PH5170 - Cloud Chamber Project

Evelyn-Andreea Ester

October 21, 2020

1 Introduction

A cloud chamber is a particle physics detector where the environment contains a supersaturated vapor. When a charged particle moves through the vapor, it ionizes the atoms in the vapor. The ions then act as a catalyst to form "clouds", which condense the vapor and leave visible trails along the path of the particle.
This project will use a C++ program to analyze cloud chamber images from a diffusion cloud chamber which creates the supersaturated vapor by cooling alcohol vapor with a condenser at the bottom of the chamber. A light illuminates the edges of the chamber so that the tracks can be seen against the black background of the bottom of the chamber from light scattering off of the condensing vapor. The cloud chamber is sensitive to several kinds of ionizing radiation: weakly ionizing and strongly ionizing particles. The first category, weakly ionizing, comprises primarily beta particles from the decay of radioactive elements in the environment, and are characterized by narrow tracks. The second category, strongly ionizing, comprises primarily alpha particles from the decay of transuranic elements in the environment. These tracks deposit more energy per unit length than the weakly ionizing particles, and therefore appear to ionize more atoms in the vapor and appear much brighter. Alphas are also much less subject to deflection and the tracks are typically straighter.[0]

2 Pre-processing

Pre-processing is the name used for operations on images at the lowest level of abstraction —both input and output are intensity images. Such images are usually of the same kind as the original data captured by the sensor, with an intensity image usually represented by a matrix or matrices of brightness values.
Pre-processing is very useful in a variety of situations since it helps to suppress information irrelevant to the specific image processing or analysis task. Therefore, the aim of pre-processing is an improvement of the image data that suppresses undesired distortions or enhances some image features important for further processing. The technique used in this project is called 'median filtering' and will be discussed below.

2.1 Median filtering

In probability theory, the median divides the higher half of a probability distribution from the lower half. For a random variable x, the median M is the value for which the probability of the outcome $x < M$ is 0.5. The median of a finite list of real numbers is simply found by ordering the list and selecting the middle member. Lists are often constructed to be odd in length to secure uniqueness.

Median filtering is a non-linear smoothing method that reduces the blurring of edges [Tyan, 1981], in which the idea is to replace the current point in the image by the median of the brightnesses in its neighborhood. The median in the neighborhood is not affected by individual noise spikes and so median smoothing eliminates impulse noise quite well. Further, as median filtering does not blur edges much, it can be applied iteratively. Clearly, performing a sort on pixels within a (possibly large) rectangular window at every pixel position may become very expensive. A more efficient approach [Huang et al., 1979; Pitas and Venetsanopoulos, 1990] is to notice that as the window moves across a row by one column, the only change to its contents is to lose the leftmost column and replace it with a new right column. For a median window of m rows and n columns, $mn - 2m$ pixels are unchanged and do not need re-sorting. The algorithm is as follows:

1. Set $t = \frac{mn}{2}$. (If m and n are both odd, round t.)

2. Position the window at the beginning of a new row and sort its contents. Construct a histogram H of the window pixels, determine the median m and record n_m, the number of pixels with intensity less than or equal to m.

3. For each pixel p in the leftmost column of intensity p_g, perform

$$H[p_g] = H[p_g] - 1. \tag{1}$$

   If $p_g \leqslant m$, set

$$n_m = n_m - 1. \tag{2}$$

4. Move the window one column right. For each pixel p in the rightmost column of intensity p_g, perform

$$H[p_g] = H[p_g] + 1. \tag{3}$$

   If $p_g \leqslant m$, set

$$n_m = n_m + 1. \tag{4}$$

5. If $n_m = t$, go to step (8).

6. If $n_m > t$, go to step (7).
   Repeat

$$m = m + 1 \tag{5}$$
$$n_m = n_m + H[m], \tag{6}$$
$$\tag{7}$$

   until $n_m \geqslant t$. Go to step (8).

7. If we are at this step, we have $n_m > t$. Repeat

$$n_m = n_m - H[m] \tag{8}$$
$$m = m - 1, \tag{9}$$

   until $n_m \leqslant t$.

8. If the right hand column of the window is not at the right-hand edge of the image, go to step (3).

9. If the bottom row of the window is not at the bottom of the image, go to step (2).

An implementation of the median filtering algorithm is as follows:

```
1  bool ProcessImage::denoise_with_median_filter(int neighbours)
2  {
3      int window[9];
4      if(DEBUG_ProcessImage_MESSAGES)
5      std::cout << "denoise with median filter " <<  neighbours << " neighbours"
   << std::endl;
6      if((neighbours!=4)||(neighbours!=8)) return false;
7      if(neighbours==8)
8      { // 8 neighbours
9          for( int j=1 ; j < image.TellHeight()-1 ; j++)
10         {
11             for( int i=1 ; i < image.TellWidth()-1 ; i++)
12             {
13                 window[0] = image(i,j-1)->Red;
14                 window[1] = image(i,j+1)->Red;
15                 window[2] = image(i,j)->Red;
16                 window[3] = image(i-1,j)->Red;
17                 window[4] = image(i+1,j)->Red;
18                 window[5] = image(i-1,j-1)->Red;
19                 window[6] = image(i+1,j-1)->Red;
20                 window[7] = image(i-1,j+1)->Red;
21                 window[8] = image(i+1,j+1)->Red;
22                 //sort window array
23                 std::vector<int> myvector(window, window+9);
24                 std::sort(myvector.begin(), myvector.end());
25                 image(i,j)->Red = myvector[4];
26                 image(i,j)->Green = myvector[4];
27                 image(i,j)->Blue = myvector[4];
28                 image(i,j)->Alpha=0;
29             }
30         }
31         return true;
32     }
```

3 Segmentation

Image segmentation is one of the most important steps leading to the analysis of processed image data. Its main goal is to divide an image into parts that have a strong correlation

with objects or areas of the real world contained in the image. We may aim for complete segmentation, which results in a set of disjoint regions corresponding uniquely with objects in the input image, or for partial segmentation, in which regions do not correspond directly with image objects. A complete segmentation of an image R is a finite set of regions $R_1, ..., R_S$, such that

$$R = \bigcup_{i=1}^{S} R_i, \quad R_i \cap R_j = \emptyset, \quad i \neq j. \tag{10}$$

Gray-level thresholding is the simplest segmentation process. Many objects or image regions are characterized by constant reflectivity or light absorption of their surfaces; then a brightness constant or threshold can be determined to segment objects and background. Thresholding is computationally inexpensive and fast—it is the oldest segmentation method and is still widely used in simple applications; it can easily be performed in real time.

3.1 Thresholding

Here, a simple global approach can be used and the complete segmentation of an image into objects and background can be obtained. Such processing is context independent; no object-related model is used, and no knowledge about expected segmentation results contributes to the final segmentation. If partial segmentation is the goal, an image is divided into separate regions that are homogeneous with respect to a chosen property such as brightness, color, reflectivity, texture, etc.

3.2 Optimal thresholding and Otsu's algorithm

The algorithm finds an optimal threshold of an image by minimizing the within-class variance, using only the gray-level histogram of the image. Image segmentation consists on separating an image into regions or contours, that generally correspond to boundaries or objects on images. Usually, segmentation is made by identifying common properties or finding differences between regions. This implies that pixels are grouped into regions or classes that share some common property (such as color, intensity, etc.). Otsu's segmentation method is a global image thresholding algorithm usually used for thresholding, binarization and segmentation. It works mainly with the image histogram, looking at the pixel values and the regions that the user wants to segment out, rather than looking at the edges of an image. It tries to segment the image making the variance on each of the classes minimal. The algorithm works well for images that contain two classes of pixels, following a bi-modal histogram distribution.

Thresholding takes a gray-scale image and replaces each pixel with a black one if its intensity is less than some fixed constant, or a white pixel if the intensity is greater than that constant. The new binary image produced separates dark from bright regions. Mainly because finding pixels that share intensity in a region is not computationally expensive, thresholding is a simple and efficient method for image segmentation.

Otsu's algorithm is a simple thresholding method for image segmentation. The algorithm divides the image histogram into two classes, by using a threshold such as the in-class

variability is very small. This way, each class will be as compact as possible. The spatial relationship between pixels is not taken into account, so regions that have similar pixel values but are in completely different locations in the image will be merged when computing the histogram, meaning that Otsu's algorithm treats them as the same.
Assuming that the pixels are categorized in two classes, the algorithm tries to minimize the weighted within-class variance $\sigma_w^2(t)$, defined by the expression below. The variable t is the threshold, which is typically a value between 0 and 255.

$$\sigma_w^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t). \tag{11}$$

The process for computing $\sigma_w^2(t)$ is as follows: a probability function P is obtained for every pixel value. First, the histogram distribution for the image is computed, then a normalization is performed in order to guarantee it follows a probability distribution. After that, the pixel values are divided into two classes C_1 and C_2 by a threshold t, using the class probability functions $q_1(t)$ and $q_2(t)$, defined in equation 12.

$$q_1(t) = \sum_{i=1}^{t} P(i), \quad q_2(t) = \sum_{i=t+1}^{I} P(i). \tag{12}$$

Class C_1 represents those pixels with intensity levels in $[1, t]$, and class C_2 represents those pixels with levels in the interval $[t+1, I]$, where I is the largest pixel value (usually 255). Then, the means for class $C_1, \sigma_1^2(t)$ and class $C_2, \sigma_2^2(t)$ are obtained:

$$\mu_1(t) = \sum_{i=1}^{t} \frac{iP(i)}{q_1(t)}, \quad \mu_2(t) = \sum_{i=t+1}^{I} \frac{iP(i)}{q_2(t)}. \tag{13}$$

Afterwards, the variances for class $C_1, \sigma_1^2(t)$ and class $C_2, \sigma_2^2(t)$ are computed:

$$\sigma_1^2(t) = \sum_{i=1}^{t} [i - \mu_1(t)]^2 \frac{P(i)}{q_1(t)}, \tag{14}$$

$$\sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu_2(t)]^2 \frac{P(i)}{q_2(t)}. \tag{15}$$

Equations (14) and (15) define the weighted within-class variance for C_1 and C_2, respectively. These are the values that Otsu's algorithm is trying to minimize. This variance is a measure of how compact each class is, meaning that if the method chooses a bad threshold, the variance for one of the classes will be large. Using equations (11), (14) and (15), the total variance is defined as the sum of the within-class variance and the between-class variance,

$$\sigma^2 = \sigma_w^2(t) + \sigma_b^2(t), \tag{16}$$

where $\sigma_b^2(t) = q_1(t)q_2(t)[\mu_1(t) - \mu_2(t)]^2$. The value σ^2 is constant, as it does not depend on the threshold, therefore the algorithm must focus on minimizing $\sigma_w^2(t)$, or maximizing $\sigma_b^2(t)$.[1],[3]

```
int pixels = (stop->i-start->i)*(stop->j-start->j);
int threshold = 0;

    if (manual_threshold != 0)
     {
    // If threshold was manually entered
    threshold = manual_threshold;
    segment_image(Output, threshold,start,stop);
    return;
  }
// Compute threshold
// Init variables

double sum = 0;
double sum_B = 0; // for background pixels
double max_variance = 0; // max variance
int p1 = 0;
int p2 = 0;

//used for computing mean for foreground
    for (int i = 0; i <= MAX_LEVEL; i++)
    {
      sum = sum+(i*histogram[i]);
    }

    for (int i = 0 ; i <= MAX_LEVEL ; i++)
    {

      p1 = p1 + histogram[i]; // how many pixels for background at each step
      if (p1 == 0) continue; // we can't divide by zero

      p2 = pixels - p1; // how many pixels for foreground at each step

      if (p2 == 0) break; // no foreground pixels, we can't divide by zero
      // Compute mean_B and mean_F
      sum_B = sum_B + (double) (i * ((int)histogram[i])); // for background
      double mean_B = sum_B / p1; // mean for background
      double mean_F = (sum - sum_B) / p2; // mean for foreground
      double variance = (double) p1 * (double) p2 * (mean_B - mean_F) *
      (mean_B- mean_F); // total variance
      if (variance > max_variance)
      {
        max_variance = variance; // new max variance
        threshold = i; // threshold is at position i in histogram of image
      }
    }
```

4 Implementation

4.1 main.cpp

The charge is shifted through the device until it is readout by some electronics. The shifting and readout gives "noise" to each pixel. Also, the top right is generally brighter than the lower left. This is due to the way that the cloud chamber is illuminated and it is an important effect that will be taken into account.
To correct for the variations in light level and for the fixed feature in the data, each run has a background image. This background image is subtracted from each raw image to produce a 'cleaned' image. We will use segmentation methods to find the objects in the image. After segmenting the image, we start track detection using the nearest neighbour algorithm.

In main.cpp, we will perform the following:

1. Invoke program on a command line window as:
<input_filename><background_filename><output_filename><width>
<height><manual_threshold>,
2. Check if the input is correct.
3. Subtract background image.
4. Denoise image using median filtering.
5. Use Otsu's Algorithm for image segmentation.
6. Make a map of objects in order to remove the tracks that are too small, extract and report valid tracks (using the nearest neighbour algorithm).
7. Write output image.

The output of this program is comprised of two categories of information: 1. an output image for the detected tracks and eventually a black and white intermediate image obtained after Otsu's algorithm (see figure 1), 2. program outputs on the command window for every detected track. If we do not impose a manual threshold level and run the program for an image containing only β particles, usually the threshold computed by Otsu's algorithm will be 2, despite the fact that β particles have pixel values with RGB colours between 8 and 30. Making the manual threshold less than 10 will tend to detect also the vapors from the upper part of the image, which is not convenient. To summarize,

- Running the program for an image containing only α particles with manual threshold= 0 will correctly detect α particles, therefore we let Otsu's algorithm to decide which is the threshold for foreground pixels (see figure 2).
- Running the program for an image containing α and β particles with manual threshold= 0 will detect α particles, but will tend to discard the tracks corresponding to β particles. For such an image, we can set manual threshold= 10 from the command line option (see figure 3).

- Running the program for an image which contains only β particles with manual threshold= 0 will give us a threshold= 2 in Otsu's algorithm and from there we can either force a manual threshold= 10, or use the variables force_threshold=true and forced_threshold=10 to impose a threshold if the value obtained in Otsu's algorithm is too low (see figure 4).

4.2 defines.h

This is used to define debugging messages and a number of parameters:

1. Width and height of the image or section of the image to be processed.
2. Max value of gray level.
3. Force_threshold if user wants to force a certain threshold in Otsu's algorithm.
4. Forced_threshold in case the threshold is too low, set to this value.
5. Sensitivity of detection algorithm.
6. Energy value for which we decide between an α and a β particle.

4.3 Runclass.cpp/Runclass.h

This is a derived class from ProcessImage and TrackFinding. Its objectives are taking as input and checking if command line parameters are valid. For input images (scenes) and background image, it validates the existence of the file and checks that the images have the dimensions defined in defines.h. It also validates input files to have only grayscale pixels. For Otsu's algorithm, we also check the validity of window size and the value of manual threshold parameter.
By calling appropriate methods, runclass will perform corrections for getting the detection and report of the tracks. We use as private variables the following pointers:

```
1         BMP *Input; // this pointer will keep original image
2         BMP *Subtract; // this pointer will keep background image to be
3                        // subtracted from original image
4         BMP *Output; // output image for detected tracks
```

4.4 EasyBMP libraries

Images for this project have been taken using an iPad camera and have been converted to a greyscale .bmp image. The .bmp format records a red, green, blue and transparency value for each pixel. We will therefore use the EasyBMP library to handle the images. This library allows to read in a bitmap file and store its pixel data as a 2D array for manipulation. It is also possible to construct an image as a 2D array of pixels and draw a representation to a bitmap (.bmp) file.

4.5 pixel_of_object.cpp/pixel_of_object.h

Keeps inormation about a pixel and the track to which it belongs. All detected tracks are stored in a vector of vectors of pixel_of_object.

4.6 ProcessImage.cpp/ProcessImage.h

1. Initialise histogram vector.
2. Load image to be analyzed.
3. Load background image and subtract it (choose either 4 or 8 neighbours for a pixel).
4. Denoise with median filtering.
5. Make histogram image for a rectangle inside input image.
6. Perform Otsu's algorithm to determine the threshold for foreground and background pixels (or force it using manual_threshold).
7. Given the threshold and the position of a rectangle inside input image, segment the image.

The final result from processimage will be stored in BMP *Output from Runclass.

4.7 TrackFinding.cpp/TrackFinding.h

After performing the segmentation of the image in the object BMP *Output from Runclass, TrackFinding will do the following:

1. Make a map of pixels associated with each object (track) member using the nearest neighbour algorithm.
2. Extract the tracks from the map into a vector of vectors.
3. Remove tracks that are too small.
4. Check if the track touches the edges.

Using the make_map_of_objects method, which traverses the segmented image from left to right and top to bottom, we build a map that contains (i,j) pair as a key and object_number as value. For every convenient pixel (white colour), which is not already in the map, we test if the pixel is not present in the map and if its neighbours are not already present in the map. If this is the case, we decide that the current pixel represents a new object (track) and we insert it into the map. We then call the insert_neighbours method with the new object number. Otherwise we call the insert_neighbours with the current object number.

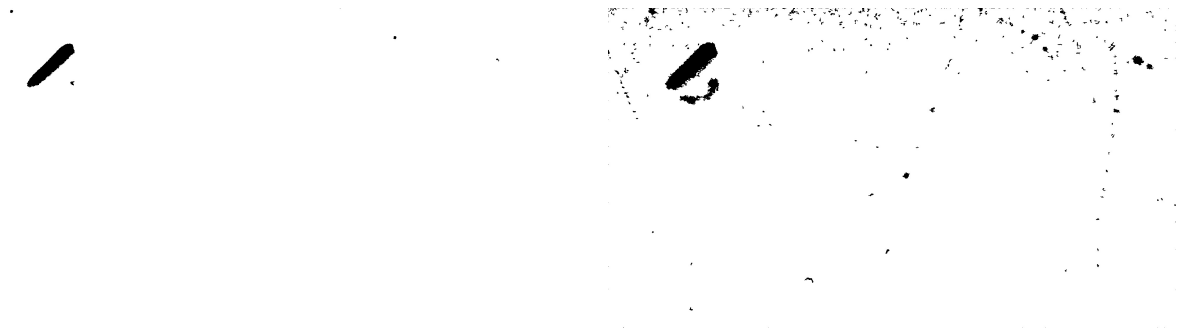
The insert_neighbour method uses the flood-fill technique (see [2]) in order to insert in the map the pixels which are neighbours for the current object number.

The extract_tracks_from_map method builds a vector of vectors containing pixel_of_object objects, which is a method of representing each pixel with its corresponding object (track) number.
The remove_tracks_too_small method deletes the tracks having number of members less than or equal to the sensitivity parameter, defined in defines.h. By trial and error method, a good value for the sensitivity parameter was found to be 20, since the noise is canceled out.

The report_tracks method displays relevant information for every detected track, such as:

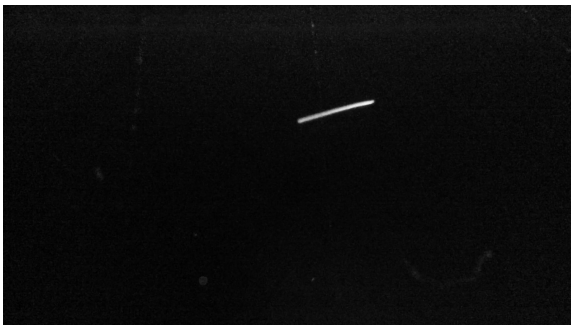
- Maximum distance between all pixels belonging to a track. This gives us information about the size of the track.
- Object location given by the pixels which have maximum distance for this object.
- Total energy of the track as sum of all pixel values, **taken from original image**.
- Total energy versus max.distance in the track.
- Total energy versus the number of pixels in the track.
- Direction angle of the track computed between the pixels which have the maximum distance in the track.
- If total energy versus max.distance in the track is less than β energy, **defined in defines.h**, we decide that this is a β particle, otherwise it is an α particle.

Using relevant methods, we also check if the track touches any of the edges using the sensitivity parameter.
Finally, in order to better see the result, we reverse the output image pixels and write an intermediate image obtained after segmentation. In this temporary image, we display all the pixels obtained after Otsu's segmentation. As a final result, in the chosen output image, we reverse all the pixels and we display all the tracks from the vector of tracks using one of the colours R,G,B, depending on the object (modulo 3).



(a) Image with threshold 0 (b) Image with threshold 10

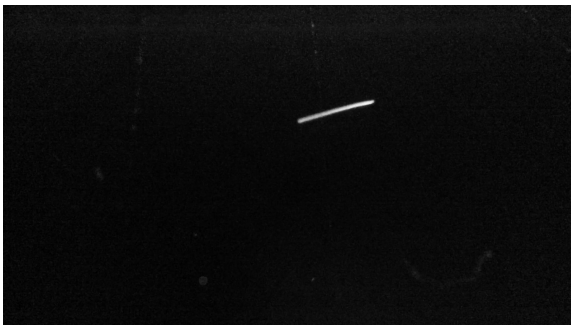
Figure 1: Black and White intermediate images for scene00186



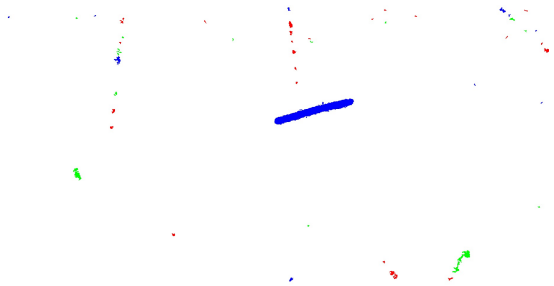
(a) Original image

(b) Image with threshold 0

Figure 2: Original image versus output for scene00518



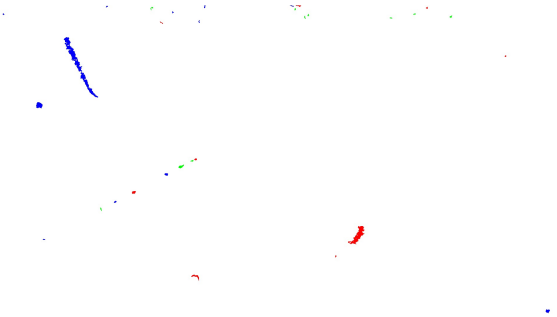
(a) Original image



(b) Image with threshold 10

Figure 3: Original image versus output for scene00518

(a) Original image



(b) Image with threshold 0

Figure 4: Original image versus output for scene00255

References

PH3170 Image Analysis in Cloud Chamber Photographs

[0] [1] Sonka, M., Hlavac, V. and Boyle, R. (2015).*Image processing, analysis and machine vision.* CENGAGE Learning.

[2] V4.software-carpentry.org. (2019). *Software Carpentry*:. [online] Available at: https://v4.software-carpentry.org/media/stars.html [Accessed 19 Dec. 2019].

[3] Balarini, J. and Nesmachnow, S. (2016). *A C++ Implementation of Otsu's Image Segmentation Method.* Image Processing On Line, 5, pp.155-164.