

# “基于自适应直方图均衡化耦合拉普拉斯变换的红外图像增强算法” 论文研究报告

ZYH

28th June 2024

## 1 论文目的

红外图像由于其其在低光照条件下的有效性，广泛应用于夜间监控、遥感探测和军事侦查等领域。然而，这些图像通常因为对比度低和细节表达不足而难以达到理想的视觉效果。尽管目前许多红外图像增强方法利用图像的灰度特征来改进图像质量，但这些方法往往忽略了图像灰度分布的不均匀性，导致细节丢失和对比度不理想的问题。针对这一问题，该研究提出了一种结合自适应直方图均衡化和拉普拉斯变换的新算法。该算法旨在通过自适应地调整图像的灰度分布和锐化图像边缘，以显著提升红外图像的对比度和清晰度，进而优化其在关键应用中的性能表现。

## 2 研究方法

### 1. 图像分割与灰度不均匀性测量

在图像分割与灰度不均匀性测量阶段，研究首先将输入的红外图像均匀分割成若干非重叠的小区块，以便于独立处理每个区块，从而减少不同区域之间的干扰，更好地保留局部细节。接下来，利用洛伦兹曲线和基尼系数对每个分割后的区块进行灰度不均匀性的测量。洛伦兹曲线是一个用于描述分布均衡性的图形工具，而基尼系数则是一个量化不均匀性的统计度量，其值越高表明图像区块的灰度分布越不均匀。通过计算每个区块的基尼系数，可以准确地评估图像中的灰度分布情况。这一测量不仅帮助识别出需要特别处理的区域，还为后续的自适应直方图均衡化提供了关键的定量依据，使得整体的图像增强过程更加精确和有效。

### 2. 自适应直方图均衡化

在自适应直方图均衡化阶段，算法根据前一步得到的灰度不均匀性测量结果，为每个图像区块构建特定的上限和下限阈值。这些阈值是基于各区

块的基尼系数计算得来，目的是为了调整每个区块内的灰度级分布，以改善图像对比度而不过度增强或减少细节信息。自适应直方图均衡化通过修改局部直方图来实现，具体方法是将每个区块的灰度级分布拉伸或压缩至新的上下限阈值范围内，从而增强了图像的局部对比度。这种方法有效地防止了由传统全局直方图均衡化引起的细节丢失问题，因为它允许在保留图像细节的同时，针对每个区块的特定需求进行调整。结果是，增强后的图像在视觉上更为平衡，对比度提高，同时更好地保留了图像的细节和纹理信息，使得图像更适合进一步的分析和处理。

### 3. 平滑滤波去噪

在平滑滤波去噪阶段，为了减少图像中的噪声并提高后续增强处理的质量，算法采用了平滑滤波技术对图像进行预处理。此步骤主要利用一个预定大小的滤波器（如常见的  $3 \times 3$  平均滤波器或高斯滤波器）直接作用于图像的每个像素。滤波器通过计算中心像素及其邻域内像素的平均值（或加权平均值，取决于滤波类型），来平滑图像区域，有效减少随机噪声的影响。这种处理不仅降低了图像的高频噪声成分，还帮助保持了图像的基本结构和边缘信息，为接下来的锐化处理和对比度增强创造了良好的基础。通过这种方法，图像的整体质量得到提升，使得细节更加清晰，同时也避免了在增强过程中可能引入的噪声放大问题，确保最终图像的视觉效果更为自然和舒适。

### 4. 拉普拉斯变换锐化处理

在拉普拉斯变换锐化处理阶段，算法通过引入拉普拉斯算子来增强图像的边缘和细节，从而提升图像的清晰度。拉普拉斯变换是一种基于二阶导数的边缘检测技术，可以强化图像中的高频信息，即图像的边缘部分。在处理过程中，首先将经过平滑滤波去噪的图像作为输入，然后应用拉普拉斯算子。这通常涉及到计算每个像素点周围邻域的灰度值差异，通过增强这些差异来突出边缘和细节。具体来说，算法采用了一个扩展的 8 邻域拉普拉斯算子，该算子不仅考虑了像素的水平和垂直邻域，还包括了对角线方向的邻域，这样可以更全面地捕捉图像的细节。通过这种方法，图像的边缘被显著增强，使得整体图像看起来更为清晰和锐利。这一步骤是图像增强过程中至关重要的，因为它直接影响到增强图像的视觉质量，特别是在边缘清晰度和细节表达上的改善。

### 5. 图像重建

在图像重建阶段，算法的目标是合并所有经过前面步骤处理的图像区块，以形成最终的增强后图像。这一步骤至关重要，因为它需要确保处理后的各个区块在视觉上无缝连接，避免产生不自然的边界或图像失真。具体操作中，首先使用双线性插值方法来平滑每个处理过的图像区块的边界，这种插值技术可以有效地平衡相邻区块间的灰度差异，从而消除或减少区块间

的可见边界。随后，将所有插值后的区块重新组合，形成一个连贯的整体图像。这个重建过程不仅提升了图像的整体视觉效果，还确保了增强的自然性和均匀性，使得图像在保持高对比度和清晰度的同时，还具有良好的视觉连贯性。通过这种综合的处理，最终得到的图像在细节上更加精细，视觉效果更为吸引，适合进行进一步的分析或直接应用。

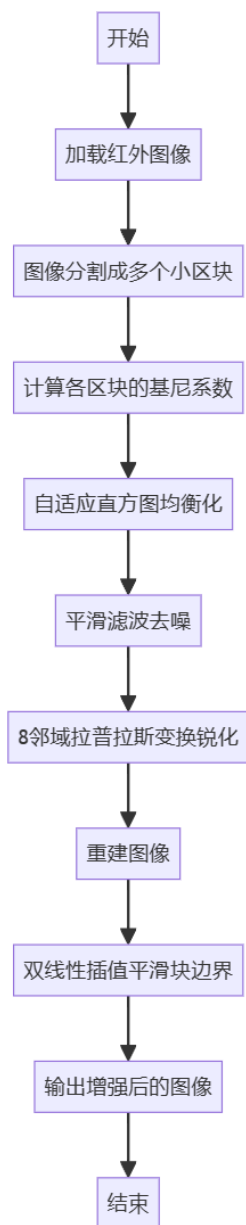


Figure 1: 实验流程图

## 3 具体实现方法

### 1. 算法设计原理

自适应直方图均衡化：基于图像各区块的灰度分布不均匀性评估结果，算法设计包括自适应地设置每个图像区块的直方图阈值。这种自适应策略的核心是使用基尼系数作为不均匀性的量化指标，根据该指标动态调整直方图的拉伸或压缩程度。这样可以在不同区域根据其灰度分布的特性进行个性化的对比度增强，避免了全局直方图均衡化中常见的细节溢出和图像过曝的问题。

拉普拉斯变换的优化实现：为了更精细地处理图像边缘和细节，算法采用了一个扩展的拉普拉斯变换，包括传统的四邻域以及对角线方向，形成 8 邻域的处理框架。这种方法可以更全面地捕捉和增强图像的细节，尤其是在边缘过渡区域，从而提高图像的总体清晰度和边缘响应。

### 2. 算法结合思路

在整体设计中，本研究算法首先通过自适应直方图均衡化来增强图像的对比度，随后利用拉普拉斯变换对细节和边缘进行锐化处理。这两个步骤相互补充：直方图均衡化处理整体灰度分布，优化图像对比度；而拉普拉斯变换则专注于边缘和高频细节的增强。此外，平滑滤波作为预处理步骤，帮助去除噪声，确保后续处理更加聚焦于真实的图像内容。通过这种结合策略，算法在增强图像质量的同时，避免了过度处理带来的负面影响，如边缘伪影和噪声放大。

### 3. 保护图像细节和边缘信息

为了保护图像细节和边缘信息，算法特别强调在增强过程中的平衡和精细处理。在自适应直方图均衡化中，通过动态设置直方图的阈值，根据每个区块的灰度不均匀性量化指标（基尼系数），精确调整图像的局部对比度，这样做可以有效保护图像中的细节，避免过度增强导致的细节损失。在使用拉普拉斯变换时，通过引入 8 邻域的考虑，增强了图像边缘的同时也保持了边缘的自然过渡，避免了锐化处理常见的过度增强问题。最后，在图像重建阶段，采用双线性插值和相邻区块像素加权平均的方法平滑处理增强后的区块边界，这一策略确保了处理后的图像在视觉上的连贯性和自然感，同时保护了边缘和细节信息。

通过这种综合的处理策略，算法不仅提升了图像的视觉效果，还确保了图像细节和边缘信息的保护，满足了红外图像在各种应用场景下的高质量需求。

## 4 图像质量评价方法

### 4.1 客观评价指标

#### 4.1.1 峰值信噪比 (Peak Signal-to-Noise Ratio, PSNR)

峰值信噪比 (PSNR) 是衡量图像恢复质量的一个常用客观指标, 广泛用于图像处理领域来评价图像压缩和图像增强技术的效果。PSNR 计算基于原始图像与处理后图像之间的均方误差 (MSE)。具体来说, 首先计算原始图像与增强图像在像素级的差异, 将这些差异的平方值求和后除以图像的总像素数, 得到 MSE。然后, PSNR 通过取信号可能的最大值 (对于 8 位图像是 255) 的平方, 除以 MSE, 再取 10 倍的对数得到。PSNR 的单位是分贝 (dB), 值越高表明图像质量越好, 意味着误差越小。在图像增强的背景下, 高 PSNR 通常指示图像的视觉质量被良好保留或提升, 特别是在处理过程中噪声得到了有效控制。这一指标尤其重要, 因为它提供了一个快速评估图像处理算法性能的方法, 帮助研究者了解增强算法在保真度方面的表现。

#### 4.1.2 结构相似性指数 (Structural Similarity Index, SSIM)

结构相似性指数 (SSIM) 是一个衡量两幅图像视觉相似度的先进指标, 特别在评价图像处理效果时非常有用, 如图像增强、压缩后的质量评估。与传统的基于像素的误差度量 (如 PSNR) 不同, SSIM 考虑的是图像的结构信息, 这更接近人眼对图像质量的感知方式。SSIM 的计算基于三个比较参数: 亮度、对比度和结构。首先, 它会计算两幅图像的亮度函数的均值, 然后评估这两个均值的相似度。接下来, 比较两幅图像的标准差 (即对比度) 以及它们标准差的相关性。最后, 通过比较两幅图像的协方差与各自方差的乘积, SSIM 评估它们的结构相似性。SSIM 的值范围从 -1 到 1, 其中 1 表示两图完全相同。在实际应用中, SSIM 可以帮助识别图像增强算法是否在改善图像视觉效果的同时保持了图像的结构特性, 使得此指标非常适合评估增强算法是否过度改变了图像的本质视觉属性。这种评价方法尤其适用于那些对图像的细节和结构保真性有高要求的应用, 如医疗影像处理和卫星图像分析。

#### 4.1.3 边缘强度 (Edge Intensity)

边缘强度是一个评估图像处理算法对图像细节和边缘增强效果的重要客观指标。它专门用来量化图像中边缘部分的清晰度和强度, 这对于图像增强技术尤其关键, 因为边缘是图像中最能表现细节和结构的区域。在计算边缘

强度时,首先需要应用边缘检测算子(如 Sobel 算子或 Canny 算子)来识别图像中的边缘区域。这些算子通过计算图像中像素点的梯度值来确定哪些区域是边缘。边缘的梯度值越高,表明该位置的像素变化越剧烈,即边缘越明显。计算完所有边缘的梯度值后,通过对这些梯度值求平均或其他统计方法(如取中位数或建立梯度直方图),可以得到整体的边缘强度指标。边缘强度的高低直接影响到图像的视觉质量,尤其是在对比度和清晰度方面。一个高的边缘强度值通常表示图像增强算法能有效地突出图像中的细节,使得整个图像看起来更为锐利和清晰。因此,边缘强度不仅是评价图像增强效果的一个重要指标,也是调整和优化图像处理算法时的关键参考数据。

#### 4.1.4 信息熵 (Entropy)

信息熵是一个衡量图像内容复杂性和信息量的重要指标,广泛用于评估图像处理算法,特别是图像增强技术的效果。信息熵反映了图像中像素值分布的随机性和多样性,一个高的信息熵值通常表示图像包含丰富的信息和细节。在图像处理中,特别是在图像增强任务中,目标之一就是提高图像的信息熵,从而增加图像的可用信息,使得图像更加清晰且细节更加丰富。

信息熵的计算涉及到统计图像中每个像素值的出现概率,然后利用这些概率值计算整个图像的熵值。具体来说,首先需要统计图像中每个灰度级的频率,然后将这些频率转换为概率,最后将所有概率值代入熵的计算公式:  $H = -\sum_{i=1}^n p_i \log_2 p_i$ , 其中  $p_i$  是某个灰度级出现的概率,  $n$  是总的灰度级数。这个公式计算的是所有可能灰度级的概率分布的加权平均信息量,得出的熵值越大,表示图像的内容越不确定,即图像的复杂度和信息量越大。

在图像增强中,通过调整图像的对比度和亮度、优化色彩分布等手段可以提升图像的信息熵,这通常意味着图像的视觉质量得到了提升,因为更高的信息熵使得观察者能从图像中得到更多的信息。因此,信息熵不仅是评估图像增强效果的一个有效指标,同时也是图像处理算法设计和优化的重要考量因素。

## 5 论文算法复现

我采用 Python 编程语言结合 OpenCV 库来尝试实现论文中的算法,但是发现结果并没有达到预期,在经历长时间的调参后也没能达到论文中的处理效果,程序的目的是通过以下步骤增强红外图像的对比度和清晰度:

1. 图像分割: 将原始图像切割成小块,以便独立地调整每块的对比度,同时尽量保留局部细节。
2. 基尼系数计算: 度量各个图像块的灰度分布不均匀性,以指导自适应

直方图均衡化的参数设置。

3. 自适应直方图均衡化：根据基尼系数调整每个图像块的直方图，以改善其对比度。

4. 平滑滤波去噪：在锐化边缘之前，先通过平滑滤波减少图像噪声。

5. 拉普拉斯锐化处理：用拉普拉斯变换增强图像的边缘和细节。

6. 图像重建：将处理过的图像块重新组合成完整图像，并通过双线性插值平滑块之间的过渡区域。

7. 输出结果：最终应输出原始图像和增强后的图像以供对比。

然而，最终的增强图像出现了一些质量问题，包括对比度和清晰度都没有达到预期效果。这可能是由于以下几个因素造成的：

1. 图像块的大小和边缘处理可能不正确，导致在图像重建时出现问题。

2. 基尼系数的计算可能未能准确反映图像块的灰度分布不均匀性，从而影响了直方图均衡化的效果。

3. 直方图均衡化的自适应阈值可能设置不当，未能达到增强对比度的目的。

4. 拉普拉斯锐化的参数可能过强或处理方式不当，引入了噪声和失真。



Figure 2: 处理前的图像

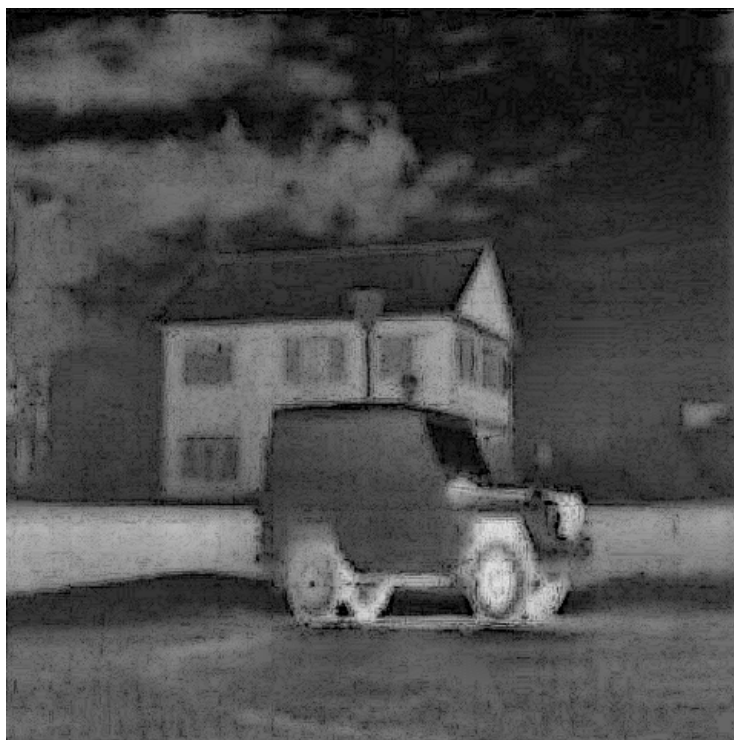


Figure 3: 处理后的图像

## 6 所得经验

论文提出的红外图像增强算法结合了自适应直方图均衡化和拉普拉斯变换锐化处理两种方法，具有理论上的优越性。自适应直方图均衡化通过考虑图像中的灰度不均匀性，使得算法能够在增强图像对比度的同时，更好地保留局部的图像细节。与此同时，拉普拉斯变换锐化处理能有效地突出图像边缘，增强图像的清晰度，特别是在细节丰富的区域表现得更加出色。

这种结合算法的优越性在于，它不仅仅局限于全局的或单一的图像处理方法，而是通过区块处理和权衡各区块之间的不同特性，达到更加细腻和个性化的增强效果。在理论上，这样的处理方法更贴合实际图像中各个区域的不同需求，尤其适用于在多变环境中获取的红外图像，其中包含了丰富的场景动态和关键信息。

在实际应用中，即便复现失败，这种结合算法的尝试也是一次宝贵的学习经验。它让我们认识到了算法在不同应用条件下的适应性，以及在实际操作中每一步精确实现的重要性。通过这次尝试，我们更加明白了在图像处理中如何根据图像的具体特点和应用场景去调整和优化参数，以及如何平衡算法中的不同组件以达到最佳的效果。

## 附件

### 1. 代码展示

以下是本研究中使用的 Python 代码片段（未实现论文效果）：

```
1 import cv2
2 import numpy as np
3
4
5 def calculate_gini_coefficient(cdf):
6     """ 计算基尼系数 """
7     n = len(cdf)
8     coefficient = 2.0 / n
9     constant = (n + 1) / n
10    weighted_sum = sum((i + 1) * val for i, val in
11                        enumerate(cdf))
12    gini = coefficient * weighted_sum / sum(cdf) -
13        constant
14    return gini
15
16 def adaptive_histogram_equalization(image,
17                                     gini_coeff):
18     """ 自适应直方图均衡化 """
19     h, w = image.shape
20     hmax = np.max(image)
21     n = w * h
22     Tu = (1 - gini_coeff) * hmax + gini_coeff * (n /
23         hmax)
24     Tl = n / hmax
25
26     image[image < Tu] = (image[image < Tu] / Tu) * Tl
27     image[image >= Tu] = Tl + (image[image >= Tu] -
28         Tu)
29     return image
```

```

28 def bilinear_interpolation(sub_blocks,
    original_shape, block_size):
29     """Use bilinear interpolation to smooth block
        edges after processing."""
30     # Determine number of blocks in each dimension
31     num_blocks_y = original_shape[0] // block_size +
        (1 if original_shape[0] % block_size != 0 else
         0)
32     num_blocks_x = original_shape[1] // block_size +
        (1 if original_shape[1] % block_size != 0 else
         0)
33
34     # Initialize empty list to store the new blocks
35     new_blocks = []
36
37     for y in range(num_blocks_y):
38         for x in range(num_blocks_x):
39             i = y * num_blocks_x + x
40             if i < len(sub_blocks):
41                 block = sub_blocks[i]
42                 # Check if the block needs padding
43                 if block.shape[0] < block_size:
44                     block = cv2.copyMakeBorder(block,
                        0, block_size -
                        block.shape[0], 0, 0,
                        cv2.BORDER_REFLECT)
45                 if block.shape[1] < block_size:
46                     block = cv2.copyMakeBorder(block,
                        0, 0, 0, block_size -
                        block.shape[1],
                        cv2.BORDER_REFLECT)
47                 new_blocks.append(block)
48             else:
49                 # Create a new block if we're out of
                    actual blocks (for edge cases)
50                 new_blocks.append(np.zeros((block_size,

```

```

        block_size), dtype=np.uint8))
51
52 def reconstruct_image_from_blocks(blocks,
    orig_shape, block_size):
53     """Reconstruct the image from its blocks
        after processing."""
54     # Calculate the number of blocks along the
        height and width
55     num_blocks_vert = int(np.ceil(orig_shape[0] /
        block_size))
56     num_blocks_horiz = int(np.ceil(orig_shape[1]
        / block_size))
57
58     # Initialize an empty image with the original
        shape
59     reconstructed = np.zeros(orig_shape,
        dtype=blocks[0].dtype)
60
61     # Place each block back into the image at the
        correct location
62     for idx, block in enumerate(blocks):
63         # Calculate the block's position in the
            image
64         row = (idx // num_blocks_horiz) *
            block_size
65         col = (idx % num_blocks_horiz) *
            block_size
66
67         # Calculate the dimensions of the block
68         block_height = min(block_size,
            orig_shape[0] - row)
69         block_width = min(block_size,
            orig_shape[1] - col)
70
71         # If the block is smaller than the
            expected block_size, resize it

```

```

72         if block.shape[0] != block_height or
           block.shape[1] != block_width:
73             block = cv2.resize(block,
                                   (block_width, block_height),
                                   interpolation=cv2.INTER_LINEAR)
74
75         # Insert the block into the reconstructed
           image
76         reconstructed[row:row + block_height,
                        col:col + block_width] = block
77
78     return reconstructed
79
80     # Usage
81     reconstructed_img =
           reconstruct_image_from_blocks(enhanced_blocks,
           original_img.shape, block_size)
82
83     # Resize to original image size using bilinear
           interpolation
84     new_image = cv2.resize(reconstructed_img,
                             (original_shape[1], original_shape[0]),
                             interpolation=cv2.INTER_LINEAR)
85     return new_image
86
87
88 # 加载图像并转换为灰度图
89 image_path = 'plane.png' # 更换为您的图像路径
90 original_img = cv2.imread(image_path, 0)
91
92 # 图像分割
93 block_size = 8 # 分割的块的大小
94 sub_blocks = [original_img[y:y + block_size, x:x +
           block_size] for x in range(0,
           original_img.shape[1], block_size) for
95                 y in range(0, original_img.shape[0],

```

```

                                block_size)]
96
97 # 处理每个块
98 enhanced_blocks = []
99 for block in sub_blocks:
100     # 计算累积分布函数(cdf)
101     hist = cv2.calcHist([block], [0], None, [256],
102                          [0, 256]).ravel()
102     cdf = hist.cumsum()
103     cdf_normalized = cdf * hist.max() / cdf.max()
104
105     # 计算基尼系数
106     gini_coeff =
107         calculate_gini_coefficient(cdf_normalized)
108
109     # 自适应直方图均衡化
110     block = adaptive_histogram_equalization(block,
111                                              gini_coeff)
112     enhanced_blocks.append(block)
113
114 # After enhancing the blocks, before reconstructing
115 # the image
116 reconstructed_img =
117     bilinear_interpolation(enhanced_blocks,
118                            original_img.shape, block_size)
119
120 # 平滑滤波去噪
121 smoothed_img = cv2.GaussianBlur(reconstructed_img,
122                                  (3, 3), 0)
123
124 # 拉普拉斯锐化处理
125 laplacian = cv2.Laplacian(smoothed_img, cv2.CV_64F)
126 sharpened_img = cv2.convertScaleAbs(smoothed_img -
127                                     0.3 * laplacian)
128

```

```
123 # 展示原始图像和增强后的图像
124 cv2.imshow('Original Image', original_img)
125 cv2.imshow('Enhanced Image', sharpened_img)
126 cv2.waitKey(0)
127 cv2.destroyAllWindows()
```