Why We Do What We Do (WWDWWD)

Jeren Suzuki

Last Edited September 5, 2013

Contents

Introduction

This document is an attempt to organize the decision behind our methods in the grand scheme plan of extracting useful data from our image. I will try to organize it in parts so that it will be easier to follow.

1 Data Acquisition

Even before we have an image, we have to talk to the camera, so to speak. This involves installing the PvPAPI SDK on an ubuntu machine connecting the machine directly to the camera through an ethernet port, then manually setting the network interface to lie on the same subnet as the camera (the camera can't change IP so it must be done on the computer). Once the IP and subnet mask (and gateway too?) are configured correctly, the camera can be set to take pictures by navigating to the `AVT GiGE SDK` → `bin-pc` → `x64` directory and running `SampleViewer`. You should see a widget appear with the camera listed in the interface. Setting up the camera is beyond the scope of this document but I have partial instructions in `sdkrefman` and Nicole has printed out instructions in one of her binders.

1.1 Setting up image

Now that we've setup the camera, we need something to take pictures of. We simulate a solar image based on the images we received from Albert. To do this, we need:

- Pixel pitch of camera sensor (physical size of pixel)
  If you don't have the pixel pitch, use the sensor dimensions against the image resolution
- Image resolution (4192x3264, etc)
- Distance from focal plane to imaged plane
- Focal length of lens

we stick them into this equation:

$$\frac{\text{pixel pitch}}{\text{focal length}} = \frac{\text{physical length of a pixel in the image}}{\text{distance from camera to sheet of paper} - \text{focal length}} \tag{1}$$

If we want the sun to be N pixels wide (corresponding to M cm), then open an image editor at 300 pixels/cm and make a sun

$$300\,\frac{\text{pixels}}{\text{cm}} \cdot M\ \text{cm} \tag{2}$$

$$300\,\frac{\text{pixels}}{\cancel{\text{cm}}} \cdot M\ \cancel{\text{cm}} \tag{3}$$

$$300 \cdot M\ \text{pixels wide} \tag{4}$$

Figure 1 compares the *fiducials* we were testing to the starting image we based them off. *Fiducials*. Not the shape/size of the sun.

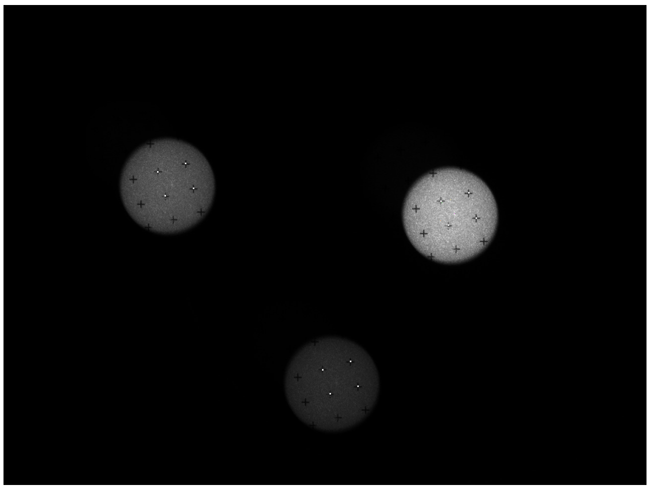
2 Handling 2D Image

2.1 File Formats

In an effort to maintain maximum portability across all platforms, we saved all our images in the *FITS* file format. Any images received for testing purposes (for example, the ones from Albert), I opened in IDL and converted into FITS.

2.1.1 Specific to Photos from a DLSR

When dealing with images generated with my DLSR when taking bundle mask images, they had to be converted from RAW into 16 bit TIFF then loaded into IDL. JPEG files introduced artifacts and the actual range of a pixel in a JPEG file wasn't as high. TIFF files were considered filled with unsigned integers while JPEG files only ranged in BYTE values. If there is the option, use TIFF files. The only problem is that TIFF files are incredibly large.


(a) The image we based the fiducial dimensions on (and only fiducial dimensions, not solar dimensions)

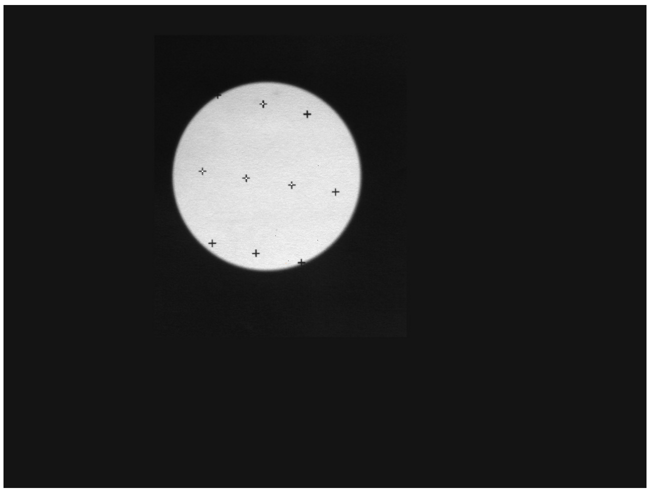

(b) Our simulated image. Ignore the hilariously large sun and notice the fiducials are about the same size.

Figure 1

Also, photos taken from my camera come with three channels, red, green, and blue. When doing image analysis, I chose the image with the most contrast when on a black-white color table. If my memory serves me right, the fits files contain the first channel of the 3 channel image when loaded with either `READ_JPEG` or `READ_TIFF`, corresponding to the red channel.

3 Skimming the Top Off

We want to eliminate any outlier pixels that are way too bright so we sort our 2D image as a 1D array and mask out the top 1% (or .1%, whatever) of the pixels for our threshold. Even though in the sample images provided by Albert there are no bright outlier pixels, eliminating the brightest pixels poses no detrimental affect to image processing.

4 Deciding Whether or Not to Use Image

Technically, this is more whether or not to *keep* the image

To make sure that we identify the appropriate centers of the sun, we make sure that we know how many are actually in the picture. First, we check if there are any pixels on the border of the image that are a significantly higher value than the mode of the image. If we see X pixels on the border, then one of the suns is cut off. If we see X and Y pixels on the border, then we take into account 2 cut-off suns. We must make sure that if we see no sun-pxiels on theborder that there isn't a chance that we missed a sun altogether and there are only 2 suns in the image. Still need to work on this.

After the centers of every solar-shaped object is found, two checks are made:

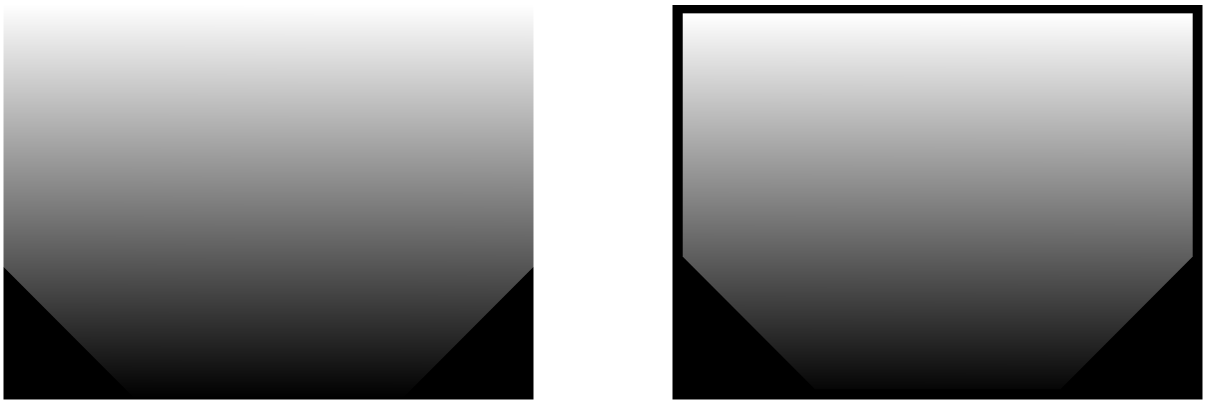
1. Is the center too close to the edge?
2. Is the center too close to certain edges after being rotated 45°?

Previous methods included:

1. Check for bright pixels on image border, if number of continuous bright pixels exceeds a threshold, the general area is considered "*bad*"
2. Count bright pixels on image border mask, if number exceeds a certain threshold, use further processing to determine where in the image the bad sun lies

We chose the rotation method because it quickly and efficiently deals with the problem that the bottom corners will never see any data. The problem with using a border mask was that it's easy to construct for a rectangle, but when you start throwing in angles, the amount of time it takes to construct the mask increases quickly. To deal with the bottom two corners, one method was to make triangle matrices with 0s and 1s and then append them to the bottom corners of a large rectangular matrix. Another method was to use `shift` to move around the rectangular matrix within a larger matrix, multiply the shifted matrices together, and then crop the final matrix down to the original size.

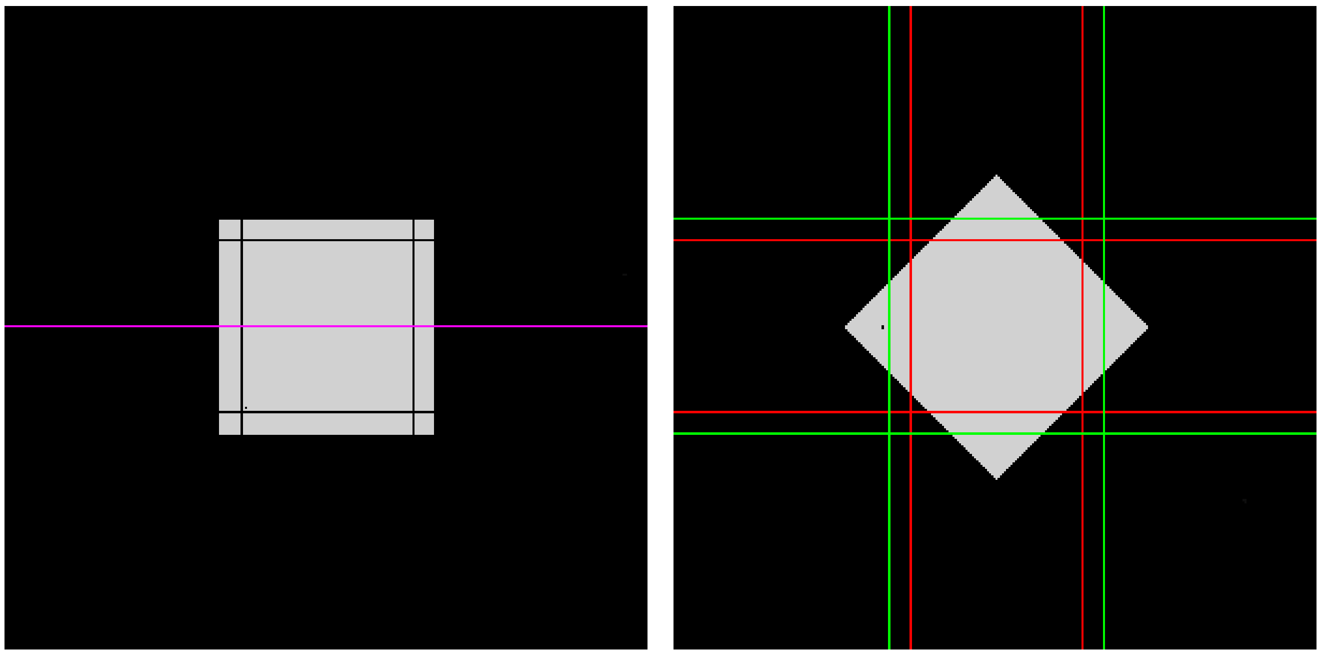
Still within the realm of rotating the image, the first attempt involved rotating the entire image and then checking the distance from the edge. This was completely unnecessary since at this point we already have the center positions of the solar shaped objects and thus only need to apply a rotation function onto a single (x, y) coordinate.



(a) The image mask with the bottom corners as no-data zones

(b) What the initial attempts at border-masking achieved, although at the heavy cost of time. If there were any solar pixels identified in that border region, then further processing would occur.

Figure 2



(a) The gray square simulates the solar image with 3 suns. The black dot is the center of a sun that would otherwise be considered "good" by the program since it doesn't take into account the bottom corners.

(b) With the image rotated 45° (but in reality, only the black dot is rotated. the entire array is rotated here to show the effect), the black dot now lies beyond the green line, which is the original border of the gray square. The left vertical and bottom horizontal green line mark the boundary regions for the bottom two corners. This method is fast.

Figure 3

5 Finding Centers of Sun(s)

We used a mask centroiding program where we take the center of mass of any shape above a certain threshold. We then zero-out a box around the sun, and look for the next shape above a threshold.

There was another method we didn't get very far on, but it had the potential to being a lot faster and *maybe* more robust. We sorted the 2d image instead of by value, but by either x or y position. In fig. 4, the top image is the 2d sorted image by either x or y position, I forget which (oops). The three sideways fingers correspond to the three solar regions. Notice how the very dark tips vertically align with each other. With the first threshold (which is defined by a vertical line and anything to the right of the line is considered above the threshold), we eliminate the rightmost finger by zeroing out anything a little before and after the x/y range of the pixels, leading to the second row. This puts the second brightest sun as the rightmost finger. Process is repeated until all suns have been cropped. This method is faster because it doesn't deal with the re-processing of a 2d image, only it zeroes out indices in a 1D array.

This method was tested because it reused the `sort()` function from making the threshold and simply applied the sort indices to x and y positions instead of pixel values. The result was a fast and robust centering method. The problem was if the suns lied in either the x or y axes (which is most probably the case). As you can see from the figure, if any of the suns share the same x or y range, even a little bit, the centering will be off. A solution was to check both the x and y indices (because you can't overlap in both), but further development was stopped in favor of a more intuitive (and easier to code) approach. I think we could've made it work though.

From the approximate centers, we crop a box around the sun and slice rows/columns of data. Next, we limb fit the edges of the sun so that we can find more accurate center positions. At first we used a 3 order fit, then a linear fit (since that's what Albert used), then to a 2nd order fit (because we thought it wasn't good enough) then back to a linear fit (because the fit was good enough).

5.1 Thresholds

We want a robust threshold so instead of using an arbitrary parameter multiplied by the max of the image, we sort the 2D image into a 1D array and return the position of the boundaries of the regions which are seen as humps.

To find the boundaries, we look at the 2nd derivative of the sorted array and find the 3 maximum peaks (in the case of 3 suns), corresponding to the boundaries of the aforementioned humps. We chose this way because thresholding the peaks requires parameterization that needs to be changed dynamically.

Several factors affecting thresholding values:

1. `smooth()` parameters
2. Unusually bright pixels
3. artifacts of smoothing

To solve 1), we used the `edge_truncate` keyword because it gave us the least amount of weird edge artifacts in the 1D array. The other options were `edge_zero` and `edge_mirror` which yielded poor results.

To solve 2), when we sorted the array by brightness, we left out the top .1% of the pixels.

To solve 3), we had to manually cut off the last 10,000 pixels. This part needs to be improved.

6 Finding Fiducial Positions

Now that we have a limb-fitted centroid, we analyze the sun for fiducials. They are characteristically dark, so we ended up looking at 1D sums in the row/column directions and looking for wide dips in brightness (where 1D sum lines up with the entire length of the fiducial). Before this method, we looked at a few other options. They included:

1. Running edge detector filters to isolate the edge of the fiducials
2. Applying convolution/cross-correlation filters

For edge detection filters, we looked at `emboss`, `shift_diff`, and `laplacian`. Using 2D filters posed a few problems. One, it was time consuming because it required processing on a 2D array. Two, the result was a 2D array that we had to further process. The basics of the edge detection filters was that when run with a kernel that emphasized an edge of the fiducial, we get an array that traces out the shape of the fiducial (to some extent). For the case of `shift_diff`, a high value corresponds to a leading edge and a low value to a falling edge. We threshold the 2D image for a high threshold so we retrieve all the leading edges in a certain direction. Next, we see where in the image the leading edges are so that we can pinpoint exact fiducial positions.

One issue was that the method used involved returning only the x and y positions of each fiducial candidate in no specific order. They weren't paired up in a certain way so we got a long list of x and y positions. We had to look at each possible fiducial pair and determine if there was actually a fiducial there or not (is the pixel value at the coordinate dim enough). Then, we checked to see if the fiducial coordinate was more than a certain distance from the solar center (because we only care about fiducials close to the center anyhow). Once it passed

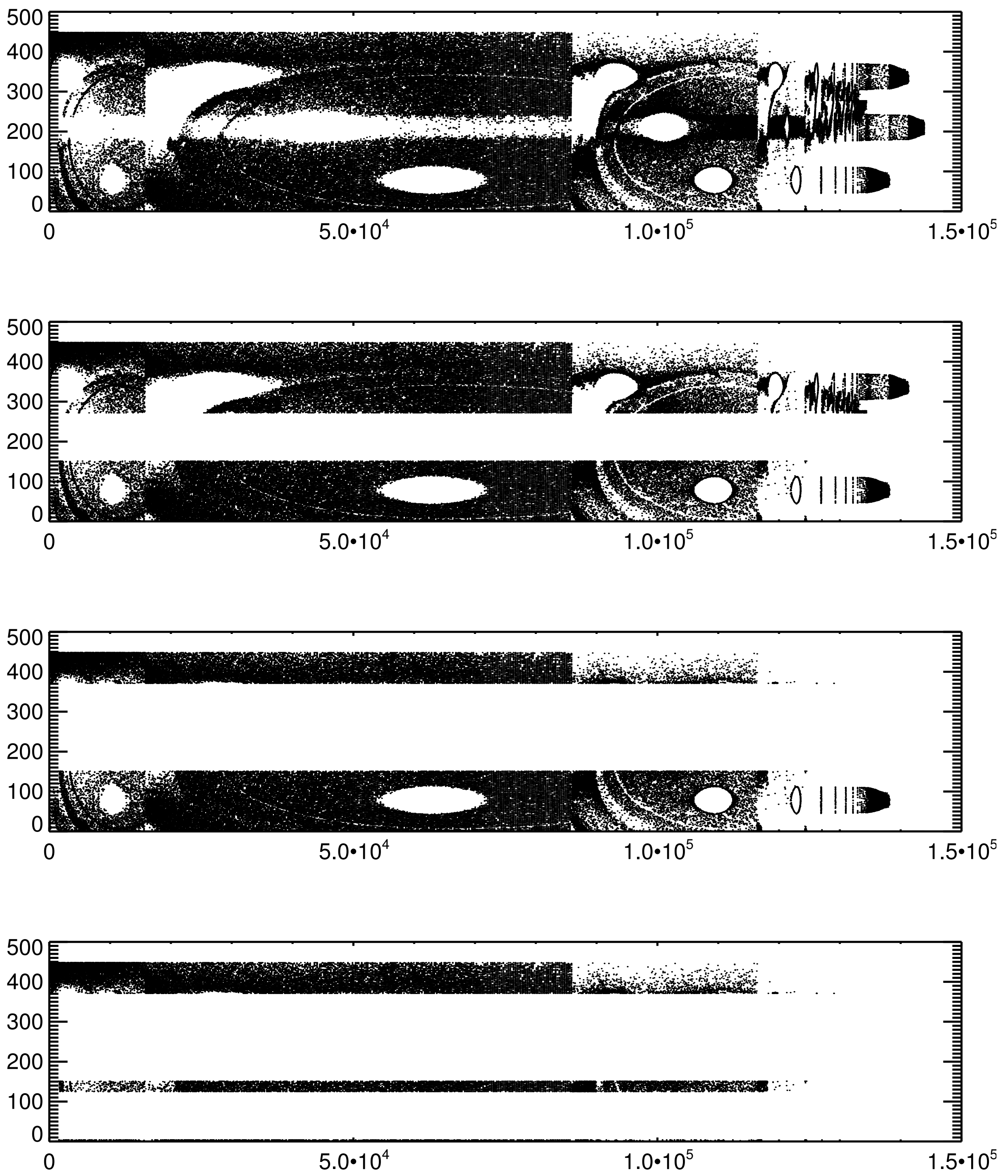


Figure 4: The 2D array sorted by x or y positions

this second check, we cropped a small area around the fiducial candidate and looked at the 1D sums per row/-column. If there is a dip (characteristic of a fiducial), then the fiducial candidate is confirmed as an actual fiducial.

In addition to using edge detection filters to find fiducials, we also looked at a convolution/cross correlation filter. I mention cross correlation because it is the method Albert used in his image analysis code. Convolution and edge detection filters only differ by the shape and content of the kernel so it required many a tweaks to find one that emphasized the shape of the fiducial while suppressing foreign noise and shapes. What we ended up with was

Another problem related to fiducial finding is dealing with fiducials on the edge of the sun. If a fiducial lies on the edge, it is hard to distinguish from the edge pixels without some sort of spatial information, like a

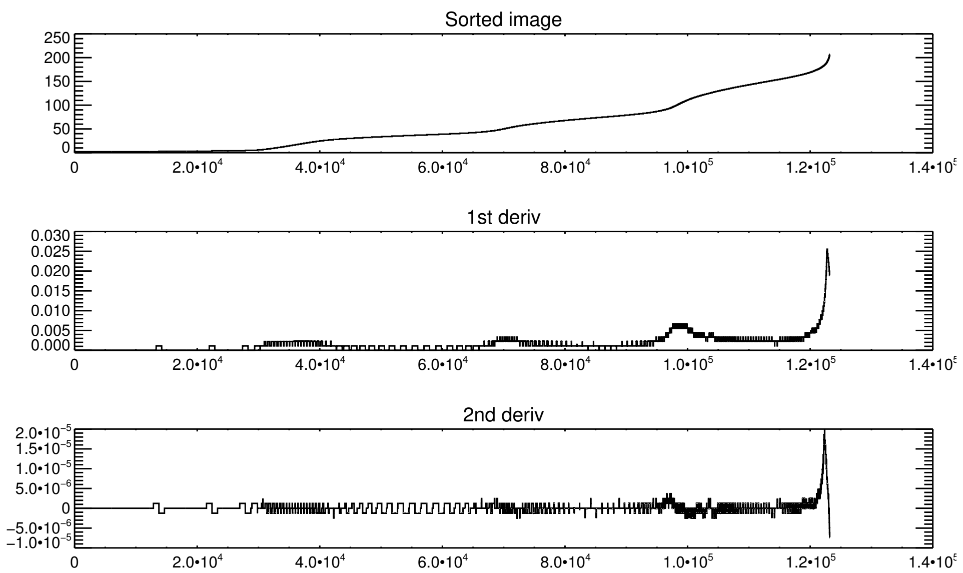


Figure 5: This is the sorted array for a 3-sun image. The peaks don't look that large in the second derivative, but that's just because of the scaling to the first peak.

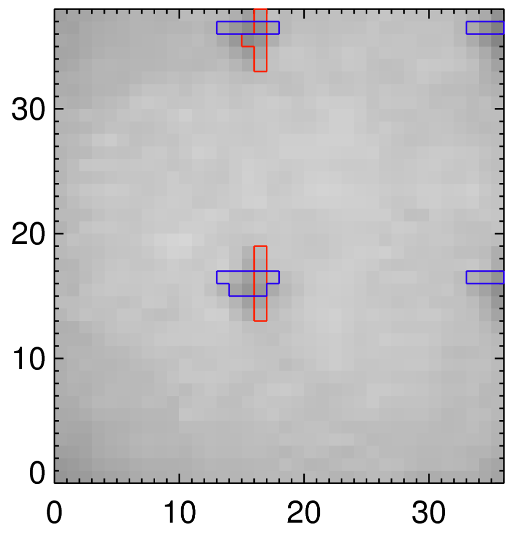


Figure 6: w

convolution filter. Currently we don't have a method to deal with dim fiducials on the edge because we only care about the 4 brightest fiducials to the center of the sun.

7 Miscellaneous

Extra

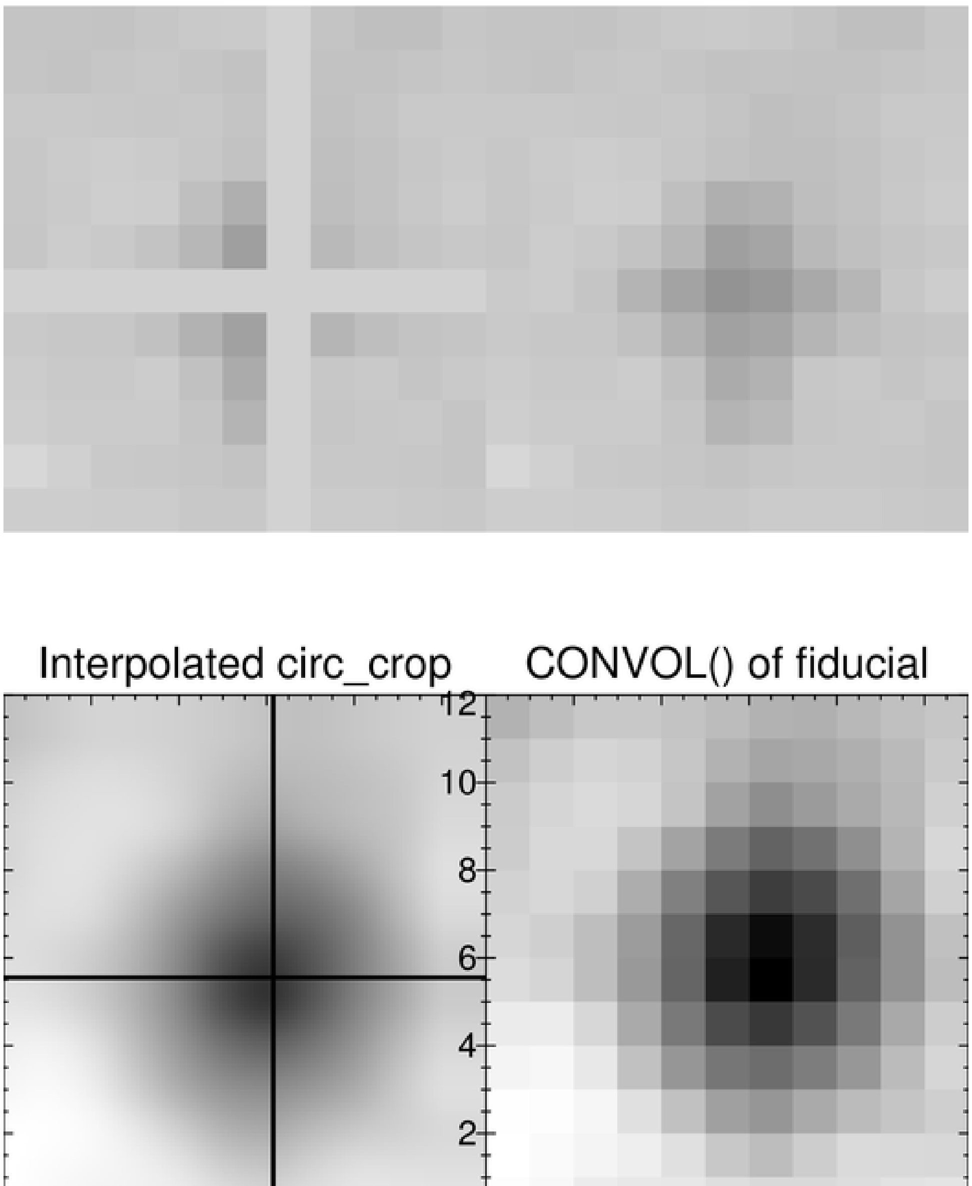

Interpolated circ_crop
CONVOL() of fiducial
0
2
4
6
8
10
12
20
40
60
80
100
0
2
4
6
8
10

Figure 7: w