Question 3 - Diffusion filtering and non-local means

November 23, 2018

1 Diffusion filtering and non-local means

We were tasked to explore a set of edge preserving denoising filters.

1.1 The code

We start by importing all libraries required for performing the filtering
1. cv2: OpenCV is a library of programming functions mainly aimed at real-time computer vision.
2. numpy: NumPy adds support for large, multi-dimensional arrays along with a large collection of functions to operate on these arrays.
3. matplotlib: This is the most widely used plotting library available for python.
4. scipy: SciPy is used for scientific computing and technical computing.

```
In [20]: import cv2
         import scipy as sc
         from scipy import ndimage
         import numpy as np
         from matplotlib import pyplot as plt
         plt.rcParams['figure.figsize'] = [15, 15]
```

We will now read the input images using `imread` method. OpenCV's `imread` reads the image in the BGR color-space by default, we converted it into grayscale by adding the parameter `0`.

```
In [11]: image = cv2.imread("/home/sarvesh/Projects/Image Processing/lenna.noise.jpg",
                            0)
         image = image.astype('float')
```

We now define three functions to perform the given task.

1. The function `anisotropic` takes source image, iterations, delta and kappa to perform anisotropic filtering along 8 neighbours.
   - We defined the gradient arrays along the 8 directions possible.
   - For each iteration we convolved the image with the gradients.
   - For the diffusion rate we used the Inverse polynomial based function with factor kappa
   - We can alternatively use exponential based function for diffusion rate
   - We updated the image with sum of product of distance, gradient and diffusion rate along with factor delta

- We can get various levels of filtering with changing iterations, delta and kappa

```
In [12]: def anisotropic(image, n, delta, kappa):
             # Initialise the output container with the given image
             out = image
             out = out.astype('float64')
             dd = np.sqrt(2)

             # Initialise gradient vectors along 8 directions
             hN = np.array([[0, 1, 0], [0, -1, 0], [0, 0, 0]], np.float64)
             hS = np.array([[0, 0, 0], [0, -1, 0], [0, 1, 0]], np.float64)
             hE = np.array([[0, 0, 0], [0, -1, 1], [0, 0, 0]], np.float64)
             hW = np.array([[0, 0, 0], [1, -1, 0], [0, 0, 0]], np.float64)
             hNE = np.array([[0, 0, 1], [0, -1, 0], [0, 0, 0]], np.float64)
             hSE = np.array([[0, 0, 0], [0, -1, 0], [0, 0, 1]], np.float64)
             hSW = np.array([[0, 0, 0], [0, -1, 0], [1, 0, 0]], np.float64)
             hNW = np.array([[1, 0, 0], [0, -1, 0], [0, 0, 0]], np.float64)

             for i in range(n):
                 # Convolution with gradient vectors
                 nN = ndimage.filters.convolve(out, hN)
                 nS = ndimage.filters.convolve(out, hS)
                 nE = ndimage.filters.convolve(out, hE)
                 nW = ndimage.filters.convolve(out, hW)
                 nNE = ndimage.filters.convolve(out, hNE)
                 nSE = ndimage.filters.convolve(out, hSE)
                 nSW = ndimage.filters.convolve(out, hSW)
                 nNW = ndimage.filters.convolve(out, hNW)

                 # Diffusion rates
                 cN = 1. / (1 + (nN / kappa)**2)
                 cS = 1. / (1 + (nS / kappa)**2)
                 cE = 1. / (1 + (nE / kappa)**2)
                 cW = 1. / (1 + (nW / kappa)**2)
                 cNE = 1. / (1 + (nNE / kappa)**2)
                 cSE = 1. / (1 + (nSE / kappa)**2)
                 cSW = 1. / (1 + (nSW / kappa)**2)
                 cNW = 1. / (1 + (nNW / kappa)**2)

                 out = out + delta * (
                     (cN * nN) + (cE * nE) + (cS * nS) +
                     (cW * nW)) + delta * (1 / (dd**2)) * (
                         (cNE * nNE) + (cSE * nSE) + (cSW * nSW) + (cNW * nNW))
             return out
```

2. The function `isotropic` takes in an image, iterations and diffusion rate lambd to perform isotropic diffusion based filtering:

   - Isotropic filtering is performed over windows of size 3x3.

- We update image with smoothing window scaled by lambd and pixel's intensity.
- Our results depend on the iterations and also the rate we have taken.

```
In [13]: def isotropic(lenna, steps, lambd):
             # Defining a laplacian window of 3*3
             window = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])

             # Make a image border of width 1
             img = cv2.copyMakeBorder(lenna, 1, 1, 1, 1, cv2.BORDER_CONSTANT, value=0)

             # Initialise the final image
             final = img

             # Iterations given by variable steps
             for k in range(steps):

                 # Run through each pixel of image
                 for i in range(lenna[0].size):
                     for j in range(lenna[0].size):
                         final[i:i + 3, j:j +
                               3] = final[i:i + 3, j:j +
                                          3] + lambd * final[i + 1, j + 1] * window
             return final
```

3. The function `gaussian` generates a 2D gaussian kernel of given length and variance

```
In [27]: def gaussian(l, sig):
             # Generate array
             ax = np.arange(-l // 2 + 1., l // 2 + 1.)
             # Generate 2D matrices by duplicating ax along two axes
             xx, yy = np.meshgrid(ax, ax)
             # kernel will be the gaussian over the 2D arrays
             kernel = np.exp(-(xx**2 + yy**2) / (2. * sig**2))
             # Normalise the kernel
             final = kernel / kernel.sum()
             return final
```

4. The function `means_filter` performs non-local means denoising on the given image and with given kernel.

   - We call the function `gaussian` to return the 7x7 gaussian kernel of variance =1
   - We denoise at every pixel of the image in patches of 7x7 around window of 5x5

```
In [72]: def means_filter(image):
             [m, n] = image.shape

             # Padding the image
             img = cv2.copyMakeBorder(image, 5, 5, 5, 5, cv2.BORDER_CONSTANT, value=255)
```

```
    # Empty output image
    out = np.zeros((m, n), dtype='float')

    # generate gaussian kernel matrix of 7*7
    kernel = gaussian(7, 1)
    h = 25
    h = h * h

    # Run the non-local means for each pixel
    for i in range(6, 512):
        for j in range(6,512):
            w1 = img[i:i + 7, j:j + 7]

            wmax = 0
            avg = 0
            sweight = 0
            rmin = i - 2
            rmax = i + 2
            cmin = j - 2
            cmax = j + 2

            # Apply Gaussian weighted square distance between
            # patches of 7*7 in a window of 5*5
            for r in range(rmin, rmax):
                for c in range(cmin, cmax):
                    w2 = img[r - 3:r + 4, c - 3:c + 4]
                    bl = w1 - w2
                    temp = np.multiply(bl, bl)
                    d = sum(sum(np.multiply(kernel, temp)))

                    w = np.exp(d / h)
                    if w > wmax:
                        wmax = w
                    sweight = sweight + w
                    avg = avg + w * img[r, c]

            avg = avg + wmax * image[i, j]
            sweight = sweight + wmax
            if sweight > 0:
                out[i-5 , j-5 ] = avg / sweight
            else:
                out[i-5 , j-5 ] = image[i, j]

    return out
```

We now perform the task. Multiple observations were made and the parameter values which gave the best results were chosen.

```
In [75]: # Call means_filter for the input image
```

```
means = means_filter(image)

# Call isotropic function with parameters:
# Iterations =10, Lambda = 0.2
iso = isotropic(image, 10, 0.2)

# Call the Anisotropic function with the parameters:
# Iterations = 10, Delta = 0.14, Kappa = 15
aniso = anisotropic(image, 10, 0.14, 15)
```

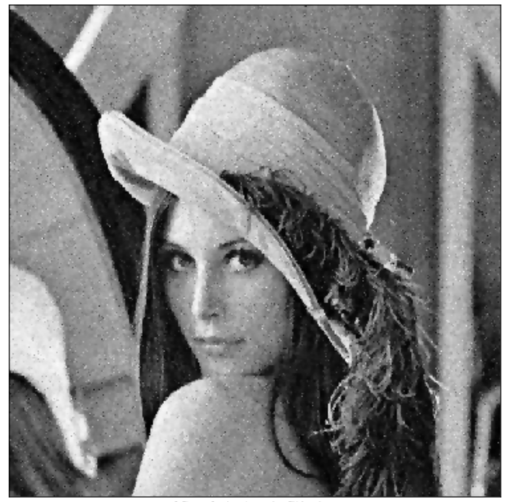
1.2 Observations and Results

The image was subjected to filtering by three different approaches.
1. Isotropic filtering denosied the image but has lead to loss of details due to extensive blurring.
2. Anisotropic filtering denoised the image and has also retained the details to a good extent.
3. Non local means filter has denoised the image and retained most of the details.

```
In [77]: plt.subplot(2, 2, 1), plt.imshow(image, cmap='gray')
         plt.xticks([]), plt.yticks([])
         plt.xlabel('Original')
         plt.subplot(2, 2, 2), plt.imshow(aniso, cmap='gray')
         plt.xticks([]), plt.yticks([])
         plt.xlabel('After Anisotropic fitlering')
         plt.subplot(2, 2, 3), plt.imshow(iso, cmap='gray')
         plt.xticks([]), plt.yticks([])
         plt.xlabel('After Isotropic filtering')
         plt.subplot(2, 2, 4), plt.imshow(means, cmap='gray')
         plt.xticks([]), plt.yticks([])
         plt.xlabel('After Non-local means denoising')

         plt.show()
```

Original



After Anisotropic fitlering

After Isotropic filtering

After Non-local means denoising

1.3 Conclusion

Diffusion filtering and non-local means was performed. The results were as expected and satisfactory.