In [ ]:
```python
# ---------- Standard library imports ----------
import os                    # For operating system file and folder handling
import sys                   # For system-specific parameters and functions
import json                  # For reading and writing JSON files
import zipfile              # For working with ZIP archives
import tempfile            # For creating temporary files and directories
import shutil              # For high-level file operations (copy, move, delete)
from io import BytesIO    # For in-memory binary streams
from collections import defaultdict  # For dictionary with default values
from pathlib import Path    # For modern object-oriented filesystem paths

# ---------- Web app framework ----------
import streamlit as st       # For creating interactive web applications

# ---------- Excel file handling ----------
import openpyxl             # For reading and writing Excel files (.xlsx)
from xlsxwriter import Workbook  # For creating more advanced Excel files

# ---------- Scientific computing ----------
import numpy as np          # For numerical computations and arrays
import pandas as pd         # For data analysis and table-like data structures
import matplotlib           # Core matplotlib configuration
import matplotlib.pyplot as plt  # For creating plots and figures
matplotlib.use("Agg")       # Use non-interactive backend (safe for headless ser

# ---------- Computer vision and image processing ----------
import cv2                  # OpenCV for image processing and computer vision
from PIL import Image       # Pillow for general image reading and manipulation

# ---------- scikit-image modules ----------
from skimage.measure import label, regionprops   # For labeling and region prope
from skimage.filters import threshold_li         # Li's thresholding
from skimage.filters import threshold_otsu       # Otsu's thresholding
from skimage.filters import threshold_isodata    # Isodata thresholding
from skimage import data, filters, measure, morphology, exposure  # General imag
from skimage.color import rgb2gray               # Convert RGB to grayscale
from skimage.morphology import opening, remove_small_objects, remove_small_holes
from skimage import color                         # Additional color space funct
from skimage.feature import peak_local_max        # Find local maxima
from skimage.segmentation import morphological_chan_vese  # Chan-Vese segmentati
from skimage.segmentation import slic             # Superpixel segmentation (SLI
from skimage.segmentation import active_contour   # Active contour segmentation
from skimage.segmentation import watershed        # Watershed segmentation
from skimage.io import imread                     # Image reading
from skimage.transform import resize              # Image resizing
from skimage import draw                          # Drawing shapes on images

# ---------- Scientific image processing with SciPy ----------
from scipy.ndimage import distance_transform_edt, label as ndi_label  # Distance
from scipy import ndimage           # General n-dimensional image processing fun
from scipy.signal import find_peaks  # Find peaks in 1D data
import scipy.ndimage as ndi          # Alternative alias for ndimage (duplicate

# ---------- Machine learning ----------
from sklearn.cluster import KMeans  # K-means clustering for segmentation or gro

# Streamlit App
st.set_page_config(layout="wide")
```

```python
st.title("Microscopy Image Processing")

# Initialize rerun flag in session_state if not present
if "rerun_flag" not in st.session_state:
    st.session_state.rerun_flag = False

# File Upload
bf_files = st.file_uploader("Upload BF Images (.tif)", type=["tif"], accept_mult
pl_files = st.file_uploader("Upload PL Images (.tif)", type=["tif"], accept_mult

# Sort uploaded files
if bf_files:
    bf_files = sorted(bf_files, key=lambda x: x.name)
if pl_files:
    pl_files = sorted(pl_files, key=lambda x: x.name)

# File Count Info
if bf_files and pl_files:
    st.success(f"Found {len(bf_files)} BF files and {len(pl_files)} PL files.")
    if len(bf_files) != len(pl_files):
        st.warning("The number of BF and PL images does not match. Only matching

    for bf, pl in zip(bf_files, pl_files):
        st.write(f"Processing: {bf.name} and {pl.name}")

# Output Directory
output_dir = "outputs"
os.makedirs(output_dir, exist_ok=True)

# Load scale settings
@st.cache_data
def load_scale_settings():
    try:
        with open('scale_map.json', 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return {"20": 1.29, "40": 5.64, "100": 13.89, "200": 4.78}

um_to_px_map = load_scale_settings()

# Sidebar Scale Input
st.sidebar.header("Scale Settings")
selected_um = st.sidebar.selectbox("Known Distance (µm):", list(um_to_px_map.key
distance_in_px = st.sidebar.text_input("Distance in Pixels:", value=str(um_to_px

# Convert to float with error handling
try:
    s_um = float(selected_um)
    d_px = float(distance_in_px)
    PIXEL_TO_UM = 1 / (s_um / d_px)
    st.success(f"Calibration result: 1 px = {PIXEL_TO_UM:.4f} µm")
    st.session_state.pixel_to_um = PIXEL_TO_UM
except ValueError:
    st.error("Please enter valid numeric values for scale calibration.")

# Add Scale Section
st.sidebar.markdown("---")
st.sidebar.subheader("Manage Scale Settings")

new_um = st.sidebar.text_input("New µm value")
```

```
new_px = st.sidebar.text_input("New pixel value")
if st.sidebar.button("➕ Add Scale"):
    try:
        new_um_f = float(new_um)
        new_px_f = float(new_px)
        um_to_px_map[str(int(new_um_f))] = new_px_f
        with open('scale_map.json', 'w') as f:
            json.dump(um_to_px_map, f, indent=4)
        st.sidebar.success(f"Added scale: {int(new_um_f)} µm = {new_px_f} px")
        st.cache_data.clear()
        # Toggle rerun_flag to trigger rerun
        st.session_state.rerun_flag = not st.session_state.rerun_flag
    except ValueError:
        st.sidebar.error("Enter valid numbers to add scale.")

# Delete Scale Option
delete_um = st.sidebar.selectbox("Select µm to delete", list(um_to_px_map.keys())
if st.sidebar.button("🗑️ Delete Scale"):
    try:
        um_to_px_map.pop(delete_um, None)
        with open('scale_map.json', 'w') as f:
            json.dump(um_to_px_map, f, indent=4)
        st.sidebar.success(f"Deleted scale: {delete_um} µm")
        st.cache_data.clear()
        # Toggle rerun_flag to trigger rerun
        st.session_state.rerun_flag = not st.session_state.rerun_flag
    except Exception as e:
        st.sidebar.error(f"Error deleting: {e}")

# Session State Initialization
if "script1_done" not in st.session_state:
    st.session_state.script1_done = False
if "script1_results" not in st.session_state:
    st.session_state.script1_results = []
if "zip_path_1" not in st.session_state:
    st.session_state.zip_path_1 = None

# Start Button
if st.button("Number of cells with crystals"):
    if not bf_files or not pl_files:
        st.warning("Please upload both BF and PL files.")
    elif len(bf_files) != len(pl_files):
        st.error("Mismatch in number of BF and PL files.")
    else:
        st.session_state.script1_done = True
        st.session_state.script1_results.clear()

# Processing Logic
if st.session_state.script1_done:
    st.write("🔄 Starting batch processing...")
    all_output_files = []
    # Placeholder for storing row data to summarize in Excel or logs
    summary_rows = []

    for bf_file, pl_file in zip(bf_files, pl_files):

        with tempfile.NamedTemporaryFile(delete=False) as bf_temp, tempfile.Name
            bf_temp.write(bf_file.read())
            pl_temp.write(pl_file.read())
            bf_path = bf_temp.name
```

```
            pl_path = pl_temp.name

        imageA = cv2.imread(bf_path)
        imageB = cv2.imread(pl_path)

        if imageA is None or imageB is None:
            st.warning(f"Unable to read {bf_file.name} or {pl_file.name}. Skippi
            continue

        grayA = rgb2gray(imageA)

        # ----- CROP SCALE BAR REGION (e.g., bottom-right %) -----
        h, w = grayA.shape
        crop_margin_h = int(0.015* h)  # % of height-0.01
        crop_margin_w = int(0.025 * w)  # % of width-0.02

        # Create a mask that excludes bottom-right corner
        mask = np.ones_like(grayA, dtype=bool)
        mask[h - crop_margin_h:, w - crop_margin_w:] = False
        grayA = grayA * mask  # Set scale bar region to 0

        grayA = exposure.equalize_adapthist(grayA)
        grayA = cv2.bilateralFilter((grayA * 255).astype(np.uint8), 9, 75, 75)
        threshold = threshold_otsu(grayA)
        binary_A = (grayA < threshold).astype(np.uint8) * 255

        # Apply morphological operations to clean up the binary mask
        binary_A = morphology.opening(binary_A)
        binary_A = morphology.remove_small_objects(binary_A.astype(bool), min_si
        binary_A = morphology.dilation(binary_A, morphology.disk(6))
        binary_A = morphology.remove_small_holes(binary_A, area_threshold=5000)
        binary_A = morphology.closing(binary_A, morphology.disk(6))
        binary_A = (binary_A > 0).astype(np.uint8) * 255

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

        # Define crop box coordinates (bottom-right crop region)
        crop_start_row = h - crop_margin_h
        crop_start_col = w - crop_margin_w

        filtered_labels = []

        # Create a mask for the crop area pixels
        crop_mask = np.zeros_like(region_labels_A, dtype=bool)
        crop_mask[crop_start_row:, crop_start_col:] = True

        for region in region_props_A:
            # Get the mask of this region (boolean)
            region_mask = (region_labels_A == region.label)

            # Check if any pixel in this region overlaps with the crop mask
            if np.any(region_mask & crop_mask):
                # Region overlaps the crop area, skip it
                continue

            filtered_labels.append(region.label)

        # Create new labeled image excluding those 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

        region_labels_A[crop_start_row:, crop_start_col:] = 0

        # Update region_labels_A and region_props_A to filtered versions
        region_labels_A = new_label_img
        region_props_A = regionprops(region_labels_A)


        # Compute average threshold based on the mean and standart desviation of
        areas = [region.area for region in region_props_A]

        mean_area = np.mean(areas)
        median_area = np.median(areas)
        std_area = np.std(areas)
        min_area = np.min(areas)

        average = median_area + std_area

        # Histogram Areas
        fig, ax = plt.subplots()
        ax.hist(areas, bins=20, color='skyblue', edgecolor='black')
        hist_path_Areas = os.path.join(output_dir, f"{os.path.splitext(bf_file.n
        fig.savefig(hist_path_Areas)
        all_output_files.append(hist_path_Areas)

        for region in region_props_A:
            if region.area < average:
                new_label_img[region.slice][region.image] = label_counter
                label_counter += 1
            else:
                # Extract the subregion
                region_mask = np.zeros_like(region_labels_A, dtype=np.uint8)
                region_mask[region.slice][region.image] = 1

                # Compute distance transform
                distance = ndi.distance_transform_edt(region_mask)

                # Detect peaks for watershed markers
                # Get coordinates
                coordinates = peak_local_max(distance, labels=region_mask, min_d

                # Create empty mask and mark coordinates
                local_maxi = np.zeros_like(distance, dtype=bool)
                local_maxi[tuple(coordinates.T)] = True

                markers = label(local_maxi)

                # Apply watershed on the distance transform
                labels_ws = watershed(-distance, markers, mask=region_mask)

                # Add the new labels to the global label image
                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

    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_A is the correct shape (resize if necessary)
    if binary_A.shape != grayA.shape:
        binary_A = resize(binary_A, grayA.shape, order=0, preserve_range=Tru

    # Convert label image to RGB for annotation
    overlay_image = cv2.cvtColor((binary_A > 0).astype(np.uint8) * 255, cv2.

    # Loop through each region and annotate label number
    for region in regionprops(region_labels_A):
        y, x = region.centroid  # Note: (row, col) = (y, x)
        label_id = region.label
        cv2.putText(
            overlay_image,
            str(label_id),
            (int(x), int(y)),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.5,
            (0, 0, 255),  # Red color for text
            1,
            cv2.LINE_AA
        )

    # Save the annotated image
    annotated_path = os.path.join(output_dir, f"{bf_file.name}_Segmented_Cel
    cv2.imwrite(annotated_path, overlay_image)
    all_output_files.append(annotated_path)

    # Generate a dataframe for cells.
    region_area_df = pd.DataFrame({
        "Region_Label": [r.label for r in region_props_A],
        "Region_Area (pixels)": [r.area for r in region_props_A],
        "Region_Area (µm²)": [r.area * (PIXEL_TO_UM ** 2) for r in region_pr
    })

    region_area_df = region_area_df[region_area_df["Region_Area (µm²)"] > 0]
    total_cells = region_area_df["Region_Label"].count()
    region_area_df.loc["Total Area"] = ["", "Total Area", region_area_df["Re
    region_area_df.loc["Total Cells"] = ["", "Total Cells", total_cells]

    excel_path = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)[
    region_area_df.to_excel(excel_path, index=False)

    # Histogram A
    fig, ax = plt.subplots()
    ax.hist(grayA.ravel(), bins=256, range=[0, 255])
    ax.axvline(threshold, color='red', linestyle='--')
    hist_path_A = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)
    fig.savefig(hist_path_A)
    all_output_files.append(hist_path_A)
```

```
        # Image B thresholding
        grayB = rgb2gray(imageB)
        grayB = exposure.equalize_adapthist(grayB)
        grayB = cv2.bilateralFilter((grayB * 255).astype(np.uint8), 9, 75, 75)
        mean_intensity = np.mean(grayB)
        std_intensity = np.std(grayB)
        dynamic_threshold = mean_intensity + 4 * std_intensity
        binary_B = (grayB > dynamic_threshold).astype(np.uint8)

        fig, ax = plt.subplots()
        ax.hist(grayB.ravel(), bins=256, range=[0, 255])
        ax.axvline(dynamic_threshold, color='red', linestyle='--')
        hist_path_B = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)
        fig.savefig(hist_path_B)
        all_output_files.append(hist_path_B)

        overlap = (np.logical_and(cv2.resize(binary_A, (2048, 2048)) > 0, cv2.re

        # Mask bottom-right to remove scale bar artifacts
        h2, w2 = overlap.shape
        overlap[h2-60:h2, w2-450:w2] = 0

        overlap_path = os.path.join(output_dir, f"{os.path.splitext(bf_file.name
        cv2.imwrite(overlap_path, overlap)
        all_output_files.append(overlap_path)

        # Region associations
        region_props = regionprops(label(overlap))
        cell_props = region_props_A
        crystal_to_cell = []
        cell_to_crystals = defaultdict(list)

        for region in region_props:
            region_coords = set(map(tuple, region.coords))
            best_match_cell = None
            max_overlap = 0
            for cell in cell_props:
                cell_coords = set(map(tuple, cell.coords))
                overlap_area = len(region_coords & cell_coords)
                if overlap_area > 0:
                    cell_to_crystals[cell.label].append(region.label)
                if overlap_area > max_overlap:
                    max_overlap = overlap_area
                    best_match_cell = cell.label
            crystal_to_cell.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)

        # Generate a dataframe for crystals.
        df_mapping = pd.DataFrame(crystal_to_cell)

        if not df_mapping.empty and "Region_Area (µm²)" in df_mapping.columns:
```

```
        df_mapping = df_mapping[(df_mapping["Region_Area (µm²)"] < 10) & (df
        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

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

    # Merge only if df_mapping has Associated_Cell
    if not df_mapping.empty and "Associated_Cell" in df_mapping.columns:
        merged_df = df_mapping.merge(region_area_df, left_on="Associated_Cel
    else:
        merged_df = pd.DataFrame()

    # Groups all datasets into a single dataset.
    grouped_xlsx_path = os.path.join(output_dir, f"{os.path.splitext(bf_file
    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)
        merged_df.to_excel(writer, sheet_name="Cells + Crystals", index=Fals
        cell_crystal_df.to_excel(writer, sheet_name="Cell-Crystal Map", inde

    # Annotated Image
    annotated_image = cv2.cvtColor(imageA, cv2.COLOR_GRAY2BGR) if imageA.ndi
    for _, mapping in df_mapping.iterrows():
        if pd.notna(mapping["Associated_Cell"]):
            region = next((r for r in region_props if r.label == mapping["Re
            if region:
                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 {int(mapping['Associated
                            cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 0, 0), 1, l

    annotated_image_path = os.path.join(output_dir, f"{os.path.splitext(bf_f
    cv2.imwrite(annotated_image_path, annotated_image)
    all_output_files.append(annotated_image_path)

    # Calculate percentage (formatted as string with 2 decimals and % sign)
    percentage = f"{(total_distinct_cells / total_cells * 100):.2f}%" if tot

    # Create a summary dataframe (one row for each file)
    summary_rows.append({
        "Day": os.path.splitext(bf_file.name)[0],
        "Total_Cells": total_cells,
        "Cells_with_Crystals": total_distinct_cells,
        "%_cells_with_crystals": percentage
    })
```

```
        # Save session result
        st.session_state.script1_results.append({
            "bf_name": bf_file.name,
            "excel_path": grouped_xlsx_path,
            "annotated_image_path": annotated_image_path,
            "overlap_path": overlap_path,
            "hist_A_path": hist_path_A,
            "hist_B_path": hist_path_B
        })

    summary_df = pd.DataFrame(summary_rows)
    #-------------------------------------------------------------------------
    summary_df["Day"] = summary_df["Day"].astype(str)
    summary_df = summary_df.sort_values(by="Day")

    # Convert percentage to float
    summary_df["%_cells_with_crystals"] = summary_df["%_cells_with_crystals"].as

    # Extract number to group
    summary_df["DAYS"] = summary_df["Day"].str.extract(r"(\d+)")

    # Group by day
    grouped_df = summary_df.groupby("DAYS").agg({
        "%_cells_with_crystals": ["mean", "std"]
    }).reset_index()
    grouped_df.columns = ["DAYS", "mean_percentage", "std_percentage"]
    grouped_df["DAYS"] = grouped_df["DAYS"].astype(int)
    grouped_df = grouped_df.sort_values(by="DAYS")
    #-------------------------------------------------------------------------
    excel_path_2 = os.path.join(output_dir, "Plot.xlsx")
    grouped_df.to_excel(excel_path_2, index=False)
    #-----------------------------------------
    # 📈 Save % cells with crystals plot
    fig, ax = plt.subplots()
    ax.errorbar(grouped_df["DAYS"], grouped_df["mean_percentage"], yerr=grouped_
    ax.set_xlabel("Days")
    ax.set_ylabel("% Cells with Crystals")
    ax.set_title("Mean % Cells with Crystals Over Time")
    ax.grid(True)

    plot_img_path = os.path.join(output_dir, "Plot.png")
    fig.savefig(plot_img_path)
    all_output_files.append(plot_img_path)
    #-----------------------------------------------------------------
    # Add summary file info to session state separately
    st.session_state.script1_results.append({
        "bf_name": bf_file.name,
        "excel_path": grouped_xlsx_path,
        "annotated_image_path": annotated_image_path,
        "overlap_path": overlap_path,
        "hist_A_path": hist_path_A,
        "hist_B_path": hist_path_B,
        "excel_path_2": excel_path_2
    })

    # Create ZIP
    zip_path_1 = os.path.join(output_dir, "All_Images_histograms.zip")
    with zipfile.ZipFile(zip_path_1, 'w') as zipf_1:
        for file_path in all_output_files:
            zipf_1.write(file_path, arcname=os.path.basename(file_path))
```

```
    st.session_state.zip_path_1 = zip_path_1
    st.success("✅ Processing complete!")

if st.session_state.script1_results:
    st.header("📦 Results")

    for idx, result1 in enumerate(st.session_state.script1_results):
        st.subheader(f"📁 {result1['bf_name']}")

        if "annotated_image_path" in result1 and "overlap_path" in result1:
            st.image(result1["annotated_image_path"], caption="Detections crysta
            st.image(result1["overlap_path"], caption="Correlation")

        # Only ONE dataset button per image
        if "excel_path" in result1:
            with open(result1["excel_path"], "rb") as f1:
                st.download_button(
                    "📊 Download Dataset",
                    f1,
                    file_name=os.path.basename(result1["excel_path"]),
                    key=f"download_button_{idx}_{os.path.basename(result1['excel
                )

    with open(st.session_state.zip_path_1, "rb") as zf_1:
        st.download_button(
            "🗂️ Download All Images and Histograms",
            zf_1,
            file_name="All_Images_histograms.zip",
            key=f"download_zip_histograms_{idx}"
        )

    if "excel_path_2" in result1:
        with open(result1["excel_path_2"], "rb") as f2:
            st.download_button(
                "📊 Download Summary Plot",
                f2,
                file_name=os.path.basename(result1["excel_path_2"]),
                key=f"download_summary_button_{idx}"
            )
#-------------------------------------------------------------------------------

# Session State Initialization
if "script2_done" not in st.session_state:
    st.session_state.script2_done = False
if "script2_results" not in st.session_state:
    st.session_state.script2_results = []
if "zip_path_2" not in st.session_state:
    st.session_state.zip_path_2 = None

# Start Button
if st.button("Areas"):
    if not bf_files or not pl_files:
        st.warning("Please upload both BF and PL files.")
    elif len(bf_files) != len(pl_files):
        st.error("Mismatch in number of BF and PL files.")
    else:
        st.session_state.script2_done = True
        st.session_state.script2_results.clear()

# Processing Logic
```

```
if st.session_state.script2_done:
    st.write("🔄 Starting batch processing...")
    all_output_files = []
    # Placeholder for storing row data to summarize in Excel or logs
    summary_rows = []

    for bf_file, pl_file in zip(bf_files, pl_files):

        with tempfile.NamedTemporaryFile(delete=False) as bf_temp, tempfile.Name
            bf_temp.write(bf_file.read())
            pl_temp.write(pl_file.read())
            bf_path = bf_temp.name
            pl_path = pl_temp.name

        imageA = cv2.imread(bf_path)
        imageB = cv2.imread(pl_path)

        if imageA is None or imageB is None:
            st.warning(f"Unable to read {bf_file.name} or {pl_file.name}. Skippi
            continue

        grayA = rgb2gray(imageA)

        # ----- CROP SCALE BAR REGION (e.g., bottom-right %) -----
        h, w = grayA.shape
        crop_margin_h = int(0.015* h)  # % of height-0.01
        crop_margin_w = int(0.025 * w)  # % of width-0.02

        # Create a mask that excludes bottom-right corner
        mask = np.ones_like(grayA, dtype=bool)
        mask[h - crop_margin_h:, w - crop_margin_w:] = False
        grayA = grayA * mask  # Set scale bar region to 0

        grayA = exposure.equalize_adapthist(grayA)
        grayA = cv2.bilateralFilter((grayA * 255).astype(np.uint8), 9, 75, 75)
        threshold = threshold_otsu(grayA)
        binary_A = (grayA < threshold).astype(np.uint8) * 255

        # Apply morphological operations to clean up the binary mask
        binary_A = morphology.opening(binary_A)
        binary_A = morphology.remove_small_objects(binary_A.astype(bool), min_si
        binary_A = morphology.dilation(binary_A, morphology.disk(6))
        binary_A = morphology.remove_small_holes(binary_A, area_threshold=5000)
        binary_A = morphology.closing(binary_A, morphology.disk(6))
        binary_A = (binary_A > 0).astype(np.uint8) * 255

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

        # Define crop box coordinates (bottom-right crop region)
        crop_start_row = h - crop_margin_h
        crop_start_col = w - crop_margin_w

        filtered_labels = []

        # Create a mask for the crop area pixels
        crop_mask = np.zeros_like(region_labels_A, dtype=bool)
        crop_mask[crop_start_row:, crop_start_col:] = True
```

```
        for region in region_props_A:
            # Get the mask of this region (boolean)
            region_mask = (region_labels_A == region.label)

            # Check if any pixel in this region overlaps with the crop mask
            if np.any(region_mask & crop_mask):
                # Region overlaps the crop area, skip it
                continue

            filtered_labels.append(region.label)

        # Create new labeled image excluding those 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

        region_labels_A[crop_start_row:, crop_start_col:] = 0

        # Update region_labels_A and region_props_A to filtered versions
        region_labels_A = new_label_img
        region_props_A = regionprops(region_labels_A)


        # Compute average threshold based on the mean and standart desviation of
        areas = [region.area for region in region_props_A]

        mean_area = np.mean(areas)
        median_area = np.median(areas)
        std_area = np.std(areas)
        min_area = np.min(areas)

        average = median_area + std_area

        # Histogram Areas
        fig, ax = plt.subplots()
        ax.hist(areas, bins=20, color='skyblue', edgecolor='black')
        hist_path_Areas = os.path.join(output_dir, f"{os.path.splitext(bf_file.n
        fig.savefig(hist_path_Areas)
        all_output_files.append(hist_path_Areas)

        for region in region_props_A:
            if region.area < average:
                new_label_img[region.slice][region.image] = label_counter
                label_counter += 1
            else:
                # Extract the subregion
                region_mask = np.zeros_like(region_labels_A, dtype=np.uint8)
                region_mask[region.slice][region.image] = 1

                # Compute distance transform
                distance = ndi.distance_transform_edt(region_mask)

                # Detect peaks for watershed markers
                # Get coordinates
                coordinates = peak_local_max(distance, labels=region_mask, min_d

                # Create empty mask and mark coordinates
                local_maxi = np.zeros_like(distance, dtype=bool)
```

```
            local_maxi[tuple(coordinates.T)] = True

            markers = label(local_maxi)

            # Apply watershed on the distance transform
            labels_ws = watershed(-distance, markers, mask=region_mask)

            # Add the new labels to the global label image
            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

    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_A is the correct shape (resize if necessary)
    if binary_A.shape != grayA.shape:
        binary_A = resize(binary_A, grayA.shape, order=0, preserve_range=Tru

    # Convert label image to RGB for annotation
    overlay_image = cv2.cvtColor((binary_A > 0).astype(np.uint8) * 255, cv2.

    # Loop through each region and annotate label number
    for region in regionprops(region_labels_A):
        y, x = region.centroid  # Note: (row, col) = (y, x)
        label_id = region.label
        cv2.putText(
            overlay_image,
            str(label_id),
            (int(x), int(y)),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.5,
            (0, 0, 255),  # Red color for text
            1,
            cv2.LINE_AA
        )

    # Save the annotated image
    annotated_path = os.path.join(output_dir, f"{bf_file.name}_Segmented_Cel
    cv2.imwrite(annotated_path, overlay_image)
    all_output_files.append(annotated_path)

    #Generate a dataframe for cells
    region_area_df = pd.DataFrame({
        "Region_Label": [r.label for r in region_props_A],
        "Region_Area (pixels)": [r.area for r in region_props_A],
        "Region_Area (µm²)": [r.area * (PIXEL_TO_UM ** 2) for r in region_pr
    })

    region_area_df = region_area_df[region_area_df["Region_Area (µm²)"] > 0]
    total_cells = region_area_df["Region_Label"].count()
    region_area_df.loc["Total Area"] = ["", "Total Area", region_area_df["Re
    region_area_df.loc["Total Cells"] = ["", "Total Cells", total_cells]
```

```
        total_area = region_area_df["Region_Area (µm²)"].sum()

        excel_path = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)[
        region_area_df.to_excel(excel_path, index=False)

        # Histogram A
        fig, ax = plt.subplots()
        ax.hist(grayA.ravel(), bins=256, range=[0, 255])
        ax.axvline(threshold, color='red', linestyle='--')
        hist_path_A = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)
        fig.savefig(hist_path_A)
        all_output_files.append(hist_path_A)

        # Image B thresholding
        grayB = rgb2gray(imageB)
        grayB = exposure.equalize_adapthist(grayB)
        grayB = cv2.bilateralFilter((grayB * 255).astype(np.uint8), 9, 75, 75)
        mean_intensity = np.mean(grayB)
        std_intensity = np.std(grayB)
        dynamic_threshold = mean_intensity + 4.6 * std_intensity
        binary_B = (grayB > dynamic_threshold).astype(np.uint8)

        fig, ax = plt.subplots()
        ax.hist(grayB.ravel(), bins=256, range=[0, 255])
        ax.axvline(dynamic_threshold, color='red', linestyle='--')
        hist_path_B = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)
        fig.savefig(hist_path_B)
        all_output_files.append(hist_path_B)

        overlap = (np.logical_and(cv2.resize(binary_A, (2048, 2048)) > 0, cv2.re

        # Mask bottom-right to remove scale bar artifacts
        h2, w2 = overlap.shape
        overlap[h2-60:h2, w2-450:w2] = 0

        overlap_path = os.path.join(output_dir, f"{os.path.splitext(bf_file.name
        cv2.imwrite(overlap_path, overlap)
        all_output_files.append(overlap_path)

        # Region associations
        region_props = regionprops(label(overlap))
        cell_props = region_props_A
        crystal_to_cell = []
        cell_to_crystals = defaultdict(list)

        for region in region_props:
            region_coords = set(map(tuple, region.coords))
            best_match_cell = None
            max_overlap = 0
            for cell in cell_props:
                cell_coords = set(map(tuple, cell.coords))
                overlap_area = len(region_coords & cell_coords)
                if overlap_area > 0:
                    cell_to_crystals[cell.label].append(region.label)
                if overlap_area > max_overlap:
                    max_overlap = overlap_area
                    best_match_cell = cell.label
            crystal_to_cell.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)

    # Generate a dataframe for crystals.
    df_mapping = pd.DataFrame(crystal_to_cell)

    if not df_mapping.empty and "Region_Area (µm²)" in df_mapping.columns:
        df_mapping = df_mapping[(df_mapping["Region_Area (µm²)"] < 6) & (df_
        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
    cell_crystal_df = pd.DataFrame([
        {
            "Cell_Label": cell_label,
            "Crystal_Labels": ", ".join(map(str, set(crystals))),  # remove
            "Crystal_Count": len(set(crystals))                    # correct
        }
        for cell_label, crystals in cell_to_crystals.items()
    ])

    # Merge only if df_mapping has Associated_Cell
    if not df_mapping.empty and "Associated_Cell" in df_mapping.columns:
        merged_df = df_mapping.merge(region_area_df, left_on="Associated_Cel
    else:
        merged_df = pd.DataFrame()

    # Groups all datasets into a single dataset.
    grouped_xlsx_path = os.path.join(output_dir, f"{os.path.splitext(bf_file
    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)
        merged_df.to_excel(writer, sheet_name="Cells + Crystals", index=Fals
        cell_crystal_df.to_excel(writer, sheet_name="Cell-Crystal Map", inde

    # Annotated Image
    annotated_image = cv2.cvtColor(imageA, cv2.COLOR_GRAY2BGR) if imageA.ndi
    for _, mapping in df_mapping.iterrows():
        if pd.notna(mapping["Associated_Cell"]):
            region = next((r for r in region_props if r.label == mapping["Re
            if region:
                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 {int(mapping['Associated
                            cv2.FONT_HERSHEY_SIMPLEX, 0.4, (255, 0, 0), 1, l

    annotated_image_path = os.path.join(output_dir, f"{os.path.splitext(bf_f
    cv2.imwrite(annotated_image_path, annotated_image)
    all_output_files.append(annotated_image_path)
```

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

        # Create a summary dataframe (one row for each file)
        summary_rows.append({
                "Day": os.path.splitext(bf_file.name)[0],           # Extract i
                "total_cells_area": total_area,                  # Sum of all cel
                "total_crystals_area": total_area_cr,            # Sum of all cry
                "%_area_crystals_cells": Percentage              # Area percentag
            })
        # Save session result
        st.session_state.script2_results.append({
            "bf_name": bf_file.name,
            "excel_path": grouped_xlsx_path,
            "annotated_image_path": annotated_image_path,
            "overlap_path": overlap_path,
            "hist_A_path": hist_path_A,
            "hist_B_path": hist_path_B
        })

    summary_df = pd.DataFrame(summary_rows)
    #-------------------------------------------
    summary_df["Day"] = summary_df["Day"].astype(str)
    summary_df = summary_df.sort_values(by="Day")

    # Convert the percentage to float (if it comes as a string with "%")
    summary_df["%_area_crystals_cells"] = summary_df["%_area_crystals_cells"].as

    # Extract the number to group (e.g., "1A" → "1")
    summary_df["DAYS"] = summary_df["Day"].str.extract(r"(\d+)")  # solo números

    # Group by group_id and calculate average and standard deviation
    grouped_df = summary_df.groupby("DAYS").agg({
        "%_area_crystals_cells": ["mean", "std"]
    }).reset_index()

    # Flatten columns
    grouped_df.columns = ["DAYS", "mean_percentage", "std_percentage"]

    # Ordenar por group_id como entero
    grouped_df["DAYS"] = grouped_df["DAYS"].astype(int)
    grouped_df = grouped_df.sort_values(by="DAYS")
    #-----------------------------------------------------------------
    excel_path_2 = os.path.join(output_dir, "Plot.xlsx")
    grouped_df.to_excel(excel_path_2, index=False)
    #-------------------------------
    # 📈 Save % cells with crystals plot
    fig, ax = plt.subplots()
    ax.errorbar(grouped_df["DAYS"], grouped_df["mean_percentage"], yerr=grouped_
    ax.set_xlabel("Days")
    ax.set_ylabel("% Area Crystals/Cells")
    ax.set_title("Mean % Area Crystals/Cells Over Time")
    ax.grid(True)

    plot_img_path = os.path.join(output_dir, "Plot.png")
    fig.savefig(plot_img_path)
    all_output_files.append(plot_img_path)
    #-------------------------------------------
    # Add summary file info to session state separately
    st.session_state.script2_results.append({
```

```
            "bf_name": bf_file.name,
            "excel_path": grouped_xlsx_path,
            "annotated_image_path": annotated_image_path,
            "overlap_path": overlap_path,
            "hist_A_path": hist_path_A,
            "hist_B_path": hist_path_B,
            "excel_path_2": excel_path_2
        })

        # Create ZIP
        zip_path_2 = os.path.join(output_dir, "All_Images_histograms.zip")
        with zipfile.ZipFile(zip_path_2, 'w') as zipf_2:
            for file_path in all_output_files:
                zipf_2.write(file_path, arcname=os.path.basename(file_path))
        st.session_state.zip_path_2 = zip_path_2
        st.success("✅ Processing complete!")

if st.session_state.script2_results:
    st.header("📦 Results")

    for idx, result2 in enumerate(st.session_state.script2_results):
        st.subheader(f"📁 {result2['bf_name']}")

        if "annotated_image_path" in result2 and "overlap_path" in result2:
            st.image(result2["annotated_image_path"], caption="Detections crysta
            st.image(result2["overlap_path"], caption="Correlation")

        # Only ONE dataset button per image
        if "excel_path" in result2:
            with open(result2["excel_path"], "rb") as f2:
                st.download_button(
                    "📊 Download Dataset",
                    f2,
                    file_name=os.path.basename(result2["excel_path"]),
                    key=f"download_button_{idx}_{os.path.basename(result2['excel
                )

    with open(st.session_state.zip_path_2, "rb") as zf_2:
        st.download_button(
            "🗂️ Download All Images and Histograms",
            zf_2,
            file_name="All_Images_histograms.zip",
            key=f"download_zip_histograms_{idx}"
        )

    first_result_2 = st.session_state.script2_results[0]
    if "excel_path_2" in first_result_2:
        with open(first_result_2["excel_path_2"], "rb") as f3:
            st.download_button(
                "📊 Download Summary Plot",
                f3,
                file_name=os.path.basename(first_result["excel_path_2"]),
                key="download_summary_button"
            )

    if "excel_path_2" in result2:
        with open(result2["excel_path_2"], "rb") as f3:
            st.download_button(
                "📊 Download Summary Plot",
                f3,
```

```
                file_name=os.path.basename(result2["excel_path_2"]),
                key=f"download_summary_button_{idx}"
            )

# Session State Initialization
if "script3_done" not in st.session_state:
    st.session_state.script3_done = False
if "script3_results" not in st.session_state:
    st.session_state.script3_results = []
if "zip_path_3" not in st.session_state:
    st.session_state.zip_path_3 = None

# Start Button
if st.button("Number of cells"):
    if not bf_files or not pl_files:
        st.warning("Please upload both BF and PL files.")
    elif len(bf_files) != len(pl_files):
        st.error("Mismatch in number of BF and PL files.")
    else:
        st.session_state.script3_done = True
        st.session_state.script3_results.clear()

# Processing Logic
if st.session_state.script3_done:
    st.write("🔄 Starting batch processing...")
    all_output_files = []

    for bf_file, pl_file in zip(bf_files, pl_files):
        with tempfile.NamedTemporaryFile(delete=False) as bf_temp, tempfile.Name
            bf_temp.write(bf_file.read())
            pl_temp.write(pl_file.read())
            bf_path = bf_temp.name
            pl_path = pl_temp.name

        imageA = cv2.imread(bf_path)
        imageB = cv2.imread(pl_path)

        if imageA is None or imageB is None:
            st.warning(f"Unable to read {bf_file.name} or {pl_file.name}. Skippi
            continue

        grayA = rgb2gray(imageA)

        # ----- CROP SCALE BAR REGION (e.g., bottom-right %) -----
        h, w = grayA.shape
        crop_margin_h = int(0.015* h)  # % of height-0.01
        crop_margin_w = int(0.025 * w)  # % of width-0.02

        # Create a mask that excludes bottom-right corner
        mask = np.ones_like(grayA, dtype=bool)
        mask[h - crop_margin_h:, w - crop_margin_w:] = False
        grayA = grayA * mask  # Set scale bar region to 0

        grayA = exposure.equalize_adapthist(grayA)
        grayA = cv2.bilateralFilter((grayA * 255).astype(np.uint8), 9, 75, 75)
        threshold = threshold_otsu(grayA)
        binary_A = (grayA < threshold).astype(np.uint8) * 255

        # Apply morphological operations to clean up the binary mask
        binary_A = morphology.opening(binary_A)
```

```
        binary_A = morphology.remove_small_objects(binary_A.astype(bool), min_si
        binary_A = morphology.dilation(binary_A, morphology.disk(6))
        binary_A = morphology.remove_small_holes(binary_A, area_threshold=5000)
        binary_A = morphology.closing(binary_A, morphology.disk(6))
        binary_A = (binary_A > 0).astype(np.uint8) * 255

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

        # Define crop box coordinates (bottom-right crop region)
        crop_start_row = h - crop_margin_h
        crop_start_col = w - crop_margin_w

        filtered_labels = []

        # Create a mask for the crop area pixels
        crop_mask = np.zeros_like(region_labels_A, dtype=bool)
        crop_mask[crop_start_row:, crop_start_col:] = True

        for region in region_props_A:
            # Get the mask of this region (boolean)
            region_mask = (region_labels_A == region.label)

            # Check if any pixel in this region overlaps with the crop mask
            if np.any(region_mask & crop_mask):
                # Region overlaps the crop area, skip it
                continue

            filtered_labels.append(region.label)

        # Create new labeled image excluding those 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

        region_labels_A[crop_start_row:, crop_start_col:] = 0

        # Update region_labels_A and region_props_A to filtered versions
        region_labels_A = new_label_img
        region_props_A = regionprops(region_labels_A)


        # Compute average threshold based on the mean and standart desviation of
        areas = [region.area for region in region_props_A]

        mean_area = np.mean(areas)
        median_area = np.median(areas)
        std_area = np.std(areas)
        min_area = np.min(areas)

        average = median_area + std_area

        # Histogram Areas
        fig, ax = plt.subplots()
        ax.hist(areas, bins=20, color='skyblue', edgecolor='black')
        hist_path_Areas = os.path.join(output_dir, f"{os.path.splitext(bf_file.n
        fig.savefig(hist_path_Areas)
```

```
        all_output_files.append(hist_path_Areas)

        for region in region_props_A:
            if region.area < average:
                new_label_img[region.slice][region.image] = label_counter
                label_counter += 1
            else:
                # Extract the subregion
                region_mask = np.zeros_like(region_labels_A, dtype=np.uint8)
                region_mask[region.slice][region.image] = 1

                # Compute distance transform
                distance = ndi.distance_transform_edt(region_mask)

                # Detect peaks for watershed markers
                # Get coordinates
                coordinates = peak_local_max(distance, labels=region_mask, min_d

                # Create empty mask and mark coordinates
                local_maxi = np.zeros_like(distance, dtype=bool)
                local_maxi[tuple(coordinates.T)] = True

                markers = label(local_maxi)

                # Apply watershed on the distance transform
                labels_ws = watershed(-distance, markers, mask=region_mask)

                # Add the new labels to the global label image
                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

        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_A is the correct shape (resize if necessary)
        if binary_A.shape != grayA.shape:
            binary_A = resize(binary_A, grayA.shape, order=0, preserve_range=Tru

        # Convert label image to RGB for annotation
        overlay_image = cv2.cvtColor((binary_A > 0).astype(np.uint8) * 255, cv2.

        # Loop through each region and annotate label number
        for region in regionprops(region_labels_A):
            y, x = region.centroid  # Note: (row, col) = (y, x)
            label_id = region.label
            cv2.putText(
                overlay_image,
                str(label_id),
                (int(x), int(y)),
                cv2.FONT_HERSHEY_SIMPLEX,
                0.5,
                (0, 0, 255),  # Red color for text
```

```
                1,
                cv2.LINE_AA
            )

        # Save the annotated image
        annotated_path = os.path.join(output_dir, f"{bf_file.name}_Segmented_Cel
        cv2.imwrite(annotated_path, overlay_image)
        all_output_files.append(annotated_path)

        # Generate a dataframe for cells.
        region_area_df = pd.DataFrame({
            "Region_Label": [r.label for r in region_props_A],
            "Region_Area (pixels)": [r.area for r in region_props_A],
            "Region_Area (µm²)": [r.area * (PIXEL_TO_UM ** 2) for r in region_pr
        })

        region_area_df = region_area_df[region_area_df["Region_Area (µm²)"] > 0]
        total_cells = region_area_df["Region_Label"].count()
        region_area_df.loc["Total Area"] = ["", "Total Area", region_area_df["Re
        region_area_df.loc["Total Cells"] = ["", "Total Cells", total_cells]

        excel_path = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)[
        region_area_df.to_excel(excel_path, index=False)

        # Histogram A
        fig, ax = plt.subplots()
        ax.hist(grayA.ravel(), bins=256, range=[0, 255])
        ax.axvline(threshold, color='red', linestyle='--')
        hist_path_A = os.path.join(output_dir, f"{os.path.splitext(bf_file.name)
        fig.savefig(hist_path_A)
        all_output_files.append(hist_path_A)

        # Save session result
        st.session_state.script3_results.append({
            "bf_name": bf_file.name,
            "annotated_path": annotated_path,
            "hist_A_path": hist_path_A,
            "hist_path_Areas": hist_path_Areas,
            "excel_path": excel_path,
        })

    # Create ZIP
    zip_path_3 = os.path.join(output_dir, "All_Images_histograms.zip")
    with zipfile.ZipFile(zip_path_3, 'w') as zipf_3:
        for file_path in all_output_files:
            zipf_3.write(file_path, arcname=os.path.basename(file_path))
    st.session_state.zip_path_3 = zip_path_3
    st.success("✅ Processing complete!")

# Display Outputs and Download Buttons
if st.session_state.script3_results:
    st.header("📦 Results")

    for idx, result3 in enumerate(st.session_state.script3_results):
        st.subheader(f"📁 {result3['bf_name']}")
        st.image(result3["annotated_path"], caption="Segmented Image")
        st.image(result3["hist_path_Areas"], caption="Areas Histogram")
        st.image(result3["hist_A_path"], caption="Pixels Intensity Histogram")

        with open(result3["excel_path"], "rb") as f3:
```

```
            st.download_button("📊 Download Dataset",f3,file_name=os.path.basena

    with open(st.session_state.zip_path_3, "rb") as zf_3:
        st.download_button("🗂️ Download All Images and Histograms", zf_3, file_
```