# Notes on Current Method Applied to New Grainy Image

Jeren Suzuki

Last Edited August 27, 2013

## Contents

# 1 Introduction

Using a new test image, we see how robust our method is applied to a different (and most likely more realistic) image. The new image measures 1296 x 966 in size, compared to out old image size of 449 x 321. The result is that out image runs slower, but not linearly so.



Figure 1 I have used a different color table here (Stern Special) because the black and white image was too dim. The fiducials are staggered and can be seen extending off the solar limbs.

# 2 How Slow is Slow

Table 1 lays out where our code takes the most time. Part of the process of making the code faster will be looking at which routines are called sparsely but still consume a lot of computing time, like `sort()`, for example.

# 3 Code Flow Chart

Table 1. Time (Total elapsed: 0.23712516 s)

| Routine | Times Called | Time Taken | Time Taken | A Number |
|---|---|---|---|---|
| SHIFT | 14 | 0.02763 | 0.02763 | 1 |
| CONVOL | 1 | 0.02647 | 0.02647 | 1 |
| SORT | 64 | 0.02610 | 0.02610 | 1 |
| SMOOTH | 2 | 0.01612 | 0.01612 | 1 |
| HISTOGRAM | 2 | 0.00705 | 0.00705 | 1 |
| LABEL_REGION | 2 | 0.00684 | 0.00684 | 1 |
| ERODE | 2 | 0.00428 | 0.00428 | 1 |
| TOTAL | 141 | 0.00425 | 0.00425 | 1 |
| FLTARR | 19 | 0.00374 | 0.00374 | 1 |
| DILATE | 2 | 0.00329 | 0.00329 | 1 |
| FLOAT | 121 | 0.00247 | 0.00247 | 1 |
| RESOLVE_ROUTINE | 1 | 0.00164 | 0.00164 | 1 |
| WHERE | 31 | 0.00159 | 0.00159 | 1 |
| CREATE_STRUCT | 13 | 0.00094 | 0.00094 | 1 |
| MAX | 6 | 0.00091 | 0.00091 | 1 |
| ISA | 9 | 0.00046 | 0.00046 | 1 |
| REBIN | 2 | 0.00046 | 0.00046 | 1 |
| MESSAGE | 1 | 0.00042 | 0.00042 | 1 |
| BYTARR | 10 | 0.00019 | 0.00019 | 1 |
| ROTATE | 1 | 0.00010 | 0.00010 | 1 |
| READF | 12 | 0.00004 | 0.00004 | 1 |
| FIX | 21 | 0.00004 | 0.00004 | 1 |
| FINITE | 12 | 0.00003 | 0.00003 | 1 |
| STRTRIM | 58 | 0.00003 | 0.00003 | 1 |
| SCOPE_VARFETCH | 14 | 0.00003 | 0.00003 | 1 |
| ON_ERROR | 70 | 0.00003 | 0.00003 | 1 |
| FILE_LINES | 1 | 0.00003 | 0.00003 | 1 |
| INDGEN | 5 | 0.00003 | 0.00003 | 1 |
| GETTOK | 2 | 0.00002 | 0.00002 | 0 |
| STRTOK | 12 | 0.00002 | 0.00002 | 1 |
| REPLICATE | 8 | 0.00002 | 0.00002 | 1 |
| PROFILER | 1 | 0.00002 | 0.00002 | 1 |
| STRMID | 34 | 0.00001 | 0.00001 | 1 |
| OPENR | 1 | 0.00001 | 0.00001 | 1 |
| DOUBLE | 12 | 0.00001 | 0.00001 | 1 |
| DEFSYSV | 2 | 0.00001 | 0.00001 | 1 |
| SQRT | 63 | 0.00001 | 0.00001 | 1 |
| STRCOMPRESS | 14 | 0.00001 | 0.00001 | 1 |
| PTR_NEW | 2 | 0.00001 | 0.00001 | 1 |
| PRINT | 1 | 0.00001 | 0.00001 | 1 |
| REFORM | 33 | 0.00001 | 0.00001 | 1 |
| N_PARAMS | 38 | 0.00001 | 0.00001 | 1 |
| ARRAY_EQUAL | 5 | 0.00001 | 0.00001 | 1 |
| MIN | 7 | 0.00001 | 0.00001 | 1 |
| STRING | 14 | 0.00001 | 0.00001 | 1 |
| FREE_LUN | 1 | 0.00001 | 0.00001 | 1 |
| STRLEN | 36 | 0.00001 | 0.00001 | 1 |
| CATCH | 10 | 0.00001 | 0.00001 | 1 |
| ABS | 13 | 0.00000 | 0.00000 | 1 |
| SKIP_LUN | 1 | 0.00000 | 0.00000 | 1 |
| PRODUCT | 5 | 0.00000 | 0.00000 | 1 |
| BYTE | 4 | 0.00000 | 0.00000 | 1 |
| SYSTIME | 2 | 0.00000 | 0.00000 | 1 |
| MAKE_ARRAY | 2 | 0.00000 | 0.00000 | 1 |
| TAG_NAMES | 1 | 0.00000 | 0.00000 | 1 |
| PTR_FREE | 1 | 0.00000 | 0.00000 | 1 |
| TRANSPOSE | 1 | 0.00000 | 0.00000 | 1 |
| FINDGEN | 2 | 0.00000 | 0.00000 | 1 |
| STRPOS | 3 | 0.00000 | 0.00000 | 1 |
| PTRARR | 1 | 0.00000 | 0.00000 | 1 |
| STRUPCASE | 1 | 0.00000 | 0.00000 | 1 |
| INTARR | 1 | 0.00000 | 0.00000 | 1 |

**Prepare parameters**

- Load paramater table
- Sort image and eliminate brightest .1% pixels
- Calculate peaks in 2nd derivative to return thresholds
- Associate suns to region numbers defined by maxima binning

**Prepare image for limb fitting**

- Mask image above thresholds to find every sun-shaped entity
- Count number of pixels
  - > 75% of typical number → Count as whole sun
  - < 75% of typical number → Mark as partial sun
- Crop remaining whole suns

**Fitting with limb strips**

- Extract strips of each cropped image
- Create limb strips of data according to where strips exceed threshold
- Apply linear fit to limb strips to find where limb crosses threshold
- Average midpoints of chord pairs to return center

**Returning Fiducial Positions**

- Load limb-fitted centers
- Cross correlate image with fiducial-shaped kernel
- Isolate local maxima of cross correlated image
- For each local maxima, fit parabola to adjacent 2 pixels in each axis to return fitted fiducial peak

Figure 2

## 4 The Dreaded Nested For Loop

This is taken straight from Albert's C++ code:

```
1  for (int m = 1; m < correlation.rows−1; m++)
2      {
3          for (int n = 1; n < correlation.cols−1; n++)
4          {
5              thisValue = correlation.at<float>(m,n);
6              if(thisValue > threshold)
7              {
8                  //Checks if cross correlated pixel is higher than adjacent pixels
9                  if((thisValue > correlation.at<float>(m, n + 1)) &
10                    (thisValue > correlation.at<float>(m, n − 1)) &
11                    (thisValue > correlation.at<float>(m + 1, n)) &
12                    (thisValue > correlation.at<float>(m − 1, n)))
13                 {
14                     redundant = false;
15                     for (unsigned int k = 0; k < pixelFiducials.size(); k++)
16                     {
17                         // Checks if previous fiducial correlation values are within 2 fiducial lengths of each other. If so, use
                               the one with a higher correlation value
18                         if (abs(pixelFiducials[k].y − m) < fiducialLength∗2 &&
19                             abs(pixelFiducials[k].x − n) < fiducialLength∗2)
20                         {
21                             redundant = true;
22                             thatValue = correlation.at<float>((int) pixelFiducials[k].y,(int) pixelFiducials[k].x);
23                             Choose the "fiducial" with a higher correlation value
24                             if ( thisValue > thatValue)
25                             {
26                                 pixelFiducials[k] = cv::Point2f(n,m);
27                             }
28                             // Break out of this because there should only be one instance of this per run
29                             break;
30                         }
31                     }
32                     // Regardless of whether or not the fiducial was replaced, break out of the loop
33                     if (redundant == true)
34                         continue;
35
36                     // If we're short a few entries for fiducials, extend the array
37                     if ( (int) pixelFiducials.size() < numFiducials)
38                     {
39                         pixelFiducials.add(n, m);
40                     }
41                     else
42                     {
43                         // Dealing with too many fiducials
44                         minValue = std::numeric_limits<float>::infinity();;
45                         minIndex = −1;
46                         for (int k = 0; k < numFiducials; k++)
47                         {
48                             if (correlation.at<float>((int) pixelFiducials[k].y,(int) pixelFiducials[k].x)
49                                 < minValue)
50                             {
51                                 minIndex = k;
52                                 minValue = correlation.at<float>((int) pixelFiducials[k].y,(int) pixelFiducials[k].x);
53                             }
54                         }
55                         if (thisValue > minValue)
56                         {
57                             pixelFiducials[minIndex] = cv::Point2f(n, m);
58                         }
59                     }
60                 }
61             }
62         }
63     }
```
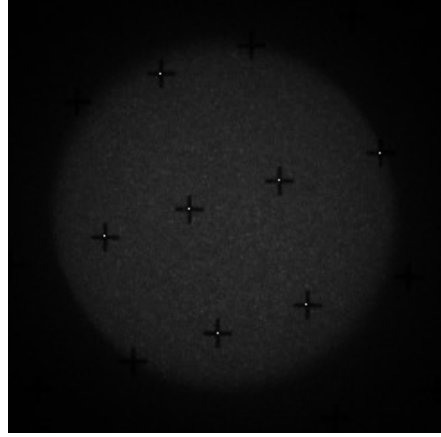
Table 2. Comparison of Fiducial Positions

| Fiducial Number | Albert's X | My X | Albert's Y | My Y |
|---|---|---|---|---|
| 0 | 674.6796 | N/A | 151.0038 | N/A |
| 1 | 796.3074 | N/A | 195.0324 | N/A |
| 2 | 740.4443 | 741.185 | 210.6342 | 211.289 |
| 3 | 690.2598 | 690.985 | 226.1973 | 226.961 |
| 4 | 643.4235 | 644.227 | 241.8869 | 242.636 |
| 5 | 755.8672 | 756.764 | 279.6622 | 280.443 |
| 6 | 706.0065 | 706.809 | 295.3022 | 295.957 |

# 5 Comparison to Albert's Code
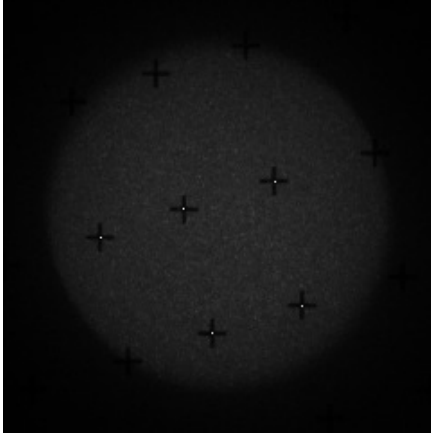


(a) The fiducials I find



(b) The fiducials Albert finds

Figure 3 My code can't pick up two fiducials due to one or many of the following factors: different kernel, different convolution method, different threshold.

In Table 2 The fiducial positions are typically within 1 pixel of Albert's calculated positions, which is pretty good.

# 6 Partial Sun Checking

We're motivated to keep some center data regardless of how cut off a sun may be. To do this, we must quantify the poorness of the fit as more sun is cut off. Figures 6 and 7 aim to quantify the worsening of the evaluated centers. We start by lining up the edge of the image to the solar limb then cropping in 10 columns.

The whole solar image consists of 26,597 pixels above the threshold.

# 7 Glaring Problems

I was having trouble with proper thresholding but it was alleviated with increasing the smoothing parameter. *Go parameter block!*

Our attempts to identify partial suns has evolved from:

1. Determining if center of sun is within certain distance of border

2. Count pixels on very border of image, if 6 consecutive pixels found, mark nearest sun as partial

Table 3. Side Crop Test for Figure 4

| Amount Cropped from Limb (pixels) | $x_{\mathrm{True}} - x_{\mathrm{Cropped}}$ | $y_{\mathrm{True}} - y_{\mathrm{Cropped}}$ |
|---|---|---|
| 10 | -1.17771 | -0.0108643 |
| 20 | -4.07970 | -0.0376663 |
| 30 | -7.63260 | -0.0522766 |
| 40 | -11.7287 | -0.0585175 |
| 50 | -16.2043 | -0.0185776 |
| 60 | -20.9117 | -0.0872879 |
| 70 | -25.7588 | -0.277687 |
| 80 | -30.8586 | -0.321724 |
| 90 | -36.1318 | -0.318489 |

Table 4. Corner Crop Test for Figure 5

| Amount Cropped from Limb (pixels) | $x_{\mathrm{True}} - x_{\mathrm{Cropped}}$ | $y_{\mathrm{True}} - y_{\mathrm{Cropped}}$ |
|---|---|---|
| 10 | -1.17902 | -1.22132 |
| 20 | -4.23825 | -4.28215 |
| 30 | -8.41805 | -8.49775 |
| 40 | -13.2540 | -13.3160 |
| 50 | -18.2548 | -18.0202 |
| 60 | -23.0267 | -22.9181 |
| 70 | -27.5755 | -27.8987 |
| 80 | -32.1102 | -32.4790 |
| 90 | -36.6139 | -37.0980 |

Table 5. Side Crop Test for Figure 6

| Amount Cropped from Limb (pixels) | $x_{\mathrm{True}} - x_{\mathrm{Cropped}}$ | $y_{\mathrm{True}} - y_{\mathrm{Cropped}}$ | $N_{\mathrm{pixels}}$ above threshold | Percentage of Total Pixels |
|---|---|---|---|---|
| 5 | -1.17771 | -0.0108643 | 26250 | 98.6953 |
| 10 | -2.52845 | -0.0205002 | 25838 | 97.1463 |
| 15 | -4.07970 | -0.0376663 | 25353 | 95.3228 |
| 20 | -5.82758 | -0.0490799 | 24795 | 93.2248 |
| 25 | -7.63260 | -0.0522766 | 24208 | 91.0178 |
| 30 | -9.60011 | -0.0380630 | 23557 | 88.5701 |

Table 6. Corner Crop Test for Figure 7

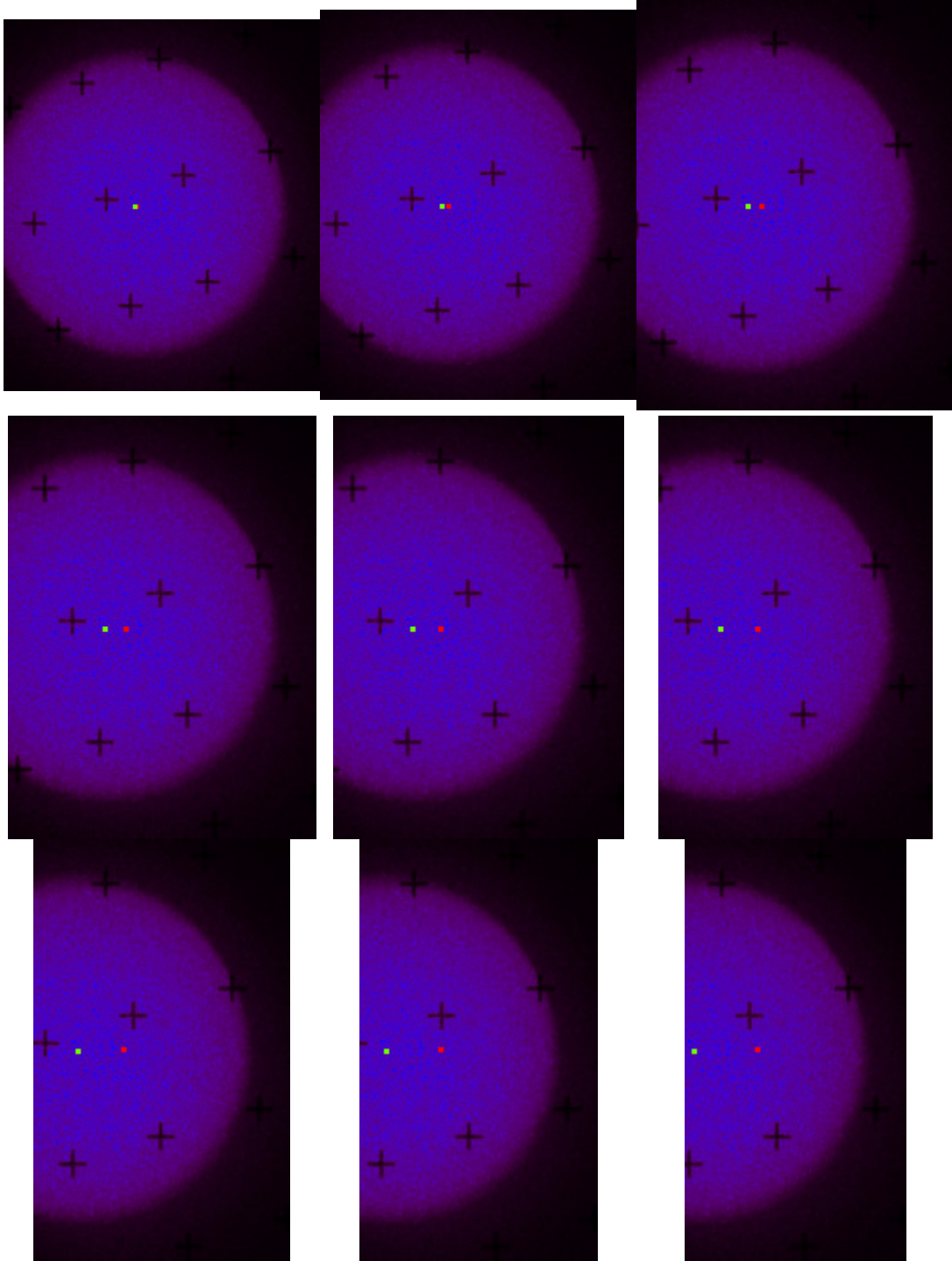| Amount Cropped from Limb (pixels) | $x_{\mathrm{True}} - x_{\mathrm{Cropped}}$ | $y_{\mathrm{True}} - y_{\mathrm{Cropped}}$ | $N_{\mathrm{pixels}}$ above threshold | Percentage of Total Pixels |
|---|---|---|---|---|
| 5 | -1.17902 | -1.22132 | 25896 | 97.3644 |
| 10 | -2.58559 | -2.60790 | 25079 | 94.2926 |
| 15 | -4.23825 | -4.28215 | 24113 | 90.6606 |
| 20 | -6.22114 | -6.27864 | 22994 | 86.4534 |
| 25 | -8.41805 | -8.49775 | 21805 | 81.9829 |
| 30 | -10.8070 | -10.8650 | 20544 | 77.2418 |

Figure 4 The green pixel is the image's true center and the red pixel is the center of the cropped image. The images are cropped 10 columns at a time.

3. Count number of solar pixels above threshold, if below a certain amount, marked as partial sun

The problem with the second method is that the shape of our mask is designed such that the bottom two corners will always be dark so scanning in a border-pattern will return ill results. We've opted to return to the first method, masking the sun, regardless of shape, and measuring the distance between the
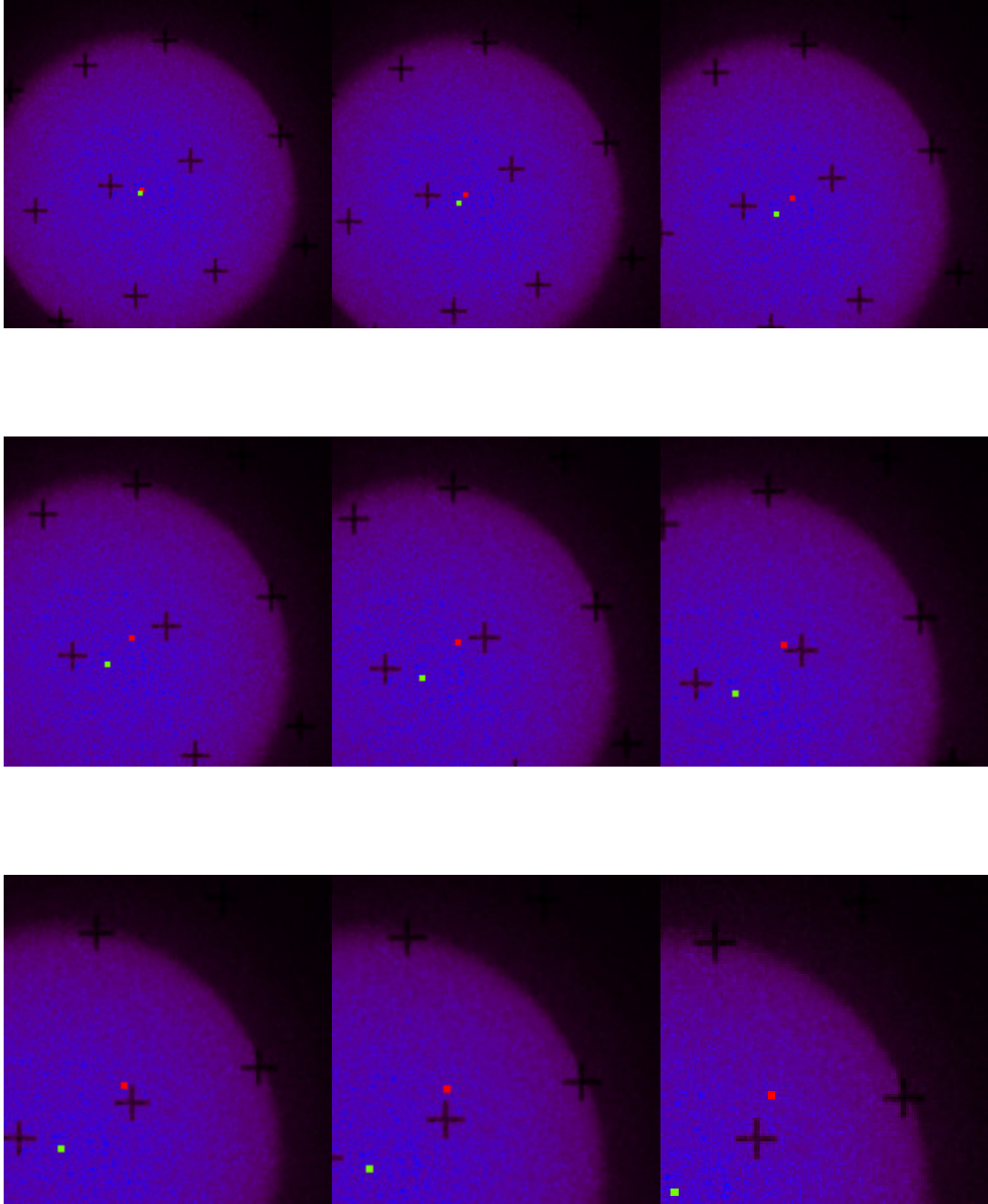
Figure 5 The green pixel is the image's true center and the red pixel is the center of the cropped image. The images are cropped 10 columns at a time.

mask center and the edge of the image.

A possible approach is to use the mask in Figure 8a and check the distance to the closest non-0 pixel; the problem with this method are the numerous `sqrt()` calls to each pixel on the border of our image.
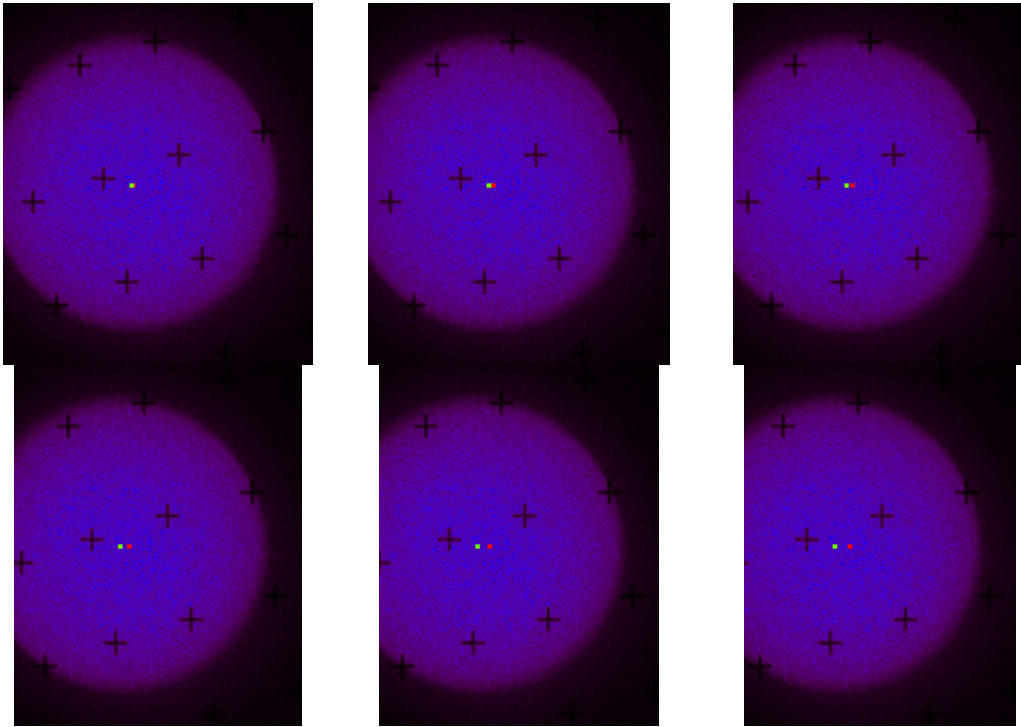
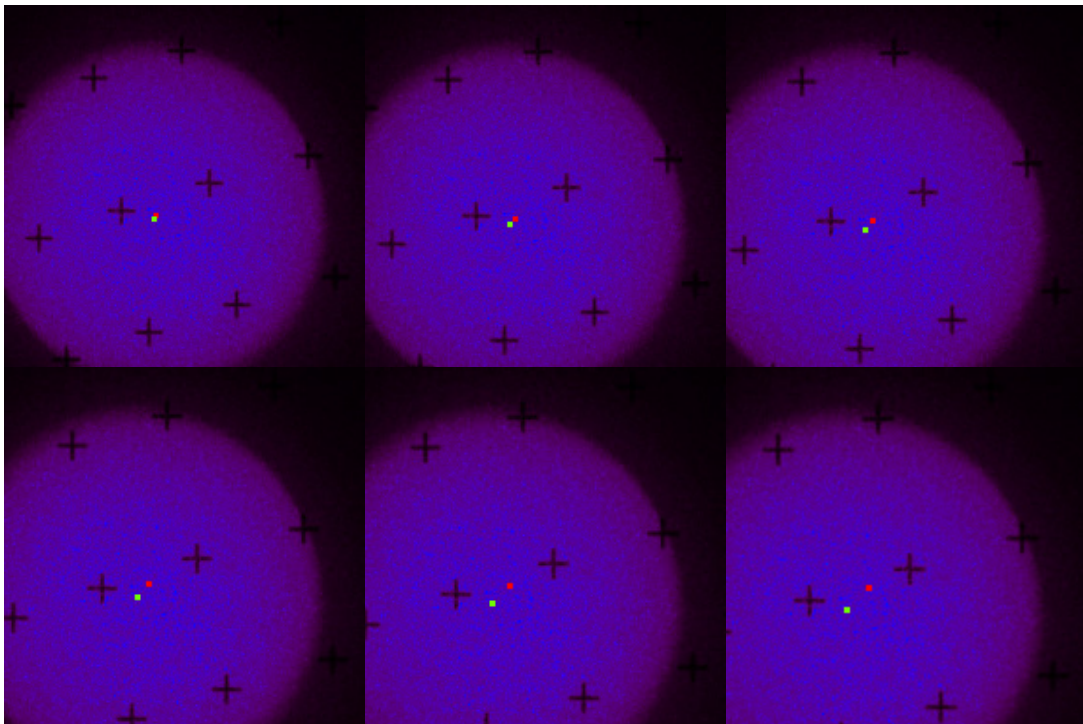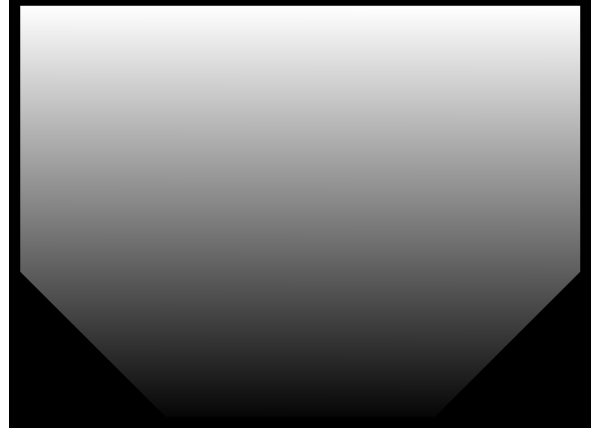Figure 6 The sun is being cropped off 5 columns at a time, opposed to the earlier Figures' 10.



Figure 7

(a) What our mask should look like - side of black triangle is 1/4 of image width



(b) A proposed mask that looks within a certain distance form the border.

Figure 8 There *used* to be a problem with the mask in 8b; it used an erode function (*which takes an ungodly amount of time*) to shrink the mask used in 8a. Now I pad the image, shift it in all 4 directions, then multiply the masks together.

Table 7.   Comparison of Center Positions

| Method | X Position | Y Position |
| --- | --- | --- |
| Mine | 710.811 | 230.695 |
| Albert's | 709.7835 | 230.1023 |