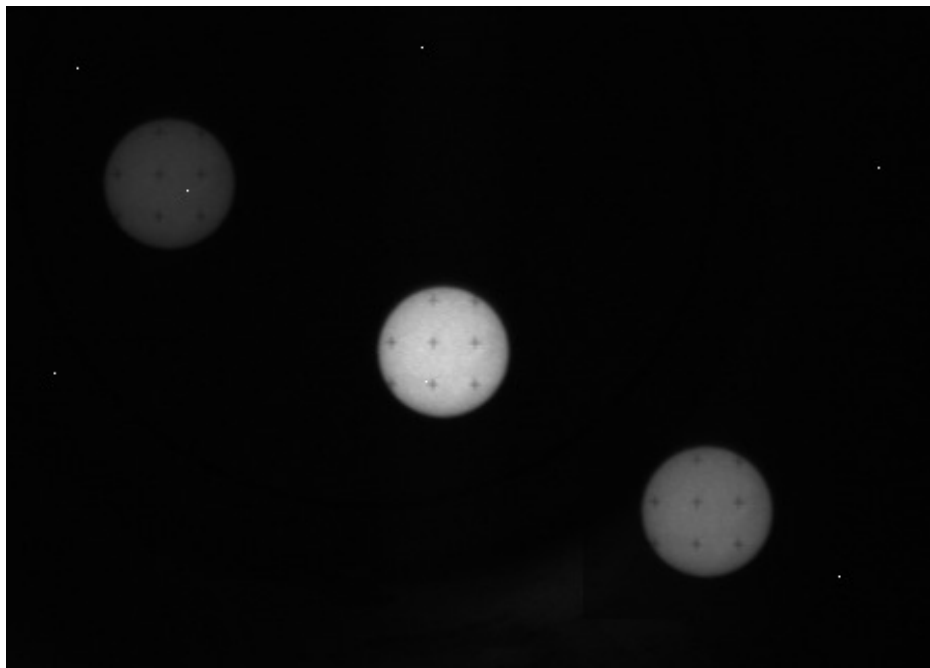We plot a 2D image as a 1D spectrum to identify the shapes in our 2D image.

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.
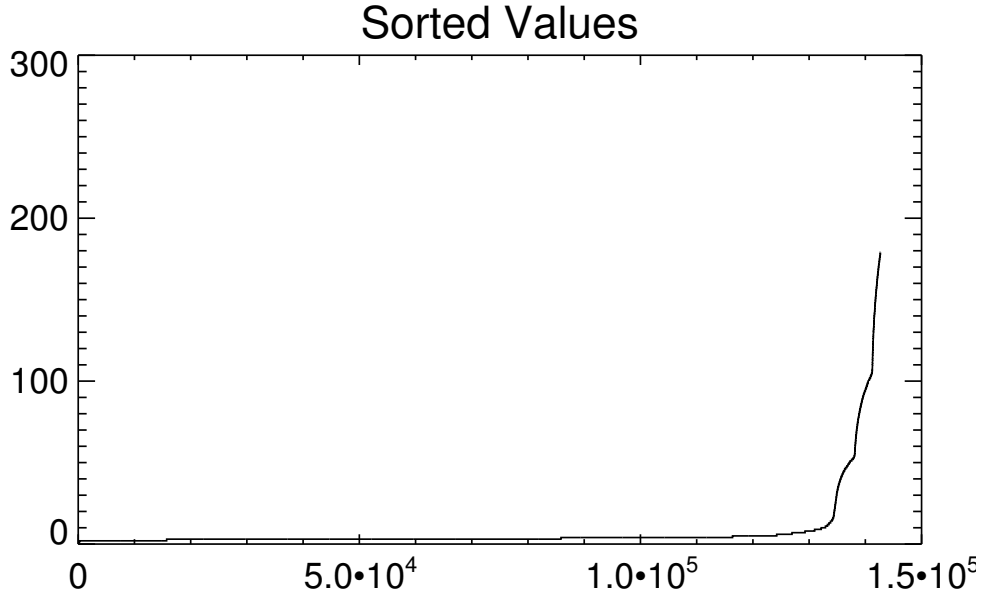
Figure 2 Lowest 99% of sorted 2D image.

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. Thus, it is safe to conclude that we cannot blindly use a sorted array without some sort of spatial reasoning. Figure 5 suggests we think harder about how to crop out auxiliary suns while operating on a single array.

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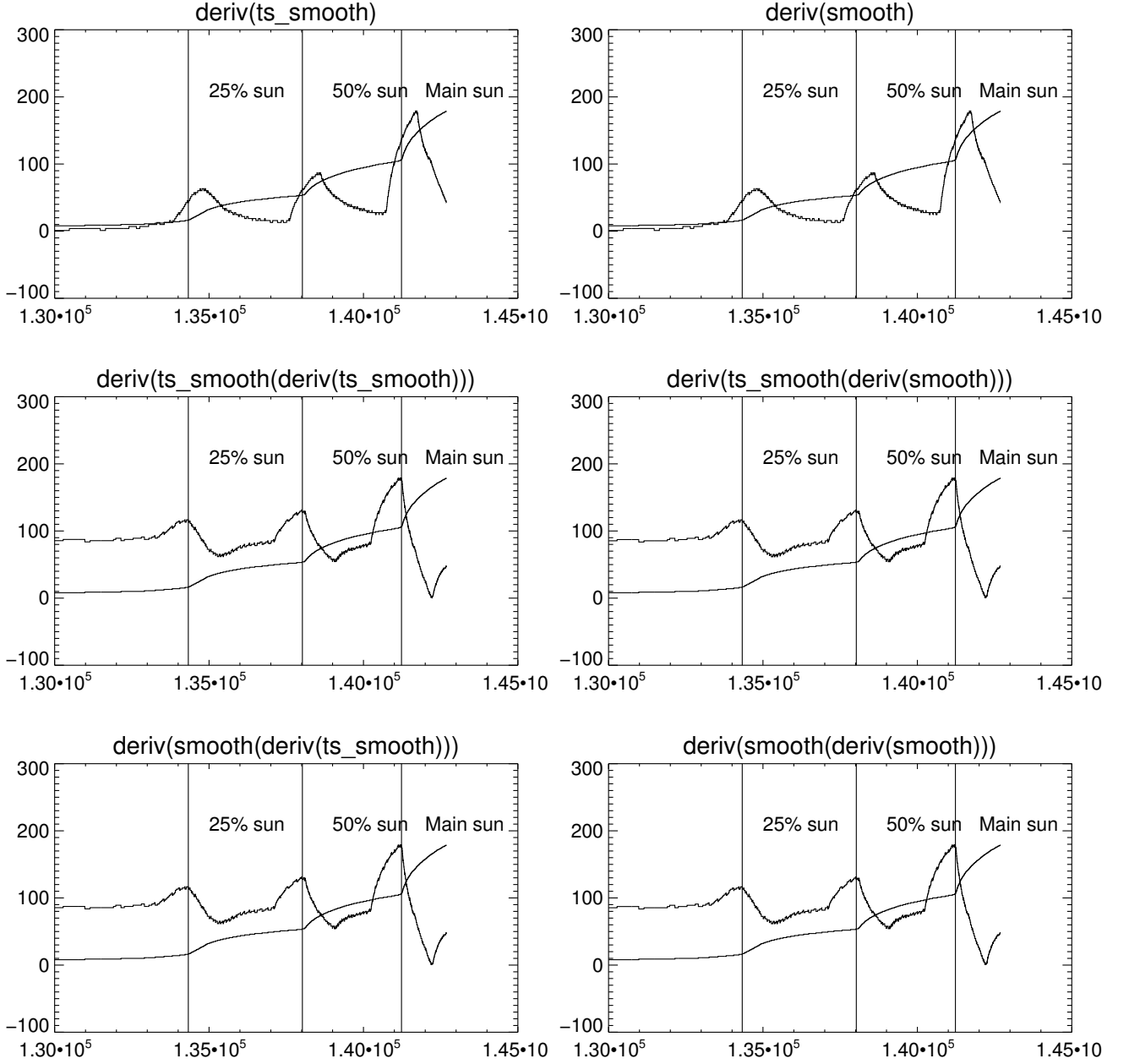
Figure 3 The vertical lines correspond to eyeballed boundaries of the sorted array. A large part of the left half of the array is cropped out to emphasize the shape of the humps and peaks. The derivative has been scaled to within the max/min of the starting master array. The width of the smoothing filter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

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 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.
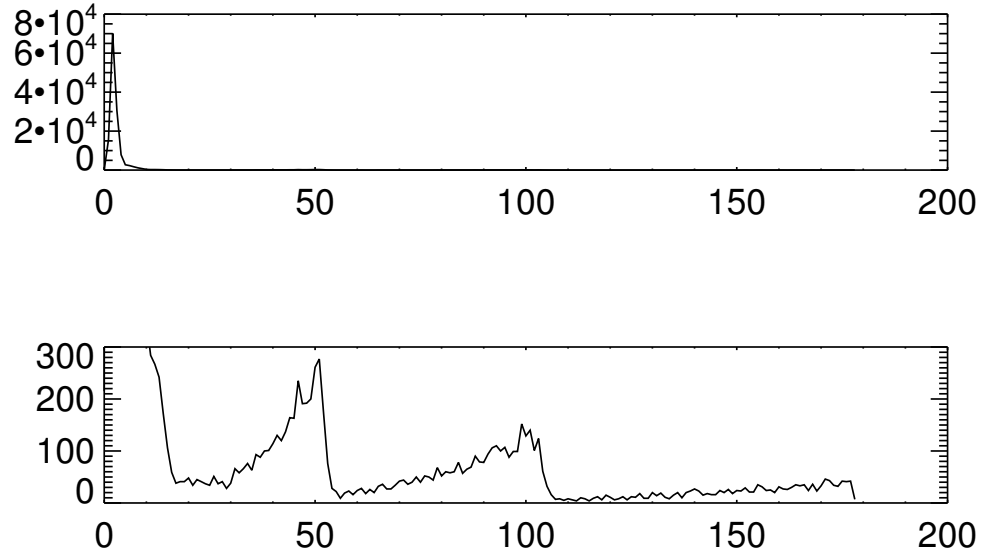
3

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].

Table 1. Center Positions Using Different Thresholds

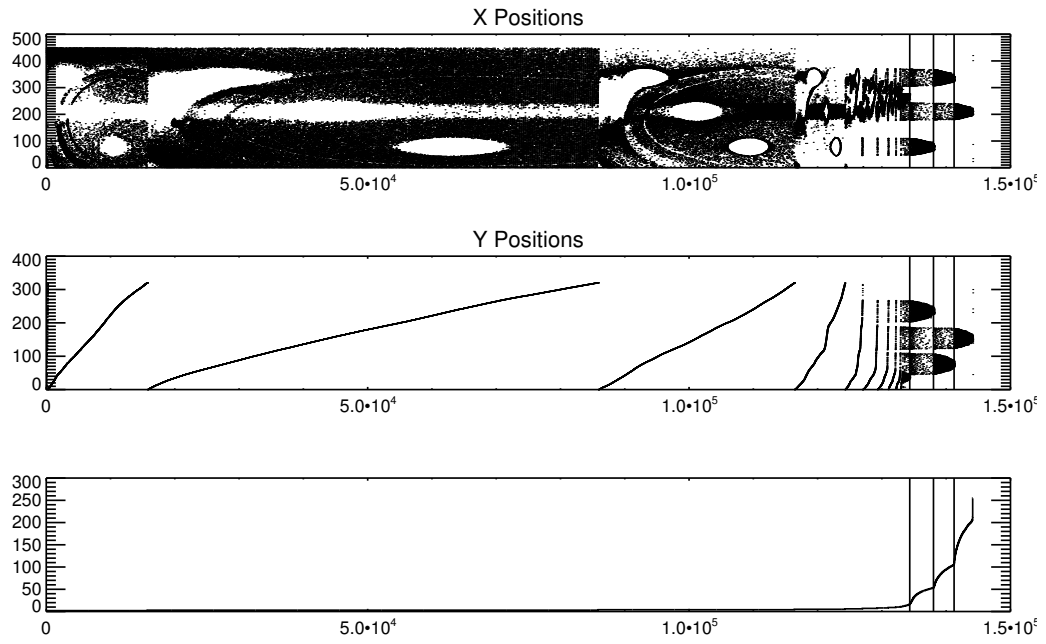| Sun Brightness (%)	X Position	Y Position	Threshold
			Old
100	210.50238	154.27054	116.35
50	337.80600	76.894958	76.5
25	78.683426	235.11536	35.8
			New
100	210.48686	154.25601	123
50	337.91956	77.234901	62
25	78.887283	234.69048	20

4

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.

5