Санкт-Петербургский Политехнический Университет Петра Великого

Институт компьютерных наук и технологий

**Курсовая работа**

**“Программа автоматической нормализации изображений листьев растений для последующей диагностики вида заболевания”**

по дисциплине “Цифровая обработка сигналов”

Выполнил студент  
группы 3530904/00103: Бабушкин Е. А.

Преподаватель: Тутыгин В. С.

Санкт-Петербург  
 2023

ЗАДАНИЕ 3

НА ВЫПОЛНЕНИЕ КУРСОВОГО ПРОЕКТА

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| студенту группы | 3530904/00103 | |  | Бабушкин Евгений Алексеевич | | | | | |
|  | *(номер группы)* | |  | *(фамилия, имя, отчество)* | | | | | |
| ***1. Тема проекта (работы)*** | | |  | | | | | | |
| 1. Программа автоматической нормализации изображений листьев растений для последующей диагностики вида заболевания. | | | | | | | | | |
| ***2. Исходные данные к работе***: Имеются аннотированные наборы ImageNet и PlantVillageDataset изображений листьев растений с признаками болезней. Требуется разработать алгоритм автоматической нормализации изображений листьев растений по размерности (300x100 пикселей) и ориентации в соответствии с эталонным образцом нормализованного изображения, создать программные средства на С++ или Python, реализующие этот алгоритм. | | | | | | | | | |
| ***3. Содержание расчетно-пояснительной записки*** | | | | | | | | |
| 3.1. Обзор методов и алгоритмов нормализации фотоизображений по размерности, ориентации, яркости и контрастности.  3.2. Разработка алгоритма и программы в среде MATLAB для автоматической нормализации изображений листьев растений.  3.3. Программная реализация разработанного алгоритма на Python и экспериментальные результаты | | | | | | | | | |
| 4. Дата получения задания: «2» февраля 2023 г. | | | | | | | | | |
| Руководитель доц.,к.т.н. | |  | | | |  | *В. С. Тутыгин* | | |
|  | | *(подпись)* | | | |  | *(инициалы, фамилия)* | | |
| Задание принял к исполнению | |  | | |  | | *Е. А. Бабушкин* | | |
|  | | *(подпись)* | | |  | | *(инициалы, фамилия)* | | |
|  | | | | | | | | 26.04.2023 | |
|  | | | | | | | | *(дата)* | |

***Содержание***

[*Введение* 4](#_Toc134320790)

[*Постановка задачи* 5](#_Toc134320791)

[*Программное обеспечение* 6](#_Toc134320792)

[*Описание алгоритмов* 7](#_Toc134320793)

[*Программная реализация* 9](#_Toc134320794)

[*Заключение* 16](#_Toc134320795)

[*Код программы* 17](#_Toc134320796)

[*Список литературы* 27](#_Toc134320797)

# *Введение*

С каждым годом количество людей на планете увеличивается, в связи с этим вопрос обеспечения продовольствием становится все более актуальным. Однако, болезни растений являются серьезной проблемой для сельского хозяйства и могут снизить урожайность и качество продукции.

Обнаружение болезней на ранней стадии имеет большое значение для борьбы с ними. Для успешной борьбы с болезнями растений необходимо уметь быстро и точно определять их виды и степень поражения. Это позволяет своевременно принимать меры по предотвращению распространения болезней, а также выбирать наиболее эффективные методы лечения. Однако, обнаружение болезней растений на ранней стадии может быть сложным из-за того, что заболевания часто проявляются не сразу или могут быть схожими с другими заболеваниями.

Поэтому, разработка автоматизированных систем, которые могут точно и быстро обнаруживать заболевания растений, является крайне важной для сельского хозяйства. Для этого необходимо разработать алгоритм, который может определить заболевание, выделив больную часть листа на изображении. Такие системы могут значительно сократить время и ресурсы, необходимые для обнаружения и борьбы с болезнями растений, и улучшить эффективность борьбы с болезнями растений, что в свою очередь повысит урожайность и качество продукции, что важно для обеспечения продовольственной безопасности населения.

# *Постановка задачи*

Для обработки массива полноразмерных изображений листьев растений необходимо разработать алгоритм автоматической нормализации, который будет приводить все изображения к одинаковому размеру (300x100 пикселей) и ориентации в соответствии с эталонным образцом нормализованного изображения. Это позволит облегчить сравнение и анализ изображений, а также снизить влияние факторов, связанных с размером и ориентацией изображения.

Для разработки алгоритма будут использоваться различные методы обработки изображений, такие как масштабирование, поворот, фильтрация шумов, сглаживание, улучшение контраста и яркости.

# 

# *Программное обеспечение*

Для реализации алгоритма были использованы:

* Язык программирования: Python 3.11.3
* Среда программирования: PyCharm
* Библиотеки: OpenCV, NumPy, PIL, math

Для эффективной работы должен соответствовать следующим требованиям:

* Центральный процессор: тактовая частота не менее 2,5 ГГц
* Графический процессор: желательно с поддержкой CUDA или OpenCL
* Оперативная память: не менее 4 ГБ оперативной памяти.
* Операционная система: Windows, Linux, MacOS.

# *Описание алгоритмов*

Представим алгоритм нормализации изображений в виде обобщенной блок-схемы:

Изображение выглядит как диаграмма

Автоматически созданное описание

В начале работы программы на вход подается изображение. В первую очередь, с ним проводится фильтрация, которая необходима для уменьшения уровня шумов на изображении и повышения его читабельности. Хотя на сегодняшний день это не является обязательным шагом, но так как качество входного изображения может сильно варьироваться, фильтрация является хорошей практикой для обеспечения более точного и стабильного результата. Для улучшения качества изображения используются операции повышения резкости и четкости изображения, а также применяется фильтрация Гаусса для сглаживания изображения.

Далее используется алгоритм Canny. Алгоритм Canny - это один из наиболее широко используемых алгоритмов для обнаружения границ на изображениях. Он основан на подавлении шума, определении градиентов яркости, пороговой фильтрации и операции свертки с фильтром подавления не-максимума. Алгоритм Canny начинается с преобразования изображения в оттенки серого, чтобы уменьшить количество данных для обработки. Затем происходит градиентный анализ, который вычисляет градиент яркости в каждой точке изображения. С помощью этого определяются направления и интенсивность границ на изображении. Далее происходит применение пороговой фильтрации, чтобы удалить нежелательные границы на изображении и сохранить только те, которые имеют высокую интенсивность. Пороговая фильтрация позволяет настроить чувствительность алгоритма к границам и отрегулировать размер области, которую он охватывает. Наконец, используется операция свертки с фильтром подавления не-максимума для тонкой настройки обнаруженных границ и удаления нежелательных. Этот шаг помогает сгладить контуры границ и сделать их более четкими.

Результатом работы алгоритма Canny является изображение с ярко выделенными границами, что делает его полезным для последующей обработки и анализа изображений.

Преобразование Хафа - это метод для обнаружения геометрических фигур на изображении, таких как линии, окружности и эллипсы. Оно используется для анализа изображений и находит группы пикселей, которые могут быть интерпретированы как линии или другие геометрические фигуры. Преобразование Хафа применяется после обнаружения границ на изображении (с помощью алгоритма Canny). Оно основано на математическом преобразовании изображения в пространство параметров линий. Каждая точка на изображении соответствует возможной линии в пространстве параметров. Процесс преобразования Хафа начинается с построения таблицы аккумулятора, которая представляет пространство параметров линий. Затем каждая точка на изображении добавляется в таблицу аккумулятора, где ей соответствует возможная линия в пространстве параметров. Когда точки, соответствующие одной линии, накапливаются в таблице аккумулятора, обнаруживается группа пикселей, которая может быть интерпретирована как линия.

Одним из основных преимуществ преобразования Хафа является его способность находить линии, даже если они не связаны и имеют прерывистую форму.

Далее происходят следующие шаги: находятся два наибольших контура на изображении, определяется минимальная ограничивающая прямоугольная область для каждого контура. Затем находится центр итоговой формы и вычисляется матрица преобразования для поворота исходного изображения на угол, определенный на основе итоговой формы. Наконец, повернутое изображение обрезается до заданных размеров.

# *Программная реализация*

# В качестве примера возьмем *лист с темно-бурой пятнистостью:*

1 шаг: увеличение резкости и четкости изображения.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

2 шаг: применение алгоритма Canny для выделения границ на изображении.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

3 шаг: применение размытия Гаусса для уменьшения шума на изображении.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

4 шаг: применение преобразования Хафа для поиска общих линий на изображении.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

5 шаг: используем полученное преобразование для выделения границ объекта.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

6 шаг: поиск двух самых больших контуров.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

7 шаг: нахождение минимальной ограничивающей прямоугольной области для каждого из контуров.

|  |  |
| --- | --- |
| Исходное изображение  Изображение выглядит как диаграмма  Автоматически созданное описание | Результат |

8 шаг: определение центра итоговой формы, поворот изображения на угол, вычисляемый по этой форме.

|  |  |
| --- | --- |
| Исходное изображение  Изображение выглядит как текст, в помещении, лазер, темный  Автоматически созданное описание | Результат |

9 шаг: обрезка повернутого изображения до заданных размеров.

|  |  |
| --- | --- |
| Исходное изображение | Результат |

Другие примеры:

*Мучнистая роса Blumeria graminis (Erysiphe graminis)*

|  |  |
| --- | --- |
| Исходное изображение | Результат |

*Ржавчина желтая Puccinia striiformis*

|  |  |
| --- | --- |
| Исходное изображение | Результат |

*Бурая ржавчина*

|  |  |
| --- | --- |
| Исходное изображение | Результат |

Несмотря на свою эффективность, алгоритм не всегда может решить задачу:

1.

|  |  |
| --- | --- |
|  | Изображение выглядит как млекопитающее  Автоматически созданное описание |
| Изображение выглядит как текст, лазер  Автоматически созданное описание |

Мы можем заметить, что на фотографии изображены три объекта, причем два из них занимают большую часть изображения слева. Однако, алгоритм считает, что оба объекта являются аномалиями, и пытается выделить их оба. В результате получается не совсем удовлетворительный результат.

|  |  |
| --- | --- |
|  |  |
|  |  |

Исходное изображение содержит множество различных объектов, и алгоритм не способен точно выделить конкретный лист. Вместо этого, он пытается охватить всю область, где находится объект. Хотя результат является относительно корректным, алгоритм не справляется.

1. Алгоритм расчитан на то, что мы приводим все изображения к размеру 300 на 100 пикселей. Но если исходное изображение будет иметь более высокое разрешение, есть вероятность, что не охватится нужная область.

*Изображение взято из интернета*

|  |  |
| --- | --- |
| Исходное изображение  Изображение выглядит как дерево, насекомое  Автоматически созданное описание | |
| 300 на 100 | 900 на 300  Изображение выглядит как растение, дерево  Автоматически созданное описание |

Из этого можно сделать вывод, что размер 300 на 100 пикселей не является оптимальным.

# *Заключение*

В обработке изображений для распознавания болезней растений важным этапом является нормализация изображений. Нормализация заключается в приведении всех изображений к единому формату и размеру, чтобы обеспечить более точную и стабильную обработку изображений.

Нормализация изображений позволяет выделить лист с растением на изображении и убрать фон, что повышает точность распознавания болезней. Для этого используются различные методы, такие как изменение размера изображения, изменение контрастности, четкости, фильтрация и другие.

Однако разработка алгоритма нормализации является сложной задачей из-за отсутствия общей частной задачи и множества граничных случаев. Например, если лист на изображении не помещается полностью, то после поворота могут появиться черные фрагменты, которые необходимо учитывать.

Также следует учитывать, что в текущей реализации алгоритма нормализации есть ограничение: фон, на котором запечатлен лист не должен быть зеленым, иначе фон может слиться с самим листом на изображении. Это ограничение связано с тем, что зеленый цвет является основным цветом листа, и наличие зеленого фона может затруднить выделение границ листа. Также это ограничение действует, если на изображении несколько крупных объектов.

При использовании алгоритма нормализации на изображениях с высоким разрешением может возникнуть проблема неполного выделения зоны поражения на листе. Это связано с тем, что мы обрезаем изображение под определенные размеры, которые, как выяснилось, не являются оптимальными.

Таким образом, нормализация изображений является важным этапом в обработке изображений для распознавания болезней растений, но требует тщательной разработки алгоритма и учета граничных случаев, а также ограничений, связанных с цветом фона и разрешением исходного изображения.

# *Код программы*

Программа 1: обрабатывает одно ненормализованное фото.

import cv2  
import math  
import numpy as np  
from PIL import Image  
from PIL import ImageEnhance  
  
# Объединение двух ограничивающих прямоугольников в один  
def defineFinalRect(box1, box2):  
 final\_shape = []  
 for point in box1:  
 final\_shape.append(point)  
 for point in box2:  
 final\_shape.append(point)  
 return final\_shape  
  
  
# Корректировка резкости изображения и сохранение результата в файл  
def adjust\_sharpness(input\_image, output\_image, factor):  
 image = Image.open(input\_image)  
 enhancer\_object = ImageEnhance.Sharpness(image)  
 out = enhancer\_object.enhance(factor)  
 out.save(output\_image)  
  
  
# Отображение изображения в окне с заданным заголовком  
def printImage(mes, img):  
 cv2.imshow(mes, img)  
 cv2.waitKey(0)  
 cv2.destroyAllWindows()  
  
  
# Поиск двух контуров с наибольшим количеством точек  
def foundContours(contours):  
 max\_index = 0  
 second\_index = 0  
 max\_value = 0  
 second\_value = 0  
 index = 0  
 for tup in contours:  
 if len(tup) >= max\_value:  
 second\_value = max\_value  
 second\_index = max\_index  
 max\_value = len(tup)  
 max\_index = index  
 elif len(tup) >= second\_value:  
 second\_value = len(tup)  
 second\_index = index  
 index += 1  
 return contours[max\_index], contours[second\_index]  
  
  
# Нахождение точки правого верхнего угла  
def findTopRight(points):  
 max\_y = 0  
 res = []  
 for point in points:  
 if point[1] > max\_y:  
 max\_y = point[1]  
 res = point  
 return res  
  
  
# Нахождение точки левого нижнего угла  
def findLeftBottom(points):  
 min\_x = math.inf  
 min\_y = math.inf  
 res = []  
 for point in points:  
 if point[1] < min\_y:  
 min\_x = point[0]  
 min\_y = point[1]  
 res = point  
 return res  
  
  
# Нахождение точки правого нижнего угла  
def findRightBottom(points):  
 lb = findLeftBottom(points)  
 tr = findTopRight(points)  
 \_, width = findHighWidth(points)  
 unknown = []  
 for p in points:  
 if p[0] != lb[0] and p[1] != lb[1] and p[0] != tr[0] and p[1] != tr[1]:  
 unknown.append(p)  
 min\_y = math.inf  
 point = []  
 for p in unknown:  
 if p[1] < min\_y:  
 min\_y = p[1]  
 point = p  
 return point  
  
  
# Нахождение точки левого верхнего угла  
def findLeftTop(points):  
 rb = findRightBottom(points)  
 lb = findLeftBottom(points)  
 tr = findTopRight(points)  
 for p in points:  
 if p[0] != rb[0] and p[1] != rb[1] and p[0] != lb[0] and p[1] != lb[1] and p[0] != tr[0] and p[1] != tr[1] :  
 return p  
  
  
# Нахождение центра прямоугольника  
def foundCenter(points):  
 top\_right = findTopRight(points)  
 left\_bottom = findLeftBottom(points)  
 return [(top\_right[0] - left\_bottom[0])/2 + left\_bottom[0], (top\_right[1] - left\_bottom[1])/2 + left\_bottom[1]]  
  
  
# Нахождение высоты и ширины прямоугольника  
def findHighWidth(points):  
 right\_top = findTopRight(points)  
 left\_bottom = findLeftBottom(points)  
 return right\_top[1]-left\_bottom[1], right\_top[0] - left\_bottom[0]  
  
  
# Проверка, является ли второй прямоугольник частью первого прямоугольника  
def isSecondPartOfFirst(rectFirst, rectSecond):  
 top\_right = findTopRight(rectFirst)  
 left\_bottom = findLeftBottom(rectFirst)  
 center = foundCenter(rectSecond)  
 print("tr ", top\_right)  
 print("lb ", left\_bottom)  
 print("center ", center)  
 return (left\_bottom[0] < center[0] < top\_right[0] and  
 left\_bottom[1] < center[1] < top\_right[1] ) or (  
 left\_bottom[0] > center[0] > top\_right[0] and  
 left\_bottom[1] > center[1] > top\_right[1]) or (  
 left\_bottom[0] > center[0] > top\_right[0] and  
 left\_bottom[1] < center[1] < top\_right[1])  
  
  
# Нахождение угла наклона прямоугольника  
def findEgle(points):  
 box = np.intp(points)  
 edge1 = np.intp((box[1][0] - box[0][0], box[1][1] - box[0][1]))  
 edge2 = np.intp((box[2][0] - box[1][0], box[2][1] - box[1][1]))  
 edgeMax = edge1  
 if cv2.norm(edge2) > cv2.norm(edge1):  
 edgeMax = edge2  
  
 reference = (1, 0)  
 angle = 180.0 / math.pi \* math.acos(  
 (reference[0] \* edgeMax[0] + reference[1] \* edgeMax[1]) / (cv2.norm(reference) \* cv2.norm(edgeMax)))  
 if edgeMax[1] > 0:  
 angle = 270 + angle  
 else:  
 angle = 90 - angle  
  
 return angle  
  
  
# Проверка, расположен ли первый прямоугольник слева от второго  
def isFirstLeft(rectFirst, rectSecond):  
 tr = findTopRight(rectFirst)  
 tr2 = findTopRight(rectSecond)  
 return tr[0] < tr2[0]  
  
  
# Нахождение наибольшего прямоугольника  
def findMaxRectangle(points):  
 min\_x = math.inf  
 min\_y = math.inf  
 max\_x = 0  
 max\_y = 0  
 for point in points:  
 if point[0] > max\_x:  
 max\_x = point[0]  
 if point[0] < min\_x:  
 min\_x = point[0]  
 if point[1] > max\_y:  
 max\_y = point[1]  
 if point[1] < min\_y:  
 min\_y = point[1]  
 return [[min\_x, min\_y],[max\_x, min\_y],[min\_x, max\_y],[max\_x, max\_y]]  
  
  
# Определение итогового прямоугольника  
def defineFinalRect(rectFirst, rectSecond):  
 print(isSecondPartOfFirst(rectFirst, rectSecond))  
 if isSecondPartOfFirst(rectFirst, rectSecond):  
 return rectFirst  
 else:  
 if findEgle(rectFirst) - findEgle(rectSecond) < 15:  
 if isFirstLeft(rectFirst, rectSecond):  
 rb1 = findRightBottom(rectFirst)  
 tr1 = findTopRight(rectFirst)  
 lt2 = findLeftTop(rectSecond)  
 lb2 = findLeftBottom(rectSecond)  
 return [rb1, tr1, lt2, lb2]  
 else:  
 rb1 = findRightBottom(rectSecond)  
 tr1 = findTopRight(rectSecond)  
 lt2 = findLeftTop(rectFirst)  
 lb2 = findLeftBottom(rectFirst)  
 return [rb1, tr1, lt2, lb2]  
 else:  
 return rectFirst  
  
  
# Обработка изображения  
def process\_image(input\_image, output\_image, target\_height, target\_width):  
 # Увеличение резкости изображения  
 adjust\_sharpness(input\_image, output\_image, 7)  
 img = cv2.imread(output\_image)  
 # Применение размытия Гаусса для сглаживания изображения  
 img = cv2.GaussianBlur(img, (3, 3), 0)  
 img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 # Применение алгоритма Canny для выделения границ на изображении  
 res = cv2.Canny(img\_gray, 100, 200)  
 # Применение размытия Гаусса для уменьшения шума на изображении  
 blur\_canny = cv2.GaussianBlur(res, (3, 3), 0)  
 # Применение преобразования Хафа для поиска линий на изображении  
 lines = cv2.HoughLinesP(blur\_canny, 1, np.pi / 180, 30, minLineLength=100, maxLineGap=10)  
 lines = lines[:, 0, :]  
 for x1, y1, x2, y2 in lines:  
 cv2.line(res, (x1, y1), (x2, y2), (255, 255, 255), 3)  
 # Применение преобразования Хафа для поиска линий на изображении  
 contours, \_ = cv2.findContours(res, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
 # Нахождение двух наибольших контуров  
 biggest, big = foundContours(contours)  
 # Создание бинаризированной маски  
 black\_img = np.zeros(img.shape, dtype=np.uint8)  
 black\_img.fill(0)  
 # Заполнение найденных контуров на черном изображении зеленым и красным цветом соответственно  
 cv2.fillPoly(black\_img, biggest, color=(0, 255, 0))  
 cv2.fillPoly(black\_img, big, color=(0, 0, 255))  
 # Нахождение минимальной ограничивающей прямоугольной области для каждого из контуров  
 rect = cv2.minAreaRect(biggest)  
 rect2 = cv2.minAreaRect(big)  
 # Нахождение минимальной ограничивающей прямоугольной области для каждого из контуров  
 box = np.intp(cv2.boxPoints(rect))  
 box2 = np.intp(cv2.boxPoints(rect2))  
 # Нахождение минимальной ограничивающей прямоугольной области для каждого из контуров  
 final\_shape = defineFinalRect(box, box2)  
  
 tr = findTopRight(final\_shape)  
 lb = findLeftBottom(final\_shape)  
 # Определение центра итоговой формы  
 center\_x = (tr[0] - lb[0]) / 2 + lb[0]  
 center\_y = (tr[1] - lb[1]) / 2 + lb[1]  
 # Чтение исходного изображения  
 img = cv2.imread(input\_image, cv2.IMREAD\_COLOR)  
 # Нахождение матрицы преобразования для поворота исходного изображения  
 # на определенный угол, вычисляемый на основе итоговой формы  
 rotate\_matrix = cv2.getRotationMatrix2D(center=(center\_x, center\_y), angle=findEgle(final\_shape), scale=1)  
 height, width = findHighWidth(findMaxRectangle(final\_shape))  
 rotated\_image = cv2.warpAffine(img, rotate\_matrix, dsize=(height, width))  
 # Обрезка повернутого изображения до заданных размеров  
 new\_img = cv2.getRectSubPix(rotated\_image, (target\_width, target\_height), (center\_x, center\_y))  
  
 return new\_img  
  
  
input\_image = '1.jpg'  
output\_image = 'output.jpg'  
target\_height = 300  
target\_width = 100  
  
normalized\_image = process\_image(input\_image, output\_image, target\_height, target\_width)

cv2.imwrite("output.jpg", normalized\_image)  
cv2.imshow("Normalized image:", normalized\_image)  
cv2.waitKey(0)  
cv2.destroyAllWindows()

Программа 2: обрабатывает датасет ненормализованных фото.

import os  
import cv2  
import math  
import numpy as np  
from PIL import Image  
from PIL import ImageEnhance  
  
# Объединение двух ограничивающих прямоугольников в один  
def defineFinalRect(box1, box2):  
 final\_shape = []  
 for point in box1:  
 final\_shape.append(point)  
 for point in box2:  
 final\_shape.append(point)  
 return final\_shape  
  
  
# Корректировка резкости изображения и сохранение результата в файл  
def adjust\_sharpness(input\_image, output\_image, factor):  
 image = Image.open(input\_image)  
 enhancer\_object = ImageEnhance.Sharpness(image)  
 out = enhancer\_object.enhance(factor)  
 out.save(output\_image)  
  
  
# Отображение изображения в окне с заданным заголовком  
def printImage(mes, img):  
 cv2.imshow(mes, img)  
 cv2.waitKey(0)  
 cv2.destroyAllWindows()  
  
  
# Поиск двух контуров с наибольшим количеством точек  
def foundContours(contours):  
 max\_index = 0  
 second\_index = 0  
 max\_value = 0  
 second\_value = 0  
 index = 0  
 for tup in contours:  
 if len(tup) >= max\_value:  
 second\_value = max\_value  
 second\_index = max\_index  
 max\_value = len(tup)  
 max\_index = index  
 elif len(tup) >= second\_value:  
 second\_value = len(tup)  
 second\_index = index  
 index += 1  
 return contours[max\_index], contours[second\_index]  
  
  
# Нахождение точки правого верхнего угла  
def findTopRight(points):  
 max\_y = 0  
 res = []  
 for point in points:  
 if point[1] > max\_y:  
 max\_y = point[1]  
 res = point  
 return res  
  
  
# Нахождение точки левого нижнего угла  
def findLeftBottom(points):  
 min\_x = math.inf  
 min\_y = math.inf  
 res = []  
 for point in points:  
 if point[1] < min\_y:  
 min\_x = point[0]  
 min\_y = point[1]  
 res = point  
 return res  
  
  
# Нахождение точки правого нижнего угла  
def findRightBottom(points):  
 lb = findLeftBottom(points)  
 tr = findTopRight(points)  
 \_, width = findHighWidth(points)  
 unknown = []  
 for p in points:  
 if p[0] != lb[0] and p[1] != lb[1] and p[0] != tr[0] and p[1] != tr[1]:  
 unknown.append(p)  
 min\_y = math.inf  
 point = []  
 for p in unknown:  
 if p[1] < min\_y:  
 min\_y = p[1]  
 point = p  
 return point  
  
# Нахождение точки левого верхнего угла  
def findLeftTop(points):  
 rb = findRightBottom(points)  
 lb = findLeftBottom(points)  
 tr = findTopRight(points)  
 for p in points:  
 if not np.array\_equal(p, rb) and not np.array\_equal(p, lb) and not np.array\_equal(p, tr):  
 return p  
  
# Нахождение центра прямоугольника  
def foundCenter(points):  
 top\_right = findTopRight(points)  
 left\_bottom = findLeftBottom(points)  
 return [(top\_right[0] - left\_bottom[0])/2 + left\_bottom[0], (top\_right[1] - left\_bottom[1])/2 + left\_bottom[1]]  
  
  
# Нахождение высоты и ширины прямоугольника  
def findHighWidth(points):  
 right\_top = findTopRight(points)  
 left\_bottom = findLeftBottom(points)  
 return right\_top[1]-left\_bottom[1], right\_top[0] - left\_bottom[0]  
  
  
# Проверка, является ли второй прямоугольник частью первого прямоугольника  
def isSecondPartOfFirst(rectFirst, rectSecond):  
 top\_right = findTopRight(rectFirst)  
 left\_bottom = findLeftBottom(rectFirst)  
 center = foundCenter(rectSecond)  
 print("tr ", top\_right)  
 print("lb ", left\_bottom)  
 print("center ", center)  
 return (left\_bottom[0] < center[0] < top\_right[0] and  
 left\_bottom[1] < center[1] < top\_right[1] ) or (  
 left\_bottom[0] > center[0] > top\_right[0] and  
 left\_bottom[1] > center[1] > top\_right[1]) or (  
 left\_bottom[0] > center[0] > top\_right[0] and  
 left\_bottom[1] < center[1] < top\_right[1])  
  
  
# Нахождение угла наклона прямоугольника  
def findEgle(points):  
 box = np.intp(points)  
 edge1 = np.intp((box[1][0] - box[0][0], box[1][1] - box[0][1]))  
 edge2 = np.intp((box[2][0] - box[1][0], box[2][1] - box[1][1]))  
 edgeMax = edge1  
 if cv2.norm(edge2) > cv2.norm(edge1):  
 edgeMax = edge2  
  
 reference = (1, 0)  
 angle = 180.0 / math.pi \* math.acos(  
 (reference[0] \* edgeMax[0] + reference[1] \* edgeMax[1]) / (cv2.norm(reference) \* cv2.norm(edgeMax)))  
 if edgeMax[1] > 0:  
 angle = 270 + angle  
 else:  
 angle = 90 - angle  
  
 return angle  
  
  
# Проверка, расположен ли первый прямоугольник слева от второго  
def isFirstLeft(rectFirst, rectSecond):  
 tr = findTopRight(rectFirst)  
 tr2 = findTopRight(rectSecond)  
 return tr[0] < tr2[0]  
  
  
# Нахождение наибольшего прямоугольника  
def findMaxRectangle(points):  
 min\_x = math.inf  
 min\_y = math.inf  
 max\_x = 0  
 max\_y = 0  
 for point in points:  
 if point[0] > max\_x:  
 max\_x = point[0]  
 if point[0] < min\_x:  
 min\_x = point[0]  
 if point[1] > max\_y:  
 max\_y = point[1]  
 if point[1] < min\_y:  
 min\_y = point[1]  
 return [[min\_x, min\_y],[max\_x, min\_y],[min\_x, max\_y],[max\_x, max\_y]]  
  
  
# Определение итогового прямоугольника  
def defineFinalRect(rectFirst, rectSecond):  
 print(isSecondPartOfFirst(rectFirst, rectSecond))  
 if isSecondPartOfFirst(rectFirst, rectSecond):  
 return rectFirst  
 else:  
 if findEgle(rectFirst) - findEgle(rectSecond) < 15:  
 if isFirstLeft(rectFirst, rectSecond):  
 rb1 = findRightBottom(rectFirst)  
 tr1 = findTopRight(rectFirst)  
 lt2 = findLeftTop(rectSecond)  
 lb2 = findLeftBottom(rectSecond)  
 return [rb1, tr1, lt2, lb2]  
 else:  
 rb1 = findRightBottom(rectSecond)  
 tr1 = findTopRight(rectSecond)  
 lt2 = findLeftTop(rectFirst)  
 lb2 = findLeftBottom(rectFirst)  
 return [rb1, tr1, lt2, lb2]  
 else:  
 return rectFirst  
  
  
# Обработка изображения  
def process\_image(input\_image, output\_image, target\_height, target\_width):  
 # Увеличение резкости изображения  
 adjust\_sharpness(input\_image, output\_image, 7)  
 img = cv2.imread(output\_image)  
 # Применение размытия Гаусса для сглаживания изображения  
 img = cv2.GaussianBlur(img, (3, 3), 0)  
 img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 # Применение алгоритма Canny для выделения границ на изображении  
 res = cv2.Canny(img\_gray, 100, 200)  
 # Применение размытия Гаусса для уменьшения шума на изображении  
 blur\_canny = cv2.GaussianBlur(res, (3, 3), 0)  
 # Применение преобразования Хафа для поиска линий на изображении  
 lines = cv2.HoughLinesP(blur\_canny, 1, np.pi / 180, 30, minLineLength=100, maxLineGap=10)  
 lines = lines[:, 0, :]  
 for x1, y1, x2, y2 in lines:  
 cv2.line(res, (x1, y1), (x2, y2), (255, 255, 255), 3)  
 # Применение преобразования Хафа для поиска линий на изображении  
 contours, \_ = cv2.findContours(res, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
 # Нахождение двух наибольших контуров  
 biggest, big = foundContours(contours)  
 # Создание бинаризированной маски  
 black\_img = np.zeros(img.shape, dtype=np.uint8)  
 black\_img.fill(0)  
 # Заполнение найденных контуров на черном изображении зеленым и красным цветом соответственно  
 cv2.fillPoly(black\_img, biggest, color=(0, 255, 0))  
 cv2.fillPoly(black\_img, big, color=(0, 0, 255))  
 # Нахождение минимальной ограничивающей прямоугольной области для каждого из контуров  
 rect = cv2.minAreaRect(biggest)  
 rect2 = cv2.minAreaRect(big)  
 # Нахождение минимальной ограничивающей прямоугольной области для каждого из контуров  
 box = np.intp(cv2.boxPoints(rect))  
 box2 = np.intp(cv2.boxPoints(rect2))  
 # Нахождение минимальной ограничивающей прямоугольной области для каждого из контуров  
 final\_shape = defineFinalRect(box, box2)  
  
 tr = findTopRight(final\_shape)  
 lb = findLeftBottom(final\_shape)  
 # Определение центра итоговой формы  
 center\_x = (tr[0] - lb[0]) / 2 + lb[0]  
 center\_y = (tr[1] - lb[1]) / 2 + lb[1]  
 # Чтение исходного изображения  
 img = cv2.imread(input\_image, cv2.IMREAD\_COLOR)  
 # Нахождение матрицы преобразования для поворота исходного изображения  
 # на определенный угол, вычисляемый на основе итоговой формы  
 rotate\_matrix = cv2.getRotationMatrix2D(center=(center\_x, center\_y), angle=findEgle(final\_shape), scale=1)  
 height, width = findHighWidth(findMaxRectangle(final\_shape))  
 rotated\_image = cv2.warpAffine(img, rotate\_matrix, dsize=(height, width))  
 # Обрезка повернутого изображения до заданных размеров  
 new\_img = cv2.getRectSubPix(rotated\_image, (target\_width, target\_height), (center\_x, center\_y))  
  
 return new\_img  
  
  
# Путь к папке с фотографиями  
photos\_folder = './photos/'  
target\_height = 300  
target\_width = 100  
  
# Итерация по файлам в папке и обработка каждой фотографии  
for filename in os.listdir(photos\_folder):  
 if filename.endswith(".jpg") or filename.endswith(".png"):  
 input\_image = os.path.join(photos\_folder, filename)  
 output\_image = os.path.join(photos\_folder, "output\_" + filename)  
 normalized\_image = process\_image(input\_image, output\_image, target\_height, target\_width)  
 cv2.imwrite(output\_image, normalized\_image)  
 cv2.imshow("Normalized image:", normalized\_image)  
 cv2.waitKey(0)  
 cv2.destroyAllWindows()

# *Список литературы*

1. Рябцев И.А. Система распознавания болезней растений по изображениям листьев на основе нечёткой логики и нейронной сети. ВКР магистра, СПб, СПбПУ, 2019.
2. Дьяконов В., Абраменкова И. MATLAB. Обработка сигналов и изображений. Специальный справочник.­- СПб.: Питер, 2002г.
3. М.Койшибаев. Болезни пшеницы. Атлас ООН. Анкара, 2018

4. Йорданка Станчева. Атлас болезней сельскохозяйственных культур. Т.3. Болезни полевых культур. Изд. ПЕНСОФТ, София, Болгария, 2003