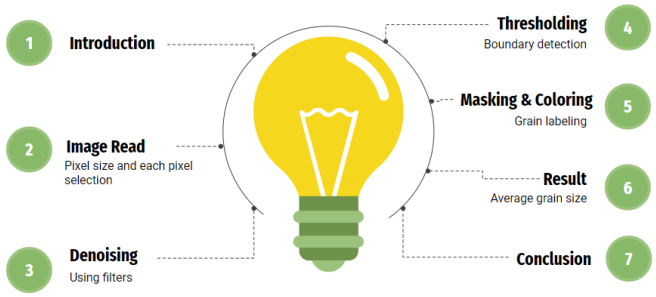
GRAIN SIZE ANALYSIS USING PYTHON

Arju B.Tech EE Part III, Khushboo Chaurasiya B.Tech EE Part III

Department of Electrical Engineering
Indian Institute of Technology Varanasi (BHU), India
Email:arju.student.eee21@itbhu.ac.in
khushboo.chaurasiya.eee21@itbhu.ac.in

Abstract*—The primary objective of this paper is to introduce an innovative approach to measure average grain area in microscopic images. In summary we read the image in grayscale after denoising using NLM filter and thresholding, masked and labeled grains with different color and using clusters with the average area and hence this will be useful for various inspection on microscopic level. Our method offers researchers and engineers an efficient tool for analyzing material microstructures, beneficial for pharmaceutical applications to analyze crystalline structures.



I. Introduction

Our project aims to develop a *Python-based solution* for grain size analysis, offering researchers and engineers a versatile and efficient tool to analyze images of material microstructures. By leveraging image processing techniques and computational algorithms, we provide a systematic approach to extracting grain size information from microscopy images.

II. Image Reading

Microscopic images are typically captured using a variety of imaging techniques, such as optical microscopy, scanning electron microscopy (SEM), or transmission electron microscopy (TEM), depending on the desired resolution and contrast requirements. These images provide detailed views of the microstructure of materials at the microscopic level.

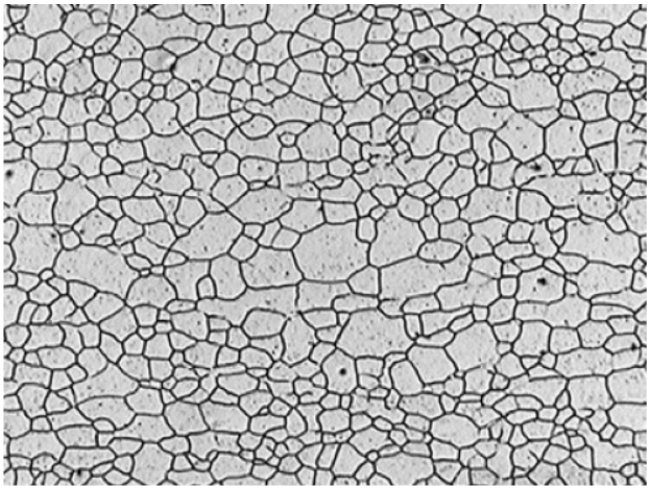


figure1. Sample Grain Image at microscopic level

Reading Microscopic Images: Once the microscopic images are acquired, our Python-based solution allows us to read these images programmatically. We use libraries such as *OpenCV* or *scikit-image* to read the image as an ndarray (N-dimensional array). The 0 argument(in code1.) indicates that the image should be read as grayscale. Grayscale images have a single channel representing

pixel intensity.Every image is made up of individual image elements known as pixels. The resolution of an image is determined by the number of pixels it contains. However, in order to relate features in an image to real-world dimensions, we need to establish a relationship between pixels and physical units such as microns.

Pixel size refers to the physical size represented by each pixel in an image. This information is often provided by the imaging device or can be determined through calibration procedures. By knowing the pixel size, we can convert measurements made in pixels to real-world units (eg, microns), allowing us to accurately quantify the dimensions of features in a material's microstructure.

By determining the pixel size, we bridge the gap between the digital representation of an image and its physical reality, facilitating quantitative analysis and comparison with experimental data or theoretical models.

```
import cv2
from skimage import img_as_float
from skimage.util import img_as_ubyte
from matplotlib import pyplot as plt
from skimage import io
import numpy as np

from scipy import ndimage as nd

# Specify the path to your image using the Kaggle dataset path
image_path = '/kaggle/input/grains/GrainSizeAnalysis_AppNote_img1.jpg'

img=img_as_float(io.imread(image_path,0))
```

Code1. reading image

II. Denoising Using Filters

In microscopy images, noise can arise due to various factors such as sensor imperfections, lighting variations, or electronic interference. This noise can degrade image quality and affect the accuracy of subsequent analysis. To address this, we employ denoising techniques to remove or reduce unwanted noise while preserving important image features.

- Types of Noise: Common types of noise in microscopy images include Gaussian noise, salt-and-pepper noise, and speckle noise. Each type of noise requires specific denoising methods tailored to its characteristics.

- Denoising Techniques: We utilize a range of denoising algorithms such as Gaussian smoothing, median filtering, bilateral filtering, non-local means (NLM) filtering, or wavelet denoising. These techniques effectively reduce noise while retaining the structural details of the image, ensuring a clearer representation of the material microstructure.

Gaussian Filter:

- ➢ Blurs or smoothes images.
- ➢ Uses a bell-shaped curve (Gaussian function) for convolution.
- ➢ The amount of blurring is controlled by the standard deviation parameter.
- ➢ Effective for reducing noise while preserving edges.

Median Filter:

- ➢ Removes noise from images.
- ➢ Replaces each pixel value with the median value of its neighbors.
- ➢ Particularly effective for removing "salt and pepper" noise.
- ➢ Preserves edges and fine details in the image.

```
#denoishing the image

#using gaussian filter
gauss_img=nd.gaussian_filter(img, sigma=2)
plt.imsave('/kaggle/working/new_img.jpg', gauss_img)

from IPython.display import Image, display

saved_image_path = '/kaggle/working/new_img.jpg'


#using median filter
median_img=nd.median_filter(img,size=2)
plt.imsave('/kaggle/working/new_img2.jpg', median_img)

saved_image2_path = '/kaggle/working/new_img2.jpg'
```

Code2. denoising using gaussian filter and median filter

```
#Display actual image
plt.imshow(img)
plt.axis('off')
plt.show()
```

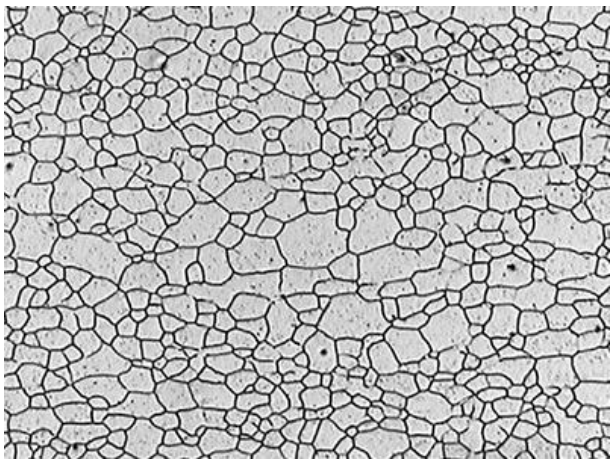


figure2. actual image without using any filer

```
# Display the median image
display(Image(filename=saved_image2_path)
```

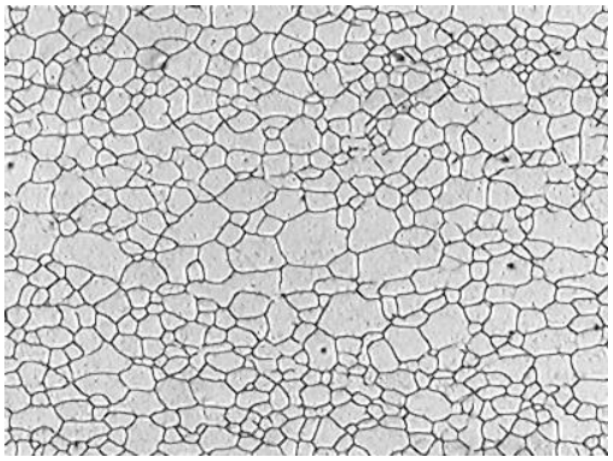


figure3. image after using median filter

```
# Display the gaussian  image
display(Image(filename=saved_image_path))
```

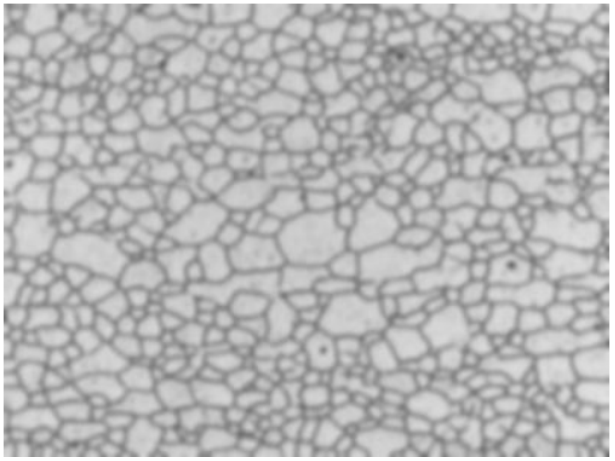


figure4. image after using gaussian filter

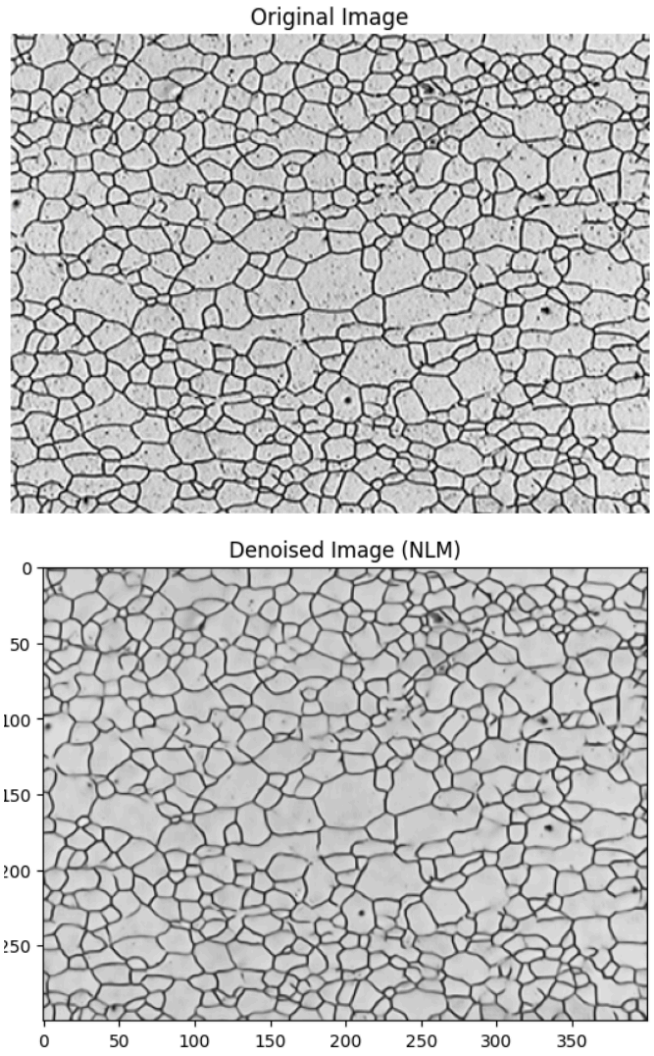
NLM Filter:

- ➢ Consider global image similarity.
- ➢ Matches patches for resemblance.
- ➢ Employs weighted averaging.
- ➢ Adapts to image content.
- ➢ Computational complexity may be high.
- ➢ Applied in medical imaging, photography, and video processing.

```
from skimage.restoration import denoise_nl_means,estimate_sigma
sigma_est = np.mean(estimate_sigma(img, channel_axis=-1))

patch_kw = dict(patch_size=5,
                patch_distance=3,)

denoise_img = denoise_nl_means(img, h=5* sigma_est,
                fast_mode=False,patch_size=5, patch_distance=3)

denoised_image = img_as_ubyte(denoise_img)
```

Code2. denoising using NLM filter

```
# Visualize the original and denoised images
#discplay actual imag
plt.imshow(img)
plt.title('Original Image')
plt.axis('off')
plt.show()
#display denoished img
plt.imshow(denoise_img)
plt.title('Denoised Image (NLM)')
plt.imsave('/kaggle/working/denoished.jpg', denoised_image)
```



Code3. Denoising using NLM filter

III. Image Thresholding

Thresholding on a gray image is a fundamental image processing technique used to separate objects or regions of interest from the background based on their pixel intensity values. In thresholding, a threshold value is defined, and each pixel in the gray image is compared to this threshold value. Pixels with intensity values above the threshold are classified as foreground (or object), while pixels with intensity values below the threshold are classified as background.
There are different methods for thresholding, including global thresholding, adaptive thresholding, and Otsu's thresholding, each suitable for different scenarios and image characteristics.

Threshold Selection: Thresholding involves selecting a threshold value to binarize the image, dividing it into foreground (grains) and background regions. The choice of threshold value depends on the intensity distribution of the image and the characteristics of the grains and background.

Types of Thresholding: Various thresholding methods are available, including global thresholding, adaptive thresholding, and Otsu's thresholding. These methods automatically determine the optimal threshold value based on the image histogram or local pixel neighborhoods, ensuring robust segmentation results.

Segmentation Accuracy: Proper thresholding is essential for accurate segmentation of grains from the background. It ensures that each grain is delineated effectively, laying the foundation for subsequent analysis such as grain size measurement and morphology characterization.

Otsu's Thresholding : Otsu's thresholding is a widely used technique for automatic image thresholding, particularly in cases where the histogram of the image is bimodal, meaning it has two peaks corresponding to the foreground and background intensities.It aims to find the optimal threshold value that separates the foreground (object of interest) from the background in an image. It does this by maximizing the between-class variance or minimizing the intra-class variance.

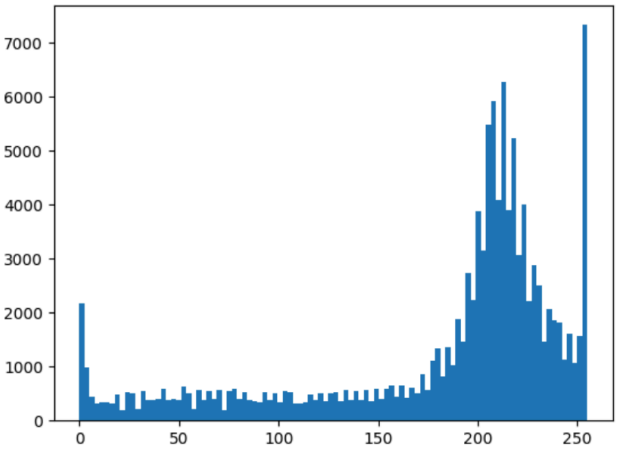


figure. histogram

- Otsu's method calculates the histogram of the image, representing the distribution of pixel

intensities.

- ➢ It then iterates through all possible threshold values and calculates the between-class variance for each threshold.
- ➢ The threshold value that maximizes the between-class variance (or minimizes the intra-class variance) is selected as the optimal threshold.

0:

- ➢ Pixels with intensity values below the threshold are set to 0.
- ➢ These pixels typically represent backgrounds or regions that do not meet the thresholding criteria.

255:

- ➢ Pixels with intensity values above the threshold are set to 255.
- ➢ These pixels typically represent foreground or regions of interest that meet the thresholding criteria.

III. Masking The Grains

Masking grainy gray images after thresholding involves using a binary mask to filter out noise or unwanted elements from the image. First, you apply a threshold to convert the gray image into a binary one. Then, we create a mask based on specific criteria, such as retaining larger features and filtering out noise. Finally, we apply this mask to the thresholded image, where areas marked '1' in the mask remain unchanged, while areas marked '0' are filtered out. This results in a clearer image with reduced noise and improved segmentation.

- ➢ After applying a threshold to the denoised grayscale image , the resulting binary image is subjected to morphological operations - erosion and dilation. These operations help in smoothing and refining the binary image by removing small noise and filling gaps in object contours.

- ➢ Next, a mask is created by checking where the dilated image has a pixel value of 255. This results in a binary mask where areas of interest are marked with '1' (white) and background or noise areas are marked with '0' (black).

- ➢ Finally, the labeled mask is generated. This function assigns a unique label to each connected component in the mask, allowing for the identification and analysis of separate objects or regions in the image.

```
from skimage import measure, color, io

# Convert the denoised image to grayscale
denoised_gray = cv2.cvtColor(denoised_image,
                             cv2.COLOR_RGB2GRAY)

# Apply Otsu's thresholding
ret, thresh = cv2.threshold(denoised_gray, 0, 255,
              cv2.THRESH_BINARY + cv2.THRESH_OTSU)
kernel = np.ones((3,3),np.uint8)
eroded = cv2.erode(thresh,kernel,iterations = 1)
dilated = cv2.dilate(eroded,kernel,iterations = 1)
mask = dilated == 255

io.imshow(mask)
s = [[1,1,1],[1,1,1],[1,1,1]]
labeled_mask, num_labels = nd.label(mask, structure=s)
#The function outputs a new image that contains a
#different integer label
#for each object, and also the number of objects found.
```

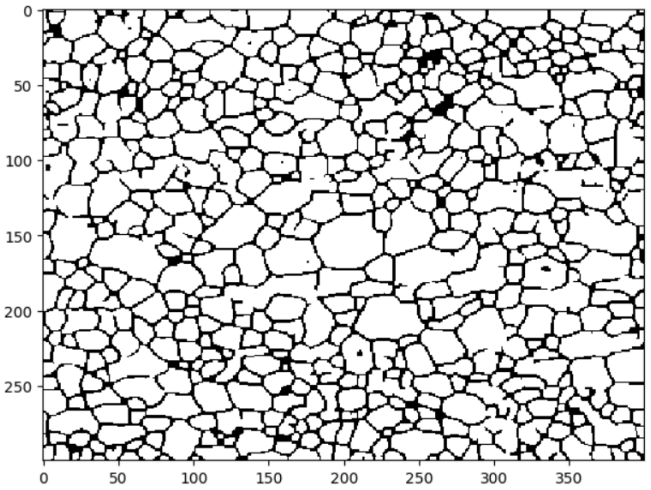


fig. thresholded image

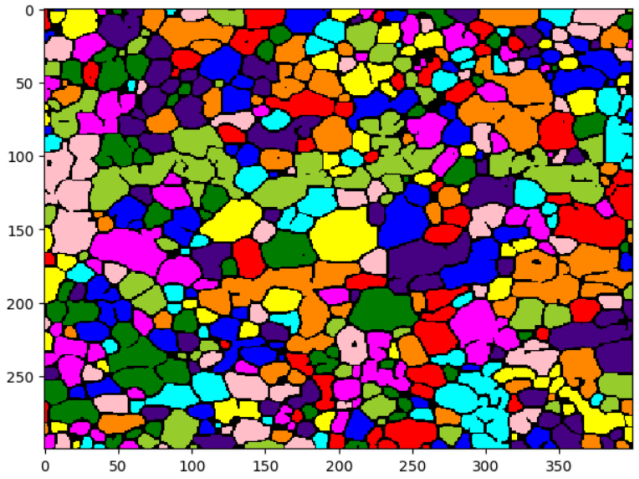
IV.Coloring The Grains

- ➢ Finally, the labeled mask is generated. This function assigns a unique label to each connected component in the mask, allowing for the identification and analysis of separate objects or regions in the image.

- ➢ By passing the *labeled_mask* to the *color.label2rgb* function, we generate a new image where each labeled region is visually distinguished by a distinct color. The *bg_label* parameter is used to specify the background label, which often corresponds to unlabeled regions or background areas in the image.

```
#Let's color the labels to see the effect
img2 = color.label2rgb(labeled_mask, bg_label=0)

io.imshow( img2)
io.show()
```

➢ **Coloring Labels for Visualization:** we enhance the visual representation of labeled regions in the image by assigning distinct colors to each label by utilizing the *color.label2rgb* function, which is commonly found in libraries like scikit-image. This function is specifically designed to assign unique colors to each label in a labeled image. This process aids in better understanding and interpretation of the segmented regions.



V. Area Calculation

The regionprops function in the skimage measure module calculates useful parameters for each object. Clusters are created using this function by providing the labeled mask and the original image for intensity measurements. A list of properties of interest, such as area, equivalent diameter, orientation, and intensity statistics, is defined. Assuming a pixel size of 1 square micrometer, the total area of all clusters is computed by iterating through each cluster and summing up the areas. Finally, the average area is determined by dividing the total area by the number of clusters.

```
  #Step 5: Measure the properties of each grain (object)

# regionprops function in skimage measure module
# calculates useful parameters for each object.

clusters = measure.regionprops(labeled_mask, img2)
#send in original image for Intensity measurements

propList = ['Area',
            'equivalent_diameter',
            'orientation',
            'MajorAxisLength',
            'MinorAxisLength',
            'Perimeter',
            'MinIntensity',
            'MeanIntensity',
            'MaxIntensity']
```

```
  #Step 5: Measure the properties of each grain (object)
# regionprops function in skimage measure module
# calculates useful parameters for each object.
clusters = measure.regionprops(labeled_mask, img2)
#send in original image for Intensity measurements

propList = ['Area',
            'equivalent_diameter',
            'orientation',
            'MajorAxisLength',
            'MinorAxisLength',
            'Perimeter',
            'MinIntensity',
            'MeanIntensity',
            'MaxIntensity']
pixel_size = 1 # tasking 1pixel size as 1um X 1um
total_area = 0
num_clusters = len(clusters)
print("No. grains: ",num_clusters)

# Iterate through each cluster and sum up the areas
for cluster_props in clusters:
    total_area+= cluster_props['Area'] * (pixel_size ** 2)
    # Assuming 'Area' is the key for area in cluster_props
# Calculate the average area
average_area = total_area / num_clusters
print("Final Average Area: ", average_area,"um^2")
```

Final Result For The Sample Image

```
No. grains:  431
Final Average Area:  218.61252900232017 um^2
```

VI. Conclusion

In summary, the model provides a holistic approach to processing and analyzing microscopy images, with a specific focus on material microstructures. By utilizing a diverse range of denoising algorithms including Gaussian smoothing, median filtering, bilateral filtering, non-local means (NLM) filtering, and wavelet denoising, the model effectively reduces noise while preserving crucial structural details. This ensures

that the resulting images accurately portray the material microstructure, facilitating further analysis. Additionally, the model incorporates techniques for defining pixel size and label coloring for visualization, enabling both quantitative analysis and qualitative inspection of microscopic features. Overall, the model serves as a robust solution for researchers and practitioners, enabling efficient and accurate extraction of insights from microscopy image.

VII. Futuristic work

In the future, we aim to make this tool universally applicable, allowing users to apply it to any image . We envision creating a user-friendly interface where individuals can upload their own sample images and experiment with modifications to achieve even more accurate results. By empowering users to tailor the analysis to their specific needs and challenges, we hope to foster a community-driven approach to refining and optimizing this tool for widespread use.