The ALPHA Experiment

Blob Analysis of Catching Trap Ions and Antiprotons

Ian Schoch

Supervisor: Phil Rubin

Mentor: William Bertsche

June 18 – August 10, 2012

Abstract

This technical note describes the MCP/Phosphor/CCD creation process of images of antiprotons and ions in the vacuum of the ALPHA catching trap. Differentiating noise from signal is a key problem that must be solved by a blobbing algorithm. Two different blobbing algorithms were developed in Matlab R2011b: one that is well suited for counting a small number (<100) of ion blobs, and another that gives a rough estimate of the number of antiproton blobs. Both algorithms rely heavily on Canny edge detection, but both also use other image processing techniques, such as defining an image threshold. Next, a description of the blobbing algorithms is given, and results are discussed. Finally, suggestions are made about possible improvements for this project. The commented code for each blobbing algorithm is given in the Appendix.
Image Creation

The ALPHA experiment uses a MCP/Phosphor/CCD system to capture images of the various particle species as shown in Figure 1. An MCP is a multichannel plate, which is an array of miniature electron multipliers. For each particle that strikes the MCP, a proportional number of electrons are emitted and excite phosphorous atoms in a phosphor screen behind the MCP. The phosphorous atoms in the screen decay producing visible light, which is reflected off a mirror and captured in a CCD camera. The camera digitizes the light and a .tiff image file is produced. An image of vacuum ions captured by the MCP/Phosphor/CCD system is shown in Figure 2. This method of image capture has proved to be a very effective particle imaging technique, and a powerful diagnostic tool. At ALPHA, images are produced with the MCP/Phosphor/CCD system to view plasmas and to diagnose vacuum conditions in the catching trap.

![Figure 1: A particle strikes the MCP that creates an avalanche of electrons, which in turn excite phosphorous atoms, which then decay producing light that is reflected off of a mirror and captured with a CCD camera.]

Obstacles

There are several obstacles to overcome when developing a blobbing algorithm on the images produced from the MCP/Phosphor/CCD system. Regardless of the image system used, there will always be noise in the image produced from various sources. For example, a stray cosmic ray, hot pixels in the camera, and defects in the materials composing the MCP/Phosphor/CCD system can all be causes of noise. In the images analyzed in this report, there existed horizontal noise lines in each image as well as hotter noise bordering the left of the image. A major obstacle is differentiating noise, or background fluctuations in pixel intensity, from signal, or particle hits. In Figure 2 there are a total of 23 ion hits, most of which are easily recognized as turquoise splotches in the image. However, there are two faint hits that can be difficult to spot by eye: one near the center of the image and one on the bottom and further left than the other ions. Figure 3 shows the results of the ion blobbing algorithm on the image in Figure 2.

1 Butler, 27-28
2 Butler, 28
Figure 2: An image, created from the MCP/Phosphor/CCD system, of a small number of ions in the catching trap.

Figure 3: Blobbing on Figure 2 by the ion blobbing algorithm. All 23 ion hits were correctly blobbed.
As a first try at creating a blobbing algorithm, one might try creating an image threshold based on the mean and standard deviation pixel intensity of the image. The results of using a threshold of 3.5 and 4 standard deviations above the mean pixel intensity are shown in Figure 4 and Figure 5 respectively. Both threshold levels have their drawbacks. Using a Threshold at 3.5 standard deviations above the mean allows noise on the bottom left of the image to be counted as ion hits, while using a threshold at 4 standard deviations above the mean misses the two faint blobs in the image. Also, a threshold level that may suit one image may not be a good fit for another image, so using an image threshold based on mean and standard deviation of the image pixel intensity cannot easily be generalized. One can however set a high threshold (5 or 6 standard deviations above the mean) and have a blobbing algorithm that is very roughly accurate and systematically undercounting the true number of ion hits. While differentiating noise from signal using pixel intensity alone may not be effective, differentiating only by pixel structure also has its drawbacks.

Figure 4: Threshold set to 3.5 standard deviations above the mean pixel intensity. With careful examination, a scattering of red pixels bordering the left side of the image can be seen. These spots are high intensity noise in the image.
Another challenge to creating a blobbing algorithm is that some of the particle hits show up as only a few pixels in the image, making identifying blobs by structure quite challenging. In a small neighborhood of pixels surrounding a high intensity pixel, noisy pixels and faint particle hits can look remarkably similar making differentiation that much harder. Also, due to the small and faint ions, it is difficult to remove background from the image without removing some of the signal simultaneously. For example, the weiner2 filter function was used on the images and did a very good job removing noise, but it would often obscure faint particle hits or cause them to disappear completely from the image as shown in Figure 6 and Figure 7. Other methods of background removal involving the Matlab imopen and medfilt commands were similarly flawed.

A further complication is added in the case of antiproton images, where the antiprotons often leave behind one or two tails of pion decays, which show up as bright lines on the image as shown in Figure 8. Some of these tails are hotter than other antiproton hits, further complicating matters. With these obstacles in mind, a blobbing algorithm that differentiated both on pixel intensity and pixel structure was sought after.
Figure 6: Original image

Figure 7: Image after filtering with a weiner2 filter. Background is removed at the cost of several faint blobs.
Figure 8: Antiproton image. Notice the bigger area hits as compared to the ion hits. Also the light tails associated with some of the hits further complicate blobbing.

Two Blobbing Algorithms

After trying various functions from the Matlab Image processing toolbox, two blobbing algorithms were developed that best blobbed the data. The blobbing algorithm designed for ions was tested on a set of 14 different ion pictures. The actual number of ions was counted by eye and compared with the number counted by each of the trial algorithms. By trial and error with implementing various functions on the blobbing algorithm, eventually an algorithm was created that correctly identified the blobs in all 14 images.

As previously mentioned, the blobbing algorithms both rely heavily on Canny edge detection. Basically each algorithm attempts to set the optimal threshold for edge detection, and then proceeds to prune the resultant image of noise using various image processing techniques. The code for both of the blobbing algorithms with comments describing exactly how they work can be found in the Appendix.

Figure 9 shows the results of the ion blobbing algorithm on a simple image. The blobbing algorithm correctly counts the exact number of blobs in the 14 trial ion images. The main component of the algorithm iterates the threshold of a Canny edge detector to an optimal value based on the extent of each of the objects produced from the edge detection. Extent is the ratio of the pixel area of the object to the area of the smallest box surrounding that blob. Further noise removal is accomplished by setting a threshold for the remaining blobs. Finally, blobs with a small extent are more closely examined and an individual threshold is set to differentiate multiple blobs that are very close together and were counted as a single blob. As you can see, the ion blobbing algorithm is fairly accurate in finding ion hits, sometimes even finding blobs that are hard to see by the naked eye.
The second blobbing algorithm roughly blobs the antiprotons in an image. As previously mentioned, there is a further difficulty in blobbing the antiproton images as they have bright tails that should not be counted as signal. However, by tweaking the ion blobbing algorithm, an antiproton algorithm was created that roughly estimates the number of antiproton hits in the image. The same Canny edge detection was used, but a new feature was added. The algorithm attempts to identify the tails in the image that are originally counted as antiprotons and exclude them from the total count. Figure 10 shows the results of the antiproton blobbing algorithm on two different images. As you can see, the antiproton blobbing algorithm finds most of the blobs, but sometimes mistakes a bright tail for a blob, or misses some very faint hits.

To prevent excessive computing time, both blobbing algorithms stop after 15 iterations of the Canny edge detector threshold. Thus, the blobbing algorithms can be automatically used on any image, even images they were not designed to handle, and not crash or take forever to compute. The typical run time of the algorithm on an image is between 0.5 and 3.0 seconds.
Figure 10: Results of the antiproton blobbing algorithm on a couple images. 82 and 51, respectively, is the determined antiproton count in each of the images.
Improvements and Further Work

Due to time constraints and lack of knowledge on both Matlab and image processing, the two blobbing algorithms, imperfect as they are, are the end result of this project. Further work might focus on improving these algorithms by tweaking certain parameters, adding or removing a filtering component, or developing an entirely new blobbing algorithm from scratch. Despite some effort, utilizing a number of “blank” images taken just before the plasma image to eliminate background was not successful in this project. However, further work might focus on finding a way to use “blank” images to filter out background. Using a 2-D Gaussian function to fit to blobs could be another possible improvement. A 2-D Gaussian fitter was found online, but proved difficult to implement into the blobbing algorithm, as it was time consuming. However, as determined by an image processing research group, studying turbulence of edge plasma in a nuclear fusion reactor, a 2-D Gaussian fitting algorithm could be very effective. Perhaps trying to fit all of the local maximums in an image above a certain threshold to a Gaussian and eliminating all those that had a poor fit could be an effective blobbing algorithm. Also, further work could be done on tweaking the antiproton blobbing algorithm to better eliminate the tails. Further work might also pursue combining the two blobbing algorithms to discriminate ions from antiprotons.

Conclusion

The two blobbing algorithms presented in this work are by no means perfect or comprehensive. While they do a decent job of identifying a small number of ions and antiprotons respectively, there is much room for improvement. The better blobbing algorithms were shown to utilize multiple techniques, and finding the best blobbing algorithms will probably involve applying the right techniques to the image in the correct order. An ideal blobbing algorithm would be able to automatically detect each of the types of particle species in any image along with a confidence level.

3 Mineault, Patrick
4 Love, Nichole
References


Appendix

Ion Blobbing Algorithm

```matlab
% Ian Schoch, August 7th 2012
% Counts all the ions in a .tiff image
% Uncomment the last 3 lines to display the image with blobs outlined.
% Close all open image tools, close all figures, clear variables.
imtool close all
close all;
clear all;
% change directory to specified directory of .tiff file.
cd 'C:\Users\potpie\Documents\MATLAB';
files=dir('*.tif');
% change following name to specified .tif file.
name='';
% This offset causes a to have "normal" intensity values.
% imtool(a,[],'Colormap',hot); Uncomment to display original image.
[junk threshold] = edge(a, 'canny'); %Guess starting threshold
fudgeFactorcanny=.75; %sigma fudge factor for Canny prefiltering
x=-1; % edge detector iteration counter
stop=false; % stop determines when to end the edge detector iteration
% 15 iteration limit prevents iterations from taking forever on nonblob
% images or the threshold from exceeding 1, causing program to crash.
while (stop==false && x<15)
    x=x+1; %This initializes x to zero on the first time through the loop
    % Create a Binary image of the edges. Start with fudgefactor=2;
    % decrease the fudge factor by .1 each iteration.
    BWs = edge(a,'canny',(2-x*.1)*threshold,sqrt(2)*fudgeFactorcanny);
    BWf=imfill(BWs,'holes'); %fills in holes in the binary image
    cc2=bwconncomp(BWf); %identifies each set of connected pixels as a distinct ob-
   ject
    stats2 = regionprops(cc2,a,'Extent'); % Get the Extent of each object in the
    % image.
    extents=[stats2.Extent]; % Store the extents of each object in an array
    sizes=cc2.NumObjects; %The number of objects
    m=1;
    %If any of the objects have an extent smaller than .34, stop iterating
```

The point: Only detect dense little blobs with high extent: ignore lines created from noise with low extent.

```matlab
while (m<=sizes && stop==false)
    if (extents(m) < .34) %.34 is empirically a good number
        stop=true;
    end
    m=m+1;
end
```

```matlab
proceed=false;
```

```matlab
% If the initial guess was too low, we need to iterate upwards.
if (x==0)
    This time we start with a mess of lines and iterate until we have a fairly clean image.
    ```matlab
    while(proceed==false && x<15)
        proceed=true;
        x=x+1;
        Similar to previous chunk of code
        BWs = edge(a,'canny',(2+x*.1)*threshold,sqrt(2)*fudgeFactorcanny);
        BWf=imfill(BWs,'holes');
        cc2=bwconncomp(BWf);
        stats2 = regionprops(cc2,a,'Extent');
        extents=[stats2.Extent];
        sizes=cc2.NumObjects;
        m=1;
        % If ANY of the objects have a small extent, we cannot proceed;
        % we keep iterating until all objects have extent>.34
        while (m<=sizes && proceed==true)
            if (extents(m) < .34)
                proceed=false;
            end
            m=m+1;
        end
end
```

```matlab
% If the appropriate edge detector threshold isn't found in 15
% iterations, we give up and move on to the next image.
if(x==15)
    disp('took more than 15 iterations. program done.')
    % imtool(a,[],'Colormap',hot);
    return;
end
```

```matlab
if (proceed==false)
    % If we had to iterate upwards, then x is already optimal.
    % If we had to iterate downwards, then x is set to 1 more than is optimal.
    BWs = edge(a,'canny',(2-(x-1)*.1)*threshold,sqrt(2)*fudgeFactorcanny);
end
```

```matlab
% Fill in the holes to see all pixels within the blob for a better read for max intensity
BWf=imfill(BWs,'holes');
cc = bwconncomp(BWf); % identify each set of connected pixels as an object
% Get array containing the Maximum Intensity pixel value of each object
stats = regionprops(cc,a,'MaxIntensity');
amean=mean2(a); %mean of original image
astd=std2(a); %Standard deviation of original image
maxes=[stats.MaxIntensity]; %Store maximum intensity pixel value of each object in an array
```
idx = find(maxes > amean+2.5*astd); % Eliminate Objects classified as noise by thresholding
BW2 = ismember(labelmatrix(cc), idx); % Create Binary image of surviving objects.
% This part removes the filled in holes in BW2.
compl = imcomplement(BW2);
BW2s(compl) = 0;
% the especially long blobs are to checked to see if
% they are actually a conglomeration of multiple blobs
cc3 = bwconncomp(BW2s);
stats = regionprops(cc3, a, 'BoundingBox', 'Perimeter');
boxes = stats.BoundingBox;
perims = stats.Perimeter;
perimstd = std(perims);
perimean = mean(perims);
idt = find(perims > perimean + perimstd); % blobs with a relatively high perimeter are considered
BWtemp = ismember(labelmatrix(cc3), idt);
% The next chunk prepares the future analysis. It ensures that the
% bounding boxes only contain pixels from its blob and not other blobs
% (This part could fail if two or more blobs with high perimeter are right next
% to eachother).
BWtemp = imfill(BWtemp, 'holes');
comb = imadd(BWtemp, BW2);
b = a;
b(comb == 1) = 0;
sizers = size(idt, 2);
extra = 0;
% For each object that has a large perimeter,
for k = 1:sizers
    % Create a tiny image of the bounding box only surrounding the blob
    index = (idt(k) - 1)*4 + 1;
    x = floor(boxes(index));
    y = floor(boxes(index + 1));
    anew = b(y:y + boxes(index + 3), x:x + boxes(index + 2)); % Threshold the image.
    anewmean = mean2(anew);
    anewstd = std2(anew);
    foreground = (anew > (max([(anewmean + anewstd) (amean + 3.5*astd)])));
    cc10 = bwconncomp(foreground);
    % imtool(foreground, [], 'Colormap', hot);
    % 1-The number of connected objects is the number of extra blobs in
    % the object.
    extra = extra + cc10.NumObjects - 1;
    if (cc10.NumObjects == 0)
        extra = extra + 1;
    end
end
% The total number of blobs is the number of blobs + the number of
% extra blobs.
blobtotal = cc3.NumObjects + extra;
disp(blobtotal);
% For Diagnostic: Create an image of the original with outlined blobs.
outlined = a;
outlined(BWs) = 0;
imtool(outlined, [], 'Colormap', hot);
% imtool(a, [], 'Colormap', hot);
Antiproton Blobbing Algorithm

% Ian Schoch, August 9th 2012
% Counts all the ions in every .tif image in a folder Has fairly good accuracy, but sometimes counts touching blobs as a single blob.
% Uncomment the last 3 lines to display the image with blobs outlined.
% Close all open image tools, close all figures, clear variables.
imtool close all
close all;
clear all;
% change the following directory to specified .tif file.
cd 'C:\Users\potpie\Documents\MATLAB';
% change the following to the specified file name.
name='';
a=imread (name);
a=a-(2^15-1); % This offset causes a to have "normal" intensity values.
% amean=mean2(a);
% astd=std2(a);
% foreground=(a>(amean+4*astd));
imtool(foreground,[],'Colormap','jet');
imtool(a,[],'Colormap','jet'); % Uncomment to display original image.
% amean=mean2(a);
% astd=std2(a);
% fore=(a<amean+2*astd);
% a(fore)=amean;
imtool(a,[],'Colormap','jet');
[junk threshold] = edge(a,'canny'); % Guess starting threshold
fudgeFactorcanny=.75; % sigma fudge factor for Canny prefiltering
x=-1; % edge detector iteration counter
excount=0;
sizes=0;
% 15 iteration limit prevents iterations from taking forever on nonblob images or the threshold from exceeding 1, causing program to crash.
while(excount<.05*sizes && x<15)
excount=0;
x=x+1; % This initializes x to zero on the first time through the loop
% Create a Binary image of the edges. Start with fudgefactor=2; decrease the fudge factor by .1 each iteration.
BW = edge(a,'canny', (2.0-x*.1)*threshold, sqrt(2)*fudgeFactorcanny);
BWf=imfill(BW,[],'holes'); % fills in holes in the binary image
% imtool(BWf,[],'Colormap','hot');
cc2=bwconncomp(BWf); % identifies each set of connected pixels as a distinct object
stats2 = regionprops(cc2,a,'Extent'); % Get the Extent of each object in the image.
% Extent is the ratio of the total area of the object and the smallest bounding rectangle of the object.
extents=[stats2.Extent]; % Store the extents of each object in an array
sizes=cc2.NumObjects; % The number of objects
m=1;
% If any of the objects have an extent smaller than .34, stop iterating
% The point: Only detect dense little blobs with high extent: ignore lines created from noise with low extent.
while (m<sizes && excount<.05*sizes)
    if (extents(m) < .34) %.34 is empirically a good number
        excount=excount+1;
    end
    m=m+1;
end

% imtool(a(m),[],'Colormap','jet');
% imtool(BWf(m),[],'Colormap','hot');
% imtool(cc2, m, 'ColorMap', 'jet');
% imtool(BWf(m),[],'Colormap','hot');
end
m=m+1;
end
end
sizes=0;
temp=excount*.7;
down=x;
x=0;

%  This time we start with a mess of lines and iterate until we have a fairly clean image
while(temp<=excount*.7 && x<15)
    temp=excount;
    x=x+1;
    % Similar to previous chunk of code
    BWs = edge(a,'canny',2.0-down*.1+x*.1)*threshold,sqrt(2)*fudgeFactorcanny);
    BWf=imfill(BWs,'holes');
    % imtool(BWf,[],'Colormap',hot);
    cc2=bwconncomp(BWf);
    stats2 = regionprops(cc2,a,'Extent');
    extents=[stats2.Extent];
    sizes=cc2.NumObjects;
    m=1;
    % If ANY of the objects have a small extent, we cannot proceed;
    % we keep iterating until all objects have extent>.34
    while (m<=sizes)
        if (extents(m) < .34 )
            excount=excount+1;
        end
        m=m+1;
    end
end
if(x==15)
    disp('took more than 15 iterations. program done.')
    imtool(a,[],'Colormap',jet);
    return;
end
imtool(BWf,[],'Colormap',hot);
% Fill in the holes to see all pixels within the blob for a better read for max intensity
cc = bwconncomp(BWf); % identify each set of connected pixels as an object
stats = regionprops(cc,a,'MaxIntensity');
amean=mean2(a); %mean of original image
astd=std2(a); %Standard deviation of original image
maxes=[stats.MaxIntensity]; %Store maximum intensity pixel value of each object in an array
idx = find(maxes > amean+4*astd); % Eliminate Objects classified as noise by thresholding
BW2 = ismember(labelmatrix(cc), idx); % Create Binary image of surviving objects.
% This part removes the filled in holes in BW2.
compl=imcomplement(BW2);
BW2=compl==0;
BW2=bwareaopen(BW2,3);
cc3 = bwconncomp(BW2);
% imtool(BW2,[],'Colormap',jet);
total=cc3.NumObjects;
% disp(total);
BWf = imfill(BWs, 'holes');
cc3 = bwconncomp(BWf);
stats = regionprops(cc3, a, 'Perimeter');
perims = [stats.Perimeter];
perimstd = std(perims);
perimean = mean(perims);
idp = find(perims > perimean + 0 * perimstd); % blobs with a relatively high perimeter are considered
longsize = size(idp);
BWtemp = ismember(labelmatrix(cc3), idp);
cc = bwconncomp(BWtemp); % identify each set of connected pixels as an object
% Get array containing the Maximum Intensity pixel value of each object
stats = regionprops(cc, a, 'MeanIntensity');
means = [stats.MeanIntensity]; % Store maximum intensity pixel value of each object in an array
idx = find(means > amean + 3 * astd); % Eliminate Objects classified as noise by thresholding
BW2 = ismember(labelmatrix(cc), idx); % Create Binary image of surviving objects.
% This part removes the filled in holes in BW2.
cc = bwconncomp(BW2);
numtails = longsize - cc.NumObjects;
numantiproton = total - numtails;
disp (numantiproton);
g = imabsdiff(BW2, BWtemp);
news = imabsdiff(g, BWf);
compl = imcomplement(news);
BWs(compl) = 0;
% outlined = a;
% outlined(BWs) = 0;
% imtool(outlined, [], 'Colormap', hot);
% imtool(a, [], 'Colormap', hot);