

```

In [ ]: # — Standard Library Imports —————
import os # Operating system interaction
import sys # Access to system-specific parameters and functions
import json # Reading and writing JSON configuration
import zipfile # Handling ZIP file creation

# — Third-party Imports —————
import cv2 # OpenCV for image processing
import numpy as np # Numerical operations
import pandas as pd # Data manipulation and tables
import openpyxl # Excel file I/O
import matplotlib.pyplot as plt # Plotting
import matplotlib # Matplotlib config
matplotlib.use("Qt5Agg") # Use Qt5Agg backend for GUI support

# — PyQt5 GUI Components —————
from PyQt5.QtWidgets import (
    QApplication, QWidget, QLabel, QLineEdit, QPushButton,
    QVBoxLayout, QHBoxLayout, QFileDialog, QMessageBox,
    QTextEdit, QInputDialog, QComboBox
)

# — Image I/O —————
import imageio.v2 as imageio # Image reading/writing (Legacy v2 API)

# — Skimage Modules for Image Processing —————
from skimage.measure import label, regionprops # Region Labeling
from skimage.filters import threshold_li, threshold_otsu, threshold_isodata # T
from skimage import data, filters, measure, morphology # Generic image ops
from skimage.color import rgb2gray # Convert RGB to grayscale
from skimage.morphology import (
    opening, remove_small_objects, remove_small_holes, disk
) # Morphological ops
from skimage import exposure, color # Image enhancement and color ops
from skimage.feature import peak_local_max # Peak detection
from skimage.segmentation import (
    morphological_chan_verse, slic, active_contour,
    watershed, random_walker
) # Various segmentation algorithms
from skimage.io import imread # Image reading
from skimage.transform import resize # Image resizing
from skimage import draw # Drawing shapes

# — SciPy for Advanced Processing —————
import scipy.ndimage as ndi # Multidimensional processing
from scipy.ndimage import distance_transform_edt, label as ndi_label # Distance
from scipy import ndimage # General ndimage support
from scipy.signal import find_peaks # Signal peak detection

# — Machine Learning —————
from sklearn.cluster import KMeans # Clustering (e.g., for region grouping)

# — Excel Writing —————
from xlswriter import Workbook # Advanced Excel writing

# — Qt Event Processing —————
QApplication.processEvents() # Process any pending GUI events

```

```

# — Threading & Event Control —————
from threading import Event # Used to signal stopping of processing

# — Utilities —————
from collections import defaultdict # Dictionary that creates default values au

import gc

# —————
# GUI Application Class for Image Processing
class ImageProcessingApp(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI() # Set up GUI layout

        # Default scale mapping (µm to pixels)
        self.um_to_px_map = {
            "40": 5.64039652,
            "100": 13.889
        }

        # Initialize folder paths and control flags
        self.bf_folder = ""
        self.pl_folder = ""
        self.output_folder = ""
        self.processing_active = False # Track if a process is currently running
        self.stop_event = Event() # Event to handle stop signal

        self.load_scale_settings() # Load saved scale mappings

    def initUI(self):
        # Create the GUI layout
        layout = QVBoxLayout()

        # Label and input for pixel distance
        self.pixel_distance_label = QLabel("Distance in pixels:")
        self.pixel_distance_input = QLineEdit()
        self.pixel_distance_input.setText("NOT VALUE")

        # Label and combo box for known µm distances
        self.known_um_label = QLabel("Known distance (µm):")
        self.known_um_combo = QComboBox()
        self.known_um_combo.setEditable(True)
        self.known_um_combo.addItem("40")
        self.known_um_combo.addItem("100")
        self.known_um_combo.setCurrentText("NOT VALUE")
        self.known_um_combo.setInsertPolicy(QComboBox.InsertAtBottom)
        self.known_um_combo.lineEdit().editingFinished.connect(self.on_custom_um)
        self.known_um_combo.currentIndexChanged.connect(self.update_pixel_distance)

        # Labels for folder selection display
        self.bf_label = QLabel("BF Folder: Not selected")
        self.pl_label = QLabel("PL Folder: Not selected")
        self.output_label = QLabel("Output Folder: Not selected")

        # Buttons for actions and controls
        self.set_scale_button = QPushButton("Set µm to px Scale")
        self.delete_scale_button = QPushButton("Delete Selected Scale")
        self.bf_button = QPushButton("Select BF Folder")
        self.pl_button = QPushButton("Select PL Folder")
        self.output_button = QPushButton("Select Output Folder")

```

```

self.process_button = QPushButton("Number of crystals")
self.process_button_2 = QPushButton("Areas")
self.process_button_3 = QPushButton("Number of cells")
self.stop_button = QPushButton("Stop Processing")
self.restart_button = QPushButton("Restart Processing")

# Log output window
self.log_output = QTextEdit()
self.log_output.setReadOnly(True)

# Connect button actions to their corresponding methods
self.set_scale_button.clicked.connect(self.set_known_um_and_px)
self.delete_scale_button.clicked.connect(self.delete_selected_scale)
self.bf_button.clicked.connect(self.select_bf_folder)
self.pl_button.clicked.connect(self.select_pl_folder)
self.output_button.clicked.connect(self.select_output_folder)
self.process_button.clicked.connect(self.start_processing)
self.process_button_2.clicked.connect(self.start_processing_2)
self.process_button_3.clicked.connect(self.start_processing_3)
self.stop_button.clicked.connect(self.stop_processing)
self.restart_button.clicked.connect(self.restart_processing)

# Add widgets to the GUI layout
layout.addWidget(self.set_scale_button)
layout.addWidget(self.delete_scale_button)
layout.addWidget(self.pixel_distance_label)
layout.addWidget(self.pixel_distance_input)
layout.addWidget(self.known_um_label)
layout.addWidget(self.known_um_combo)
layout.addWidget(self.bf_label)
layout.addWidget(self.bf_button)
layout.addWidget(self.pl_label)
layout.addWidget(self.pl_button)
layout.addWidget(self.output_label)
layout.addWidget(self.output_button)
layout.addWidget(self.process_button)
layout.addWidget(self.process_button_2)
layout.addWidget(self.process_button_3)
layout.addWidget(self.log_output)
layout.addWidget(self.stop_button)
layout.addWidget(self.restart_button)

# Finalize window settings
self.setLayout(layout)
self.setWindowTitle("Batch Image Processing")
self.resize(500, 400)

def log(self, message):
    # Append a Log message to the Log output display (likely a QTextEdit or
    self.log_output.append(message)

def on_custom_um_entered(self):
    # Handle user entering a custom  $\mu\text{m}$  value in the combo box
    text = self.known_um_combo.currentText().strip()

    # If the entered text is not already in the combo box, add it
    if text not in [self.known_um_combo.itemText(i) for i in range(self.know
        self.known_um_combo.addItem(text)

def update_pixel_distance(self):

```

```

# Update the pixel distance input field based on the selected scale
text = self.known_um_combo.currentText()

# If the scale is known, set the corresponding px value; otherwise clear
if text in self.um_to_px_map:
    self.pixel_distance_input.setText(str(self.um_to_px_map[text]))
else:
    self.pixel_distance_input.clear()

def select_bf_folder(self):
    # Prompt user to select a folder for BF (Brightfield) images
    self.bf_folder = QFileDialog.getExistingDirectory(self, "Select BF Folder")
    self.bf_label.setText(f"BF Folder: {self.bf_folder}")

def select_pl_folder(self):
    # Prompt user to select a folder for PL (Polarized Light) images
    self.pl_folder = QFileDialog.getExistingDirectory(self, "Select PL Folder")
    self.pl_label.setText(f"PL Folder: {self.pl_folder}")

def select_output_folder(self):
    # Prompt user to select a folder to save outputs
    self.output_folder = QFileDialog.getExistingDirectory(self, "Select Output Folder")
    self.output_label.setText(f"Output Folder: {self.output_folder}")

def stop_processing(self):
    # Set the stop event flag to signal that processing should stop
    self.stop_event.set()
    self.log("Stopping process...")

def restart_processing(self):
    # Stop current process and then start Script 3 again
    self.stop_processing()
    self.log("Restarting processing...")
    self.start_processing_3()

def save_scale_settings(self):
    # Save the scale mapping dictionary to a JSON file
    with open('scale_map.json', 'w') as f:
        json.dump(self.um_to_px_map, f)

def load_scale_settings(self):
    # Load scale mapping from a JSON file; use defaults if not found
    try:
        with open('scale_map.json', 'r') as f:
            self.um_to_px_map = json.load(f)
    except FileNotFoundError:
        # Fallback to default values if file is missing
        self.um_to_px_map = {
            "20": 1.29,
            "40": 5.64,
            "100": 13.89,
            "200": 4.78
        }

    # Clear and update the known  $\mu\text{m}$  combo box with loaded values
    self.known_um_combo.clear()
    self.known_um_combo.addItems(self.um_to_px_map.keys())

def set_known_um_and_px(self):
    # Prompt user to input a known real-world micrometer value

```

```

known_um, ok1 = QDialog.getDouble(self, "Known  $\mu$ m", "Enter known mi
if not ok1:
    return

# Prompt user to input the corresponding pixel distance
distance_px, ok2 = QDialog.getDouble(self, "Distance in Pixels", "E
if not ok2 or distance_px == 0:
    return

# Calculate  $\mu$ m per pixel ratio
um_per_px = known_um / distance_px
name = f"{known_um}"

# Save this new scale in the map and refresh the combo box
self.um_to_px_map[name] = um_per_px
self.save_scale_settings()
self.load_scale_settings()
self.known_um_combo.setCurrentText(name)

# Notify user that scale was saved
QMessageBox.information(self, "Saved", f"Added mapping '{name}' = {um_pe

def load_scales_from_json(self):
    # Load scales from a predefined JSON file, fallback to default if failed
    try:
        with open("scales.json", "r") as f:
            scales = json.load(f)
        return scales
    except Exception:
        return {"20": 1.29, "40": 5.64, "100": 13.89, "200": 4.78}

def add_new_scale(self, scale_name, scale_value):
    # Add new scale mapping and save it
    self.um_to_px_map[scale_name] = scale_value
    self.save_scale_settings()

def delete_selected_scale(self):
    # Delete selected scale from the combo box and mapping
    selected_scale = self.known_um_combo.currentText()

    # Only allow deletion of user-defined scales, not defaults
    if selected_scale in self.um_to_px_map and selected_scale not in ["20",
        confirm = QMessageBox.question(
            self,
            "Confirm Deletion",
            f"Are you sure you want to delete the scale '{selected_scale}'?"
            QMessageBox.Yes | QMessageBox.No
        )
        if confirm == QMessageBox.Yes:
            del self.um_to_px_map[selected_scale]
            self.save_scale_settings()
            self.load_scale_settings()
            self.pixel_distance_input.clear()
            self.known_um_combo.setCurrentText("NOT VALUE")
            self.log(f"Deleted scale '{selected_scale}'")
        else:
            # Warn if trying to delete a protected or non-existing scale
            QMessageBox.warning(self, "Not Found", f"The scale '{selected_scale}

def start_processing(self):

```

```

# Flag to indicate that processing is active
self.processing_active = True

# Reset the stop event in case it was triggered during a previous run
self.stop_event.clear()

# Validate that all necessary folders (BF, PL, and Output) have been selected
if not self.bf_folder or not self.pl_folder or not self.output_folder:
    self.log("Please select all folders before starting.")
    return
try:
    # Read user input for scale calibration
    distance_in_px = float(self.pixel_distance_input.text()) # Distance
    known_um = float(self.known_um_combo.currentText()) # Known re

    # Prevent division by zero when calculating pixel-to-micron scale
    if distance_in_px == 0:
        raise ValueError("Distance in pixels cannot be zero.")

    # Compute pixel-to-micrometer conversion factor
    pixel_to_um = 1 / (known_um / distance_in_px)
except ValueError:
    # Show warning if input is invalid or conversion fails
    QMessageBox.warning(self, "Input Error", "Please enter valid numeric")
    return None

# Create the output directory if it doesn't already exist
os.makedirs(self.output_folder, exist_ok=True)

# Collect and sort all .tif files in both BF and PL folders
bf_files = sorted([f for f in os.listdir(self.bf_folder) if f.endswith('.tif')])
pl_files = sorted([f for f in os.listdir(self.pl_folder) if f.endswith('.tif')])

# Check that the number of BF and PL images match for paired processing
if len(bf_files) != len(pl_files):
    raise ValueError("Mismatch in the number of BF and PL .tif files.")

# List to keep track of output files generated during processing
all_output_files = []

# Placeholder for storing row data to summarize in Excel or Logs
summary_rows = []

# Batch process each pair of Brightfield (BF) and Polarized Light (PL) images
for bf_file, pl_file in zip(bf_files, pl_files):
    print(f"Processing: {bf_file} and {pl_file}")

    # Allow user to stop processing midway
    if self.stop_event.is_set():
        self.log("Processing stopped.")
        return

    self.log(f"Processing {bf_file} and {pl_file}...")

    # Load BF and PL images
    bf_image_path = os.path.join(self.bf_folder, bf_file)
    pl_image_path = os.path.join(self.pl_folder, pl_file)
    imageA = cv2.imread(bf_image_path)
    imageB = cv2.imread(pl_image_path)

```

```

# Skip if images failed to load
if imageA is None or imageB is None:
    print(f"Skipping {bf_file} or {pl_file}: Unable to load image.")
    continue

# Convert BF image to grayscale
grayA = rgb2gray(imageA)

# --- Remove bottom-right scale bar region to avoid false detections
h, w = grayA.shape
crop_margin_h = int(0.015 * h) # ~1.5% of height
crop_margin_w = int(0.025 * w) # ~2.5% of width

# Mask the scale bar region (bottom-right) from analysis
mask = np.ones_like(grayA, dtype=bool)
mask[h - crop_margin_h:, w - crop_margin_w:] = False
grayA = grayA * mask # Apply mask to grayscale image

# Enhance contrast using adaptive histogram equalization
grayA = exposure.equalize_adapthist(grayA)

# Denoise the image using bilateral filtering
grayA = cv2.bilateralFilter((grayA * 255).astype(np.uint8), 9, 75, 7)

# Segment the image using Otsu's thresholding
threshold = threshold_otsu(grayA)
binary_A = (grayA < threshold).astype(np.uint8) * 255

# Apply morphological operations to clean segmentation
binary_A = morphology.opening(binary_A)
binary_A = morphology.remove_small_objects(binary_A.astype(bool), min_size=100)
binary_A = morphology.dilation(binary_A, morphology.disk(6))
binary_A = morphology.remove_small_holes(binary_A, area_threshold=50)
binary_A = morphology.closing(binary_A, morphology.disk(6))
binary_A = (binary_A > 0).astype(np.uint8) * 255

# Label connected regions
region_labels_A = label(binary_A)
region_props_A = regionprops(region_labels_A)

# Create mask for excluding cropped scale bar area
crop_start_row = h - crop_margin_h
crop_start_col = w - crop_margin_w
crop_mask = np.zeros_like(region_labels_A, dtype=bool)
crop_mask[crop_start_row:, crop_start_col:] = True

# Filter out regions that intersect with the cropped area
filtered_labels = []
for region in region_props_A:
    region_mask = (region_labels_A == region.label)
    if not np.any(region_mask & crop_mask):
        filtered_labels.append(region.label)

# Generate new Label image without excluded regions
new_label_img = np.zeros_like(region_labels_A, dtype=np.int32)
label_counter = 1
for lbl in filtered_labels:
    new_label_img[region_labels_A == lbl] = label_counter
    label_counter += 1

```



```

# Refresh region labels and properties
region_labels_A = new_label_img
region_props_A = regionprops(region_labels_A)

# Calculate region area statistics for filtering/splitting
areas = [region.area for region in region_props_A]
media_area = np.median(areas)
std_area = np.std(areas)
average = media_area + std_area # Adaptive threshold

# --- Save histogram of region areas ---
plt.figure(figsize=(8, 5))
plt.hist(areas, bins=20, color='skyblue', edgecolor='black')
plt.title("Histogram of Region Areas")
plt.xlabel("Area (pixels)")
plt.ylabel("Frequency")
plt.grid(True)
plt.tight_layout()
hist_areas_image_path = os.path.join(self.output_folder, f"{os.path.
plt.savefig(hist_areas_image_path, dpi=300, bbox_inches='tight')
plt.pause(0.001)
QApplication.processEvents()
plt.close()

print(f"Saved histogram for {bf_file} to {hist_areas_image_path}")
all_output_files.append(hist_areas_image_path)

# Refine Label image: keep small regions, split large ones using wat
for region in region_props_A:
    if region.area < average:
        new_label_img[region.slice][region.image] = label_counter
        label_counter += 1
    else:
        region_mask = np.zeros_like(region_labels_A, dtype=np.uint8)
        region_mask[region.slice][region.image] = 1
        distance = ndi.distance_transform_edt(region_mask)
        coordinates = peak_local_max(distance, labels=region_mask, m
        local_maxi = np.zeros_like(distance, dtype=bool)
        local_maxi[tuple(coordinates.T)] = True
        markers = label(local_maxi)
        labels_ws = watershed(-distance, markers, mask=region_mask)
        for ws_label in np.unique(labels_ws):
            if ws_label == 0:
                continue
            mask = labels_ws == ws_label
            new_label_img[mask] = label_counter
            label_counter += 1

# Final Labeled image after splitting
region_labels_A = new_label_img
region_props_A = regionprops(region_labels_A)

# 🔥 Reset Labels to start from 1
region_labels_A = label(region_labels_A > 0)
region_props_A = regionprops(region_labels_A)

# Ensure binary mask matches grayscale shape
if binary_A.shape != grayA.shape:
    binary_A = resize(binary_A, grayA.shape, order=0, preserve_range

```



```

# --- Visualize segmentation ---
plt.figure(figsize=(8, 8))
plt.imshow(region_labels_A, cmap='nipy_spectral')
plt.title('Segmentation')
plt.axis('off')
plt.pause(0.001)
QApplication.processEvents()
plt.close()

# Annotate region labels on binary image
overlay_image = cv2.cvtColor((binary_A > 0).astype(np.uint8) * 255,
for region in regionprops(region_labels_A):
    y, x = region.centroid
    label_id = region.label
    cv2.putText(overlay_image, str(region.label), (int(x), int(y)), c

# Save annotated segmentation image
annotated_path = os.path.join(self.output_folder, f"{os.path.splitext
cv2.imwrite(annotated_path, overlay_image)
print(f"Saved annotated image with labels to {annotated_path}")
all_output_files.append(annotated_path)

# Create binary mask with only valid detected regions
filtered_binary_A = np.zeros_like(binary_A)
for prop in region_props_A:
    if prop.area > 0:
        min_row, min_col, max_row, max_col = prop.bbox
        filtered_binary_A[min_row:max_row, min_col:max_col] = (
            region_labels_A[min_row:max_row, min_col:max_col] == prop
        )
filtered_binary_A = (filtered_binary_A > 0).astype(np.uint8) * 255

# --- Save region statistics to Excel ---
region_area = pd.DataFrame({
    "Region_Label": [region.label for region in region_props_A],
    "Region_Area (pixels)": [region.area for region in region_props_
    "Region_Area (μm²)": [region.area * (pixel_to_um ** 2) for regio
})

# Filter out regions with non-positive area (shouldn't happen, but s
region_area_df = region_area[region_area["Region_Area (μm²)"] > 0]

total_area = region_area_df["Region_Area (μm²)"].sum()
total_cells = region_area_df["Region_Label"].count()

# Append summary rows
region_area_df.loc["Total Area"] = ["", "Total Area", total_area]
region_area_df.loc["Total Cells"] = ["", "Total Cells", total_cells]

# Save region stats to Excel
region_area_excel_path = os.path.join(self.output_folder, f"{os.path
print(f"Saved region areas for {bf_file} to {region_area_excel_path}

# --- Plot histogram of pixel intensities ---
plt.figure(figsize=(8, 6))
plt.hist(grayA.ravel(), bins=256, range=[0, 255], color='blue', alph
plt.axvline(threshold, color='red', linestyle='dashed', linewidth=2,
plt.title('Histogram of Pixel Intensities')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')

```

```

plt.legend()

# Save the pixel intensity histogram
hist_cells_image_path = os.path.join(self.output_folder, f"{os.path.
plt.savefig(hist_cells_image_path, dpi=300, bbox_inches='tight')
plt.pause(0.001)
QApplication.processEvents()
plt.close()
print(f"Saved histogram for {bf_file} to {annotated_path}")
all_output_files.append(hist_cells_image_path)

# Convert BF image to grayscale and enhance contrast
grayB = rgb2gray(imageB)

grayB = exposure.equalize_adapthist(grayB)

# Apply bilateral filter to reduce noise
grayB = cv2.bilateralFilter((grayB * 255).astype(np.uint8), 9, 75, 7

# Calculate dynamic threshold
mean_intensity = np.mean(grayB)
std_intensity = np.std(grayB)

#ORIGINAL WITH VALUE 4
dynamic_threshold = mean_intensity + 4 * std_intensity

# Apply dynamic threshold
binary_B = (grayB > dynamic_threshold).astype(np.uint8)

plt.figure(figsize=(8, 6))
plt.hist(grayB.ravel(), bins=256, range=[0, 255], color='blue', alph
plt.axvline(dynamic_threshold, color='red', linestyle='dashed', line
plt.title('Histogram of Pixel Intensities')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.legend()

# Save the histogram image
hist_crystals_image_path = os.path.join(self.output_folder, f"{os.pa
plt.savefig(hist_crystals_image_path, dpi=300, bbox_inches='tight')
#plt.show()
plt.pause(0.001)
QApplication.processEvents() # Refresh PyQt GUI
plt.close()
print(f"Saved histogram for {bf_file} to {hist_crystals_image_path}")
all_output_files.append(hist_crystals_image_path)

QApplication.processEvents() # Refresh PyQt GUI

# Resize for alignment
filtered_binary_A_resized = cv2.resize(binary_A, (2048, 2048), inter
binary_B_resized = cv2.resize(binary_B, (2048, 2048), interpolation=

# Overlap calculation
overlap = (np.logical_and(filtered_binary_A_resized > 0, binary_B_re

# ☒ Mask the scale bar in bottom-right (adjust size as needed)
h2, w2 = overlap.shape
overlap[h2-60:h2, w2-450:w2] = 0 # adjust 50 and 100 depending on t

```

```

# Save overlap results
overlap_path = os.path.join(self.output_folder, f"{os.path.splitext(
cv2.imwrite(overlap_path, overlap)
all_output_files.append(overlap_path)

# Save clustering information
region_to_cell_mapping = []
cell_labels = label(filtered_binary_A_resized)
cell_props = regionprops(cell_labels)
region_labels = label(overlap)
region_props = regionprops(region_labels)

cell_to_crystals = defaultdict(list)

for region in region_props:
    region_coords = set(tuple(coord) for coord in region.coords)
    best_match_cell = None
    max_overlap = 0
    for cell in cell_props:
        cell_coords = set(tuple(coord) for coord in cell.coords)
        overlap_area = len(region_coords & cell_coords)
        if overlap_area > max_overlap:
            max_overlap = overlap_area
            best_match_cell = cell.label
    region_to_cell_mapping.append({
        "Region_Label": region.label,
        "Associated_Cell": best_match_cell,
        "Overlap (pixels)": max_overlap,
        "Region_Area (pixels)": region.area,
        "Region_Area (μm²)": region.area * (pixel_to_um ** 2)
    })

# ✅ Store the crystal label for the matched cell
if best_match_cell is not None:
    cell_to_crystals[best_match_cell].append(region.label)

# Save region-to-cell mapping as CSV
df_mapping = pd.DataFrame(region_to_cell_mapping)

if not df_mapping.empty and "Region_Area (μm²)" in df_mapping.columns:
    df_mapping = df_mapping[(df_mapping["Region_Area (μm²)"] < 10) &
df_mapping["Associated_Cell_Count"] = df_mapping["Associated_Cel
total_distinct_cells = df_mapping["Associated_Cell"].nunique()
df_mapping["Total_Cells_with_crystals"] = total_distinct_cells
total_area_cr = df_mapping["Region_Area (μm²)"].sum()
total_row = ["", "", "", "Total Area Crystals", total_area_cr, ""
df_mapping.loc["Total"] = total_row
else:
    total_distinct_cells = 0

# Save cell-to-crystal list (for debugging or export) ---
cell_crystal_df = pd.DataFrame([
    {
        "Cell_Label": cell_label,
        "Crystal_Labels": ", ".join(map(str, crystals)),
        "Crystal_Count": len(crystals)
    }
    for cell_label, crystals in cell_to_crystals.items()
])

```

```

# --- Save Excel ---
mapping_excel_path = os.path.join(self.output_folder, f"{os.path.spli

grouped_xlsx_path = os.path.join(self.output_folder, f"{os.path.spli

with pd.ExcelWriter(grouped_xlsx_path, engine='xlsxwriter') as writer:
    region_area_df.to_excel(writer, sheet_name='Cells', index=False)
    df_mapping.to_excel(writer, sheet_name='Crystals', index=False)
    cell_crystal_df.to_excel(writer, sheet_name='Cell-to-crystal map

print(f"Saved results for {bf_file} to {grouped_xlsx_path}")
#-----
# Visualization
annotated_image = imageA.copy()
for mapping in region_to_cell_mapping:
    region_label = mapping["Region_Label"]
    associated_cell = mapping["Associated_Cell"]
    if associated_cell:
        region = next(r for r in region_props if r.label == region_l
        min_row, min_col, max_row, max_col = region.bbox
        cv2.rectangle(annotated_image, (min_col, min_row), (max_col,
        cv2.putText(
            annotated_image,
            f"Cell {associated_cell}",
            (min_col, min_row - 5),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.3,
            (255, 0, 0),
            1
        )

# Plot both binary_A and binary_B
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

# Show detections
ax[0].imshow(annotated_image, cmap='gray')
ax[0].set_title('Detections')
ax[0].axis('off') # Hide axes

# Show coincidences
ax[1].imshow(overlap, cmap='gray')
ax[1].set_title('Coincidences')
ax[1].axis('off') # Hide axes

plt.tight_layout()
plt.pause(0.001)
QApplication.processEvents() # Refresh PyQt GUI
plt.close()

# Save annotated image
annotated_image_path = os.path.join(self.output_folder, f"{os.path.s
cv2.imwrite(annotated_image_path, annotated_image)

print(f"Saved results for {bf_file} to {self.output_folder}")

all_output_files.append(annotated_image_path)

del grayA, binary_A, region_labels_A, region_props_A, overlay_image,
gc.collect()

```

```

# ----- Summary -----

# Calculate the percentage of cells that contain at least one crystal
Percentage = f"{{(total_distinct_cells / total_cells * 100):.2f}}%" if

# Append summary statistics for this image to the report
summary_rows.append({
    "Day": os.path.splitext(bf_file)[0],      # Use base filename (w
    "total_cells": total_cells,                # Total number of segm
    "cells_with_crystals": total_distinct_cells, # Number of cells
    "%_cells_with_crystals": Percentage        # Percent of cells wit
})

# ----- Generate Summary Plot -----

# Create a DataFrame from the collected summary information
summary_df = pd.DataFrame(summary_rows)

# Ensure the "Day" column is treated as a string for proper sorting
summary_df["Day"] = summary_df["Day"].astype(str)
summary_df = summary_df.sort_values(by="Day")

# Convert percentage column from string (e.g., "12.5%") to float (e.g.,
summary_df["%_cells_with_crystals"] = summary_df["%_cells_with_crystals"]

# Extract numeric part from the "Day" string for grouping (e.g., "3A" →
summary_df["DAYS"] = summary_df["Day"].str.extract(r"(\d+)") # Only dig

# Group by numeric day and compute mean and standard deviation of the pe
grouped_df = summary_df.groupby("DAYS").agg({
    "%_cells_with_crystals": ["mean", "std"]
}).reset_index()

# Flatten multi-level column names after aggregation
grouped_df.columns = ["DAYS", "mean_percentage", "std_percentage"]

# Convert DAYS to integer for proper numerical sorting
grouped_df["DAYS"] = grouped_df["DAYS"].astype(int)
grouped_df = grouped_df.sort_values(by="DAYS")

# Determine the Y-axis limit (max percentage + buffer, capped at 100%)
max_percentage = grouped_df["mean_percentage"].max()
y_max_limit = min(100, max_percentage + 10)

# Plot average % of cells with crystals per day
plt.figure(figsize=(10, 6))
plt.plot(
    grouped_df["DAYS"],
    grouped_df["mean_percentage"],
    marker='o',
    linestyle='-',
    color='blue',
    linewidth=2,
    label="Average"
)

# Draw vertical lines for ±1 standard deviation
for x, y, std in zip(grouped_df["DAYS"], grouped_df["mean_percentage"],
    plt.vlines(

```

```

        x=x,
        ymin=y - std,
        ymax=y + std,
        color='blue',
        alpha=0.7,
        linewidth=2,
        label='±1 STD' if x == grouped_df["DAYS"].iloc[0] else ""
    )

    # Avoid duplicate Legend entries
    handles, labels = plt.gca().get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    plt.legend(by_label.values(), by_label.keys())

    plt.title("Average % Cells with Crystals", fontsize=14)
    plt.xlabel("Day", fontsize=12)
    plt.ylabel("% Cells with Crystals", fontsize=12)
    plt.ylim(0, y_max_limit)
    plt.grid(True)
    plt.pause(0.001)
    QApplication.processEvents()

    # Save the plot image
    plot_path = os.path.join(self.output_folder, "Plot.png")
    plt.savefig(plot_path, dpi=300)
    plt.pause(0.001)
    QApplication.processEvents()
    plt.close()

    # Save the grouped summary data to Excel
    grouped_df.to_excel(os.path.join(self.output_folder, "Plot.xlsx"), index

    self.log("Processing complete!")

    # ----- Zip Result Files -----
    # Create a ZIP archive with all saved result images
    zip_path = os.path.join(self.output_folder, "All_Images_histograms.zip")
    with zipfile.ZipFile(zip_path, 'w') as zipf:
        for file_path in all_output_files:
            zipf.write(file_path, arcname=os.path.basename(file_path))

    # Delete the individual files after zipping
    for file_path in all_output_files:
        if os.path.exists(file_path):
            os.remove(file_path)

def start_processing_2(self):
    # Flag to indicate that processing is active
    self.processing_active = True

    # Reset the stop event in case it was triggered during a previous run
    self.stop_event.clear()

    # Validate that all necessary folders (BF, PL, and Output) have been sel
    if not self.bf_folder or not self.pl_folder or not self.output_folder:
        self.log("Please select all folders before starting.")
        return
    try:
        # Read user input for scale calibration
        distance_in_px = float(self.pixel_distance_input.text()) # Distance

```

```

        known_um = float(self.known_um_combo.currentText()) # Known re

        # Prevent division by zero when calculating pixel-to-micron scale
        if distance_in_px == 0:
            raise ValueError("Distance in pixels cannot be zero.")

        # Compute pixel-to-micrometer conversion factor
        pixel_to_um = 1 / (known_um / distance_in_px)
    except ValueError:
        # Show warning if input is invalid or conversion fails
        QMessageBox.warning(self, "Input Error", "Please enter valid numeric")
        return None

    # Create the output directory if it doesn't already exist
    os.makedirs(self.output_folder, exist_ok=True)

    # Collect and sort all .tif files in both BF and PL folders
    bf_files = sorted([f for f in os.listdir(self.bf_folder) if f.endswith('.tif')])
    pl_files = sorted([f for f in os.listdir(self.pl_folder) if f.endswith('.tif')])

    # Check that the number of BF and PL images match for paired processing
    if len(bf_files) != len(pl_files):
        raise ValueError("Mismatch in the number of BF and PL .tif files.")

    # List to keep track of output files generated during processing
    all_output_files = []

    # Placeholder for storing row data to summarize in Excel or Logs
    summary_rows = []

    # Batch process each pair of Brightfield (BF) and Polarized Light (PL) images
    for bf_file, pl_file in zip(bf_files, pl_files):
        print(f"Processing: {bf_file} and {pl_file}")

        # Allow user to stop processing midway
        if self.stop_event.is_set():
            self.log("Processing stopped.")
            return

        self.log(f"Processing {bf_file} and {pl_file}...")

        # Load BF and PL images
        bf_image_path = os.path.join(self.bf_folder, bf_file)
        pl_image_path = os.path.join(self.pl_folder, pl_file)
        imageA = cv2.imread(bf_image_path)
        imageB = cv2.imread(pl_image_path)

        # Skip if images failed to load
        if imageA is None or imageB is None:
            print(f"Skipping {bf_file} or {pl_file}: Unable to load image.")
            continue

        # Convert BF image to grayscale
        grayA = rgb2gray(imageA)

        # --- Remove bottom-right scale bar region to avoid false detections
        h, w = grayA.shape
        crop_margin_h = int(0.015 * h) # ~1.5% of height
        crop_margin_w = int(0.025 * w) # ~2.5% of width

```



```

# Mask the scale bar region (bottom-right) from analysis
mask = np.ones_like(grayA, dtype=bool)
mask[h - crop_margin_h:, w - crop_margin_w:] = False
grayA = grayA * mask # Apply mask to grayscale image

# Enhance contrast using adaptive histogram equalization
grayA = exposure.equalize_adapthist(grayA)

# Denoise the image using bilateral filtering
grayA = cv2.bilateralFilter((grayA * 255).astype(np.uint8), 9, 75, 7)

# Segment the image using Otsu's thresholding
threshold = threshold_otsu(grayA)
binary_A = (grayA < threshold).astype(np.uint8) * 255

# Apply morphological operations to clean segmentation
binary_A = morphology.opening(binary_A)
binary_A = morphology.remove_small_objects(binary_A.astype(bool), min_area=10)
binary_A = morphology.dilation(binary_A, morphology.disk(6))
binary_A = morphology.remove_small_holes(binary_A, area_threshold=50)
binary_A = morphology.closing(binary_A, morphology.disk(6))
binary_A = (binary_A > 0).astype(np.uint8) * 255

# Label connected regions
region_labels_A = label(binary_A)
region_props_A = regionprops(region_labels_A)

# Create mask for excluding cropped scale bar area
crop_start_row = h - crop_margin_h
crop_start_col = w - crop_margin_w
crop_mask = np.zeros_like(region_labels_A, dtype=bool)
crop_mask[crop_start_row:, crop_start_col:] = True

# Filter out regions that intersect with the cropped area
filtered_labels = []
for region in region_props_A:
    region_mask = (region_labels_A == region.label)
    if not np.any(region_mask & crop_mask):
        filtered_labels.append(region.label)

# Generate new Label image without excluded regions
new_label_img = np.zeros_like(region_labels_A, dtype=np.int32)
label_counter = 1
for lbl in filtered_labels:
    new_label_img[region_labels_A == lbl] = label_counter
    label_counter += 1

# Refresh region labels and properties
region_labels_A = new_label_img
region_props_A = regionprops(region_labels_A)

# Calculate region area statistics for filtering/splitting
areas = [region.area for region in region_props_A]
media_area = np.median(areas)
std_area = np.std(areas)
average = media_area + std_area # Adaptive threshold

# --- Save histogram of region areas ---
plt.figure(figsize=(8, 5))
plt.hist(areas, bins=20, color='skyblue', edgecolor='black')

```

```

plt.title("Histogram of Region Areas")
plt.xlabel("Area (pixels)")
plt.ylabel("Frequency")
plt.grid(True)
plt.tight_layout()
hist_areas_image_path = os.path.join(self.output_folder, f"{os.path.
plt.savefig(hist_areas_image_path, dpi=300, bbox_inches='tight')
plt.pause(0.001)
QApplication.processEvents()
plt.close()
print(f"Saved histogram for {bf_file} to {hist_areas_image_path}")
all_output_files.append(hist_areas_image_path)

# Refine Label image: keep small regions, split large ones using wat
for region in region_props_A:
    if region.area < average:
        new_label_img[region.slice][region.image] = label_counter
        label_counter += 1
    else:
        region_mask = np.zeros_like(region_labels_A, dtype=np.uint8)
        region_mask[region.slice][region.image] = 1
        distance = ndi.distance_transform_edt(region_mask)
        coordinates = peak_local_max(distance, labels=region_mask, m
        local_maxi = np.zeros_like(distance, dtype=bool)
        local_maxi[tuple(coordinates.T)] = True
        markers = label(local_maxi)
        labels_ws = watershed(-distance, markers, mask=region_mask)
        for ws_label in np.unique(labels_ws):
            if ws_label == 0:
                continue
            mask = labels_ws == ws_label
            new_label_img[mask] = label_counter
            label_counter += 1

# Final Labeled image after splitting
region_labels_A = new_label_img
region_props_A = regionprops(region_labels_A)

# 🔥 Reset Labels to start from 1
region_labels_A = label(region_labels_A > 0)
region_props_A = regionprops(region_labels_A)

# Ensure binary mask matches grayscale shape
if binary_A.shape != grayA.shape:
    binary_A = resize(binary_A, grayA.shape, order=0, preserve_range

# --- Visualize segmentation ---
plt.figure(figsize=(8, 8))
plt.imshow(region_labels_A, cmap='nipy_spectral')
plt.title('Segmentation')
plt.axis('off')
plt.pause(0.001)
QApplication.processEvents()
plt.close()

# Annotate region labels on binary image
overlay_image = cv2.cvtColor((binary_A > 0).astype(np.uint8) * 255,
for region in regionprops(region_labels_A):
    y, x = region.centroid
    label_id = region.label

```

```

        cv2.putText(overlay_image, str(region.label), (int(x), int(y)), c

# Save annotated segmentation image
annotated_path = os.path.join(self.output_folder, f"{os.path.splitext
cv2.imwrite(annotated_path, overlay_image)
print(f"Saved annotated image with labels to {annotated_path}")
all_output_files.append(annotated_path)

# Create binary mask with only valid detected regions
filtered_binary_A = np.zeros_like(binary_A)
for prop in region_props_A:
    if prop.area > 0:
        min_row, min_col, max_row, max_col = prop.bbox
        filtered_binary_A[min_row:max_row, min_col:max_col] = (
            region_labels_A[min_row:max_row, min_col:max_col] == prop
        )
filtered_binary_A = (filtered_binary_A > 0).astype(np.uint8) * 255

# --- Save region statistics to Excel ---
region_area = pd.DataFrame({
    "Region_Label": [region.label for region in region_props_A],
    "Region_Area (pixels)": [region.area for region in region_props_
    "Region_Area (μm²)": [region.area * (pixel_to_um ** 2) for regio
})

# Filter out regions with non-positive area (shouldn't happen, but s
region_area_df = region_area[region_area["Region_Area (μm²)"] > 0]

total_area = region_area_df["Region_Area (μm²)"].sum()
total_cells = region_area_df["Region_Label"].count()

# Append summary rows
region_area_df.loc["Total Area"] = ["", "Total Area", total_area]
region_area_df.loc["Total Cells"] = ["", "Total Cells", total_cells]

# Save region stats to Excel
region_area_excel_path = os.path.join(self.output_folder, f"{os.path
print(f"Saved region areas for {bf_file} to {region_area_excel_path}

# --- Plot histogram of pixel intensities ---
plt.figure(figsize=(8, 6))
plt.hist(grayA.ravel(), bins=256, range=[0, 255], color='blue', alph
plt.axvline(threshold, color='red', linestyle='dashed', linewidth=2,
plt.title('Histogram of Pixel Intensities')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.legend()

# Save the pixel intensity histogram
hist_cells_image_path = os.path.join(self.output_folder, f"{os.path.
plt.savefig(hist_cells_image_path, dpi=300, bbox_inches='tight')
plt.pause(0.001)
QApplication.processEvents()
plt.close()
print(f"Saved histogram for {bf_file} to {annotated_path}")
all_output_files.append(hist_cells_image_path)

# Convert BF image to grayscale and enhance contrast
grayB = rgb2gray(imageB)

```

```

grayB = exposure.equalize_adapthist(grayB)

# Apply bilateral filter to reduce noise
grayB = cv2.bilateralFilter((grayB * 255).astype(np.uint8), 9, 75, 7)

# Calculate dynamic threshold
mean_intensity = np.mean(grayB)
std_intensity = np.std(grayB)

#ORIGINAL WITH VALUE 4
dynamic_threshold = mean_intensity + 4.6 * std_intensity

# Apply dynamic threshold
binary_B = (grayB > dynamic_threshold).astype(np.uint8)

binary_B = opening(binary_B) # Remove small noise
binary_B = (binary_B > 0).astype(np.uint8) * 255 # Convert back to b

plt.figure(figsize=(8, 6))
plt.hist(grayB.ravel(), bins=256, range=[0, 255], color='blue', alph
plt.axvline(dynamic_threshold, color='red', linestyle='dashed', line
plt.title('Histogram of Pixel Intensities')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.legend()

# Save the histogram image
hist_crystals_image_path = os.path.join(self.output_folder, f"{os.pa
plt.savefig(hist_crystals_image_path, dpi=300, bbox_inches='tight')
plt.pause(0.001)
QApplication.processEvents() # Refresh PyQt GUI
plt.close()
print(f"Saved histogram for {bf_file} to {hist_crystals_image_path}")
all_output_files.append(hist_crystals_image_path)

QApplication.processEvents() # Refresh PyQt GUI

# Resize for alignment
filtered_binary_A_resized = cv2.resize(binary_A, (2048, 2048), inter
binary_B_resized = cv2.resize(binary_B, (2048, 2048), interpolation=

# Overlap calculation
overlap = (np.logical_and(filtered_binary_A_resized > 0, binary_B_re

# ☒ Mask the scale bar in bottom-right (adjust size as needed)
h2, w2 = overlap.shape
overlap[h2-60:h2, w2-450:w2] = 0 # adjust 50 and 100 depending on t


# Save overlap results
overlap_path = os.path.join(self.output_folder, f"{os.path.splitext(
cv2.imwrite(overlap_path, overlap)
all_output_files.append(overlap_path)

# Save clustering information
region_to_cell_mapping = []
cell_labels = label(filtered_binary_A_resized)
cell_props = regionprops(cell_labels)
region_labels = label(overlap)
region_props = regionprops(region_labels)
cell_to_crystals = defaultdict(list)

```

```

for region in region_props:
    region_coords = set(tuple(coord) for coord in region.coords)
    best_match_cell = None
    max_overlap = 0
    for cell in cell_props:
        cell_coords = set(tuple(coord) for coord in cell.coords)
        overlap_area = len(region_coords & cell_coords)
        if overlap_area > max_overlap:
            max_overlap = overlap_area
            best_match_cell = cell.label
    region_to_cell_mapping.append({
        "Region_Label": region.label,
        "Associated_Cell": best_match_cell,
        "Overlap (pixels)": max_overlap,
        "Region_Area (pixels)": region.area,
        "Region_Area (μm²)": region.area * (pixel_to_um ** 2)
    })

#  Store the crystal label for the matched cell
if best_match_cell is not None:
    cell_to_crystals[best_match_cell].append(region.label)

# Save region-to-cell mapping as CSV
df_mapping = pd.DataFrame(region_to_cell_mapping)

if not df_mapping.empty and "Region_Area (μm²)" in df_mapping.columns:
    df_mapping = df_mapping[(df_mapping["Region_Area (μm²)"] < 6) &
    df_mapping["Associated_Cell_Count"] = df_mapping["Associated_Cell"]
    total_distinct_cells = df_mapping["Associated_Cell"].nunique()
    df_mapping["Total_Cells_with_crystals"] = total_distinct_cells
    total_area_cr = df_mapping["Region_Area (μm²)"].sum()
    total_row = ["", "", "", "Total Area Crystals", total_area_cr, ""
    df_mapping.loc["Total"] = total_row
else:
    total_distinct_cells = 0

# --- Optional: Save cell-to-crystal list (for debugging or export)
cell_crystal_df = pd.DataFrame([
    {
        "Cell_Label": cell_label,
        "Crystal_Labels": ", ".join(map(str, crystals)),
        "Crystal_Count": len(crystals)
    }
    for cell_label, crystals in cell_to_crystals.items()
])

# --- Save Excel ---
mapping_excel_path = os.path.join(self.output_folder, f"{os.path.splitext(
grouped_xlsx_path = os.path.join(self.output_folder, f"{os.path.splitext(

with pd.ExcelWriter(grouped_xlsx_path, engine='xlsxwriter') as writer:
    region_area_df.to_excel(writer, sheet_name='Cells', index=False)
    df_mapping.to_excel(writer, sheet_name='Crystals', index=False)
    cell_crystal_df.to_excel(writer, sheet_name='Cell-to-crystal map

print(f"Saved results for {bf_file} to {grouped_xlsx_path}")
#-----
# Visualization

```

```

annotated_image = imageA.copy()
for mapping in region_to_cell_mapping:
    region_label = mapping["Region_Label"]
    associated_cell = mapping["Associated_Cell"]
    if associated_cell:
        region = next(r for r in region_props if r.label == region_label)
        min_row, min_col, max_row, max_col = region.bbox
        cv2.rectangle(annotated_image, (min_col, min_row), (max_col,
        cv2.putText(
            annotated_image,
            f"Cell {associated_cell}",
            (min_col, min_row - 5),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.3,
            (255, 0, 0),
            1
        )

# Plot both binary_A and binary_B
fig, ax = plt.subplots(1, 2, figsize=(12, 6))

# Show detections
ax[0].imshow(annotated_image, cmap='gray')
ax[0].set_title('Detections')
ax[0].axis('off') # Hide axes

# Show coincidences
ax[1].imshow(overlap, cmap='gray')
ax[1].set_title('Coincidences')
ax[1].axis('off') # Hide axes

plt.tight_layout()
plt.pause(0.001)
QApplication.processEvents() # Refresh PyQt GUI
plt.close()

# Save annotated image
annotated_image_path = os.path.join(self.output_folder, f"{os.path.splitext(bf_file)[0]}_annotated.png")
cv2.imwrite(annotated_image_path, annotated_image)

print(f"Saved results for {bf_file} to {self.output_folder}")

all_output_files.append(annotated_image_path)

del grayA, binary_A, region_labels_A, region_props_A, overlay_image,
gc.collect()

# Calculate the percentage of crystal-covered area relative to total
Percentage = f"{{(total_area_cr / total_area * 100):.2f}}%" if total_area_cr > 0 else "0%"

# Append summary information for this image to the report
summary_rows.append({
    "Day": os.path.splitext(bf_file)[0], # Extract image
    "total_cells_area": total_area, # Sum of all cell areas
    "total_crystals_area": total_area_cr, # Sum of all crystal areas
    "%_area_crystals_cells": Percentage # Area percentage
})

# Create a DataFrame from all summarized results
summary_df = pd.DataFrame(summary_rows)

```

```

# Ensure 'Day' is treated as a string for consistent sorting
summary_df["Day"] = summary_df["Day"].astype(str)
summary_df = summary_df.sort_values(by="Day")

# Convert percentage string to float if needed (e.g., "23.5%" → 23.5)
summary_df["%_area_crystals_cells"] = summary_df["%_area_crystals_cells"]

# Extract numeric portion of the day (e.g., "1A" → 1) to group by day
summary_df["DAYS"] = summary_df["Day"].str.extract(r"(\d+)") # Extract

# Group by day number and compute mean and standard deviation of percent
grouped_df = summary_df.groupby("DAYS").agg({
    "%_area_crystals_cells": ["mean", "std"]
}).reset_index()

# Flatten multi-index column names
grouped_df.columns = ["DAYS", "mean_percentage", "std_percentage"]

# Convert DAYS to integer and sort numerically
grouped_df["DAYS"] = grouped_df["DAYS"].astype(int)
grouped_df = grouped_df.sort_values(by="DAYS")

# Determine Y-axis limit for the plot
max_percentage = grouped_df["mean_percentage"].max()
y_max_limit = min(100, max_percentage + 4) # Cap at 100%

# Plot average % of cells with crystals per day
plt.figure(figsize=(10, 6))
plt.plot(
    grouped_df["DAYS"],
    grouped_df["mean_percentage"],
    marker='o',
    linestyle='-',
    color='blue',
    linewidth=2,
    label="Average"
)

# Draw vertical lines for ±1 standard deviation
for x, y, std in zip(grouped_df["DAYS"], grouped_df["mean_percentage"],
                    grouped_df["std_percentage"]):
    plt.vlines(
        x=x,
        ymin=y - std,
        ymax=y + std,
        color='blue',
        alpha=0.7,
        linewidth=2,
        label='±1 STD' if x == grouped_df["DAYS"].iloc[0] else ""
    )

# Avoid duplicate legend entries
handles, labels = plt.gca().get_legend_handles_labels()
by_label = dict(zip(labels, handles))
plt.legend(by_label.values(), by_label.keys())

plt.ylim(0, y_max_limit)
plt.xlabel("Days")
plt.ylabel("% Area Crystals / Cells")
plt.title("Average % Area Crystals/Cells per Day")

```



```

plt.grid(True)
plt.pause(0.001)
QApplication.processEvents() # Update PyQt GUI

# Save the plot as PNG
plot_path = os.path.join(self.output_folder, "Plot.png")
plt.savefig(plot_path, dpi=300)
plt.pause(0.001)
QApplication.processEvents()
plt.close()

# Export grouped summary data to Excel
grouped_df.to_excel(os.path.join(self.output_folder, "Plot.xlsx"), index

self.log("Processing complete!")

# -----
# Create a ZIP archive with all output histogram and annotated image fil
zip_path = os.path.join(self.output_folder, "All_Images_histograms.zip")
with zipfile.ZipFile(zip_path, 'w') as zipf:
    for file_path in all_output_files:
        zipf.write(file_path, arcname=os.path.basename(file_path))

# Remove the original files after archiving
for file_path in all_output_files:
    if os.path.exists(file_path):
        os.remove(file_path)

def start_processing_3(self):
    # Flag to indicate that processing is active
    self.processing_active = True

    # Reset the stop event in case it was triggered during a previous run
    self.stop_event.clear()

    # Validate that all necessary folders (BF, PL, and Output) have been sel
    if not self.bf_folder or not self.pl_folder or not self.output_folder:
        self.log("Please select all folders before starting.")
        return
    try:
        # Read user input for scale calibration
        distance_in_px = float(self.pixel_distance_input.text()) # Distance
        known_um = float(self.known_um_combo.currentText()) # Known re

        # Prevent division by zero when calculating pixel-to-micron scale
        if distance_in_px == 0:
            raise ValueError("Distance in pixels cannot be zero.")

        # Compute pixel-to-micrometer conversion factor
        pixel_to_um = 1 / (known_um / distance_in_px)
    except ValueError:
        # Show warning if input is invalid or conversion fails
        QMessageBox.warning(self, "Input Error", "Please enter valid numeric
        return None

    # Create the output directory if it doesn't already exist
    os.makedirs(self.output_folder, exist_ok=True)

    # Collect and sort all .tif files in both BF and PL folders
    bf_files = sorted([f for f in os.listdir(self.bf_folder) if f.endswith('

```

```

pl_files = sorted([f for f in os.listdir(self.pl_folder) if f.endswith('.tif')])

# Check that the number of BF and PL images match for paired processing
if len(bf_files) != len(pl_files):
    raise ValueError("Mismatch in the number of BF and PL .tif files.")

# List to keep track of output files generated during processing
all_output_files = []

# Batch process each pair of Brightfield (BF) and Polarized Light (PL) images
for bf_file, pl_file in zip(bf_files, pl_files):
    print(f"Processing: {bf_file} and {pl_file}")

    # Allow user to stop processing midway
    if self.stop_event.is_set():
        self.log("Processing stopped.")
        return

    self.log(f"Processing {bf_file} and {pl_file}...")

    # Load BF and PL images
    bf_image_path = os.path.join(self.bf_folder, bf_file)
    pl_image_path = os.path.join(self.pl_folder, pl_file)
    imageA = cv2.imread(bf_image_path)
    imageB = cv2.imread(pl_image_path)

    # Skip if images failed to load
    if imageA is None or imageB is None:
        print(f"Skipping {bf_file} or {pl_file}: Unable to load image.")
        continue

    # Convert BF image to grayscale
    grayA = rgb2gray(imageA)

    # --- Remove bottom-right scale bar region to avoid false detections
    h, w = grayA.shape
    crop_margin_h = int(0.015 * h) # ~1.5% of height
    crop_margin_w = int(0.025 * w) # ~2.5% of width

    # Mask the scale bar region (bottom-right) from analysis
    mask = np.ones_like(grayA, dtype=bool)
    mask[h - crop_margin_h:, w - crop_margin_w:] = False
    grayA = grayA * mask # Apply mask to grayscale image

    # Enhance contrast using adaptive histogram equalization
    grayA = exposure.equalize_adapthist(grayA)

    # Denoise the image using bilateral filtering
    grayA = cv2.bilateralFilter((grayA * 255).astype(np.uint8), 9, 75, 7)

    # Segment the image using Otsu's thresholding
    threshold = threshold_otsu(grayA)
    binary_A = (grayA < threshold).astype(np.uint8) * 255

    # Apply morphological operations to clean segmentation
    binary_A = morphology.opening(binary_A)
    binary_A = morphology.remove_small_objects(binary_A.astype(bool), min_size=100)
    binary_A = morphology.dilation(binary_A, morphology.disk(6))
    binary_A = morphology.remove_small_holes(binary_A, area_threshold=50)
    binary_A = morphology.closing(binary_A, morphology.disk(6))

```

```

binary_A = (binary_A > 0).astype(np.uint8) * 255

# Label connected regions
region_labels_A = label(binary_A)
region_props_A = regionprops(region_labels_A)

# Create mask for excluding cropped scale bar area
crop_start_row = h - crop_margin_h
crop_start_col = w - crop_margin_w
crop_mask = np.zeros_like(region_labels_A, dtype=bool)
crop_mask[crop_start_row:, crop_start_col:] = True

# Filter out regions that intersect with the cropped area
filtered_labels = []
for region in region_props_A:
    region_mask = (region_labels_A == region.label)
    if not np.any(region_mask & crop_mask):
        filtered_labels.append(region.label)

# Generate new Label image without excluded regions
new_label_img = np.zeros_like(region_labels_A, dtype=np.int32)
label_counter = 1
for lbl in filtered_labels:
    new_label_img[region_labels_A == lbl] = label_counter
    label_counter += 1

# Refresh region Labels and properties
region_labels_A = new_label_img
region_props_A = regionprops(region_labels_A)

# Calculate region area statistics for filtering/splitting
areas = [region.area for region in region_props_A]
media_area = np.median(areas)
std_area = np.std(areas)
average = media_area + std_area # Adaptive threshold

# --- Save histogram of region areas ---
plt.figure(figsize=(8, 5))
plt.hist(areas, bins=20, color='skyblue', edgecolor='black')
plt.title("Histogram of Region Areas")
plt.xlabel("Area (pixels)")
plt.ylabel("Frequency")
plt.grid(True)
plt.tight_layout()
hist_areas_image_path = os.path.join(self.output_folder, f"{os.path.
plt.savefig(hist_areas_image_path, dpi=300, bbox_inches='tight')
plt.pause(0.001)
QApplication.processEvents()
plt.close()
print(f"Saved histogram for {bf_file} to {hist_areas_image_path}")
all_output_files.append(hist_areas_image_path)

# Refine Label image: keep small regions, split large ones using wat
for region in region_props_A:
    if region.area < average:
        new_label_img[region.slice][region.image] = label_counter
        label_counter += 1
    else:
        region_mask = np.zeros_like(region_labels_A, dtype=np.uint8)
        region_mask[region.slice][region.image] = 1

```

```

        distance = ndi.distance_transform_edt(region_mask)
        coordinates = peak_local_max(distance, labels=region_mask, m
        local_maxi = np.zeros_like(distance, dtype=bool)
        local_maxi[tuple(coordinates.T)] = True
        markers = label(local_maxi)
        labels_ws = watershed(-distance, markers, mask=region_mask)
        for ws_label in np.unique(labels_ws):
            if ws_label == 0:
                continue
            mask = labels_ws == ws_label
            new_label_img[mask] = label_counter
            label_counter += 1

# Final Labeled image after splitting
region_labels_A = new_label_img
region_props_A = regionprops(region_labels_A)

# 🔥 Reset Labels to start from 1
region_labels_A = label(region_labels_A > 0)
region_props_A = regionprops(region_labels_A)

# Ensure binary mask matches grayscale shape
if binary_A.shape != grayA.shape:
    binary_A = resize(binary_A, grayA.shape, order=0, preserve_range

# --- Visualize segmentation ---
plt.figure(figsize=(8, 8))
plt.imshow(region_labels_A, cmap='nipy_spectral')
plt.title('Segmentation')
plt.axis('off')
plt.pause(0.001)
QApplication.processEvents()
plt.close()

# Annotate region Labels on binary image
overlay_image = cv2.cvtColor((binary_A > 0).astype(np.uint8) * 255,
for region in regionprops(region_labels_A):
    y, x = region.centroid
    label_id = region.label
    cv2.putText(overlay_image, str(region.label), (int(x), int(y)), c

# Save annotated segmentation image
annotated_path = os.path.join(self.output_folder, f"{os.path.splitext
cv2.imwrite(annotated_path, overlay_image)
print(f"Saved annotated image with labels to {annotated_path}")
all_output_files.append(annotated_path)

# Create binary mask with only valid detected regions
filtered_binary_A = np.zeros_like(binary_A)
for prop in region_props_A:
    if prop.area > 0:
        min_row, min_col, max_row, max_col = prop.bbox
        filtered_binary_A[min_row:max_row, min_col:max_col] = (
            region_labels_A[min_row:max_row, min_col:max_col] == pro
        )
filtered_binary_A = (filtered_binary_A > 0).astype(np.uint8) * 255

# --- Save region statistics to Excel ---
region_area = pd.DataFrame({
    "Region_Label": [region.label for region in region_props_A],

```

```

        "Region_Area (pixels)": [region.area for region in region_props_
        "Region_Area (μm²)": [region.area * (pixel_to_um ** 2) for regio
    })

    # Filter out regions with non-positive area (shouldn't happen, but s
    region_area_df = region_area[region_area["Region_Area (μm²)"] > 0]

    total_area = region_area_df["Region_Area (μm²)"].sum()
    total_cells = region_area_df["Region_Label"].count()

    # Append summary rows
    region_area_df.loc["Total Area"] = ["", "Total Area", total_area]
    region_area_df.loc["Total Cells"] = ["", "Total Cells", total_cells]

    # Save region stats to Excel
    region_area_excel_path = os.path.join(self.output_folder, f"{os.path
    region_area_df.to_excel(region_area_excel_path, index=False)
    print(f"Saved region areas for {bf_file} to {region_area_excel_path}")

    # --- Plot histogram of pixel intensities ---
    plt.figure(figsize=(8, 6))
    plt.hist(grayA.ravel(), bins=256, range=[0, 255], color='blue', alph
    plt.axvline(threshold, color='red', linestyle='dashed', linewidth=2,
    plt.title('Histogram of Pixel Intensities')
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')
    plt.legend()

    # Save the pixel intensity histogram
    hist_cells_image_path = os.path.join(self.output_folder, f"{os.path
    plt.savefig(hist_cells_image_path, dpi=300, bbox_inches='tight')
    plt.pause(0.001)
    QApplication.processEvents()
    plt.close()
    print(f"Saved histogram for {bf_file} to {annotated_path}")
    all_output_files.append(hist_cells_image_path)

    del grayA, binary_A, region_labels_A, region_props_A, overlay_image,
    gc.collect()

    self.log("Processing complete!")

    # -----
    # Create a ZIP archive with all output histogram and annotated image fil
    zip_path = os.path.join(self.output_folder, "All_Images_histograms.zip")
    with zipfile.ZipFile(zip_path, 'w') as zipf:
        for file_path in all_output_files:
            zipf.write(file_path, arcname=os.path.basename(file_path))

    # Remove the original files after archiving
    for file_path in all_output_files:
        if os.path.exists(file_path):
            os.remove(file_path)

    # Entry point of the application
    if __name__ == "__main__":
        # Create a Qt application instance
        app = QApplication(sys.argv)

        # Instantiate the main window (custom image processing GUI)

```

```
window = ImageProcessingApp()

# Show the main window
window.show()

# Execute the Qt event loop and exit the application when it's closed
sys.exit(app.exec_())
```

In [ ]: