PyPETANA: Python Tool for Image Extraction and Dynamic Feature Analysis in Petridish Biological Systems

Sanghita Sengupta

Howard University, 2400 6^{th} St NW, Washington, DC 20059

August 1, 2024

Abstract

This tutorial presents an overview of the Python tool, PyPETANA, designed for image extraction and dynamic feature analysis in petridish biological systems. The tool includes various functionalities such as curve transformation, radius masking, and feature extraction from images. Key features include the ability to preprocess images, extract and mask Petri dishes, and identify and analyze dynamic features like circularity and fractal dimensions related to morphological growth. Additionally, this work provides detailed descriptions and code snippets for critical functions, alongside usage instructions and an outline of the code workflow. This tool aims to streamline image analysis processes and enhance research capabilities in biological studies involving petridish experiments for dynamical physical parameters.

Contents

1 Introduction

This work aims to describe the general code workflow, functionalities, and features of the Python-based software tool `PyPETANA` (3). Understanding the growth morphology and behavioral dynamics of biological organisms on petri dishes is crucial, as these aspects are influenced by environmental conditions. A quantifiable approach to studying these features requires both physical and computational toolkits. In this version of `PyPETANA`, we address these needs by providing advanced image processing techniques and dynamic quantifications, including area, perimeter, circularity, and fractal dimensions.

1.1 System Requirements

The script is designed to run with `Python 3`. Ensure `Python 3` is installed on your system. The following Python packages are required to run the script:

- `opencv-python` (cv2): Used for image processing and contour analysis.
- `numpy`: Provides support for numerical operations and array manipulations.
- `docopt`: Used for command-line argument parsing.
- `matplotlib`: Used for plotting and visualization.

1.2 Routines developed

Two specific routines are developed, (i) Image Extraction Routine & (2) Dynamical Feature Extraction Routine. Details of code workflow, functions, main functionalities and code snippets for these routines are provided in Sec. 2.1 and Sec. 2.2, respectively. Next we provide well-known definitions of mathematical entities such as `Circularity` and `fractal dimension` that are implemented in our core code capabilities. The general idea of the code is to extract the images of biological organisms capturing their growth dynamics (see Secs. [3 -8] for details) and then implement and extract the parameters of the aforementioned mathematical entities.

1.3 `Circularity`

Growth morphology can be characterized by mathematical quantifier such as `Circularity` which is dependent on parameters such as the `area` and `perimeter` of the biological organism. Often used as a shape factor, `Circularity` is defined as,

$$\text{Circularity} = 4\pi \left[\frac{\text{Area}}{\text{Perimeter}^2} \right] \tag{1}$$

Circularity ~ 1 implies a circular/radial morphology. The core code capability in `PyPETANA` allows for the computation of the `area`, `perimeter` and the corresponding `Circularity` of the biological organism. Morphological changes related to orientation or chirality such as *symmetrical/asymmetrical* growth features as a function of time can also be explored within the core code functionalities. Details pertaining to the implementation of `Circularity` is given in Secs. [9 - 15].

1.4 `Fractal Dimension`

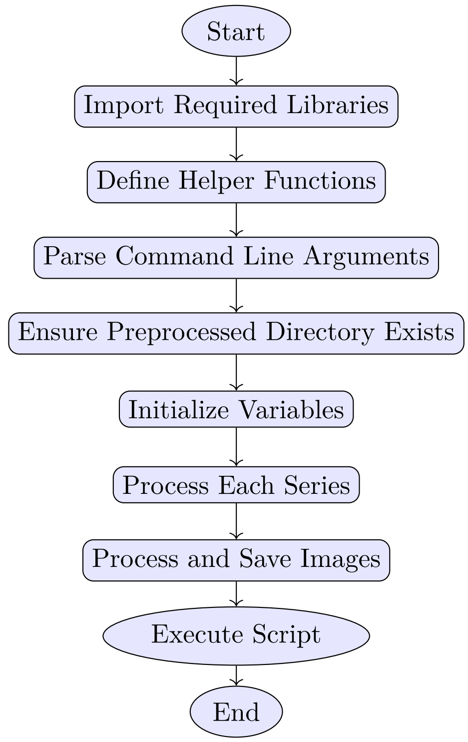
The complexity of geometrical shapes are quantified and characterized by `fractal dimension`(2). We use a `Box-Counting` algorithm to compute the `fractal dimension`. Details pertaining to the implementation of the box-algorithm and computation of fractal dimension is given in Secs. [9 - 15].

2 Code Features & Routines

2.1 Image Extraction Routines

The script targets a series of images. It extracts Petri dishes, masks objects, and writes new processed images for each dish. Additionally, it rotates Petri dishes based on the position and count of objects on petridishes to align them correctly. Below we provide a general workflow gives an overview of the main steps and functionality of the script.

2.1.1 Code Workflow



2.1.2 Import Required Libraries

Import necessary libraries: `os, glob, cv2, numpy, and docopt`.

2.1.3 Define Helper Functions

- `ensure_directory_exists(directory)`: Ensures the specified directory exists.
- `apply_curve_transformation(image, transformation_function, lower_threshold, upper_threshold)`: Applies a transformation to the image.
- `linear_transformation(pixel_values)`: Example of a simple linear transformation.
- `piecewise_linear_transformation(pixel_values_in, lower_threshold, upper_threshold)`: Applies a piecewise linear transformation to the image.
- `eliminate_duplicate_circles(circles, center_threshold)`: Eliminates duplicate circles based on a center threshold.
- `extract_radius_mask(filepath, save=None)`: Extracts and processes the radii mask from an image.

Detailed explanations of functions given in Secs. [3 - 7].

2.1.4 Main Functionality

- *Parse Command Line Arguments:* Use `docopt` to parse command line arguments and options.
- *Ensure Preprocessed Directory Exists:* Create a directory to store preprocessed images if it doesn't already exist.
- *Initialize Variables:* Set initial values for `min_radius` and `min_count`.
- *Process Each Series:*
  - Loop through directories to identify image series.
  - For each identified series, extract relevant information such as date, scanner, experiment number, and image files.
  - Extract radius mask and other relevant information for each series.
  - Update `min_radius` and `min_count` based on the processed series.
- *Process and Save Images*
  - Loop through the identified series.
  - For each series, process and save images based on extracted masks and transformations.
  - Apply transformations and rotations as needed.
  - Save the processed images to the preprocessed directory.

2.1.5 Execute the Script

If the script is run as the main module, execute the command line parsing and start processing the image series.

2.1.6 Core Code Functions & Snippets

- **Ensure Directory Exists:** This function ensures a specified directory exists, creating it if necessary.

```python
def ensure_directory_exists(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)
```

- **Apply Curve Transformation:** Applies a specified transformation function to an image.

```
def apply_curve_transformation(image, transformation_function, lower_threshold,
↪   upper_threshold):
    float_image = image.astype(np.float32) / 255.0
    transformed_image = transformation_function(float_image, lower_threshold,
    ↪   upper_threshold)
    transformed_image = np.clip(transformed_image, 0, 1) * 255
    transformed_image = transformed_image.astype(np.uint8)
    return transformed_image
```

- **Linear Transformation:** A simple linear transformation example.

```
def linear_transformation(pixel_values):
    return 1 - pixel_values
```

- **Piecewise Linear Transformation:** Applies a piecewise linear transformation to pixel values.

```
def piecewise_linear_transformation(pixel_values_in, lower_threshold,
↪   upper_threshold):
    invert=True
    pixel_values = 1 - pixel_values_in if invert else pixel_values_in
    clipped_values = np.clip(pixel_values, lower_threshold, upper_threshold)
    scale = 1.0 / (upper_threshold - lower_threshold)
    transformed = (clipped_values - lower_threshold) * scale
    transformed = 1 - transformed if invert else transformed
    return transformed
```

- **Eliminate Duplicate Circles:** Eliminates duplicate circles based on a center threshold.

```
def eliminate_duplicate_circles(circles, center_threshold):
    unique_indices = []
    for index, circle in enumerate(circles):
        x, y, r = circle
        duplicate_found = False
```

```python
        for index2 in unique_indices:
            ux, uy, ur = circles[index2]
            distance = np.sqrt((ux - x)**2 + (uy - y)**2)
            if distance < center_threshold:
                duplicate_found = True
                break
        if not duplicate_found:
            unique_indices.append(index)
    return np.array(unique_indices, dtype=int)
```

- **Extract Radius Mask:** Extracts and masks Petri dishes and weights from an image.

```python
def extract_radius_mask(filepath, save=None):
    frame = cv2.imread(filepath)
    modified_frame = apply_curve_transformation(frame,
    ↪  piecewise_linear_transformation, 0.00, 0.70)
    gray = cv2.cvtColor(modified_frame, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (9, 9), 2)
    dish_circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT, dp=1,
    ↪  minDist=300,
                                   param1=20, param2=29, minRadius=1150,
                                   ↪  maxRadius=1350)
    unique_indices = eliminate_duplicate_circles(dish_circles[0],
    ↪  center_threshold=300)
    dish_circles = np.round(dish_circles[0, unique_indices]).astype("int")
    height, width, channels = frame.shape
    left_dish_circles = dish_circles[np.where(dish_circles[:,0] < 6*width/10)]
    right_dish_circles = dish_circles[np.where(dish_circles[:,0] >= 6*width/10)]
    left_dish_circles = left_dish_circles[np.argsort(left_dish_circles[:,1])]
    right_dish_circles = right_dish_circles[np.argsort(right_dish_circles[:,1])]
    dish_circles = np.concatenate((left_dish_circles, right_dish_circles),
    ↪  axis=0)
    mask = np.zeros_like(frame)
    for (x, y, r) in dish_circles:
        cv2.circle(mask, (x, y), r, (255, 255, 255), -1)
    masked_frame = cv2.bitwise_and(modified_frame, mask)
    blurred = cv2.cvtColor(masked_frame, cv2.COLOR_BGR2GRAY)
    weight_circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT, dp=1,
    ↪  minDist=300,
```

```
                                   param1=20, param2=29, minRadius=10,
                                   ↪  maxRadius=100)
    if weight_circles is not None:
        weight_circles = np.round(weight_circles[0, :]).astype("int")
        unique_indices = eliminate_duplicate_circles(weight_circles,
        ↪  center_threshold=30)
        weight_circles = np.round(weight_circles[unique_indices]).astype("int")
        for (x, y, r) in weight_circles:
            cv2.circle(mask, (x, y), r, (0, 0, 255), -1)
    rot_angles = []
    for (x, y, r) in dish_circles:
        angle = calculate_rotation_angle(x, y)  # Assume this function is
        ↪  implemented elsewhere
        rot_angles.append(angle)
    if save is not None:
        cv2.imwrite(save, masked_frame)
    return mask, dish_circles, rot_angles
```

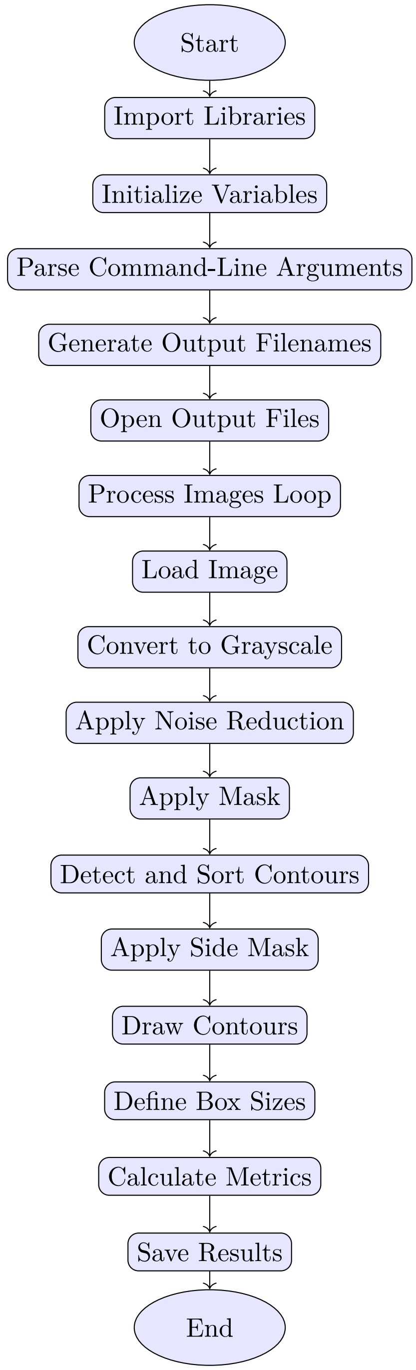
- **Process Series** Processes a series of images to extract and save preprocessed images.

```
def process_series(series_prefix, series, min_radius, min_count):
    for dish_coords in series:
        print("Processing:", dish_coords)
    first_good_image = None
    for frame in files:
        if first_good_image is None:
            first_good_image = cv2.imread(frame)
        frame_number = int(re.search(r'(\d+)', frame).group(0))
        if frame_number < 3:
            continue
        output_path = os.path.join(output_dir,
        ↪  f"{series_prefix}_{frame_number}.png")
        mask, dish_circles, rot_angles = extract_radius_mask(frame,
        ↪  save=output_path)
        if len(dish_circles) < min_count:
            continue
        print("Processed:", frame)
```

Detailed explanations of functions and main execution block are given in Secs. [3 - 8].

2.2 Dynamical Parameters Extraction Routines

2.2.1 Code Workflow



2.2.2 Import Required Libraries

Import necessary libraries: `os`, `glob`, `cv2`, `numpy`, and `docopt`, `matplotlib.pyplot`.

2.2.3 Define Helper Functions

- `ensure_directory_exists(directory)`: Ensures that the specified directory exists; creates it if it doesn't.
- `is_contour_inside(contour1, contour2)`: Checks if all points of `contour1` are inside `contour2`.
- `generate_points(last_point, point, N)`: Generates `N` evenly spaced points between `last_point` and `point`.
- `is_contour_partially_inside(contour1, contour2, side_width)`: Checks if `contour1` is partially inside `contour2`, including points generated between contour points.
- `is_contour_intersecting(contour1, contour2, side_width)`: Checks if `contour1` intersects `contour2`, including points generated between contour points.
- `points_in_contour(points, contour1)`: Gathers points that are inside `contour1`.
- `find_and_sort_contours(gray_image, min_thresh, max_thresh, last_center)`: Finds and sorts contours in the image based on area and proximity to the last center.

Detailed explanations of functions given in Secs. [9 - 14].

2.2.4 Main Functionality

- *Import and Initialization*
  - Check if the script is run directly.
  - Parse command-line arguments using `docopt`.
  - Store and convert command-line arguments to variables.
  - Ensure the existence of the output directory.
- *Output File Naming*
  - Construct output filenames based on the first image path.
  - Prefix filenames with `side` if specified.
- *File Handling and Initialization*
  - Open output files for writing.
  - Define headers for the output files.
  - Initialize variables for processing.
- *Image Processing Loop*

  - Loop through each image in `image_paths`.
  - Load the image and convert it to grayscale.
  - Initialize the center of the image if not already set.

- *Noise Reduction and Mask Application*
  - Apply a median filter to reduce noise.
  - Create and apply a mask to remove edges from the image.

- *Contour Detection and Sorting*
  - Find and sort contours for the first image.
  - Generate a vertical split line if `side` is specified.

- *Side Mask Application*
  - Apply a mask to remove either the left or right side of the image based on `side`.

- *Contour Sorting and Drawing*
  - Find and sort contours again after applying the side mask.
  - Draw the sorted contours on the image.

- *Quadrant Definition and Box Sizes*
  - Define quadrants and groups based on image height and `side`.
  - Choose box sizes based on the dimensions of the image.

- *Contour Processing and Metrics Calculation*
  - Create a mask for biological entity and initialize metrics.
  - Process each contour to compute area, perimeter, and other metrics.
  - Draw contours and calculate fractal dimensions and circularity.
  - Format the results for output.

- *Saving Output and Processed Image*
  - Save the processed image to the output directory.
  - Write computed metrics and data to the output files.

2.2.5 Core Code Functions & Snippets

- **Ensure Directory Exists:** This function ensures a specified directory exists, creating it if necessary.

```
def ensure_directory_exists(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)
```

- **Check if Contour is Inside:** Checks if all points of one contour are inside another contour.

```
def is_contour_inside(contour1, contour2):
    for i in range(len(contour1)):
        point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
        if cv2.pointPolygonTest(contour2, point, False) < 0:
            return False
    return True
```

- **Generate Points Between Two Points:** Generates a specified number of points between two given points.

```
def generate_points(last_point, point, N):
    x1, y1 = last_point
    x2, y2 = point
    points = []
    for i in range(1, N + 1):
        t = i / (N + 1)
        x = x1 + t * (x2 - x1)
        y = y1 + t * (y2 - y1)
        points.append((int(round(x)), int(round(y))))
    return points
```

- **Check if Contour is Partially Inside:** Checks if any part of one contour is inside another contour.

```python
def is_contour_partially_inside(contour1, contour2, side_width):
    last_point = (int(contour1[-1][0][0]), int(contour1[-1][0][1]))
    for i in range(len(contour1)):
        point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
        if cv2.pointPolygonTest(contour2, point, False) > 0:
            return True
        points = generate_points(last_point, point, int(side_width / 2) + 1)
        for new_point in points:
            if cv2.pointPolygonTest(contour2, new_point, False) > 0:
                return True
        last_point = point
    return False
```

- **Check if Contour is Intersecting:** Checks if any part of one contour intersects with another contour.

```python
def is_contour_intersecting(contour1, contour2, side_width):
    last_point = (int(contour1[-1][0][0]), int(contour1[-1][0][1]))
    for i in range(len(contour1)):
        point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
        if cv2.pointPolygonTest(contour2, point, False) == 0:
            return True
        points = generate_points(last_point, point, side_width)
        for new_point in points:
            if cv2.pointPolygonTest(contour2, new_point, False) == 0:
                return True
        last_point = point
    return False
```

- **Gather Points Inside Contour:** Gathers points that lie inside a given contour.

```python
def points_in_contour(points, contour1):
    new_points = []
    for i in range(len(points)):
        point = points[i]
        if cv2.pointPolygonTest(contour1, point, False) >= 0:
```

```
            new_points.append(point)
    return new_points
```

- **Find and Sort Contours:** Finds and sorts contours based on area and distance from the last center.

```
def find_and_sort_contours(gray_image, min_thresh, max_thresh, last_center):
    range_threshold_image = cv2.inRange(gray_image, min_thresh, max_thresh)
    range_threshold_image = cv2.medianBlur(range_threshold_image, 9)
    kernel_size = 9
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    range_threshold_image = cv2.morphologyEx(range_threshold_image,
    ↪   cv2.MORPH_CLOSE, kernel)
    contours, _ = cv2.findContours(range_threshold_image, cv2.RETR_TREE,
    ↪   cv2.CHAIN_APPROX_SIMPLE)
    sorted_contours = sorted(contours, key=cv2.contourArea, reverse=True)
    grouped_contours = []
    weights = []
    index = 0
    while index < len(sorted_contours):
        contour = sorted_contours[index]
        contour_list = []
        contour_list.append(contour)
        M = cv2.moments(contour)
        if M["m00"] != 0:
            centerX = int(M["m10"] / M["m00"])
            centerY = int(M["m01"] / M["m00"])
        else:
            centerX, centerY = 0, 0
        area = cv2.contourArea(contour)
        if area > 0:
            perimeter = cv2.arcLength(contour, True)
            index_next = index + 1
            while index_next < len(sorted_contours):
                contour_next = sorted_contours[index_next]
                area2 = cv2.contourArea(contour_next)
                if area2 > 0:
                    perimeter2 = cv2.arcLength(contour_next, True)
                    if is_contour_inside(contour_next, contour):
```

```python
                    if args['--include_holes']:
                        perimeter += perimeter2
                        area -= area2
                        contour_list.append(contour_next)
                    sorted_contours.pop(index_next)
                else:
                    index_next += 1
            else:
                sorted_contours.pop(index_next)
        grouped_contours.append([np.array([centerX, centerY]), area,
        ↪  perimeter, contour_list])
        weights.append(np.sqrt((centerX - last_center[0])**2 + (centerY -
        ↪  last_center[1])**2) / area)
        index += 1
    else:
        sorted_contours.pop(index)
sorted_indices = np.argsort(np.array(weights))
last_center = grouped_contours[sorted_indices[0]][0]
return sorted_contours, grouped_contours, sorted_indices, last_center,
↪  weights
```

- **Main Execution Block:** Handles command-line arguments, processes images, and saves results.

```python
if __name__ == '__main__':
    args = docopt(__doc__)
    image_paths = sorted(args['<filenames>'])
    min_thresh = int(args['--min_thresh'])
    max_thresh = int(args['--max_thresh'])
    side = args['--side']

    ensure_directory_exists('PROCESSED')

    fileout_filename = 'nh_fracdim_' +
    ↪  '_'.join(image_paths[0].split('/')[-1].replace('.jpg','').split('_')[:-1])
    ↪  + '.dat'
    if side in ['left', 'right']:
        fileout_filename = side + '_' + fileout_filename
```

```
    fileout_filename_frac = 'nh_fracdim_' +
    ↪   '_'.join(image_paths[0].split('/')[-1].replace('.jpg','').split('_')[:-1])
    ↪   + '_frac.dat'
    with open('PROCESSED' + '/' + fileout_filename, 'w') as fileout,
    ↪   open('PROCESSED' + '/' + fileout_filename_frac, 'w') as fileout_frac:
        output = (f'#{"frame":>8}  {"area":>15}  {"perimeter":>15}
        ↪   {"circularity":>15}  {"fracdim":>15}\r\n')
        output_frac = (f'#{"frame":>8}  {"BSR":>15}  {"NA":>15}\r\n')
        fileout.write(output)
        fileout_frac.write(output_frac)

        for image_path in image_paths:
            gray_image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
            sorted_contours, grouped_contours, sorted_indices, last_center,
            ↪   weights = find_and_sort_contours(gray_image, min_thresh,
            ↪   max_thresh, last_center)

            for contour_index in range(len(sorted_indices)):
                contour_info = grouped_contours[sorted_indices[contour_index]]
                contour = contour_info[3]
                if len(contour) > 0:
                    area = contour_info[1]
                    perimeter = contour_info[2]
                    circularity = (4 * np.pi * area) / (perimeter * perimeter)

                    ↪   fileout.write(f'{image_path.split("/")[-1].replace(".jpg","")}:>8}
                    ↪   {area:>15.2f}  {perimeter:>15.2f}  {circularity:>15.2f}
                    ↪   {np.log(area/perimeter):>15.2f}\r\n')

                    frac_dim = np.log(area / perimeter) / np.log(side_width)

                    ↪   fileout_frac.write(f'{image_path.split("/")[-1].replace(".jpg","")}:>8}
                    ↪   {frac_dim:>15.2f}  {frac_dim:>15.2f}\r\n')
```

Detailed explanations of functions and main execution block are given in Secs. [9 - 14 and 15].

3 Image Extraction Function: apply_curve_transformation

3.1 Description

The apply_curve_transformation function applies a specified transformation function to an image. It ensures that the image is in a float format to avoid clipping during the transformation, applies the transformation, clips the values to the 0-1 range, and converts the image back to an 8-bit format.

3.2 Function Definition

```
def apply_curve_transformation(image, transformation_function, lower_threshold,
↪ upper_threshold):
    # Ensure image is in a float format to avoid clipping during the transformation
    float_image = image.astype(np.float32) / 255.0

    # Apply the transformation function
    transformed_image = transformation_function(float_image, lower_threshold,
    ↪ upper_threshold)

    # Clip values to the 0-1 range and convert back to an 8-bit format
    transformed_image = np.clip(transformed_image, 0, 1) * 255
    transformed_image = transformed_image.astype(np.uint8)

    return transformed_image
```

3.3 Function Explanation

3.3.1 Step-by-Step Breakdown

Function 1: Ensure Image is in Float Format

Ensure the image is in a float format to avoid clipping during the transformation.

```
# Ensure image is in a float format to avoid clipping during the transformation
float_image = image.astype(np.float32) / 255.0
```

Explanation: The image is first converted to a floating-point format by dividing the pixel values by 255. This normalization step ensures that the pixel values range between 0 and 1. This is important to avoid any clipping issues that might occur during the transformation process.

Function 2: Apply the Transformation Function

Apply the specified transformation function.

```
# Apply the transformation function
transformed_image = transformation_function(float_image, lower_threshold,
↪  upper_threshold)
```

Explanation: The transformation function is applied to the normalized image. The `transformation_function` is passed along with the `float_image`, `lower_threshold`, and `upper_threshold` parameters. This function modifies the pixel values according to the specified transformation logic.

Function 3: Clip Values and Convert Back to 8-Bit Format

Clip the transformed image values to the 0-1 range and convert back to an 8-bit format.

```
# Clip values to the 0-1 range and convert back to an 8-bit format
transformed_image = np.clip(transformed_image, 0, 1) * 255
transformed_image = transformed_image.astype(np.uint8)
```

Explanation: After applying the transformation, the pixel values are clipped to the range [0, 1] to ensure they stay within valid bounds. The image is then multiplied by 255 to convert it back to the 8-bit format and cast back to an unsigned 8-bit integer type (`np.uint8`).

3.4 Conclusion

The `apply_curve_transformation` function is designed to facilitate image preprocessing by applying a specified transformation function to an image. It ensures the image is in a float format to avoid clipping during the transformation, applies the transformation, and then clips and converts the image back to an 8-bit format.

4 Image Extraction Function: `piecewise_linear_transformation`

4.1 Description

The `piecewise_linear_transformation` function applies a piecewise linear transformation to pixel values. It inverts the pixel values if specified, clips them at given thresholds, scales them linearly to a range of [0, 1], and optionally inverts the scaled values back.

4.2 Function Definition

```python
def piecewise_linear_transformation(pixel_values_in, lower_threshold,
↪ upper_threshold):
    invert = True

    # Invert it
    pixel_values = 1 - pixel_values_in if invert else pixel_values_in

    # Clip values at the lower and upper thresholds
    clipped_values = np.clip(pixel_values, lower_threshold, upper_threshold)

    # Linear scaling
    # Scale the range [lower_threshold, upper_threshold] to [0, 1]
    scale = 1.0 / (upper_threshold - lower_threshold)
    transformed = (clipped_values - lower_threshold) * scale

    # Invert it back
    transformed = 1 - transformed if invert else transformed

    return transformed
```

4.3 Function Explanation

4.3.1 Step-by-Step Breakdown

Function 1: Invert Pixel Values

Invert the pixel values if the `invert` flag is set to `True`.

```
# Invert it
pixel_values = 1 - pixel_values_in if invert else pixel_values_in
```

Explanation: If the `invert` flag is set to `True`, the pixel values are inverted by subtracting them from 1. This step prepares the pixel values for subsequent transformations.

Function 2: Clip Values

Clip the pixel values to lie within the specified `lower_threshold` and `upper_threshold`.

```
# Clip values at the lower and upper thresholds
clipped_values = np.clip(pixel_values, lower_threshold, upper_threshold)
```

Explanation: The pixel values are clipped to ensure they fall within the specified threshold range. This prevents values outside the specified bounds from affecting the linear scaling.

Function 3: Linear Scaling

Scale the clipped pixel values from the range `[lower_threshold, upper_threshold]` to `[0, 1]`.

```
# Linear scaling
# Scale the range [lower_threshold, upper_threshold] to [0, 1]
scale = 1.0 / (upper_threshold - lower_threshold)
transformed = (clipped_values - lower_threshold) * scale
```

Explanation: The clipped pixel values are scaled to the [0, 1] range. This is done by first computing a scaling factor and then applying this factor to adjust the values accordingly.

Function 4: Invert Back

Invert the scaled pixel values back if the `invert` flag is set to `True`.

```
# Invert it back
transformed = 1 - transformed if invert else transformed
```

Explanation: If the `invert` flag was set to `True` initially, the scaled values are inverted back by subtracting them from 1. This restores the original inversion state if needed.

4.4 Conclusion

The `piecewise_linear_transformation` function processes pixel values by applying an inversion, clipping, linear scaling, and optional inversion. This function is useful for adjusting image contrast and brightness through piecewise linear transformations.

5 Image Extraction Function: `eliminate_duplicate_circles`

5.1 Description

The `eliminate_duplicate_circles` function removes duplicate circles from a list based on their centers. It uses a distance threshold to determine if two circles are considered duplicates. Only unique circles are retained, and their indices are returned.

5.2 Function Definition

```
def eliminate_duplicate_circles(circles, center_threshold):
    unique_indices = []

    for index, circle in enumerate(circles):
        x, y, r = circle
        duplicate_found = False

        for index2 in unique_indices:
            ux, uy, ur = circles[index2]
            distance = np.sqrt((ux - x)**2 + (uy - y)**2)

            if distance < center_threshold:
                duplicate_found = True
                break

        if not duplicate_found:
            unique_indices.append(index)
```

```
    return np.array(unique_indices, dtype=int)
```

5.3 Function Explanation

5.3.1 Step-by-Step Breakdown

Function 1: Initialize Unique Indices

Initialize an empty list to store indices of unique circles.

```
unique_indices = []
```

Explanation: An empty list `unique_indices` is created to keep track of the indices of circles that are identified as unique. This list will be used to filter out duplicate circles.

Function 2: Iterate Through Circles

Iterate through each circle and determine if it is a duplicate.

```
for index, circle in enumerate(circles):
    x, y, r = circle
    duplicate_found = False
```

Explanation: The function loops through each circle in the `circles` list. For each circle, it initializes a flag `duplicate_found` to `False` to check if the circle is a duplicate.

Function 3: Check for Duplicates

Compare the current circle with previously found unique circles to check for duplicates.

```
for index2 in unique_indices:
    ux, uy, ur = circles[index2]
    distance = np.sqrt((ux - x)**2 + (uy - y)**2)
```

```
        if distance < center_threshold:
            duplicate_found = True
            break
```

Explanation: The function compares the current circle's center with those of previously identified unique circles. It calculates the distance between centers and checks if it is less than the `center_threshold`. If a duplicate is found, the `duplicate_found` flag is set to `True` and the loop exits.

Function 4: Store Unique Circle

If the circle is not a duplicate, add its index to the `unique_indices` list.

```
if not duplicate_found:
    unique_indices.append(index)
```

Explanation: If no duplicates are found for the current circle, its index is added to the `unique_indices` list, marking it as unique.

Function 5: Return Unique Circle Indices

Return an array of indices for the unique circles.

```
return np.array(unique_indices, dtype=int)
```

Explanation: The function converts the `unique_indices` list to a NumPy array of integer type and returns it. This array contains the indices of the circles that are considered unique.

5.4 Conclusion

The `eliminate_duplicate_circles` function is designed to identify and remove duplicate circles based on their centers. It ensures that only unique circles are retained and provides their indices in the output array.

6 Image Extraction Function:extract_radius_mask

6.1 Description

The `extract_radius_mask` function processes an image to identify and mask circles representing dishes and weights. It extracts circles, filters out duplicates, sorts them, and applies transformations to create a final image with masked dishes and weights.

6.2 Function Definition

```
def extract_radius_mask(filepath, save=None):
    frame = cv2.imread(filepath)
    modified_frame = apply_curve_transformation(frame,
    ↪  piecewise_linear_transformation, 0.00, 0.70)
    gray = cv2.cvtColor(modified_frame, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (9, 9), 2)

    dish_circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT, dp=1, minDist=300,
                           param1=20, param2=29, minRadius=1150, maxRadius=1350)
    unique_indices = eliminate_duplicate_circles(dish_circles[0],
    ↪  center_threshold=300)
    dish_circles = np.round(dish_circles[0, unique_indices]).astype("int")

    # sort the dish circles left to right first, then top to bottom
    height, width, channels = frame.shape
    left_dish_circles = dish_circles[np.where(dish_circles[:,0] < 6*width/10)]
    right_dish_circles = dish_circles[np.where(dish_circles[:,0] >= 6*width/10)]
    left_dish_circles = left_dish_circles[np.argsort(left_dish_circles[:,1])]
    right_dish_circles = right_dish_circles[np.argsort(right_dish_circles[:,1])]
    dish_circles = np.concatenate((left_dish_circles, right_dish_circles), axis=0)

    # Draw the filled dish circle on the mask as white (keeping)
    mask = np.zeros_like(frame)
    for (x, y, r) in dish_circles:
        cv2.circle(mask, (x, y), r, (255, 255, 255), -1)

    # Mask the image before searching for the weights
    masked_frame = cv2.bitwise_and(modified_frame, mask)
    blurred = cv2.cvtColor(masked_frame, cv2.COLOR_BGR2GRAY)

    weight_circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT_ALT, dp=1,
    ↪  minDist=1,
```

```
                              param1=60, param2=0.30, minRadius=90,
                              ↪  maxRadius=140)[0]

# sort by radius here
radii_indices = np.argsort(weight_circles[:,2])[::-1]
weight_circles = weight_circles[radii_indices]

unique_indices = eliminate_duplicate_circles(weight_circles,
↪  center_threshold=180)
weight_circles = np.round(weight_circles[unique_indices]).astype("int")

# Draw the filled weight circle on the mask as black (discard)
for (x, y, r) in weight_circles:
    in_center = False
    for (x2, y2, r2) in dish_circles:
        if np.sqrt((x-x2)**2+(y-y2)**2) < 800:
            in_center = True
            break
    if in_center == False:
        cv2.circle(mask, (x, y), r+20, (0, 0, 0), -1)

# Now for each dish we calculate necessary rotation to place the
# three masses on the right (centered at 0 radians) and the single
# mass on the left (centered at pi radians)
rot_angles = []
for i, (x1, y1, r2) in enumerate(dish_circles):
    left_weights_x = []
    left_weights_y = []
    right_weights_x = []
    right_weights_y = []
    for j, (x2, y2, r2) in enumerate(weight_circles):
        if np.sqrt((x1-x2)**2+(y1-y2)**2) < 800:
            continue
        if (x1-x2)**2+(y1-y2)**2 <= 1250*1250:
            if x2 < x1:
                left_weights_x.append(x2)
                left_weights_y.append(y2)
            else:
                right_weights_x.append(x2)
                right_weights_y.append(y2)
    if len(left_weights_x) > 0 and len(right_weights_x) > 0:
```

```
            xl, yl = np.average(left_weights_x), np.average(left_weights_y)
            xr, yr = np.average(right_weights_x), np.average(right_weights_y)
            theta = np.arctan2(yr-y1,xr-x1)
            if len(left_weights_x) > len(right_weights_x):
                theta += np.pi
            rot_angles.append(theta)
        else:
            rot_angles.append(0)

    # optionally output the masked first frame
    if save is not None:
        final_image = np.zeros((height, width * 2, channels), dtype=np.uint8)

        final_image[0:height, 0:width] = frame
        masked_frame = cv2.bitwise_and(frame, mask)
        final_image[0:height, width:2*width] = masked_frame
        for i, (x, y, r) in enumerate(dish_circles):
            # extract the single dish
            dish = masked_frame[y - r:y + r, x - r:x + r]
            # rotate it
            rotation_matrix = cv2.getRotationMatrix2D((int(r),int(r)),
            ↪   rot_angles[i]*180/np.pi, 1)
            rotated_dish = cv2.warpAffine(dish, rotation_matrix, (2*int(r),
            ↪   2*int(r)))
            # re-insert it
            final_image[y - r:y + r, x - r + width:x + r + width] = rotated_dish
            cv2.putText(final_image, f'd{i+1:02}', (x, y), cv2.FONT_HERSHEY_SIMPLEX,
            ↪   4, (255, 255, 255), 2)
            cv2.putText(final_image, f'r{rot_angles[i]*180/np.pi:5.2}', (x, y+100),
            ↪   cv2.FONT_HERSHEY_SIMPLEX, 4, (255, 255, 255), 2)
        cv2.imwrite(save, final_image)

    return mask, dish_circles, rot_angles
```

6.3 Function Explanation

6.3.1 Step-by-Step Breakdown

Function 1: Read and Transform Image

Read the image from the specified file path and apply a curve transformation to it.

```
frame = cv2.imread(filepath)
modified_frame = apply_curve_transformation(frame, piecewise_linear_transformation,
↪  0.00, 0.70)
```

Explanation: The function reads the image from the specified `filepath` and applies a curve transformation to it using `apply_curve_transformation`. The transformed image is stored in `modified_frame`.

Function 2: Convert and Blur Image

Convert the transformed image to grayscale and apply Gaussian blur.

```
gray = cv2.cvtColor(modified_frame, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (9, 9), 2)
```

Explanation: The image is converted to grayscale and then blurred using Gaussian blur to prepare it for circle detection.

Function 3: Detect and Sort Dish Circles

Detect dish circles using the Hough Circle Transform, filter duplicates, and sort them.

```
dish_circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT, dp=1, minDist=300,
                               param1=20, param2=29, minRadius=1150, maxRadius=1350)
unique_indices = eliminate_duplicate_circles(dish_circles[0], center_threshold=300)
dish_circles = np.round(dish_circles[0, unique_indices]).astype("int")

height, width, channels = frame.shape
```

```python
left_dish_circles = dish_circles[np.where(dish_circles[:,0] < 6*width/10)]
right_dish_circles = dish_circles[np.where(dish_circles[:,0] >= 6*width/10)]
left_dish_circles = left_dish_circles[np.argsort(left_dish_circles[:,1])]
right_dish_circles = right_dish_circles[np.argsort(right_dish_circles[:,1])]
dish_circles = np.concatenate((left_dish_circles, right_dish_circles), axis=0)
```

Explanation: Detect circles representing dishes, filter out duplicates, and sort them first by their x-coordinate and then by y-coordinate.

Function 4: Create Dish Mask

Create a mask for the detected dish circles and apply it to the modified image.

```python
mask = np.zeros_like(frame)
for (x, y, r) in dish_circles:
    cv2.circle(mask, (x, y), r, (255, 255, 255), -1)

masked_frame = cv2.bitwise_and(modified_frame, mask)
blurred = cv2.cvtColor(masked_frame, cv2.COLOR_BGR2GRAY)
```

Explanation: Create a binary mask for the dish circles and apply it to the image. This mask is used to isolate the dish regions for weight detection.

Function 5: Detect Weights and Sort

Detect weights using the Hough Circle Transform, sort them by radius, and filter duplicates.

```python
weight_circles = cv2.HoughCircles(blurred, cv2.HOUGH_GRADIENT_ALT, dp=1, minDist=1,
                              param1=60, param2=0.30, minRadius=90,
                              ↪  maxRadius=140)[0]

radii_indices = np.argsort(weight_circles[:,2])[::-1]
weight_circles = weight_circles[radii_indices]

unique_indices = eliminate_duplicate_circles(weight_circles, center_threshold=180)
weight_circles = np.round(weight_circles[unique_indices]).astype("int")
```

Explanation: Detect circles representing weights, sort them by their radius, and filter out duplicates.

Function 6: Update Mask for Weights

Update the mask to discard weight circles that overlap with dish circles.

```
for (x, y, r) in weight_circles:
    in_center = False
    for (x2, y2, r2) in dish_circles:
        if np.sqrt((x-x2)**2+(y-y2)**2) < 800:
            in_center = True
            break
    if in_center == False:
        cv2.circle(mask, (x, y), r+20, (0, 0, 0), -1)
```

Explanation: Update the mask to exclude weight circles that are close to dish circles by drawing them in black.

Function 7: Calculate Rotation Angles

Calculate the rotation angles for each dish to align weights correctly.

```
rot_angles = []
for i, (x1, y1, r2) in enumerate(dish_circles):
    left_weights_x = []
    left_weights_y = []
    right_weights_x = []
    right_weights_y = []
    for j, (x2, y2, r2) in enumerate(weight_circles):
        if np.sqrt((x1-x2)**2+(y1-y2)**2) < 800:
            continue
        if (x1-x2)**2+(y1-y2)**2 <= 1250*1250:
            if x2 < x1:
                left_weights_x.append(x2)
                left_weights_y.append(y2)
            else:
                right_weights_x.append(x2)
```

```
                right_weights_y.append(y2)
    if len(left_weights_x) > 0 and len(right_weights_x) > 0:
        xl, yl = np.average(left_weights_x), np.average(left_weights_y)
        xr, yr = np.average(right_weights_x), np.average(right_weights_y)
        theta = np.arctan2(yr-y1,xr-x1)
        if len(left_weights_x) > len(right_weights_x):
            theta += np.pi
        rot_angles.append(theta)
    else:
        rot_angles.append(0)
```

Explanation: Calculate the rotation angles needed for each dish to align weights properly based on their positions.

Function 8: Save and Return Results

Optionally save the final image and return the mask, dish circles, and rotation angles.

```
if save is not None:
    final_image = np.zeros((height, width * 2, channels), dtype=np.uint8)

    final_image[0:height, 0:width] = frame
    masked_frame = cv2.bitwise_and(frame, mask)
    final_image[0:height, width:2*width] = masked_frame
    for i, (x, y, r) in enumerate(dish_circles):
        dish = masked_frame[y - r:y + r, x - r:x + r]
        rotation_matrix = cv2.getRotationMatrix2D((int(r),int(r)),
        ↪  rot_angles[i]*180/np.pi, 1)
        rotated_dish = cv2.warpAffine(dish, rotation_matrix, (2*int(r), 2*int(r)))
        final_image[y - r:y + r, x - r + width:x + r + width] = rotated_dish
        cv2.putText(final_image, f'd{i+1:02}', (x, y), cv2.FONT_HERSHEY_SIMPLEX, 4,
        ↪  (255, 255, 255), 2)
        cv2.putText(final_image, f'r{rot_angles[i]*180/np.pi:5.2}', (x, y+100),
        ↪  cv2.FONT_HERSHEY_SIMPLEX, 4, (255, 255, 255), 2)
    cv2.imwrite(save, final_image)

return mask, dish_circles, rot_angles
```

Explanation: If a `save` path is provided, save the final image showing both the original and masked images. The function returns the mask, dish circles, and rotation angles.

7 Image Extraction Function: `process_series`

7.1 Overview

The `process_series` function processes a series of images, applying transformations and saving the results. It uses the following parameters:

- `series_prefix`: Prefix for output file names.
- `series`: Tuple containing series information including file paths, mask, dish coordinates, etc.
- `min_radius`: Minimum radius for Petri dish extraction.
- `min_count`: Minimum count for filtering files.

7.2 Function Definition

- Function Header

```
def process_series(series_prefix, series, min_radius, min_count):
```

- Unpack Series Information

```
date, scanner, experiment, files, mask, dish_coords, dish_angles, first_frame,
↪ last_frame = series
for i, (x, y, r) in enumerate(dish_coords):
↪ print(f'output/{series_prefix}_{i+1:02}_d03.jpg')
```

Explanation: Unpack the tuple `series` into its components and print file paths for each dish coordinate.

- Check Dish Count

```
if len(dish_coords) < 6:
    print(f'WARNING: PREPROCESSED/{series_prefix}_%d_%03d has {len(dish_coords)}
    ↪ dishes, expected 6')
```

Explanation: Issue a warning if the number of detected dishes is less than 6. Depending on the number of dishes on the raw image, change the number.

- Get the First Good Image

```
last_frame = {}
for filepath in files:
    try:
        frame = cv2.imread(filepath)
        masked_frame = cv2.bitwise_and(frame, mask)
        for i, (x, y, r) in enumerate(dish_coords):
            x, y, r = max(x, min_radius), max(y, min_radius), min(min_radius,
            ↪ min(x, y, masked_frame.shape[1] - x, masked_frame.shape[0] - y))
            dish = masked_frame[y - min_radius:y + min_radius, x - min_radius:x
            ↪ + min_radius]
            dish = apply_curve_transformation(dish,
            ↪ piecewise_linear_transformation, 0.0, 0.85)
            last_frame[i] = dish
    except:
        continue
```

Explanation: Read each image, apply the mask, extract and transform the Petri dish regions, and store them in `last_frame`.

- Process Each Image

```
this_frame_index = 1
frame_index = 0
for filepath in files:
    frame = cv2.imread(filepath)
    frame_index = int(filepath.split('_')[-1].split('.')[0])
    if frame is None:
        continue
    while this_frame_index < frame_index:
        for i, (x, y, r) in enumerate(dish_coords):
            this_last_frame = last_frame[i].copy()
            cv2.putText(this_last_frame, f'f{this_frame_index:03}', (105, 105),
            ↪ cv2.FONT_HERSHEY_SIMPLEX, 3, (255, 255, 255), 2)
            cv2.circle(this_last_frame, (50, 50), 40, (50, 50, 250), -1)
```

```
            ↪  cv2.imwrite(f'PREPROCESSED/{series_prefix}_{i+1:02}_{this_frame_index:03}.jpg',
            ↪  this_last_frame)
        this_frame_index += 1
```

Explanation: For each file, create images for missing frames (provision implemented in case of experimental data collection hiccups) by using the last good frame data.

- Process Current Frame

```
if dish_coords is not None and mask is not None:
    masked_frame = cv2.bitwise_and(frame, mask)
    for i, (x, y, r) in enumerate(dish_coords):
        x, y, r = max(x, min_radius), max(y, min_radius), min(min_radius, min(x,
        ↪  y, masked_frame.shape[1] - x, masked_frame.shape[0] - y))
        dish = masked_frame[y - min_radius:y + min_radius, x - min_radius:x +
        ↪  min_radius]
        if dish_angles[i] != 0:
            rotation_matrix =
            ↪  cv2.getRotationMatrix2D((int(min_radius),int(min_radius)),
            ↪  dish_angles[i]*180/np.pi, 1)
            dish = cv2.warpAffine(dish, rotation_matrix, (2*int(min_radius),
            ↪  2*int(min_radius)))
        if dish.size > 0 and dish.shape[0] > 0 and dish.shape[1] > 0:
            dish = apply_curve_transformation(dish,
            ↪  piecewise_linear_transformation, 0.0, 0.85)
            last_frame[i] = dish.copy()
            cv2.circle(dish, (50, 50), 40, (50, 250, 50), -1)
            cv2.putText(dish, f'f{frame_index:03}', (105, 105),
            ↪  cv2.FONT_HERSHEY_SIMPLEX, 3, (255, 255, 255), 2)

            ↪  cv2.imwrite(f'PREPROCESSED/{series_prefix}_{i+1:02}_{frame_index:03}.jpg',
            ↪  dish)
    this_frame_index += 1
```

Explanation: For the current frame, apply the mask, extract and transform the Petri dish regions, rotate if necessary, and save the processed image.

- Main Execution Block

```
if __name__ == '__main__':
    args = docopt(__doc__)
    patterns = args['<patterns>']

    ensure_directory_exists('PREPROCESSED')

    min_radius = 0
    min_count = 0

    series_dict = {}
    for date_dir in sorted(next(os.walk('.'))[1]):
        if 'git' not in date_dir and 'PROCESSED' not in date_dir:
            print(date_dir)
            for scanner_dir in sorted(next(os.walk(date_dir))[1]):
                pattern = date_dir + '/' + scanner_dir + '/*'
                files = sorted(glob.glob(pattern))
                if len(files) > 0:
                    experiment_number = files[0].split('.')[0].split('/')[-1]
                    key = date_dir + '_' + scanner_dir + '_' + experiment_number
                    if sum([p in key for p in patterns]):
                        first_frame = int(files[0].split('_')[-1].split('.')[0])
                        last_frame = int(files[-1].split('_')[-1].split('.')[0])
                        mask, dish_circles, rot_angles =
                        ↪   extract_radius_mask(files[0],
                        ↪   save=f'PREPROCESSED/preprocess_{key}.jpg')
                        series_dict[key] = [
                                date_dir,
                                scanner_dir,
                                experiment_number,
                                files,
                                mask,
                                dish_circles,
                                rot_angles,
                                first_frame,
                                last_frame
                        ]
                        if max(dish_circles[:,2]) > min_radius:
                            min_radius = max(dish_circles[:,2])
                        if len(files) > min_count:
```

```
                            min_count = len(files)

    for key, series in series_dict.items():
        process_series(key, series, min_radius, min_count)
```

Explanation: The script initializes parameters, collects series data from directories, and processes each series using `process_series`.

8 Petri Dish Image Processing: Main Execution Block

8.1 Import and Initialization

```
def process_series(series_prefix, series, min_radius, min_count):
    date, scanner, experiment, files, mask, dish_coords, dish_angles, first_frame,
    ↪  last_frame = series
    for i, (x, y, r) in enumerate(dish_coords):
        print(f'output/{series_prefix}_{i+1:02}_d03.jpg')

    if len(dish_coords) < 6:
        print(f'WARNING: PREPROCESSED/{series_prefix}_%d_%03d has {len(dish_coords)}
        ↪  dishes, expected 6')
```

Explanation:

- Initializes the `process_series` function with parameters `series_prefix`, `series`, `min_radius`, and `min_count`.
- Prints output file names based on `series_prefix` and dish coordinates.
- Issues a warning if fewer than 6 dishes (number of petridishes in the initial raw image) are found.

8.2 Image Handling and Initial Frame Extraction

```
    last_frame = {}
    for filepath in files:
        try:
            frame = cv2.imread(filepath)
```

```
        masked_frame = cv2.bitwise_and(frame, mask)
        for i, (x, y, r) in enumerate(dish_coords):
            x, y, r = max(x, min_radius), max(y, min_radius), min(min_radius,
            ↪  min(x, y, masked_frame.shape[1] - x, masked_frame.shape[0] - y))
            dish = masked_frame[y - min_radius:y + min_radius, x - min_radius:x +
            ↪  min_radius]
            dish = apply_curve_transformation(dish,
            ↪  piecewise_linear_transformation, 0.0, 0.85)
            last_frame[i] = dish
    except:
        continue
```

Explanation:

- Initializes `last_frame` to store images.
- Iterates over file paths to read and mask images.
- Extracts Petri dish regions from each image, applies curve transformation, and updates `last_frame`.

8.3 Frame Processing Loop

```
this_frame_index = 1
frame_index = 0
for filepath in files:
    frame = cv2.imread(filepath)
    frame_index = int(filepath.split('_')[-1].split('.')[0])
    if frame is None:
        continue

    while this_frame_index < frame_index:
        for i, (x, y, r) in enumerate(dish_coords):
            this_last_frame = last_frame[i].copy()
            cv2.putText(this_last_frame, f'f{this_frame_index:03}', (105, 105),
            ↪  cv2.FONT_HERSHEY_SIMPLEX, 3, (255, 255, 255), 2)
            cv2.circle(this_last_frame, (50, 50), 40, (50, 50, 250), -1)

            ↪  cv2.imwrite(f'PREPROCESSED/{series_prefix}_{i+1:02}_{this_frame_index:03}.jpg',
            ↪  this_last_frame)
        this_frame_index += 1
```

Explanation:

- Initializes `this_frame_index` and `frame_index` to track frames.
- Processes each file, skipping empty frames.
- Saves missing frames by creating copies from `last_frame` with timestamps (provision inclusion in case of missing timeframes in experimental data collection.)

8.4 Mask Application and Rotation

```
        if dish_coords is not None and mask is not None:
            masked_frame = cv2.bitwise_and(frame, mask)
            for i, (x, y, r) in enumerate(dish_coords):
                x, y, r = max(x, min_radius), max(y, min_radius), min(min_radius,
                ↪  min(x, y, masked_frame.shape[1] - x, masked_frame.shape[0] - y))
                dish = masked_frame[y - min_radius:y + min_radius, x - min_radius:x +
                ↪  min_radius]
                if dish_angles[i] != 0:
                    rotation_matrix = cv2.getRotationMatrix2D((int(min_radius),
                    ↪  int(min_radius)), dish_angles[i]*180/np.pi, 1)
                    dish = cv2.warpAffine(dish, rotation_matrix, (2*int(min_radius),
                    ↪  2*int(min_radius)))
                if dish.size > 0 and dish.shape[0] > 0 and dish.shape[1] > 0:
                    dish = apply_curve_transformation(dish,
                    ↪  piecewise_linear_transformation, 0.0, 0.85)
                    last_frame[i] = dish.copy()
```

Explanation:

- Applies mask to each frame and extracts Petri dish regions.
- Rotates dishes if necessary based on `dish_angles`.
- Applies curve transformation to the dishes and updates `last_frame`.

8.5 Output Saving

```
                # Save preprocessed images

                ↪  cv2.imwrite(f'PREPROCESSED/{series_prefix}_{i+1:02}_{frame_index:03}.jpg',
                ↪  dish)
```

Explanation:

- Saves preprocessed Petri dish images to the 'PREPROCESSED' directory with appropriate filenames.

8.6 Explanation Summary

The `process_series` function processes a series of Petri dish images by extracting and transforming regions of interest, applying masks, and rotating images as needed. It handles missing frames by generating them based on the previous frames and saves all processed images to the specified output directory. The function ensures that the images are properly masked and transformed, adhering to specified parameters for dish coordinates and rotation angles.

9 Dynamical Feature Extraction Function: is_contour_inside

9.1 Description

The `is_contour_inside` function determines whether all points of one contour (`contour1`) are inside another contour (`contour2`). It uses OpenCV's `cv2.pointPolygonTest` to perform this check.

9.2 Function Definition

```python
def is_contour_inside(contour1, contour2):
    # Check if all points of contour1 are inside contour2
    for i in range(len(contour1)):
        # Using cv2.pointPolygonTest to check each point of contour1 against contour2
        point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
        if cv2.pointPolygonTest(contour2, point, False) < 0:
            return False
    return True
```

9.3 Function Explanation

9.3.1 Step-by-Step Breakdown

Function 1: Check Containment

Check if all points of `contour1` are inside `contour2`.

```python
for i in range(len(contour1)):
    # Using cv2.pointPolygonTest to check each point of contour1 against contour2
    point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
    if cv2.pointPolygonTest(contour2, point, False) < 0:
        return False
```

Explanation: The function iterates through each point in `contour1` and uses `cv2.pointPolygonTest` to check if the point is inside `contour2`. If any point is found outside, the function returns `False`. If all points are inside, it returns `True`.

9.4 Conclusion

The `is_contour_inside` function provides a way to verify the spatial relationship between two contours by checking if all points of one contour are contained within another contour. This can be useful for various image processing and geometric applications.

10 `Dynamical Feature Extraction` Function: `generate_points`

10.1 Description

The `generate_points` function generates a list of `N` points evenly spaced between two given points, `last_point` and `point`. The generated points are rounded to the nearest integer coordinates.

10.2 Function Definition

```python
def generate_points(last_point, point, N):
    """Generate N points between last_point and point."""
    x1, y1 = last_point
    x2, y2 = point

    points = []
    for i in range(1, N + 1):
        t = i / (N + 1)
        x = x1 + t * (x2 - x1)
        y = y1 + t * (y2 - y1)
        points.append((int(round(x)), int(round(y))))

    return points
```

10.3 Function Explanation

10.3.1 Step-by-Step Breakdown

Function 1: Calculate Interpolation Factor

Calculate the interpolation factor `t` for each point.

```
t = i / (N + 1)
```

Explanation: The interpolation factor `t` is computed for each point to be generated. It ensures that points are evenly spaced between `last_point` and `point`.

Function 2: Calculate Coordinates

Calculate the coordinates of each generated point.

```
x = x1 + t * (x2 - x1)
y = y1 + t * (y2 - y1)
```

Explanation: The coordinates of each point are calculated by interpolating between the `last_point` and `point` using the interpolation factor `t`. The results are rounded to the nearest integer.

Function 3: Round and Append

Round the coordinates to the nearest integer and append to the list of points.

```
points.append((int(round(x)), int(round(y))))
```

Explanation: The calculated coordinates are rounded to the nearest integers to ensure they represent pixel locations or discrete points. These coordinates are then added to the list of generated points.

10.4 Conclusion

The `generate_points` function efficiently generates a specified number of evenly spaced points between two given coordinates. This function is useful for tasks requiring interpolation between points in various applications, including graphics and data analysis.

11 Dynamical Feature Extraction Function: is_contour_partially_inside

11.1 Description

The `is_contour_partially_inside` function checks whether any part of `contour1` is inside `contour2`. It does this by checking each point of `contour1` as well as points along the segments connecting consecutive points of `contour1`. It uses OpenCV's `cv2.pointPolygonTest` for the containment checks and considers a width around the contour segments for partial inclusion.

11.2 Function Definition

```python
def is_contour_partially_inside(contour1, contour2, side_width):
    # Check if all points of contour1 are inside contour2
    last_point = (int(contour1[-1][0][0]), int(contour1[-1][0][1]))
    for i in range(len(contour1)):
        # Using cv2.pointPolygonTest to check each point of contour1 against contour2
        point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
        if cv2.pointPolygonTest(contour2, point, False) > 0:
            return True
        points = generate_points(last_point, point, int(side_width/2)+1)
        for new_point in points:
            if cv2.pointPolygonTest(contour2, new_point, False) > 0:
                return True
        last_point = point
    return False
```

11.3 Function Explanation

11.3.1 Step-by-Step Breakdown

Function 1: Initialize Last Point

Initialize `last_point` with the coordinates of the last point of `contour1`.

```
last_point = (int(contour1[-1][0][0]), int(contour1[-1][0][1]))
```

Explanation: The function starts by setting `last_point` to the last point of `contour1`. This point will be used to generate intermediate points along the contour segments.

Function 2: Check Point Inside

Check if each point of `contour1` is inside `contour2`.

```
for i in range(len(contour1)):
    point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
    if cv2.pointPolygonTest(contour2, point, False) > 0:
        return True
```

Explanation: The function iterates over each point in `contour1` and uses `cv2.pointPolygonTest` to determine if the point is inside `contour2`. If any point is found inside, the function returns `True`.

Function 3: Generate Intermediate Points

Generate intermediate points along the line segment between `last_point` and `point`.

```
points = generate_points(last_point, point, int(side_width/2)+1)
```

Explanation: Intermediate points are generated along the line segment connecting `last_point` and `point` to account for partial inclusion. The number of intermediate points is determined by `side_width`.

Function 4: Check Intermediate Points

Check if any of the generated intermediate points are inside `contour2`.

```
for new_point in points:
    if cv2.pointPolygonTest(contour2, new_point, False) > 0:
        return True
```

Explanation: The function checks each of the generated intermediate points to see if they lie inside `contour2`. If any intermediate point is found inside, the function returns `True`.

Function 5: Update Last Point

Update `last_point` to the current `point` and continue checking the next segment.

```
last_point = point
```

Explanation: The `last_point` is updated to the current `point` after processing, so that the next segment can be checked.

11.4 Conclusion

The `is_contour_partially_inside` function checks if any part of `contour1` is inside `contour2`. It considers both direct point inclusion and partial inclusion along the segments connecting points of `contour1`, making it suitable for more comprehensive geometric containment checks.

12 Dynamical Feature Extraction Function: is_contour_intersecting

12.1 Description

The `is_contour_intersecting` function determines if `contour1` intersects `contour2`. It checks each point of `contour1` as well as points along the segments connecting consecutive points of `contour1`. The function uses OpenCV's `cv2.pointPolygonTest` to check if points are on the boundary of `contour2` and considers a width around the contour segments for intersection detection.

12.2 Function Definition

```
def is_contour_intersecting(contour1, contour2, side_width):
    # Check if all points of contour1 are inside contour2
```

```
    last_point = (int(contour1[-1][0][0]), int(contour1[-1][0][1]))
    for i in range(len(contour1)):
        # Using cv2.pointPolygonTest to check each point of contour1 against contour2
        point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
        if cv2.pointPolygonTest(contour2, point, False) == 0:
            return True
        points = generate_points(last_point, point, side_width)
        for new_point in points:
            if cv2.pointPolygonTest(contour2, new_point, False) == 0:
                return True
        last_point = point
    return False
```

12.3 Function Explanation

12.3.1 Step-by-Step Breakdown

Function 1: Initialize Last Point

Initialize `last_point` with the coordinates of the last point of `contour1`.

```
last_point = (int(contour1[-1][0][0]), int(contour1[-1][0][1]))
```

Explanation: The function initializes `last_point` as the last point of `contour1`. This serves as a reference for generating intermediate points along the contour segments.

Function 2: Check Point Intersection

Check if each point of `contour1` is on the boundary of `contour2`.

```
for i in range(len(contour1)):
    point = (int(contour1[i][0][0]), int(contour1[i][0][1]))
    if cv2.pointPolygonTest(contour2, point, False) == 0:
        return True
```

Explanation: The function iterates through each point in `contour1` and checks if any of these points are on the boundary of `contour2` using `cv2.pointPolygonTest`. If a point is found on the boundary, the function returns `True`.

Function 3: Generate Intermediate Points

Generate intermediate points along the line segment between `last_point` and `point`.

```
points = generate_points(last_point, point, side_width)
```

Explanation: Intermediate points are generated along the line segment connecting `last_point` and `point`. This is done to account for any potential intersection along the segment width specified by `side_width`.

Function 4: Check Intermediate Points

Check if any of the generated intermediate points are on the boundary of `contour2`.

```
for new_point in points:
    if cv2.pointPolygonTest(contour2, new_point, False) == 0:
        return True
```

Explanation: The function checks each intermediate point to see if it is on the boundary of `contour2`. If any intermediate point is found on the boundary, the function returns `True`.

Function 5: Update Last Point

Update `last_point` to the current `point` and continue checking the next segment.

```
last_point = point
```

Explanation: The `last_point` is updated to the current `point` after processing, so that the next segment can be checked.

12.4 Conclusion

The `is_contour_intersecting` function determines if `contour1` intersects `contour2` by checking direct point intersections as well as potential intersections along the segments between points of `contour1`. It is useful for detecting if two contours cross each other or touch.

13 Dynamical Feature Extraction Function: points_in_contour

13.1 Description

The `points_in_contour` function identifies which points from a given list `points` lie within a specified contour `contour1`. It uses OpenCV's `cv2.pointPolygonTest` to determine if each point is inside or on the boundary of the contour and collects these points.

13.2 Function Definition

```
def points_in_contour(points, contour1):
    # gather any points in the contour
    new_points = []
    for i in range(len(points)):
        # Using cv2.pointPolygonTest to check each point of contour1 against contour2
        point = points[i]
        if cv2.pointPolygonTest(contour1, point, False) >= 0:
            new_points.append(point)
    return new_points
```

13.3 Function Explanation

13.3.1 Step-by-Step Breakdown

Function 1: Initialize List

Initialize an empty list `new_points` to store points that lie within or on the boundary of the contour.

```
new_points = []
```

Explanation: The function starts by initializing an empty list `new_points` which will be used to store the points that are either inside the contour or on its boundary.

Function 2: Check Point Position

Check if each point from the `points` list is inside or on the boundary of `contour1`.

```
for i in range(len(points)):
    point = points[i]
    if cv2.pointPolygonTest(contour1, point, False) >= 0:
        new_points.append(point)
```

Explanation: The function iterates through each point in `points` and uses `cv2.pointPolygonTest` to check if the point lies inside or on the boundary of `contour1`. If the point meets this criterion, it is added to `new_points`.

Function 3: Return Points

Return the list `new_points` containing all points that are inside or on the boundary of the contour.

```
return new_points
```

Explanation: The function returns the list `new_points` which includes all points from the original list that were found to be inside or on the boundary of `contour1`.

13.4 Conclusion

The `points_in_contour` function is used to filter out and collect points from a given list that are within or on the boundary of a specified contour. This can be useful for various tasks in image processing and contour analysis.

14 Dynamical Feature Extraction Function: find_and_sort_contours

14.1 Description

The `find_and_sort_contours` function processes a grayscale image to identify, group, and sort contours based on their area and perimeter. The function also calculates distances of contour centers from a given reference point and sorts the contours accordingly.

14.2 Function Definition

```
def find_and_sort_contours(gray_image, min_thresh, max_thresh, last_center):
    # Apply range threshold
    range_threshold_image = cv2.inRange(gray_image, min_thresh, max_thresh)
    range_threshold_image = cv2.medianBlur(range_threshold_image, 9)

    # Fill in holes
    kernel_size = 9
    kernel = np.ones((kernel_size, kernel_size), np.uint8)

    # Apply the closing operation
    range_threshold_image = cv2.morphologyEx(range_threshold_image, cv2.MORPH_CLOSE,
    ↪  kernel)

    # Find contours
    contours, _ = cv2.findContours(range_threshold_image, cv2.RETR_TREE,
    ↪  cv2.CHAIN_APPROX_SIMPLE)
    sorted_contours = sorted(contours, key=cv2.contourArea, reverse=True)

    # sort contours
    grouped_contours = []
    weights = []
    index = 0
    while index < len(sorted_contours):
        contour = sorted_contours[index]

        contour_list = []
        contour_list.append(contour)

        # calculate the center of the contour
        M = cv2.moments(contour)
        # Calculate the center (centroid) of the contour
        if M["m00"] != 0:
            centerX = int(M["m10"] / M["m00"])
            centerY = int(M["m01"] / M["m00"])
        else:
            centerX, centerY = 0, 0  # Set default values if m00 is zero to avoid
            ↪  division by zero
```

```
    # Calculate area
    area = cv2.contourArea(contour)

    # Calculate perimeter
    if area > 0:
        perimeter = cv2.arcLength(contour, True)  # True indicates the contour is
        ↪  closed

        index_next = index + 1
        while index_next < len(sorted_contours):
            contour_next = sorted_contours[index_next]
            area2 = cv2.contourArea(contour_next)
            if area2 > 0:
                perimeter2 = cv2.arcLength(contour_next, True)  # True indicates
                ↪  the contour is closed
                if is_contour_inside(contour_next, contour):
                    if args['--include_holes']:
                        perimeter += perimeter2  # True indicates the contour is
                        ↪  closed
                        area -= area2
                        contour_list.append(contour_next)
                    sorted_contours.pop(index_next)
                else:
                    index_next += 1
            else:
                sorted_contours.pop(index_next)

        grouped_contours.append([np.array([centerX, centerY]), area, perimeter,
        ↪  contour_list])

        # calculate distance from center/area, g
        weights.append(np.sqrt((centerX-last_center[0])**2 +
        ↪  (centerY-last_center[1])**2)/area)
        index += 1
    else:
        sorted_contours.pop(index)

sorted_indices = np.argsort(np.array(weights))
last_center = grouped_contours[sorted_indices[0]][0]
```

```
    return sorted_contours, grouped_contours, sorted_indices, last_center, weights
```

14.3 Function Explanation

14.3.1 Step-by-Step Breakdown

Function 1: Apply Range Threshold

Apply a range threshold to the grayscale image to create a binary image where pixels within the specified threshold range are set to 255 (white), and others are set to 0 (black). Apply median blur to reduce noise.

```
range_threshold_image = cv2.inRange(gray_image, min_thresh, max_thresh)
range_threshold_image = cv2.medianBlur(range_threshold_image, 9)
```

Explanation: The function thresholds the grayscale image to segment features of interest and reduces noise using a median blur.

Function 2: Fill in Holes

Use morphological operations to fill in small holes in the thresholded image and apply a closing operation with a specified kernel size.

```
kernel_size = 9
kernel = np.ones((kernel_size, kernel_size), np.uint8)
range_threshold_image = cv2.morphologyEx(range_threshold_image, cv2.MORPH_CLOSE,
↪  kernel)
```

Explanation: Morphological closing is used to clean up the thresholded image by closing small gaps and holes.

Function 3: Find and Sort Contours

Find contours in the processed image, sort them by area in descending order, and group contours based on containment.

```
contours, _ = cv2.findContours(range_threshold_image, cv2.RETR_TREE,
↪  cv2.CHAIN_APPROX_SIMPLE)
sorted_contours = sorted(contours, key=cv2.contourArea, reverse=True)
```

Explanation: Contours are identified and sorted by their area to prioritize larger contours.

Function 4: Group Contours

Group contours based on their containment and calculate their center, area, and perimeter. Store weights based on distance from a reference center and area.

```
grouped_contours = []
weights = []
index = 0
while index < len(sorted_contours):
    contour = sorted_contours[index]
    contour_list = []
    contour_list.append(contour)
    M = cv2.moments(contour)
    if M["m00"] != 0:
        centerX = int(M["m10"] / M["m00"])
        centerY = int(M["m01"] / M["m00"])
    else:
        centerX, centerY = 0, 0
    area = cv2.contourArea(contour)
    if area > 0:
        perimeter = cv2.arcLength(contour, True)
        index_next = index + 1
        while index_next < len(sorted_contours):
            contour_next = sorted_contours[index_next]
            area2 = cv2.contourArea(contour_next)
            if area2 > 0:
                perimeter2 = cv2.arcLength(contour_next, True)
                if is_contour_inside(contour_next, contour):
                    if args['--include_holes']:
                        perimeter += perimeter2
                        area -= area2
                        contour_list.append(contour_next)
```

```
                sorted_contours.pop(index_next)
            else:
                index_next += 1
        else:
            sorted_contours.pop(index_next)
    grouped_contours.append([np.array([centerX, centerY]), area, perimeter,
    ↪   contour_list])
    weights.append(np.sqrt((centerX-last_center[0])**2 +
    ↪   (centerY-last_center[1])**2)/area)
    index += 1
else:
    sorted_contours.pop(index)
```

Explanation: Contours are grouped if one is contained within another. The function calculates the center, area, and perimeter for each group. Weights are computed based on the distance from a reference point and the contour area.

Function 5: Sort by Weights

Sort the grouped contours based on their calculated weights and update the last center.

```
sorted_indices = np.argsort(np.array(weights))
last_center = grouped_contours[sorted_indices[0]][0]
```

Explanation: Contours are sorted based on their weights (distance-to-area ratio). The 'last_center' is updated to the center of the highest-weight contour.

14.4 Conclusion

The `find_and_sort_contours` function processes a grayscale image to find, group, and sort contours by their area and perimeter. It also calculates and uses weights based on distance and area to determine the significance of each contour.

15 Dynamical Feature Extraction: Main Execution Block

15.1 Import and Initialization

```
# Get the number of series
if __name__ == '__main__':
    args = docopt(__doc__)
    image_paths = sorted(args['<filenames>'])
    min_thresh = int(args['--min_thresh'])
    max_thresh = int(args['--max_thresh'])
    side = args['--side']

    ensure_directory_exists('PROCESSED')
```

Explanation:

- This block starts by checking if the script is being run directly.
- It uses `docopt` to parse command-line arguments.
- `image_paths` stores the list of image file paths sorted.
- `min_thresh`, `max_thresh`, and `side` are command-line arguments converted to appropriate types.
- `ensure_directory_exists('PROCESSED')` ensures that the output directory exists.

15.2 Output File Naming

```
fileout_filename =
↪  'nh_fracdim_'+'_'.join(image_paths[0].split('/')[-1].replace('.jpg','').split('_')[:-1])+'.dat'
if side in ['left','right']:
    fileout_filename = side + '_' + fileout_filename

fileout_filename_frac =
↪  'nh_fracdim_'+'_'.join(image_paths[0].split('/')[-1].replace('.jpg','').split('_')[:-1])+'_frac.dat'
```

Explanation:

- This block constructs output filenames based on the first image path.
- If `side` is specified (left or right), it prefixes the filename with it.

- Two filenames are created: one for general output (`fileout_filename`) and another for fractional dimension data (`fileout_filename_frac`).

15.3 File Handling and Initialization

```
with open('PROCESSED' + '/' + fileout_filename, 'w') as fileout, open('PROCESSED' +
↪  '/' + fileout_filename_frac, 'w') as fileout_frac:
    output = (f'#{"frame":>8}  {"area":>15}  {"perimeter":>15}  {"circularity":>15}
    ↪  {"fracdim":>15}\r\n')
    output_frac = (f'#{"frame":>8}  {"BSR":>15}  {"NA":>15}  {"NP":>15}\r\n')

    last_center = None
    check_split = None
```

Explanation:

- The output files are opened for writing.
- The headers for the output files are defined.
- `last_center` and `check_split` are initialized to `None`.

15.4 Image Processing Loop

```
for image_index, image_path in enumerate(image_paths):
    # Load the image
    image = cv2.imread(image_path)
    physarum_only_image = cv2.imread(image_path)
    frame = int(image_path.split('.')[-2].split('_')[-1])

    # Convert to grayscale
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    height, width = gray_image.shape
    channels = 3

    if last_center is None:
        last_center = (int(width/2), int(height/2))
```

Explanation:

- Loop through each image in `image_paths`.

- Load the image using `cv2.imread`.
- Convert the image to grayscale.
- Get the image dimensions.
- Initialize `last_center` to the center of the image if it's `None`.

15.5 Noise Reduction and Mask Application

```
# Apply a median filter to reduce noise and increase connectivity
gray_image = cv2.medianBlur(gray_image, 9)

# Generate and apply mask to remove edges
mask = np.zeros_like(gray_image)
cv2.circle(mask, (int(width/2), int(height/2)), int(0.465*width), (255), -1)
gray_image = cv2.bitwise_and(gray_image, mask)
```

Explanation:

- Apply a median filter to reduce noise.
- Create a mask to remove the edges of the image.
- Apply the mask to the grayscale image.

15.6 Contour Detection and Sorting

```
if image_index == 0:
    first_sorted_contours, first_grouped_contours, first_sorted_indices,
    ↪ first_center, _= find_and_sort_contours(gray_image, min_thresh, max_thresh,
    ↪ last_center)
if side in ['right','left']:
    first_split_y = [i for i in range(100,int(height-100))]
    first_split = []
    for i, val in enumerate(first_split_y):
        if side == 'right':
            first_split.append([int(first_center[0]), val])
        else:
            first_split.append([int(first_center[0]), val])
```

Explanation:

- For the first image, find and sort contours based on the threshold values and center.
- If `side` is specified, generate a vertical split line through the center.

15.7 Side Mask Application

```
# Generate an additional mask to remove the left or right side of the image,
↪  optionally
if side == 'right':
    mask = np.zeros_like(gray_image)
    cv2.rectangle(mask, (int(first_center[0]), 0), (int(width), int(height)), (255),
    ↪  -1)
    gray_image = cv2.bitwise_and(gray_image, mask)
if side == 'left':
    mask = np.zeros_like(gray_image)
    cv2.rectangle(mask, (0, 0), (int(first_center[0]), int(height)), (255), -1)
    gray_image = cv2.bitwise_and(gray_image, mask)
```

Explanation:

- Apply a mask to remove either the left or right side of the image based on the `side` argument.
- For `side == 'right'`, the mask is applied to keep only the right side of the image.
- For `side == 'left'`, the mask is applied to keep only the left side of the image.

15.8 Contour Sorting and Drawing

```
# Find and sort contours again after applying the side mask
sorted_contours, grouped_contours, sorted_indices, last_center, weights =
↪  find_and_sort_contours(gray_image, min_thresh, max_thresh, last_center)

# Draw the sorted contours on the image
cv2.drawContours(image, sorted_contours, -1, (255, 0, 0), 3)
```

Explanation:

- Find and sort contours again after applying the side mask using the `find_and_sort_contours` function.
- Draw the sorted contours on the image using `cv2.drawContours` with a red color and a thickness of 3.

15.9 Quadrant Definition and Box Sizes

```
# Create quadrants based on image height
if side in ['left','right']:
    top_group, bottom_group = split_group_contours(physarum_only_image,
    ↪  grouped_contours, side, first_split)
    include_group = True
else:
    include_group = False

# Choose box sizes based on the dimensions of the image
if (height > 1000 and width > 1000):
    box_sizes = np.array([32, 48, 64, 80, 96, 112, 128, 160, 192, 224])
else:
    box_sizes = np.array([20, 30, 40, 50, 60, 70, 80, 100, 120, 140])
```

Explanation:

- Define quadrants for the image based on its height and the `side` argument, using the `split_group_contours` function.
- Choose box sizes based on the dimensions of the image to adjust the analysis scale.

15.10 Contour Processing and Metrics Calculation

```
# Create a mask for physarum only
physarum_mask = np.zeros_like(gray_image)
total_area = 0
total_perimeter = 0
points = []

for i, contour in enumerate(sorted_contours):
    M = cv2.moments(contour)
    cX = int(M["m10"] / M["m00"])
    cY = int(M["m01"] / M["m00"])
    if include_group:
        include = check_contour_in_group(cX, cY, top_group, bottom_group)
    else:
        include = True
```

```
    if include:
        cv2.drawContours(physarum_mask, [contour], -1, (255), thickness=-1)
        cv2.drawContours(physarum_only_image, [contour], -1, (255), thickness=-1)
        total_area += cv2.contourArea(contour)
        total_perimeter += cv2.arcLength(contour, True)

        epsilon = 0.01 * cv2.arcLength(contour, True)
        simplified_contours = cv2.approxPolyDP(contour, epsilon, True)
        simplified_points = np.array(simplified_contours).reshape(-1, 2)
        if len(points) == 0:
            points = simplified_points.tolist()
        else:
            points = points + simplified_points.tolist()

# Compute metrics
logbsr, logN = fractal_dimension_from_contour(physarum_mask, box_sizes)
slope, intercept, r_value, p_value, std_err = stats.linregress(logbsr, logN)
fracdim = slope

circularity = 0
if total_perimeter > 0:
    circularity = (4*math.pi*total_area)/(total_perimeter**2)
else:
    circularity = 0

if include_group:
    output += (f'{frame:8d}  {total_area:15.4f}  {total_perimeter:15.4f}
    ↪  {circularity:15.4f}  {fracdim:15.4f}\r\n')

output_frac += (f'{frame:8d}  {" ".join([str(elem) for elem in logbsr]):15}  {"
↪  ".join([str(elem) for elem in logN]):15}\r\n')
```

Explanation:

- Create a mask for the physarum image to isolate contours.
- Process each contour to compute its center, area, perimeter, and check if it should be included.
- Draw contours on the mask and the image.
- Calculate metrics such as area, perimeter, circularity, and fractal dimension, and format results for output.

15.11 Saving Output and Processed Image

```
cv2.imwrite(f'PROCESSED/{image_path.split("/")[-1]}', image)

# Write the output to file
fileout.write(output)
fileout_frac.write(output_frac)
```

Explanation:

- Save the processed image to the `PROCESSED` directory using the filename extracted from the image path.
- Write the computed metrics and fractal dimension data to the respective output files.

15.12 Explanation Summary

This script processes a series of images by applying filters and masks, detecting and sorting contours, calculating various metrics such as area, perimeter, circularity, and fractal dimension. It then saves the processed images and computed data to files. The command-line arguments allow for customization of thresholds and side masking options.

16 Discussion & Outlook

In this version of `PyPETANA`, we have implemented core code capabilities that allow for image extraction, masking petridishes and extraction of biological entity for further explorations of dynamical parameters related to growth dynamics, fractal dimensions and exploratory quantifiers such as area and perimeter. One of the core features of the `Image_Extraction_Routine` is to produce images that can be used as inputs in (1) for network dynamics studies in bio-organisms such as *Physarum*. In future work, we aim to implement visualization tools and enhancements related to statistical physics parameters which are crucial to understand critical phenomena in biological systems.

17 Acknowledgments

This research was undertaken in part thanks to funding from the Alfred P. Sloan Foundation (Matter-to-Life program). I thank Nirosha J. Murugan and Philip Kurian for helpful discussions with regards to Fricker Software and fractal dimensions, respectively. I am grateful to Benjamin E. Himberg for his technical assistance and valuable advice regarding software implementation techniques.

18 References

Bibliography

[1] Mark D Fricker, Dai Akita, Luke LM Heaton, Nick Jones, Boguslaw Obara, and Toshiyuki Nakagaki. Automated analysis of physarum network structure and dynamics. *Journal of Physics D: Applied Physics*, 50(25):254005, jun 2017.

[2] B.B. Mandelbrot. *The Fractal Geometry of Nature*. Einaudi paperbacks. Henry Holt and Company, 1983.

[3] Sanghita Sengupta. Py`PETANA`. 2024.