HOMEWORK Final Project
ECE/CS 8690 2302 Computer Vision

Coffee Grains Size &Distribution Recognition Using OpenCV

Xuanbo Miao

14422044

xmiao@mail.missouri.edu

Introduction - Problem Being Solved

Dataset: I have get one of my dataset like below, since I am doing CV project, I won't needa a huge dataset currently, I'll just use it to make some basic code.

They are at different grinding size:

Coding: a part of coding is accomplished:

```python
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Constants
IMAGE_DIR = "./CV2023_FinalProj/testData_1/"


def load_images(directory):
    # Load images from the directory
    images = []
    for file in os.listdir(directory):
        if file.endswith((".jpg", ".jpeg", ".png")):
            img_path = os.path.join(directory, file)
            img = cv2.imread(img_path)
            images.append(img)
    return images


def unify_contrast_brightness(image):
    # Compute the minimum and maximum pixel values in the image
    min_val, max_val, _, _ = cv2.minMaxLoc(image)

    # Normalize the pixel values to span the full range of 0 to 255
    if max_val > min_val:
        normalized = ((image - min_val) / (max_val - min_val)) * 255
    else:
        normalized = image.copy()

    # Convert the pixel values to the range of 0 to 255
    normalized = np.uint8(normalized)

    # Apply a histogram equalization to improve contrast
    equalized = cv2.equalizeHist(normalized)   # type: ignore

    return equalized


def auto_correlation_matrix(image, window_size=3):

    # Compute the derivatives of the image using Sobel kernels
    Ix = cv2.Sobel(image, cv2.CV_32F, 1, 0, ksize=3)
    Iy = cv2.Sobel(image, cv2.CV_32F, 0, 1, ksize=3)

    # Compute the elements of the structure tensor
    auto_rr_matrix_xx = cv2.GaussianBlur(Ix**2, (window_size,
window_size), 0)
    auto_rr_matrix_yy = cv2.GaussianBlur(Iy**2, (window_size,
window_size), 0)
    auto_rr_matrix_xy = cv2.GaussianBlur(Ix*Iy, (window_size,
window_size), 0)

    auto_rr_matrix = [auto_rr_matrix_xx, auto_rr_matrix_yy,
auto_rr_matrix_xy]

    return auto_rr_matrix


def preprocess_image(image):
    # Convert the image to Lab color space
    lab_img = cv2.cvtColor(image, cv2.COLOR_BGR2Lab)

    # Split the Lab image into L, a, and b channels
    L_channel, _, _ = cv2.split(lab_img)

    # Apply Gaussian blur to reduce noise
    blurred_img = cv2.GaussianBlur(L_channel, (3, 3), 0)

    # Apply contrast and brightness enhancement
    enhanced_image = unify_contrast_brightness(blurred_img)

    # Compute the auto-correlation matrix
    auto_corr_matrix = auto_correlation_matrix(enhanced_image)

    return enhanced_image, auto_corr_matrix


def DetectAndCompute(image, max_points):

    sift = cv2.SIFT_create()
    keypoints = sift.detect(image)
    keypoints = sorted(keypoints, key=lambda x: x.response, reverse=True)
    keypoints = keypoints[:max_points]

    keypoints, descriptors = sift.compute(image, keypoints)
    descriptors = descriptors[:max_points]

    return keypoints, descriptors


def extract_features(image):
    max_points = 1000
    keypoints, descriptors = DetectAndCompute(image, max_points)

    # Calculate the average area of keypoints
    areas = [kp.size for kp in keypoints]
    mean_area = np.mean(areas)

    # Calculate the average color values of keypoints
    colors = [image[int(kp.pt[1]), int(kp.pt[0])] for kp in keypoints]
    mean_color = np.mean(colors, axis=0)

    features = {
        'mean_area': mean_area,
        'mean_color': mean_color,
        'descriptors_size': descriptors.shape[0]
    }

    return features


def analyze_features(features):
    # Analyze the features to determine particle size, distribution, and
color value

    # Calculate particle size statistics
    mean_size = features["mean_area"]  # Use 'mean_area' instead of
'areas'
    std_dev_size = np.std(features["mean_area"])

    # Calculate color value statistics
    mean_color = features["mean_color"]

    # Create a dictionary to store the results
    results = {
        "mean_size": mean_size,
        "std_dev_size": std_dev_size,
        "mean_color": mean_color,
    }

    return results


def display_results(results):
    # Display the results in a user interface

    # Print results to the console (simple example)
    print(f"Mean Particle Size: {results['mean_size']:.2f}")
    print(
        f"Standard Deviation of Particle Size:
{results['std_dev_size']:.2f}")
    print(f"Mean Color Value (B, G, R): {tuple(results['mean_color'])}")


def main():
    # Step 1: Image Acquisition
    images = load_images(IMAGE_DIR)

    # Step 2: Image Preprocessing
    preprocessed_images = [preprocess_image(img)[0] for img in images]

    # Step 3: Feature Extraction
    feature_list = [extract_features(img) for img in preprocessed_images]

    # Step 4: Data Analysis
    analysis_results = [analyze_features(features)
                        for features in feature_list]

    # Step 5: Display the results
    for idx, results in enumerate(analysis_results):
        print(f"Results for Image {idx + 1}:")
        display_results(results)
        print()

    # Step 6: Testing and Validation
    # Compare the system's results to manual measurements
    pass


if __name__ == "__main__":
    main()
```

Background - Previous Work

Image Processing Techniques

- Image segmentation
- Edge detection
- Morphological operations

Subtitle: Particle Sizing Methods

- Watershed segmentation: divides image into regions based on pixel intensity
- Connected component labeling: detects groups of connected pixels and assigns unique labels

Feature Extraction Methods

- SIFT (Scale-Invariant Feature Transform): detects and describes local features in images, robust to changes in scale, rotation, and illumination
- ORB (Oriented FAST and Rotated BRIEF): an alternative to SIFT, faster and more efficient, but less robust to scale changes

Challenges in Previous Work

- Noise: image degradation due to sensor noise or external factors
- Varying particle sizes: handling particles with different sizes in the same image
- Irregular shapes: accurately identifying and measuring particles with non-uniform shapes

Visuals

- Include examples of watershed segmentation and connected component labeling results
- Display SIFT and ORB keypoints on images
- Show examples of images with noise, varying particle sizes, and irregular shapes

Proposed Solution - Algorithm and Novelty

Steps

1. Image Acquisition: Load images from a specified directory.
2. Image Preprocessing:
   - Convert the image to Lab color space.
   - Apply Gaussian blur to reduce noise.
   - Enhance contrast and brightness using histogram equalization.
   - Compute the auto-correlation matrix.
3. Feature Extraction:
   - Use SIFT to extract keypoints and descriptors.
   - Calculate the average area and color values of keypoints.
4. Data Analysis:
   - Calculate particle size statistics (mean, standard deviation).
   - Determine mean color value.
5. Display the results.
6. Testing and Validation: Compare the system's results to manual measurements.

Novelty

- Unified contrast and brightness enhancement to improve the quality of images with varying illumination and poor contrast.
- The use of the auto-correlation matrix to identify regions of interest in the images, improving the efficiency and accuracy of the feature extraction process.
- Robust feature extraction using SIFT, allowing for more reliable analysis of particles in complex images.

Visuals

- Flowchart or diagram illustrating the steps in the algorithm.
- Example input image and preprocessed output.
- SIFT keypoints and descriptors overlaid on an image.
- Example of displayed results.

```python
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Constants
IMAGE_DIR = "./CV2023_FinalProj/testData_1/"


def load_images(directory):
    # Load images from the directory
    images = []
    for file in os.listdir(directory):
        if file.endswith((".jpg", ".jpeg", ".png")):
            img_path = os.path.join(directory, file)
            img = cv2.imread(img_path)
            images.append(img)
    return images


def unify_contrast_brightness(image):
    # Compute the minimum and maximum pixel values in the image
    min_val, max_val, _, _ = cv2.minMaxLoc(image)

    # Normalize the pixel values to span the full range of 0 to 255
    if max_val > min_val:
        normalized = ((image - min_val) / (max_val - min_val)) * 255
    else:
        normalized = image.copy()

    # Convert the pixel values to the range of 0 to 255
    normalized = np.uint8(normalized)

    # Apply a histogram equalization to improve contrast
    equalized = cv2.equalizeHist(normalized)   # type: ignore

    return equalized


def auto_correlation_matrix(image, window_size=3):

    # Compute the derivatives of the image using Sobel kernels
    Ix = cv2.Sobel(image, cv2.CV_32F, 1, 0, ksize=3)
    Iy = cv2.Sobel(image, cv2.CV_32F, 0, 1, ksize=3)

    # Compute the elements of the structure tensor
    auto_rr_matrix_xx = cv2.GaussianBlur(Ix**2, (window_size,
window_size), 0)
    auto_rr_matrix_yy = cv2.GaussianBlur(Iy**2, (window_size,
window_size), 0)
    auto_rr_matrix_xy = cv2.GaussianBlur(Ix*Iy, (window_size,
window_size), 0)

    auto_rr_matrix = [auto_rr_matrix_xx, auto_rr_matrix_yy,
auto_rr_matrix_xy]

    return auto_rr_matrix


def preprocess_image(image):
    # Convert the image to Lab color space
    lab_img = cv2.cvtColor(image, cv2.COLOR_BGR2Lab)

    # Split the Lab image into L, a, and b channels
    L_channel, _, _ = cv2.split(lab_img)

    # Apply Gaussian blur to reduce noise
    blurred_img = cv2.GaussianBlur(L_channel, (3, 3), 0)

    # Apply contrast and brightness enhancement
    enhanced_image = unify_contrast_brightness(blurred_img)

    # Compute the auto-correlation matrix
    auto_corr_matrix = auto_correlation_matrix(enhanced_image)

    return enhanced_image, auto_corr_matrix


def DetectAndCompute(image, max_points):

    sift = cv2.SIFT_create()
    keypoints = sift.detect(image)
    keypoints = sorted(keypoints, key=lambda x: x.response, reverse=True)
    keypoints = keypoints[:max_points]

    keypoints, descriptors = sift.compute(image, keypoints)
    descriptors = descriptors[:max_points]

    return keypoints, descriptors


def extract_features(image):
    max_points = 1000
    keypoints, descriptors = DetectAndCompute(image, max_points)

    # Calculate the average area of keypoints
    areas = [kp.size for kp in keypoints]
    mean_area = np.mean(areas)

    # Calculate the average color values of keypoints
    colors = [image[int(kp.pt[1]), int(kp.pt[0])] for kp in keypoints]
    mean_color = np.mean(colors, axis=0)

    features = {
        'mean_area': mean_area,
        'mean_color': mean_color,
        'descriptors_size': descriptors.shape[0]
    }

    return features


def analyze_features(features):
    # Analyze the features to determine particle size, distribution, and
color value

    # Calculate particle size statistics
    mean_size = features["mean_area"]  # Use 'mean_area' instead of
'areas'
    std_dev_size = np.std(features["mean_area"])

    # Calculate color value statistics
    mean_color = features["mean_color"]

    # Create a dictionary to store the results
    results = {
        "mean_size": mean_size,
        "std_dev_size": std_dev_size,
        "mean_color": mean_color,
    }

    return results


def display_results(results):
    # Display the results in a user interface

    # Print results to the console (simple example)
    print(f"Mean Particle Size: {results['mean_size']:.2f}")
    print(
        f"Standard Deviation of Particle Size:
{results['std_dev_size']:.2f}")
    print(f"Mean Color Value (B, G, R): {tuple(results['mean_color'])}")


def main():
    # Step 1: Image Acquisition
    images = load_images(IMAGE_DIR)

    # Step 2: Image Preprocessing
    preprocessed_images = [preprocess_image(img)[0] for img in images]

    # Step 3: Feature Extraction
    feature_list = [extract_features(img) for img in preprocessed_images]

    # Step 4: Data Analysis
    analysis_results = [analyze_features(features)
                        for features in feature_list]

    # Step 5: Display the results
    for idx, results in enumerate(analysis_results):
        print(f"Results for Image {idx + 1}:")
        display_results(results)
        print()

    # Step 6: Testing and Validation
    # Compare the system's results to manual measurements
    pass


if __name__ == "__main__":
    main()
```

Evaluation - Testing and Validation

Testing and Validation Process

1. Collect a dataset of particle images with varying sizes, shapes, and color values.
2. Manually measure and annotate the particle sizes and color values in the dataset.
3. Run the proposed algorithm on the dataset and compare the results to the manual measurements.
4. Compute performance metrics such as Mean Absolute Error and Root Mean Squared Error for particle size and color value estimation.
5. Test the robustness of the algorithm under different conditions such as noise, poor illumination, and varying particle sizes.

Success Criteria

- The algorithm should provide accurate estimations of particle sizes and color values, with low error values compared to manual measurements.
- The algorithm should be robust to different image conditions and provide consistent results across various particle characteristics.
- The algorithm should demonstrate better performance than existing methods in terms of accuracy, efficiency, and robustness.

Visuals

- Graphs or plots showing the comparison between the algorithm's results and manual measurements.
- Performance metrics (e.g., MAE and RMSE) displayed in a table or graph format.
- Examples of input images with varying conditions and the corresponding algorithm's output.