Final Summary: Anisotropic Diffusion - The Perona Malik PDE Applied

Alles Rebel
MATH 693B Adv Computational PDE
Professor Nguyen-Truc-Dao Nguyen

May 5, 2024

Introduction

Computer vision is rich with applications where edge detection is crucial for enabling computers to interpret the world. One notable application is Simultaneous Localization and Mapping (SLAM), where machines process a series of 2D images captured from sensors to perceive their own location and the surroundings. This capability is fundamental to autonomous systems and AR/VR applications, which require either comprehension of walls and their relative positions or the ability to avoid obstacles. We'll discuss these concepts, along with a Partial Differential Equation called the Perona Malik [1], and it's implementation as a finite difference scheme.

Background

Most SLAM algorithms employ a pipeline that extracts features (e.g., perspective-invariant aspects or corners) from images, creates descriptors of these features, and matches them across various images as time progresses. Descriptors are methods that describe a feature using its local spatial information. By matching these feature descriptors and using a relative sensor such as an accelerometer, trajectories are constructed. Similar to the operation of a Kalman filter, the absolute sensor (the camera) is enhanced by the relative acceleration measurements provided by the accelerometer.

Edges further enhance this process by reducing error magnitudes over time. Edges are detected in the current image, and then a descriptor is generated along the edge. Instead of focusing solely at the feature point, an edge descriptor considers the area around the entire edge. Subsequent images then undergo the same matching procedure for any detected edges. Edges can be linked together, facilitating the generation of bounding boxes or even predicting occlusion.

How to Detect Edges

Edge detection is one of the most extensively studied problems in computer vision. Edges can arise for various reasons, such as intensity variation, depth differences, or texture changes. The simplest approach to edge detection involves using a gradient, or a first-order derivative in the spatial direction of the image. As discussed in class, whenever we encounter a derivative, we can apply a partial difference scheme. Here, we use the forward difference operator to approximate the gradient, which is implemented via one of the most fundamental computer vision operations: 2D convolution filtering. If derivatives in both directions are needed, this can be achieved by convolving the respective kernels, as convolution is associative and commutative.

However, the first derivative is highly sensitive to noise. Every image captured by a sensor inherently contains noise, which can be amplified by this method. To address this, we often resort to the second derivative for detecting rapid intensity changes, known as the Laplacian. Using finite difference schemes, we can implement a first-order central difference approximation for the second derivative. This can be converted into a convolution kernel, which is a well-known kernel among computer vision students. The Laplacian kernel, which is central to this approach, is given by:

$$\text{Laplacian Kernel} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

This kernel is used to detect areas of rapid intensity change in all directions in an image.

Additionally, the Sobel operator, often used to calculate the gradient of the image intensity, uses two separate kernels to capture horizontal and vertical derivative approximations. The kernels are defined as:

$$\text{Sobel Kernel X} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \text{Sobel Kernel Y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

These kernels help in emphasizing edges that are vertically and horizontally oriented by computing the gradient in the X and Y directions, respectively.

Gradient calculations are at the heart of one of the most common edge detection algorithms: the Canny Edge Detector. In addition to gradient calculations, noise reduction methods are applied, along with thresholding of intensities, and edge tracking techniques. For the experiment below, we'll utilize the Canny Edge Detector, implemented using the standard accelerated convolutions provided by OpenCV and see the effect of applying Anisotropic Diffusion.

Anisotropic Diffusion / Perona-Malik

The value of edge detection and its challenges, such as the difficulty in distinguishing closely spaced edges or following a curve, lead to the development of methods like anisotropic diffusion. Perona and Malik applied the diffusive properties of the heat equation to the intensity values of an image's pixels, but restricted its application to areas not deemed edges. This method, known as anisotropic diffusion, smooths out the areas away from edges while preserving the edges themselves.

The PDE

The Perona-Malik anisotropic diffusion is a partial differential equation (PDE) that modifies the image intensity as a function of both space and time. The PDE is given by:

$$\frac{\partial I}{\partial t} = \nabla \cdot (c(\nabla I)\nabla I)$$

where I represents the image intensity, ∇I is the gradient of the intensity, and $c(\nabla I)$ is the diffusion coefficient that is a function of the intensity gradient.

The Finite Difference Scheme

Perona and Malik proposed a finite difference scheme for approximating the above PDE, expressed as follows:

$$I_{i,j}^{n+1} = I_{i,j}^{n} + \lambda \left[c_{i+1,j}(I_{i+1,j}^{n} - I_{i,j}^{n}) + c_{i-1,j}(I_{i-1,j}^{n} - I_{i,j}^{n}) + c_{i,j+1}(I_{i,j+1}^{n} - I_{i,j}^{n}) + c_{i,j-1}(I_{i,j-1}^{n} - I_{i,j}^{n})\right]$$

where λ is a time-step factor, and $c_{i,j}$ is the diffusion coefficient at pixel location (i, j).

Stability analysis by the authors led to the condition:

$$\lambda \leq \frac{1}{4}$$

Intuition of Edge Preservation

The intuition behind the edge preservation capability of the Perona-Malik scheme arises from several observations:

- A correction term is added to the previous value of the pixel, which accounts for the local variations in intensity.
- The correction term incorporates the diffusion coefficient $c(\nabla I)$, which is calculated as:

  $$c(\nabla I) = e^{-\left(\frac{|\nabla I|^2}{K^2}\right)}$$

  where K is a constant that controls the sensitivity of the diffusion process to edges.
- The diffusion coefficient $c(\nabla I)$ is designed to be small at edges (where the gradient $|\nabla I|$ is large), thus reducing diffusion across edges while allowing it in homogenous regions.

- This differential diffusion effectively preserves edges, as areas of high gradient undergo minimal smoothing, maintaining the sharpness and integrity of edges.

Note: The authors present two methods for implementing the correction term, both have the same intuition.

Example Results

To demonstrate both the convolutional kernel method discussed earlier and the implementation of the Finite Difference scheme, a Python implementation utilizing OpenCV is shown in Appendix .

Utilizing the code, an input image is passed in via the command line when invoking the Python script, shown in Figure 1.

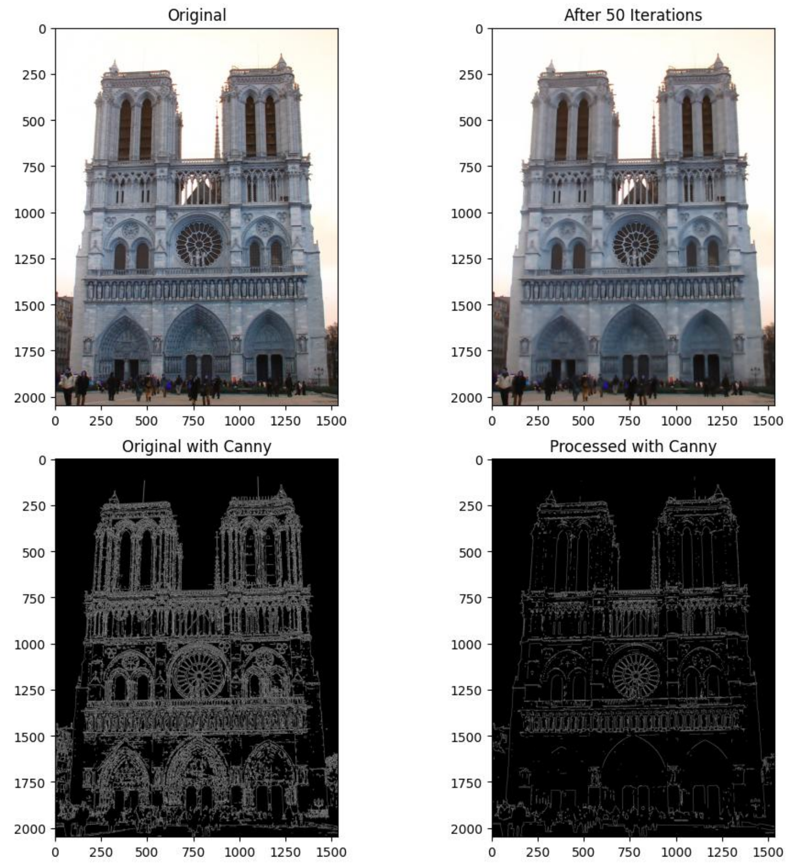

Figure 1: Comparison of original and processed images. Top left: Original image. Top right: Image after 50 iterations of the Perona-Malik finite difference scheme. Bottom left: Canny edge detector applied to the original image. Bottom right: Canny edge detector applied to the processed image.

Conclusion

Convolution kernels are a fundamental concept that nearly every computer science student encounters during their studies. However, the mathematical foundations often remain an afterthought. I have frequently used these kernels without realizing that they were actually implementations of a finite difference scheme, complete with the same stability constraints as any other finite difference scheme. In addition, utilizing standardized accelerated function calls significantly speed up computation, often an afterthought mathematical communities.

Appendix

All code written in python3.10, using matplot lib:

Reference Python Implementation

```
import sys
import matplotlib.pyplot as plt
import numpy as np
import cv2

# Implementation of the Perona-Malik paper -
# Anisotropic Diffusion!

def apply_canny_edge_detector(image, low_threshold=50,
    high_threshold=150):
    if image.dtype != np.uint8:
        image = np.uint8(image * 255)

    edges = cv2.Canny(image, low_threshold,
       high_threshold)
    return edges

def g(diff, k):
    # This function calculates the gradient modulus
       function, which controls diffusion rate
    return np.exp(-(np.abs(diff) ** 2) / (k ** 2)).
       astype(np.float32)

def aniso_diff_channel(channel, iter, l, k):
    # Convert the input channel to float32 to prevent
       type mismatch during calculations
    tmp = np.float32(channel)

    # Define kernels to find gradients in four
       directions
    n_ker = np.array([[0, 0, 0], [0, -1, 0], [0, 1,
       0]], dtype=np.float32)
    e_ker = np.array([[0, 0, 0], [1, -1, 0], [0, 0,
       0]], dtype=np.float32)
    w_ker = np.array([[0, 0, 0], [0, -1, 1], [0, 0,
       0]], dtype=np.float32)
    s_ker = np.array([[0, 1, 0], [0, -1, 0], [0, 0,
       0]], dtype=np.float32)

    for i in range(iter):
        n_diff = cv2.filter2D(tmp, -1, n_ker,
           borderType=cv2.BORDER_REFLECT)
        e_diff = cv2.filter2D(tmp, -1, e_ker,
           borderType=cv2.BORDER_REFLECT)
        s_diff = cv2.filter2D(tmp, -1, s_ker,
           borderType=cv2.BORDER_REFLECT)
        w_diff = cv2.filter2D(tmp, -1, w_ker,
           borderType=cv2.BORDER_REFLECT)

        c_n = g(n_diff, k)
        c_e = g(e_diff, k)
        c_s = g(s_diff, k)
        c_w = g(w_diff, k)

        # Update tmp with the diffusion calculated
           using the conductance terms
        tmp = tmp + l * (c_n * n_diff + c_e * e_diff +
            c_s * s_diff + c_w * w_diff)

    # Convert the float32 image back to uint8 for
       proper image format
    return np.clip(tmp, 0, 255).astype(np.uint8)

def aniso_diff_color(in_img, iter, l, k):
    if len(in_img.shape) == 2:
        return aniso_diff_channel(in_img, iter, l, k)
    elif len(in_img.shape) == 3:
        channels = cv2.split(in_img)
        processed_channels = [aniso_diff_channel(ch,
           iter, l, k) for ch in channels]
        return cv2.merge(processed_channels)

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print(f"Usage:␣python3␣{sys.argv[0]}␣<
           image_path>")
        sys.exit(1)

    image_path = sys.argv[1]
    image = cv2.imread(image_path, cv2.IMREAD_COLOR)
    print(image.shape)

    # Parameters
    nIterations = 50 # Number of iterations
    LAMBDA = 0.25 # Lambda, needs to be between 0 ->
       1/4
```

```
k = 15 # K, edge threshold parameter

# Perform anisotropic diffusion on color image
processed_image = aniso_diff_color(image,
    nIterations, LAMBDA, k)

# Process the image with anisotropic diffusion
    from cv2
#processed_image_check = cv2.ximgproc.
    anisotropicDiffusion(np.copy(image), LAMBDA, K,
     nIterations)

# Plot the original and processed images
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
axs[0, 0].set_title('Original')
axs[0, 0].imshow(image, cmap='gray')
axs[0, 1].set_title(f'After␣{nIterations}␣
    Iterations')
axs[0, 1].imshow(processed_image, cmap='gray')

# Apply Canny (edge) detection to both images
image_with_edges = apply_canny_edge_detector(np.
    copy(image))
processed_with_edges = apply_canny_edge_detector(
    np.copy(processed_image))

# Plot images with Canny
axs[1, 0].set_title('Original␣with␣Canny')
axs[1, 0].imshow(image_with_edges, cmap='gray')
axs[1, 1].set_title('Processed␣with␣Canny')
axs[1, 1].imshow(processed_with_edges, cmap='gray'
    )

plt.tight_layout()
plt.show()
```

References

[1] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," in *Proceedings of IEEE Computer Society Workshop on Computer Vision*, pp. 16–22, November 1987.