# 电 子 科 技 大 学

# 实 验 报 告

**学生姓名：谢卿云　　学号：2022010910017　指导教师：沈复民**

## 一、 实验项目名称：

Image Filtering and Hybrid Images

## 二、 实验原理：

本实验使用 OpenCV 和 Dlib 库实现面部检测和换脸。下面将介绍面部检测的换脸的核心技术。

### （一）面部检测和关键点定位

面部检测用于在图像中定位人脸。Dlib 通常采用基于方向梯度直方图（HOG）特征结合支持向量机（SVM）分类器的方法。这使得系统能够识别图像中可能包含人脸的区域。检测到人脸后，使用 Dlib 预训练的形状预测器来定位检测到的人脸上的 68 个特定的面部关键点（标志点）。这些标志点包括眉毛、眼睛、鼻子、嘴巴和下颌线上的点，提供了人脸详细的结构表示。这些精确的坐标对于后续步骤至关重要。

### （二）面部网格化

在获得源人物和目标人物面部的关键点后，使用这些关键点作为顶点在面部区域上创建三角网格。Delaunay 三角剖分是常用的算法之一，它将平面上一组离散的点连接起来形成互不重叠的三角形的过程，其中一个关键性质是：任何一个三角形的外接圆内不包含点集中的任何其他点。这个性质保证了生成的三角形尽可能地“饱满”，避免出现狭长锐角的病态三角形，这在面部网格化中非常重要，可以减少图像变形时的扭曲。增量算法是构建 Delaunay 三角剖分的一种常用方法。它生成一组覆盖面部区域的互不重叠的三角形。这种网格结构非常重要，因为它允许对人脸进行分段仿射变换，在变形过程中保持特征之间的局部关系。

**（三）面部对齐**

根据各自的关键点对源人物的面部进行对齐。这涉及到计算一个仿射变换（包括平移、旋转和缩放），以最佳地将源关键点映射到目标关键点。然后使用这个矩阵对源面部图像进行变换。

**（四）变形**

对齐后的源面部进行变形，使其更精确地匹配目标面部的形状。这通过三角网格来完成。对于目标面部网格中的每个三角形，找到源面部网格中对应的三角形。然后为每对三角形计算一个仿射变换。

**（五）无缝融合**

将变形后的源面部区域融合到目标图像中。为了避免明显的接缝并创建自然的效果，通常使用泊松图像编辑技术。这种方法侧重于融合源区域和目标区域的梯度而不是简单的像素值，从而实现更平滑的过渡。

借助 Dlib 强大的人脸分析能力和 OpenCV 强大的图像处理能力，可以有效地实现两张图像之间的面部换脸。

## 三、 实验目的：

1. 掌握面部检测与关键点定位技术，学习并理解 Dlib 库在人脸检测应用，包括 HOG 特征和 SVM 分类器的工作原理。
2. 理解 Delaunay 三角剖分在图像处理中的应用学习如何利用面部关键点进行 Delaunay 三角剖分，构建面部网格，并理解其在图像变形中的重要性。
3. 实践图像仿射变换与变形掌握 OpenCV 中仿射变换的原理和应用，能够根据关键点计算变换矩阵，并对图像区域进行精确扭曲。
4. 学习图像无缝融合技术，理解泊松图像编辑的原理，并将其应用于面部融合，以实现自然、无痕的换脸效果。

## 四、 实验内容：

1. 基于 OpenCV 和 Dlib 库，实现面部检测、关键点定位、Delaunay 三角剖分、面部对齐、图像变形以及无缝融合等核心步骤，完成两张图像之间的面部换脸。

2. 试用 ModelScope 平台上的大模型换脸 Demo，体验其换脸效果，并与手写实现进行对比。

## 五、 实验步骤：

### （一）手写面部识别

本实验基于 OpenCV 和 Dlib 库实现面部检测和换脸，主要包含以下步骤：

1. **环境配置和导入**安装实验所需的库，特别是 Dlib 库，这通常需要预先安装 Visual Studio 和 CMake。导入必要的 Python 库，例如 `cv2`, `numpy`, `dlib`, `PIL`, `matplotlib.pyplot` 读取事先准备用于换脸的两张人物图像，并将它们的大小统一调整为 (300, 300) 像素。
2. **面部检测和关键点定位**：利用 Dlib 的 `get_frontal_face_detector()` 在图像中定位人脸区域，并使用预训练的 `shape_predictor` 检测出人脸的 68 个关键点。此步骤通过实现 `get_landmarks` 函数完成，该函数负责检测人脸并返回关键点坐标。同时，实现 `get_face_mask` 函数，通过计算面部关键点的凸包来生成面部遮罩，确保后续操作只在面部区域进行。
3. **面部网格化**：以检测到的面部关键点作为顶点，对源图像和目标图像的面部区域进行 Delaunay 三角剖分，构建面部网格。此步骤通过实现 `get_delaunay_triangulation` 函数完成，它基于关键点生成面部的三角形网格。
4. **面部对齐**：根据源图像和目标图像的关键点，计算仿射变换矩阵，对源图像的面部进行对齐。这通过实现 `transformation_from_landmarks` 函数完成，该函数计算一个部分仿射变换矩阵，将源面部关键点映射到目标面部关键点。
5. **变形**：对齐后的源面部图像根据目标面部的网格进行仿射变形，使其形状与目标面部更匹配。这通过实现 `warp_img` 函数完成，该函数使用仿射变换矩阵对图像进行几何变换。
6. **无缝融合**：将变形后的源面部区域融合到目标图像中。首先，遍历 Delaunay 三角剖分得到的每个三角形，对源图像中的对应三角形区域进行仿射变换，将其扭曲到目标图像中相应三角形的位置。然后，使用 OpenCV 的 `cv2.seamlessClone` 函数，将扭曲后的面部区域无缝地融合到目标图像中，完成换脸。`seamlessClone` 通过泊松图像编辑技术，确保融合区域的梯度平滑过渡，避免明显的接缝。

### （二）试用大模型 demo 换脸技术

1. 登录网站 https://modelscope.cn/studios/Hardwell/hardwell_face_fusion_light_-release；
2. 上传想要的两张图片并保存输出图像

## 六、 实验数据及结果分析：

### （一）手写面部识别结果

通过运行 proj3.ipynb 文件，我们成功实现了基于 Dlib 和 OpenCV 的面部识别与换脸功能。我们选取 Elon 面部作为源图像，Trump 面部作为目标图像，实验结果展示了从源图像（person1.jpg）到目标图像（person2.jpg）的面部无缝融合。

#### 6.1.1. 面部关键点检测与三角网格可视化

在换脸过程开始时，需要先检查可视化面部关键点检测和 Delaunay 三角剖分的结果，结果如图 1,2 所示。

#### 6.1.2. 换脸结果

图 3 是未经处理的换脸结果。

最终的换脸结果通过泊松融合技术实现，确保了源面部与目标图像背景之间的平滑过渡，减少了视觉上的不自然感，如图 4 所示。

从结果可以看出，面部特征（如眼睛、鼻子、嘴巴）被成功地从源图像转移到目标图像上，并且与目标图像的肤色、光照等环境因素进行了较好的融合。

### （二）大模型换脸

我们试用了 ModelScope 平台上的大模型换脸 Demo。经过大模型处理后的交换融合结果如图 5 所示。与手写实现相比，大模型在细节处理和融合自然度方面表现出更高的水平，尤其是在处理复杂光照和面部表情时。

图 1: Elon 的面部特征提取

图 2: Trump 的面部特征提取

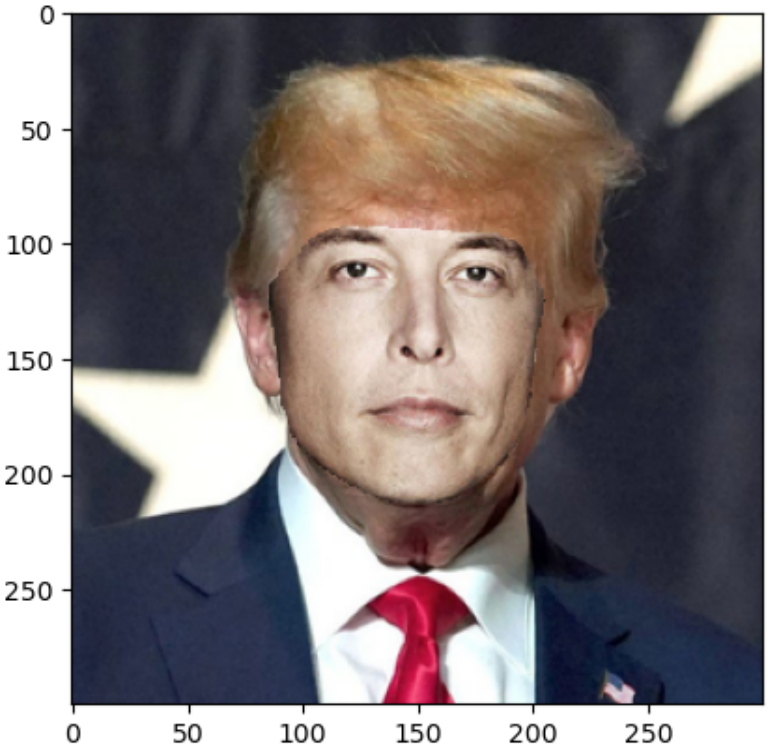


图 3: 简单换脸

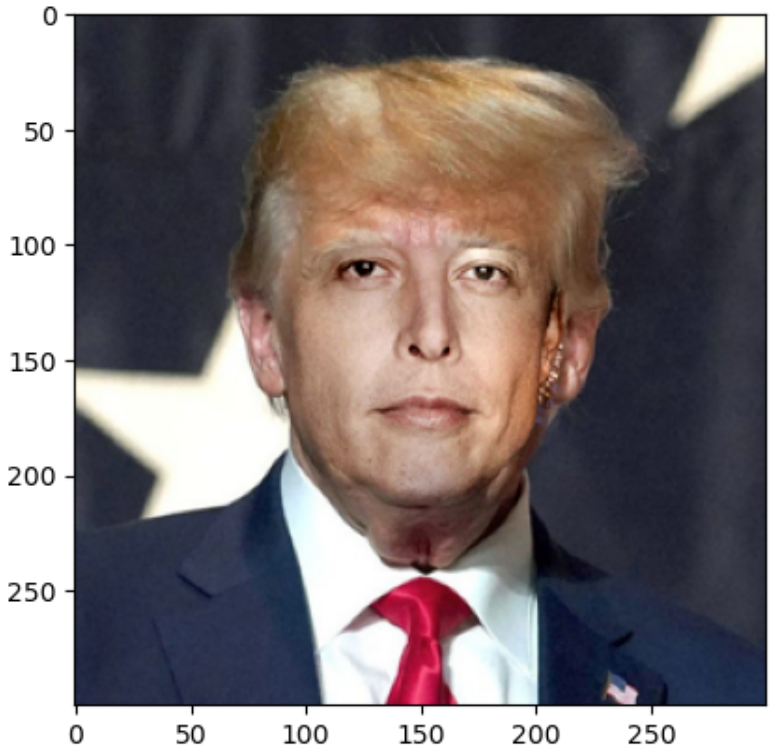


图 4: 经过泊松融合之后的结果

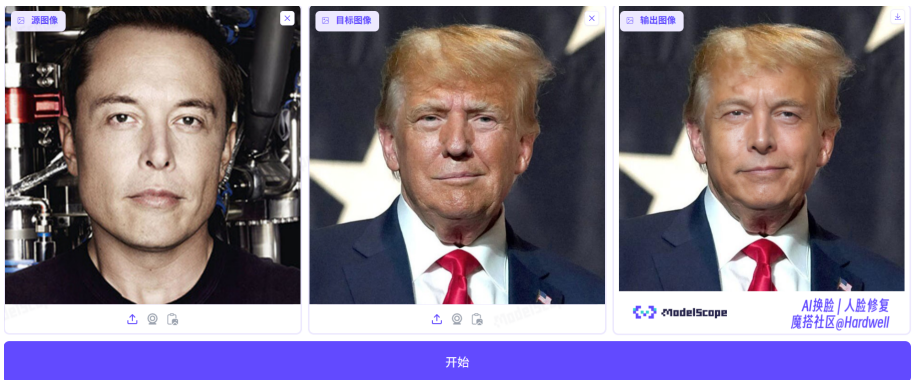


图 5: 大模型换脸结果

## 七、 实验结论：

从图可以看到，手写实现的面部对换

优点在于对算法原理有深入理解和控制，适合学习和研究。缺点是实现复杂，对图像质量和面部姿态要求较高，融合效果可能不如先进模型自然。

从图可以看到，大模型的换脸效果相当出色，Trump 的脸相较原图片更加明亮，看起来也更年轻，皱纹也少了许多，眼神也相当有神，额头也更加宽广，鼻梁

和 Alon 差不多大小，眉毛也和 Alon 形状和浓淡比较相近，也形成了和 Alon 一样的酒窝。说明当下大模型换脸技术已经非常成熟，但还是有一点瑕疵，比如头发的变化并不自然等。

大模型 Demo 优点是操作简便，融合效果通常更自然、更逼真，能够处理更多样化的输入。缺点是缺乏对底层算法的直接控制，且依赖于外部平台。

总体而言，手写实现为我们提供了面部换脸技术的基础知识和实践经验，而大模型 Demo 则展示了当前 AI 技术在图像处理领域的强大能力和未来潜力。

## 八、 总结及心得体会：

通过本次面部识别与换脸的实验，我获得了以下心得体会：

我通过亲手实现面部检测、关键点定位、Delaunay 三角剖分、仿射变换和图像融合等步骤，让我对这些图像处理的核心概念有了更深刻的理解。特别是 Delaunay 三角剖分在保持面部结构完整性方面的巧妙应用，以及泊松融合在实现无缝过渡中的重要性，都让我印象深刻，让我深入了理解图像处理核心概念。

在实验过程中，我大量使用了 OpenCV 和 Dlib 库的各种函数。这极大地提升了我对这两个库的熟练程度和实际应用能力。从零开始构建一个相对复杂的图像处理应用，让我学会了如何将一个大问题分解为多个小模块，并逐一攻克。在调试过程中，也锻炼了我的问题定位和解决能力。

通过手写实现与大模型 Demo 的对比，我直观地感受到了传统图像处理方法在特定场景下的局限性，以及深度学习模型在处理复杂图像任务（如面部表情、光照变化）时所展现出的强大能力和更高的自然度。这促使我思考未来在图像处理领域，如何更好地结合传统算法和深度学习技术。

总而言之，本次实验不仅成功实现了面部换脸功能，更重要的是，它为我提供了一个将理论知识应用于实践的宝贵机会，加深了我对计算机视觉和图像处理领域的理解，并激发了我对未来进一步探索的兴趣。

## 九、 对本实验过程及方法的改进建议：

尽管本次实验成功实现了面部识别与换脸的基本功能，但在实际应用中仍有许多可以改进的方向，以提升其鲁棒性、自然度和用户体验：

1. 现有方法在处理面部姿态（如侧脸、低头）或夸张表情差异较大的图像时，换脸效果可能不佳。可以引入更复杂的 3D 面部模型或姿态归一化技术，将人脸对齐到标准姿态，以提高换脸的鲁棒性。

2. 融合后的面部可能与目标图像的原始光照条件和肤色不完全匹配，导致不自然。可以引入更高级的颜色校正和光照估计技术，如直方图匹配、颜色迁移算法或基于深度学习的光照重建，使融合更加自然。
3. 将当前针对单张图像的换脸技术扩展到视频序列。这需要考虑帧与帧之间的一致性、时间平滑性以及实时处理能力。

**报告评分：**
**指导教师签字：**

# 附录一 代码示例

如代码 1 所示。

代码 1: student.py

```
import cv2
import numpy as np
import dlib
from PIL import Image
import matplotlib.pyplot as plt

image1 = Image.open('../data/person1.jpg')
image1 = image1.resize((300,300))

image2 = Image.open('../data/person2.jpg')
image2 = image2.resize((300,300))

# Converting image to array and converting them to grayscale
img1 = np.array(image1)
img1_gray = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
img2 = np.array(image2)
img2_gray = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

# Initalizing frontal face detector and shape predictor
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor("../data/pretrained_weights/
    shape_predictor_68_face_landmarks.dat")

def visulize_face_landmarks(img, mask, landmarks, triangles):
    plt.figure(figsize=(100, 300))
    plt.subplot(1, 3, 1)
    plt.imshow(img)
    plt.axis('off')

    plt.subplot(1, 3, 2)
    face = cv2.bitwise_and(img, img, mask=mask)
    plt.imshow(face)
    plt.axis('off')

    plt.subplot(1, 3, 3)
    img_landmark = img.copy()

    for triangle in triangles:
        cv2.polylines(img_landmark, [np.array(triangle, np.int32).reshape
            ((-1, 1 ,2))], True, (255, 255, 255), 1)
    for i, landmark_point in enumerate(landmarks):
        x, y = landmark_point[0, 0], landmark_point[0, 1]
        cv2.circle(img_landmark, (x, y), 3, (0, 255, 0), -1)


    plt.imshow(img_landmark)
    plt.axis('off')

    plt.show()

class TooManyFaces(Exception):
    pass

class NoFaces(Exception):
    pass

def get_landmarks(detector, predictor, img):
    faces = detector(img, 1)

    if len(faces) > 1:
        raise TooManyFaces
    if len(faces) == 0:
```

```
            raise NoFaces

        landmarks = np.matrix([[p.x, p.y] for p in predictor(img, faces[0]).
            parts()])
        return landmarks

    def get_face_mask(img, landmarks):
        convexhull = cv2.convexHull(landmarks)
        mask = np.zeros(img.shape, dtype=img.dtype)
        cv2.fillConvexPoly(mask, convexhull, (255, 255, 255))

        return convexhull, mask

    def get_delaunay_triangulation(landmarks, convexhull):
        rect = cv2.boundingRect(convexhull)
        subdiv = cv2.Subdiv2D(rect)

        # Insert points into the subdivision
        for p in landmarks:
            # Need to insert as integer tuples
            subdiv.insert((int(p[0,0]), int(p[0,1])))

        triangles = []
        # Get triangle list from the subdivision
        triangleList = subdiv.getTriangleList()

        # Find the indices of the points forming each triangle
        points = [(int(p[0,0]), int(p[0,1])) for p in landmarks] # Convert
            landmarks to list of tuples for easier lookup

        for t in triangleList:
            pt1 = (int(t[0]), int(t[1]))
            pt2 = (int(t[2]), int(t[3]))
            pt3 = (int(t[4]), int(t[5]))

            # Find the index of each point in the original landmarks
            # Check if the point is within the convex hull to avoid triangles
                outside the face
            if cv2.pointPolygonTest(convexhull, pt1, False) >= 0 and \
               cv2.pointPolygonTest(convexhull, pt2, False) >= 0 and \
               cv2.pointPolygonTest(convexhull, pt3, False) >= 0:

                try:
                    idx1 = points.index(pt1)
                    idx2 = points.index(pt2)
                    idx3 = points.index(pt3)
                    triangles.append([idx1, idx2, idx3])
                except ValueError:
                    # This might happen if the triangulation includes points
                        not exactly matching landmarks
                    # For simplicity, we'll skip such triangles in this
                        example
                    pass

        return triangles

    def transformation_from_landmarks(landmarks1, landmarks2):
        """
        Return an affine transformation [s * R | T] such that:
            sum ||s*R*p1,i + T - p2,i||^2
        is minimized.
        """
        # Solve the procrustes problem by subtracting centroids, scaling by
            the
        # standard deviation, and then using the SVD to calculate the rotation
            . See
        # the following for more details:
        #   https://en.wikipedia.org/wiki/Orthogonal_Procrustes_problem

        M, _ = cv2.estimateAffinePartial2D(np.array(landmarks1), np.array(
```

```
            landmarks2))
        return M

    def warp_img(img, M, dshape):
        warped_img = cv2.warpAffine(img, M, (dshape[1], dshape[0]), flags=cv2.
            INTER_LINEAR, borderMode=cv2.BORDER_REFLECT_101)
        return warped_img

    # Face 1
    landmarks1 = get_landmarks(detector, predictor, img1_gray)

    # Mask
    convexhull1, mask1 = get_face_mask(img1_gray, landmarks1)
    face1 = cv2.bitwise_and(img1, img1, mask=mask1)

    # Delaunay triangulation
    triangles1 = get_delaunay_triangulation(landmarks1, convexhull1)

    # Face 2
    landmarks2 = get_landmarks(detector, predictor, img2_gray)

    # Mask
    convexhull2, mask2 = get_face_mask(img2_gray, landmarks2)

    # Delaunay triangulation
    triangles2 = get_delaunay_triangulation(landmarks2, convexhull2)

    visulize_face_landmarks(img1, mask1, landmarks1, triangles1)
    visulize_face_landmarks(img2, mask2, landmarks2, triangles2)

    '''
    img2: target img (person2)
    warped_img1: warped face img of person1
    face_mask1: warped face mask of person1
    '''
    # Create an empty image for the warped face
    warped_img1 = np.zeros(img2.shape, dtype=img2.dtype)
    face_mask1 = np.zeros(img2.shape, dtype=img2.dtype)

    points1 = np.array(landmarks1)
    points2 = np.array(landmarks2)

    for i, tri_idx in enumerate(triangles1):
        # Get points for img1 using the indices
        pts1 = []
        for j in range(3):
            pts1.append(points1[tri_idx[j]]) # Use index to get the point from
                landmarks1

        # Get points for img2 using the corresponding indices
        pts2 = []
        for j in range(3):
            # Assuming triangles2 has the same structure of indices as
                triangles1
            # This is a simplification, a more robust approach might be needed
                for complex cases
            pts2.append(points2[tri_idx[j]]) # Use the same index to get the
                corresponding point from landmarks2


        # Convert to numpy arrays
        pts1 = np.array(pts1, dtype=np.float32)
        pts2 = np.array(pts2, dtype=np.float32)

        # Get bounding rectangles
        r1 = cv2.boundingRect(pts1)
        r2 = cv2.boundingRect(pts2)

        # Get triangles relative to bounding rects
        tri1Cropped = []
```

```
        tri2Cropped = []

        for j in range(0, 3):
            tri1Cropped.append(((pts1[j][0] - r1[0]), (pts1[j][1] - r1[1])))
            tri2Cropped.append(((pts2[j][0] - r2[0]), (pts2[j][1] - r2[1])))

        # Apply affine transformation
        img1Cropped = img1[r1[1]:r1[1] + r1[3], r1[0]:r1[0] + r1[2]]

        M = cv2.getAffineTransform(np.float32(tri1Cropped), np.float32(
            tri2Cropped))
        warped_triangle = cv2.warpAffine(img1Cropped, M, (r2[2], r2[3]), None,
             flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REFLECT_101)

        # Create mask for the warped triangle
        mask_triangle = np.zeros((r2[3], r2[2], 3), dtype=np.uint8)
        cv2.fillConvexPoly(mask_triangle, np.int32(tri2Cropped), (255, 255,
            255))

        # Copy the warped triangle into the target image
        warped_img1[r2[1]:r2[1]+r2[3], r2[0]:r2[0]+r2[2]] = warped_img1[r2[1]:
            r2[1]+r2[3], r2[0]:r2[0]+r2[2]] * (1 - mask_triangle / 255) +
            warped_triangle * (mask_triangle / 255)
        face_mask1[r2[1]:r2[1]+r2[3], r2[0]:r2[0]+r2[2]] = face_mask1[r2[1]:r2
            [1]+r2[3], r2[0]:r2[0]+r2[2]] * (1 - mask_triangle / 255) +
            mask_triangle * (mask_triangle / 255)


    output_img = img2 * (1 - face_mask1 / 255) + warped_img1 * (face_mask1 /
        255)
    plt.imshow(output_img)

    # Seamless cloning
    center = ((int(convexhull2.mean(axis=0)[0][0]), int(convexhull2.mean(axis
        =0)[0][1])))
    seamlessclone = cv2.seamlessClone(warped_img1, img2, mask2, center, cv2.
        NORMAL_CLONE)
    plt.imshow(seamlessclone)
```