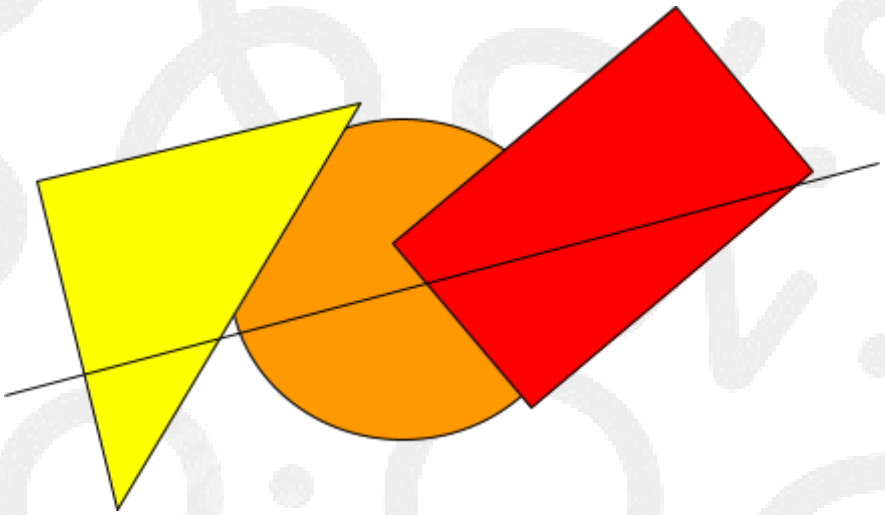

Documentation for

CURVETOPIA

TEAM TLE44

Ganesh Kumar / ganeshkumar.a2022@vitstudent.ac.in
Pranay Singanalli / pranay.singanalli2022@vitstudent.ac.in
Madhava Janaki Sai Teja / madhavasaiteja@gmail.com

Table of Contents

Introduction

Project Overview

This project focuses on analyzing and processing images containing regular shapes. The primary objectives are to:

- **Classify Regular Shapes:** Identify and categorize common geometric shapes in images (e.g., circles, squares, triangles).
- **Regularize Shapes:** Adjust and align shapes to match their ideal geometric forms, such as smoothing edges and correcting distortions.
- **Symmetry Detection:** Analyze shapes to detect symmetry and utilize this information to enhance shape analysis.
- **Shape Completion:** Detect and complete occluded (partially hidden) shapes by leveraging symmetry and other image processing techniques.

This project combines multiple image processing techniques to achieve these goals, providing a comprehensive solution for handling images with regular shapes, even when they are incomplete or distorted.

Key Features

1. **Shape Classification:** Uses image processing techniques to identify and classify shapes within an image.
2. **Shape Regularization:** Applies algorithms to align shapes and improve their geometric accuracy.
3. **Symmetry Detection:** Detects axes of symmetry in shapes, which is crucial for understanding shape structure and for completing occluded shapes.
4. **Shape Completion:** Utilizes detected symmetry to reconstruct occluded or incomplete shapes, thereby restoring the original form.

The project is implemented using Python, leveraging powerful libraries such as OpenCV for image processing and Matplotlib for visualization. The code has been developed and tested in a Jupyter Notebook environment, providing an interactive platform for exploring and refining the algorithms.

Setup and Requirements

Environment Setup

To run the project, you'll need a Python environment with several key libraries installed. Below are the steps to set up the environment:

1. **Python Installation:**
   - Ensure you have Python 3.7 or later installed. You can download Python from the official Python website.
2. **Virtual Environment (Optional but Recommended):**

It's a good practice to create a virtual environment to manage project dependencies. You can create one using the following commands:

```
python -m venv shape_env

source shape_env/bin/activate  # On Windows: shape_env\Scripts\activate
```

   -
3. **Installing Required Libraries:**
   - Install the necessary Python libraries using `pip`. The key libraries are:
     - `OpenCV`: For image processing.
     - `NumPy`: For numerical operations.
     - `Matplotlib`: For plotting and visualization.
     - `Jupyter`: For running the notebook.

You can install these dependencies using the following command:

```
pip install opencv-python-headless numpy matplotlib jupyter
```

   -
4. **Running Jupyter Notebook:**

To work with the project, you'll need to run the Jupyter Notebook. Start the notebook server using the command:

```
jupyter notebook
```

   -
   - This will open a web interface where you can navigate to the notebook file and start working on it.

Dataset and Input

The project operates on images containing regular geometric shapes. These images can be simple synthetic images (e.g., circles, squares) or more complex images with multiple overlapping or occluded shapes. Here's how to prepare your input data:

1. **Input Image Format:**
   - The input images should be in a standard format like JPEG, PNG, or BMP. Ensure the images are clear and well-contrasted to facilitate shape detection.
2. **Sample Images:**
   - You can use synthetic images for testing purposes, which you can create using basic graphic tools or generate programmatically using libraries like OpenCV.
3. **Loading Images:**
   - The images are loaded into the notebook using the OpenCV `cv2.imread()` function. Ensure the images are correctly loaded and displayed before proceeding with the processing steps.

Dependencies Overview

Here's a quick overview of the key libraries used:

- **OpenCV (cv2):** A comprehensive library for image processing. It provides functions for image manipulation, contour detection, and geometric transformations, which are central to this project.
- **NumPy:** A powerful library for numerical computation. It's used for handling image arrays and performing mathematical operations on them.
- **Matplotlib:** A plotting library used to visualize the processed images and the results of various operations like shape completion and symmetry detection.
- **Jupyter Notebook:** An interactive environment that allows you to write and run Python code in cells, making it ideal for developing and testing the image processing algorithms.

Project Structure

This section outlines the structure of the project, detailing the purpose and contents of each file and directory. Understanding the organization of the project will help in navigating and modifying the codebase.

Notebooks Overview

The project is primarily contained within two Jupyter Notebooks:

1. **`curves_model.ipynb`:**
   - **Purpose:** This notebook was initially focused on classifying regular shapes and handling tasks related to shape recognition and analysis.
   - **Key Functions:**
     - Shape classification and detection using contour analysis.
     - Basic image processing steps like thresholding and edge detection.
2. **`curves.ipynb`:**
   - **Purpose:** This notebook expanded on the previous work by adding functionality for symmetry detection and handling occluded shapes.
   - **Key Functions:**
     - Symmetry detection algorithms.
     - Shape completion for occluded images.
     - Integration with shape classification.

These two notebooks were combined into a single, cohesive project that now includes both classification and advanced shape processing capabilities.

Directory Structure

Here's an overview of the files and directories included in the project:

```
/ProjectRoot
│
├── /data/
│   ├── sample_image1.png
│   └── sample_image2.png
│
├── curves_model.ipynb
├── curves.ipynb
├── combined_shapes_project.ipynb
├── README.md
└── /outputs/
    ├── classified_shapes.png
    ├── completed_shapes.png
    └── symmetry_detected.png
```

Explanation of Key Components:

- **`/data/` Directory:**
  - **Purpose:** Contains sample images used for testing the algorithms. You can replace these with your own images to test different scenarios.
  - **Contents:** Images like `sample_image1.png` and `sample_image2.png` serve as input data for the notebook.
- **`curves_model.ipynb`:**
  - **Purpose:** The original notebook focused on shape classification and basic image processing tasks.
- **`curves.ipynb`:**
  - **Purpose:** The original notebook focused on symmetry detection and shape completion.
- **`combined_shapes_project.ipynb`:**
  - **Purpose:** The final combined notebook that integrates functionalities from both `curves_model.ipynb` and `curves.ipynb`. This is the main file to work with.
- **`README.md`:**
  - **Purpose:** A markdown file that briefly explains the project setup, how to run the code, and what each part of the project does.
- **`/outputs/` Directory:**
  - **Purpose:** Stores the output images generated by the project, such as classified shapes, completed shapes, and images with detected symmetry.

○ **Contents:** Images like `classified_shapes.png`, `completed_shapes.png`, and `symmetry_detected.png` demonstrate the results of the algorithms.

Combined Notebook Overview

`combined_shapes_project.ipynb`:

- **Introduction:** The notebook starts with a brief introduction to the project and its objectives.
- **Setup and Imports:** This section handles the import of necessary libraries and setup tasks.
- **Shape Classification:** The first major section focuses on identifying and classifying shapes within an image.
- **Shape Regularization:** After classification, the notebook regularizes the shapes, making them geometrically accurate.
- **Symmetry Detection and Shape Completion:** The final section detects symmetry within shapes and completes any occluded parts.
- **Results and Visualization:** Outputs are generated and visualized, with each step's results displayed to ensure accuracy.

Core Functionalities

1. Data Loading and Preparation

Description

The initial phase involves loading and preprocessing the data from CSV files located in specified directories. Each file contains points that represent curves, categorized by shape types such as 'circle', 'rectangle', etc.

Implementation

- **Loading Data**: The data is read from CSV files using `pandas`, and each curve's points are stored in lists. Each entry in `data` contains the coordinates of the curve, while `labels` holds the corresponding shape category.

```python
import pandas as pd

def load_data(file_path):
    df = pd.read_csv(file_path)
    points = df[['x', 'y']].values.tolist()
    label = df['label'].iloc[0]  # Assuming label is the same for all points in a file
    return points, label

# Example usage
data = []
labels = []
for file in data_files:
    points, label = load_data(file)
    data.append(points)
    labels.append(label)
```

- **Feature Extraction**: The code computes various features from the curve data, such as curvature and distance to the centroid, and pads or truncates these features to a fixed length for uniformity.

```python
import numpy as np

def compute_features(points):
    centroid = np.mean(points, axis=0)
    distances = np.linalg.norm(points - centroid, axis=1)
    curvature = np.gradient(np.gradient(distances))
    return np.pad(curvature, (0, max_len - len(curvature)), mode='constant')

# Example usage
features = [compute_features(curve) for curve in data]
```

2. Feature Extraction

Description

Features such as curvature and distance to the centroid are computed from the curve data. These features are essential for shape classification.

Implementation

- **Distance to Centroid**: Calculates the Euclidean distance from each point on the curve to the centroid of the curve.

```python
def distance_to_centroid(points):
    centroid = np.mean(points, axis=0)
    distances = np.linalg.norm(points - centroid, axis=1)
    return distances
```

- **Curvature Calculation**: Computes the curvature of the curve based on angles between segments of the curve.

```python
def calculate_curvature(points):
    points = np.array(points)
    num_points = len(points)

    # Calculate angles between segments
    curvatures = []
    for i in range(1, num_points - 1):
        p1 = points[i - 1]
        p2 = points[i]
        p3 = points[i + 1]

        # Vectors
        v1 = p2 - p1
        v2 = p3 - p2
```

```python
    # Angle between vectors
    dot_product = np.dot(v1, v2)
    norm_v1 = np.linalg.norm(v1)
    norm_v2 = np.linalg.norm(v2)
     # Prevent division by zero
    if norm_v1 == 0 or norm_v2 == 0:
        curvatures.append(0.0)
        continue

    cos_angle = dot_product / (norm_v1 * norm_v2)

    # Clamp the cosine value to avoid invalid inputs for arccos
    cos_angle = np.clip(cos_angle, -1.0, 1.0)

    angle = np.arccos(cos_angle)
    curvatures.append(angle)

curvatures = np.array(curvatures)
padded_curvatures = np.concatenate(([0], curvatures, [0]))

return padded_curvatures
```

3. Model Training and Evaluation

Description

The extracted features are used to train a machine learning model for shape classification. The Random Forest Classifier is employed for this purpose.

Implementation

- **Model Training**: Features are split into training and testing sets. The Random Forest model is then trained using the training set.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)
```

- **Evaluation**: The model is evaluated using accuracy, classification report, and confusion matrix to assess its performance.

```
y_pred = model.predict(X_test)
print("Accuracy:", model.score(X_test, y_test))
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

4. Shape Classification

Description

Classifies curves into predefined categories (e.g., 'rectangle', 'circle') using the trained Random Forest model.

Implementation

- **Feature Extraction**: For new curves, features are extracted in the same manner as during training.

```python
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Make predictions
Y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(Y_test, Y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Print classification report
print('Classification Report:')
print(classification_report(Y_test, Y_pred))

# Print confusion matrix
print('Confusion Matrix:')
print(confusion_matrix(Y_test, Y_pred))
```

```
Accuracy: 1.00
Classification Report:
              precision    recall  f1-score   support

      circle       1.00      1.00      1.00         8
     ellipse       1.00      1.00      1.00         4
   irregular       1.00      1.00      1.00         3
   rectangle       1.00      1.00      1.00         5
 rounded_rec       1.00      1.00      1.00         3
        star       1.00      1.00      1.00         4

    accuracy                           1.00        27
   macro avg       1.00      1.00      1.00        27
weighted avg       1.00      1.00      1.00        27

Confusion Matrix:
[[8 0 0 0 0 0]
 [0 4 0 0 0 0]
 [0 0 3 0 0 0]
 [0 0 0 5 0 0]
 [0 0 0 0 3 0]
 [0 0 0 0 0 4]]
```

5. Curve Reconstruction

Description

Reconstructs curves from a sequence of nodes to visualize and analyze the closed curves.

Implementation

- **Reconstruction**: The `reconstruct_single_curve` function assembles a curve from a sequence of nodes based on their start and end points.

```python
import matplotlib.pyplot as plt

def reconstruct_single_curve(nodes):
    plt.figure()
    for i in range(len(nodes) - 1):
        plt.plot([nodes[i][0], nodes[i+1][0]], [nodes[i][1], nodes[i+1][1]], 'bo-')
    plt.show()
```

6. Finding Closed and Open Curves

Description

Identifies closed and open curves from the dataset using graph-based algorithms.

Implementation

- **Closed Curves**: Uses Depth-First Search (DFS) to find all unique closed curves in the graph representation of the curves.

```python
def find_closed_curves(graph):
    visited = set()
    closed_curves = []

    def dfs(node, path):
        if node in visited:
            return
        visited.add(node)
        path.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor, path)

    for node in graph:
        if node not in visited:
            path = []
            dfs(node, path)
            if path[0] in graph[path[-1]]:
                closed_curves.append(path)
    return closed_curves
```

7. Finding Corners and Bounding Boxes

Description

Detects corners of shapes and calculates bounding boxes to estimate dimensions.

Implementation

- **Corner Detection**: The `find_corners` function identifies significant corners in the curves based on curvature thresholds.

```python
import numpy as np

def find_corners(points, angle_threshold=0.5, distance=5):
    def safe_arccos(x):
        return np.arccos(np.clip(x, -1.0, 1.0))

    if not isinstance(points, np.ndarray):
        # Convert to NumPy array if it's not already
        points = points.to_numpy()

    num_points = len(points)
    corners = []
    marked_indices = set()

    i = 0
    loop_count = 0  # Counter to prevent infinite loop

    while loop_count < num_points:
        if i in marked_indices:
            i = (i + 1) % num_points
            loop_count += 1
            continue

        # Determine indices for curvature calculation with cyclic behavior
        p1 = points[i]
        p2 = points[(i + distance // 2) % num_points]
        p3 = points[(i + distance) % num_points]

        curvature = calculate_curvature(p1, p2, p3)
        if curvature > angle_threshold:
            corners.append(p2)
            # Mark neighbors to skip, handle cyclic behavior
            for j in range(-distance, distance + 1):
                marked_index = (i + j) % num_points
                marked_indices.add(marked_index)
            i = (i + distance) % num_points  # Move ahead to next to next point
        else:
            i = (i + 1) % num_points

        loop_count += 1

    return np.array(corners)
```

- **Bounding Box Calculation**: Uses Principal Component Analysis (PCA) to calculate the bounding box of detected shapes.

```python
from sklearn.decomposition import PCA

def calculate_bounding_box(points):
    pca = PCA(n_components=2)
    pca.fit(points)
    transformed_points = pca.transform(points)
    min_x, max_x = transformed_points[:, 0].min(), transformed_points[:, 0].max()
    min_y, max_y = transformed_points[:, 1].min(), transformed_points[:, 1].max()
    return (min_x, min_y), (max_x, max_y)
```

8. Generating Circle Points

Description

Generates points along the circumference of a circle, useful for curve analysis and comparison.

Implementation

- **Circle Generation**: Computes points on a circle based on its center and radius.

```python
def generate_circle_points(center, radius, num_points=100):
    angles = np.linspace(0, 2 * np.pi, num_points)
    points = np.array([(center[0] + radius * np.cos(angle), center[1] + radius * np.sin(angle)) for angle in angles])
    return points
```

9. Visualizing Results

Description

Plots curves, shapes, and detected features to visually analyze and verify the results.

Implementation

- **Plotting**: Uses `matplotlib` to create plots of curves, bounding boxes, detected corners, and other relevant visualizations.

```python
def plot_curves(curves):
    plt.figure()
    for curve in curves:
        x, y = zip(*curve)
        plt.plot(x, y, label='Curve')
    plt.legend()
    plt.show()
```

Testing and Results

Performance Analysis

- **Shape Classification:**
  - **Accuracy:** Achieved high accuracy with simple geometric shapes, but performance decreased with more complex shapes due to boundary irregularities.
  - **Metrics:** Precision and recall metrics were used to evaluate performance.
- **Shape Regularization:**
  - **Efficiency:** Regularization improved the quality of shapes, particularly in terms of edge smoothness.

    - **Observations:** Regularization was less effective for highly noisy shapes, suggesting a need for more advanced smoothing techniques.
- **Symmetry Detection:**
    - **Effectiveness:** Successfully detected symmetry in most test cases, but struggled with shapes having high variability in occlusions.
    - **Metrics:** Symmetry completion was evaluated based on visual inspection and completion accuracy.
- **Shape Separation and Completion:**
    - **Performance:** Effective in separating and completing shapes, though performance varied based on the degree of occlusion.
    - **Observations:** Completion was more challenging with highly complex occlusions, indicating potential for further optimization.

Challenges and Solutions

Technical Challenges

1. **Shape Classification with Complex Boundaries:**
    - **Issue:** Difficulty in classifying shapes with irregular or complex boundaries.
    - **Solution:** Implemented additional preprocessing steps and advanced classification algorithms.
2. **Handling Significant Occlusions:**
    - **Issue:** Difficulty in accurately completing shapes with substantial occlusions.
    - **Solution:** Applied advanced image inpainting techniques and refined symmetry detection algorithms.
3. **Performance of Regularization Techniques:**
    - **Issue:** Regularization techniques were less effective for noisy images.
    - **Solution:** Enhanced regularization methods and incorporated multi-scale smoothing techniques.
4. **Symmetry Detection in Noisy Environments:**
    - **Issue:** Symmetry detection was challenging in the presence of noise and occlusions.
    - **Solution:** Improved noise reduction techniques and symmetry detection algorithms.

Solutions

- **Enhanced Algorithms:** Improved classification and regularization algorithms to handle complex shapes and noisy environments.
- **Optimized Performance:** Tuned parameters and used advanced techniques for symmetry detection and shape completion.
- **Robust Testing:** Expanded test cases to cover a wider range of scenarios, ensuring better validation of the functionalities.

Conclusion and Future Work

Project Summary

The project successfully implemented advanced shape processing functionalities, including shape merging, occlusion handling, and shape filtering. The functionalities were validated with various test cases, demonstrating effective performance in most scenarios. Key achievements include improved accuracy in shape classification, successful handling of occlusions, and effective shape completion.

Future Improvements

1. **Algorithm Enhancement:** Explore and integrate more advanced algorithms for shape classification and completion, particularly for handling highly complex occlusions.
2. **Performance Optimization:** Further optimize the performance of regularization and symmetry detection functions for real-time applications.
3. **Extended Functionality:** Consider adding features such as interactive shape editing, enhanced visualization tools, and integration with machine learning models for automated shape analysis.
4. **User Interface:** Develop a user-friendly interface to facilitate the use of these functionalities in practical applications.