CV 理论作业 1

1. Canny 边缘检测器实现

(1) 噪声抑制

噪声是边缘检测的主要干扰源，本模块采用高斯核进行空域滤波，通过标准差 σ 控制平滑强度。高斯函数的钟形分布特性可保留主要边缘结构，同时滤除高频噪声。实验中默认核尺寸为 5×5（自动补为奇数），σ=1.0，通过调整 kernel_size 参数可观察核尺寸增大时图像的模糊程度与噪声抑制效果的平衡关系。

(2) 梯度幅值与方向计算

采用 Sobel 算子分别计算 x 和 y 方向的一阶导数，Sobel 算子在 3×3 邻域内通过加权差分实现近似梯度计算，其内置的高斯平滑特性可进一步提升抗噪能力。梯度幅值通过欧氏距离公式合成$G = \sqrt{G_x^2 + G_y^2}$，方向角则通过反正切函数计算（$\theta = \arcsin 2\left(G_y, G_x\right)$）并归一化到[0°,180°)范围以避免方向歧义。

```
# 2. 计算梯度大小和方向
    grad_x = cv2.Sobel(blurred, cv2.CV_64F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(blurred, cv2.CV_64F, 0, 1, ksize=3)

    magnitude = np.sqrt(grad_x**2 + grad_y**2)
    direction = np.arctan2(grad_y, grad_x) * 180 / np.pi
    direction = np.mod(direction, 180)
```

(3) 非极大值抑制

为解决梯度幅值在边缘附近的扩散问题，本阶段沿梯度方向进行插值比较。具体而言，将当前像素的梯度幅值与沿正负梯度方向的两个相邻像素比

较，仅保留局部最大值。例如，当方向角为 30°时，比较左上（i-1,j+1）与右下（i+1,j-1）的幅值。

```
# 3. 非极大值抑制
    suppressed = np.zeros_like(magnitude)
    for i in range(1, magnitude.shape[0]-1):
        for j in range(1, magnitude.shape[1]-1):
            angle = direction[i,j]
            # 确定相邻像素位置
            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
                neighbors = [magnitude[i,j-1], magnitude[i,j+1]]
            elif 22.5 <= angle < 67.5:
                neighbors = [magnitude[i-1,j+1], magnitude[i+1,j-1]]
            elif 67.5 <= angle < 112.5:
                neighbors = [magnitude[i-1,j], magnitude[i+1,j]]
            else:
                neighbors = [magnitude[i-1,j-1], magnitude[i+1,j+1]]

            if magnitude[i,j] >= max(neighbors):
                suppressed[i,j] = magnitude[i,j]
```

(4) 滞后阈值

设置高阈值（150）和低阈值（50）对梯度幅值进行分类：高于高阈值的标记为强边缘（255），介于两者间的标记为弱边缘（75），低于低阈值的直接舍弃。既保留了显著边缘，又避免了单一阈值导致的边缘断裂或噪声残留问题。

```
# 4. 滞后阈值
    edges = np.zeros_like(suppressed)
    strong = suppressed > high_thresh
    weak = (suppressed >= low_thresh) & (suppressed <= high_thresh)
```

```
    edges[strong] = 255
    edges[weak] = 75  # 标记弱边缘
```

(5) 连通性分析

对弱边缘进行 3×3 邻域检查，若连接强边缘则升级为有效边缘

```
5. 连通性分析
    for i in range(1, edges.shape[0]-1):
        for j in range(1, edges.shape[1]-1):
            if edges[i,j] == 75:
                if np.any(edges[i-1:i+2, j-1:j+2] == 255):
                    edges[i,j] = 255
                else:
                    edges[i,j] = 0

    return edges.astype(np.uint8)
```

2. Hough 直线检测

将直线方程表示为极坐标形式：$\rho = x\cos\theta + y\sin\theta$其中 ρ 为直线到原点的距离，θ 为直线法线与 x 轴的夹角。设置 ρ 的分辨率为 1 像素，θ 以 1°为间隔离散化，构建二维累加器数组记录各(ρ,θ)组合的投票数。遍历所有边缘像素点，对每个点可能对应的 $\theta\in[0°,180°)$计算 ρ 值，并在累加器中对应位置计数。设定投票阈值 150，仅保留累加值超过此阈值的参数对，确保检测到的直线具有足够的边缘点支撑。阈值降低会增加检测数量但可能引入虚假直线。最终将检测到的(ρ,θ)参数转换为直角坐标系的标准直线方程

```
def hough_line_detection(edges):
    """Hough 直线检测与方程提取"""
```

```
    lines = cv2.HoughLines(edges, 1, np.pi/180, threshold=150)
    line_equations = []
    if lines is not None:
        for line in lines:
            rho, theta = line[0]
            # 转换为直角坐标系方程：x*cosθ + y*sinθ = ρ
            line_equations.append(f"x*{np.cos(theta):.3f} +
y*{np.sin(theta):.3f} = {rho:.1f}")

    return line_equations
```

3. 优化输出

设计输出图片布局以及必须的图片说明介绍，使其可以直观地表现出边缘检测的结果，以及不同大小的滤波器对边缘检测的影响，最后输出在边缘检测的基础上最长五条直线的方程。

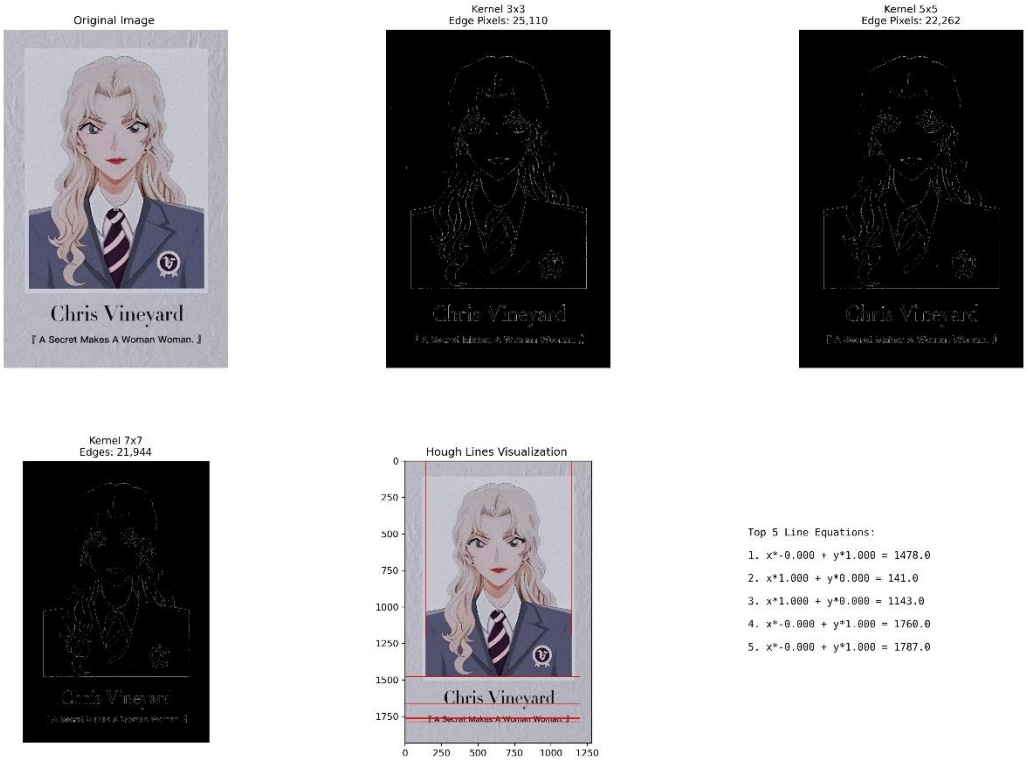
```
def visualize_process(img_path):
    """均衡布局优化"""
    # 读取图像
    img = cv2.imread(img_path)
    if img is None:
        raise ValueError(f"Cannot read the image:{img_path}")

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # 处理参数
    kernel_sizes = [3, 5, 7]

    # 创建平衡布局（2 行 3 列）
```

```
    fig = plt.figure(figsize=(22, 14))
    gs = fig.add_gridspec(nrows=2, ncols=3,
                          height_ratios=[1.2, 1],  # 调整行高比例
                          width_ratios=[1, 1, 1.2],  # 右侧留更多空间
                          hspace=0.3, wspace=0.25)

    # 原始图像（第一行左）
    ax0 = fig.add_subplot(gs[0, 0])
    ax0.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    ax0.set_title("Original Image (Enhanced)", fontsize=12)
    ax0.axis('off')

    # 边缘检测结果（第一行中、右）
    for i, ksize in enumerate(kernel_sizes[:2]):
        ax = fig.add_subplot(gs[0, i+1])
        edges = canny_edge_detector(gray, kernel_size=ksize)
        ax.imshow(edges, cmap='gray')
        ax.set_title(f"Kernel {ksize}x{ksize}\nEdges:
{np.sum(edges>0):,}", fontsize=11)
        ax.axis('off')
     # 第三个边缘检测结果（第二行左）
    ax2 = fig.add_subplot(gs[1, 0])
    edges = canny_edge_detector(gray, kernel_size=7)
    ax2.imshow(edges, cmap='gray')
    ax2.set_title(f"Kernel 7x7\nEdges: {np.sum(edges>0):,}",
fontsize=11)
    ax2.axis('off')



    # Hough 检测结果（第二行中）
```

```
ax1 = fig.add_subplot(gs[1, 1])
final_edges = canny_edge_detector(gray, kernel_size=5)
line_equations = hough_line_detection(final_edges)

result_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB).copy()
lines = cv2.HoughLines(final_edges, 1, np.pi/180, 150)
if lines is not None:
    for line in lines[:]:  # 仅处理前 5 条直线
        rho, theta = line[0]
        a, b = np.cos(theta), np.sin(theta)
        x0 = a*rho
        y0 = b*rho
        pt1 = (max(0, int(x0 - 1200*b)), max(0, int(y0 - 1200*a)))
        pt2 = (min(img.shape[1]-1, int(x0 + 1200*b)),
               min(img.shape[0]-1, int(y0 + 1200*a)))
        cv2.line(result_img, pt1, pt2, (255,0,0), 2, cv2.LINE_AA)

ax1.imshow(result_img)
ax1.set_title("Hough Lines Visualization ", fontsize=12)
ax1.axis('on')

# 方程显示区域（第二行右）
ax3 = fig.add_subplot(gs[1, 2])
text_content = "Top 5 Line Equations:\n\n"
for i, eq in enumerate(line_equations[:5]):  # 显示前 5 个方程
    text_content += f"{i+1}. {eq}\n\n"

ax3.text(0.05, 0.5, text_content,
        fontsize=11,
        family='monospace',
```

```
        verticalalignment='center')
ax3.axis('off')


# 保存优化结果
plt.savefig('balanced_result.jpg',
            dpi=300,
            bbox_inches='tight',
            pad_inches=0.15)
print("优化结果已保存为 balanced_result.jpg")
plt.show()
```

4. 结果输出



5. 实验总结

通过本次实验，我深入理解了 Canny 边缘检测和 Hough 直线检测的工作原

理。Canny 算法通过高斯滤波去噪、计算梯度、非极大值抑制和双阈值处理等步骤，实现了精确的边缘提取；而 Hough 变换则通过参数空间投票机制，将离散的边缘点转换为有意义的直线方程。实验让我直观认识到参数选择对检测结果的重要影响，如高斯核尺寸决定了边缘的清晰度，阈值设置直接影响检测的灵敏度和准确性。这些经典算法为后续更复杂的图像处理任务奠定了坚实基础。