Image and Video Processing Lab
Lab 5: Bilateral Filter and Non Local Means filter

Aruna Shaju Kollannur (SC21M111)

Sub: Image and Video Processing Lab
Date of Lab sheet: October 26, 2021
Department: Avionics(DSP)
Date of Submission: November 15, 2021

Question 1: Modify Gaussian filter to include an additional weight based on the color difference between pixels of image

$G(p) = 1/w^* \sum_{q \in S} \mathcal{N}(|p-q|)(|I_p - I_q|)I_q$

Aim

Modifying the Gaussian Low pass filter to include the weight based on the linear color difference between pixels of image

Discussion

Gaussian filter, G(p), is determined by the distance between pixel p and q, that is, —p -q—.

$G(p) = 1/w^* \sum_{q \in S} \mathcal{N}(|p-q|)I_q$

To preserve detail around edges, an additional term can be incorporated that accounts for the color variance around p as well.One controllable way to modulate filtering in edge regions is to weight the contribution of each Iq to G(p) by the color difference between p and q.

$G(p) = 1/w^* \sum_{q \in S} \mathcal{N}(|p-q|)(|I_p - I_q|)I_q$

Algorithm

```
Step 1: Start
Step 2: Read the image
Step 3: Obtain the height and width of the image.
Step 4: Calculate size of filter according to sigma parameter chosen.
Step 5: Run a loop to traverse through each pixel in image
       Step 6: Initialize weight variables (w) and gaussian functions coefficients (g) as zero.
               Step 7: Run a nested loop to create window around pixel (centre)
               Step 8: Obtain coefficients g1 as difference between pixel positions.
               Step 9: Obtain coefficients g2 as difference between pixel intensities.
               Step 9: Multiply g1 and g2 to form g.
               Step 10: Multiply g with current pixel under consideration. Sum individual g
               coefficients
               over loop.
               Step 11: Sum value of g over loop and assign to w
     Step 12: Normalise result of Step 10 with result of Step 11.
  Step 13: Display the filtered image.
```

Program Code

```
# Main File
import numpy as np
from PIL import Image
import filt

img = np.array(Image.open('rubiks_cube.png'), dtype=np.float32)
img_filtered = np.asarray(filt.gaussian(img, 30))
img_filtered = np.clip(img_filtered, 0, 255)
Image.fromarray(img_filtered.astype(np.uint8)).show()
Image.fromarray(img_filtered.astype(np.uint8)).save('Gaussian-Intensity-Linear-Filtered.png')

#Cython file for Gaussian filter with added linear weights according to difference in
pixel intensities

from libc.math cimport exp
from libc.math cimport sqrt

import numpy as np

def gaussian(float[:, :, :] im, double sigma):
    cdef int height = im.shape[0]  # cdef int tells Cython that this variable should be
        converted to a C int

    cdef int width = im.shape[1]   #

    # cdef double[:, :, :] to store this as a 3D array of doubles
    cdef double[:, :, :] img_filtered = np.zeros([height, width, 3])

    # A Gaussian has infinite support, but most of it's mass lies within
    # three standard deviations of the mean. The standard deviation is
    # the square of the variance, sigma.
    cdef int n = np.int(sqrt(sigma) * 3)

    cdef int p_y, p_x, i, j, q_y, q_x

    cdef double g1, g2r, g2g, g2b, gpr, gpg, gpb, gr, gg, gb
    cdef double wr = 0
    cdef double wg = 0
    cdef double wb = 0

    # Explicitly assigned the r, g, and b channels
    for p_y in range(height):
        for p_x in range(width):
            gpr = 0
            gpg = 0
            gpb = 0

            wr= 0
            wb=0
            wg=0

            for i in range(-n, n):
                for j in range(-n, n):
```

```
                q_y = max([0, min([height - 1, p_y + i])])
                q_x = max([0, min([width - 1, p_x + j])])

                g1 = exp( -((q_x - p_x)**2 + (q_y - p_y)**2) / (2 * sigma**2) )

                g2r = abs(im[p_y,p_x,0]- im[q_y,q_x,0])
                g2g = abs(im[p_y,p_x,1]- im[q_y,q_x,1])
                g2b = abs(im[p_y,p_x,2]- im[q_y,q_x,2])
                gr = g1 *g2r;
                gg = g1 *g2g;
                gb = g1 *g2b;

                gpr += gr * im[q_y, q_x, 0]
                gpg += gg * im[q_y, q_x, 1]
                gpb += gb * im[q_y, q_x, 2]

                wr += gr
                wg += gg
                wb += gb

        img_filtered[p_y, p_x, 0] = gpr / (wr + 1e-5)
        img_filtered[p_y, p_x, 1] = gpg / (wb + 1e-5)
        img_filtered[p_y, p_x, 2] = gpb / (wg + 1e-5)

return img_filtered
```

Result

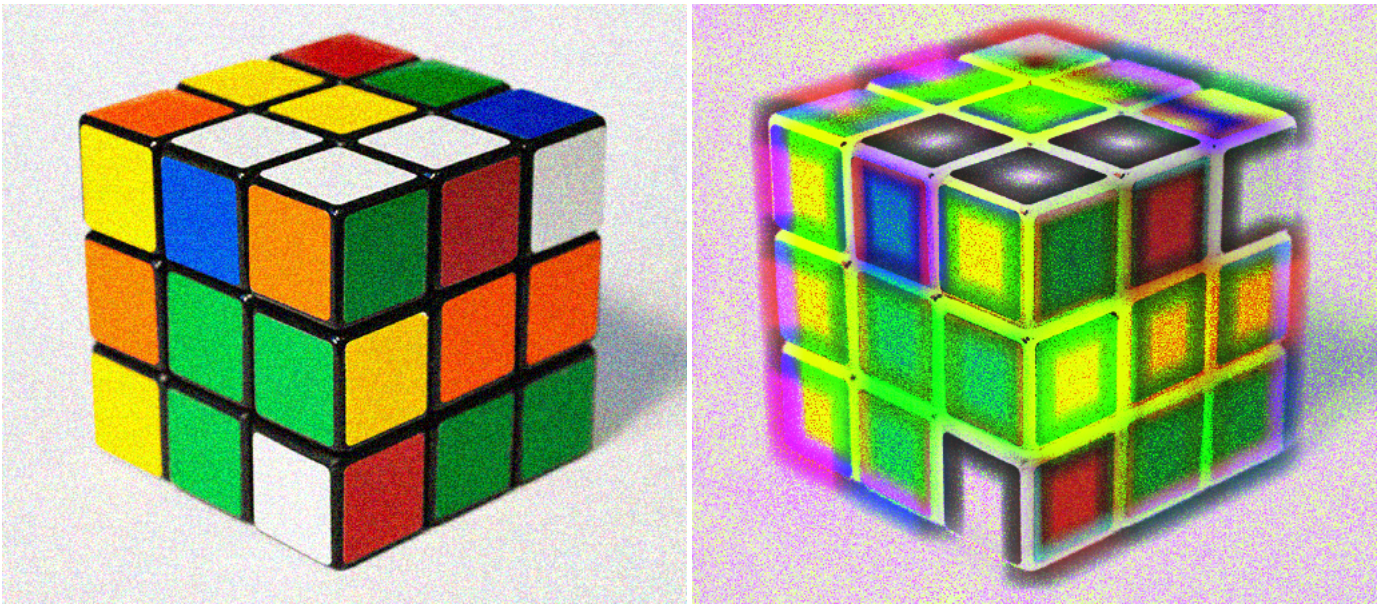

Figure 1: Rubiks Cube Image: (a)Noisy Image(b)Filtered Image

Inference

The weights for normalizing for each R,G,B frame is obtained by separately computing coefficients for each frame.

The Gaussian filter modified with pixel intensity difference gives more importance to pixels with high pixel differences in neighbourhoods.

Question 2: Modify Gaussian filter to include an additional weight which is exponentially decreasing with respect to the difference in color

$$\mathrm{G(p)} = 1/\mathrm{w}^* \sum_{q \in S} \mathcal{N}(|p-q|) exp^{(} - (|I_p - I_q|)) I_q$$

Aim

Modifying the Gaussian Low pass filter to include the weight based on exponentially decreasing function with respect to the difference in intensities between pixels of image

Discussion

To preserve detail around edges, an additional term can be incorporated that accounts for the color variance around p as well.Another controllable way to modulate filtering in edge regions is to weight the contribution of each Iq to G(p) by the decreasing exponential color difference between p and q.

$$\mathrm{G(p)} = 1/\mathrm{w}^* \sum_{q \in S} \mathcal{N}(|p-q|) exp^{(} - (|I_p - I_q|)) I_q$$

Algorithm

```
Step 1: Start
Step 2: Read the image
Step 3: Obtain the height and width of the image.
Step 4: Calculate size of filter according to sigma parameter chosen.
Step 5: Run a loop to traverse through each pixel in image
       Step 6: Initialize weight variables (w) and gaussian functions coefficients (g) as zero.
               Step 7: Run a nested loop to create window around pixel (centre)
               Step 8: Obtain coefficients g1 as difference between pixel positions.
               Step 9: Obtain coefficients g2 as decreasing exponential difference
               between pixel intensities.
               Step 9: Multiply g1 and g2 to form g.
               Step 10: Multiply g with current pixel under consideration. Sum individual g
               coefficients
               over loop.
               Step 11: Sum value of g over loop and assign to w
      Step 12: Normalise result of Step 10 with result of Step 11.
  Step 13: Display the filtered image.
```

Program Code

```
# Main File
import numpy as np
from PIL import Image
import filt

img = np.array(Image.open('rubiks_cube.png'), dtype=np.float32)
img_filtered = np.asarray(filt.gaussian(img, 30))
img_filtered = np.clip(img_filtered, 0, 255)
```

```
Image.fromarray(img_filtered.astype(np.uint8)).show()
Image.fromarray(img_filtered.astype(np.uint8)).save('Exponential-Intensity-Linear-Filtered.png')

from libc.math cimport exp
from libc.math cimport sqrt

import numpy as np

def gaussian(float[:, :, :] im, double sigma):
    cdef int height = im.shape[0]  # cdef int tells Cython that
    this variable should be converted to a C int
    cdef int width = im.shape[1]   #

    # cdef double[:, :, :] to store this as a 3D array of doubles
    cdef double[:, :, :] img_filtered = np.zeros([height, width, 3])

    # A Gaussian has infinite support, but most of it's mass lies within
    # three standard deviations of the mean. The standard deviation is
    # the square of the variance, sigma.
    cdef int n = np.int(sqrt(sigma) * 3)

    cdef int p_y, p_x, i, j, q_y, q_x

    cdef double g1, g2r, g2g, g2b, gpr, gpg, gpb, gr, gg, gb
    cdef double wr = 0
    cdef double wg = 0
    cdef double wb = 0

    # The rest of the code is similar, only now we have to
    explicitly assign the r, g, and b channels
    for p_y in range(height):
        for p_x in range(width):
            gpr = 0
            gpg = 0
            gpb = 0

            wr= 0
            wb=0
            wg=0

            for i in range(-n, n):
                for j in range(-n, n):
                    q_y = max([0, min([height - 1, p_y + i])])
                    q_x = max([0, min([width - 1, p_x + j])])

                    g1 = exp( -((q_x - p_x)**2 + (q_y - p_y)**2) / (2 * sigma**2) )

                    g2r = np.exp(-(abs(im[p_y,p_x,0]- im[q_y,q_x,0])))
                    g2g = np.exp(-(abs(im[p_y,p_x,1]- im[q_y,q_x,1])))
                    g2b = np.exp(-(abs(im[p_y,p_x,2]- im[q_y,q_x,2])))


                    gr = g1 *g2r;
                    gg = g1 *g2g;
                    gb = g1 *g2b;


                    gpr += gr * im[q_y, q_x, 0]
```

```
                    gpg += gg * im[q_y, q_x, 1]
                    gpb += gb * im[q_y, q_x, 2]

                    wr += gr
                    wg += gg
                    wb += gb

            img_filtered[p_y, p_x, 0] = gpr / (wr + 1e-5)
            img_filtered[p_y, p_x, 1] = gpg / (wb + 1e-5)
            img_filtered[p_y, p_x, 2] = gpb / (wg + 1e-5)

    return img_filtered
```

Result

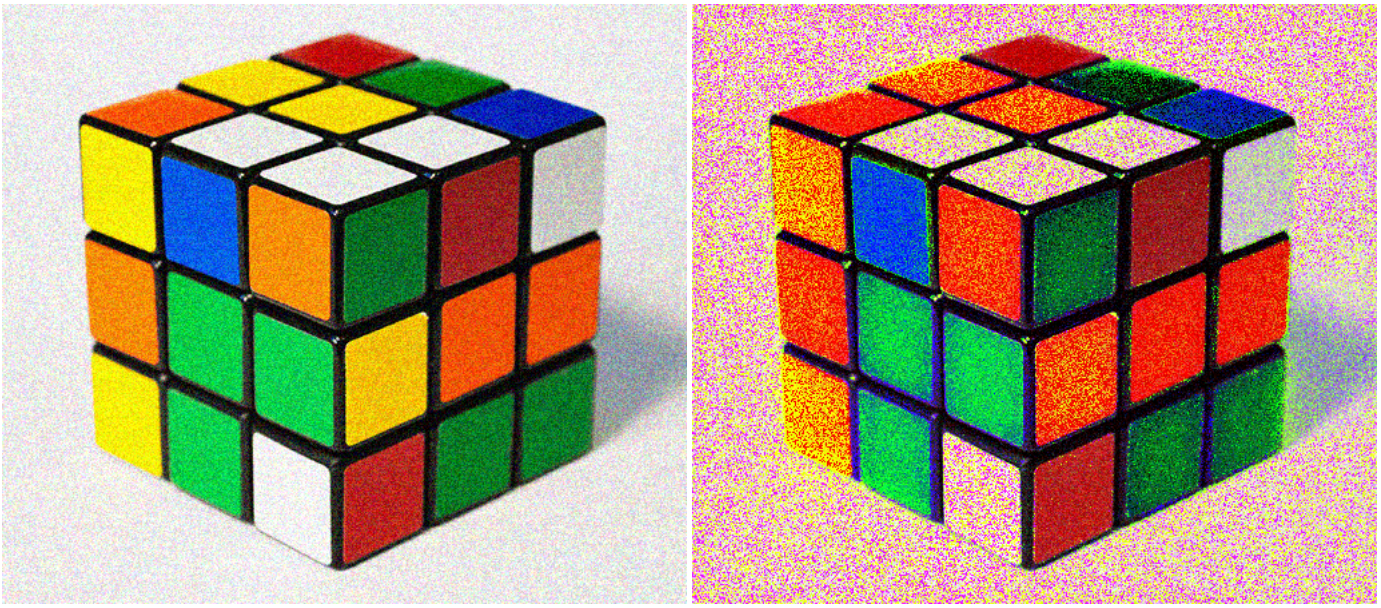

Figure 2: Rubiks Cube Image: (a)Noisy Image(b)Filtered Image

Inference

The Gaussian filter modified with exponential decreasing pixel intensity difference allocates more weight to pixels having lesser neighborhood pixel intensity difference and lesser weights to pixels having higher neighborhood pixel intensity difference, thus preserving the high frequency components of image (edges).

Question 3: Implement Bilateral filter

Aim

Implementing Bilateral Filter

Discussion

To preserve detail around edges, an additional term can be incorporated that accounts for the color variance around p as well.Another controllable way to modulate filtering in edge regions is to weight the contribution of each Iq to G(p) by the gaussian color difference between p and q. This is called Bilateral Filtering.

$$G(p) = 1/w^* \sum_{q \in S} \mathcal{N}(|p-q|)\mathcal{N}(-(|I_p - I_q|))I_q$$

Algorithm

```
Step 1: Start
Step 2: Read the image
Step 3: Obtain the height and width of the image.
Step 4: Calculate size of filter according to sigma parameter chosen.
Step 5: Run a loop to traverse through each pixel in image
       Step 6: Initialize weight variables (w) and gaussian functions coefficients (g) as zero.
               Step 7: Run a nested loop to create window around pixel (centre)
               Step 8: Obtain coefficients g1 as difference between pixel positions.
               Step 9: Obtain coefficients g2 as decreasing exponential difference
               between pixel intensities.
               Step 9: Multiply g1 and g2 to form g.
               Step 10: Multiply g with current pixel under consideration. Sum individual g
               coefficients
               over loop.
               Step 11: Sum value of g over loop and assign to w
      Step 12: Normalise result of Step 10 with result of Step 11.
  Step 13: Display the filtered image.
```

Program Code

```
# Main File
import numpy as np
from PIL import Image
import filt

img = np.array(Image.open('rubiks_cube.png'), dtype=np.float32)
img_filtered = np.asarray(filt.gaussian(img,30))
img_filtered = np.clip(img_filtered, 0, 255)
Image.fromarray(img_filtered.astype(np.uint8)).show()
Image.fromarray(img_filtered.astype(np.uint8)).save('Bilateral-Filtered-Filtered.png')

from libc.math cimport exp
from libc.math cimport sqrt

import numpy as np

def gaussian(float[:, :, :] im, double sigma):
    cdef int height = im.shape[0]  # cdef int tells C
    ython that this variable should be converted to a C int
    cdef int width = im.shape[1]   #

    # cdef double[:, :, :] to store this as a 3D array of doubles
    cdef double[:, :, :] img_filtered = np.zeros([height, width, 3])

    # A Gaussian has infinite support, but most of it's mass lies within
```

```
# three standard deviations of the mean. The standard deviation is
# the square of the variance, sigma.
cdef int n = np.int(sqrt(sigma) * 3)

cdef int p_y, p_x, i, j, q_y, q_x

cdef double g1, g2r, g2g, g2b, gpr, gpg, gpb, gr, gg, gb
cdef double wr = 0
cdef double wg = 0
cdef double wb = 0

# The rest of the code is similar, only now we have to explicitly
assign the r, g, and b channels
for p_y in range(height):
    for p_x in range(width):
        gpr = 0
        gpg = 0
        gpb = 0

        wr= 0
        wb=0
        wg=0

        for i in range(-n, n):
            for j in range(-n, n):
                q_y = max([0, min([height - 1, p_y + i])])
                q_x = max([0, min([width - 1, p_x + j])])

                g1 = exp( -((q_x - p_x)**2 + (q_y - p_y)**2) / (2 * sigma**2) )

                g2r = exp(-((im[p_y,p_x,0]- im[q_y,q_x,0])**2)/(2 * sigma**2))
                g2g = exp(-((im[p_y,p_x,1]- im[q_y,q_x,1])**2)/(2 * sigma**2))
                g2b = exp(-((im[p_y,p_x,2]- im[q_y,q_x,2])**2)/(2 * sigma**2))

                gr = g1 *g2r;
                gg = g1 *g2g;
                gb = g1 *g2b;

                gpr += gr * im[q_y, q_x, 0]
                gpg += gg * im[q_y, q_x, 1]
                gpb += gb * im[q_y, q_x, 2]

                wr += gr
                wg += gg
                wb += gb

        img_filtered[p_y, p_x, 0] = gpr / (wr + 1e-5)
        img_filtered[p_y, p_x, 1] = gpg / (wb + 1e-5)
        img_filtered[p_y, p_x, 2] = gpb / (wg + 1e-5)

return img_filtered
```

Result

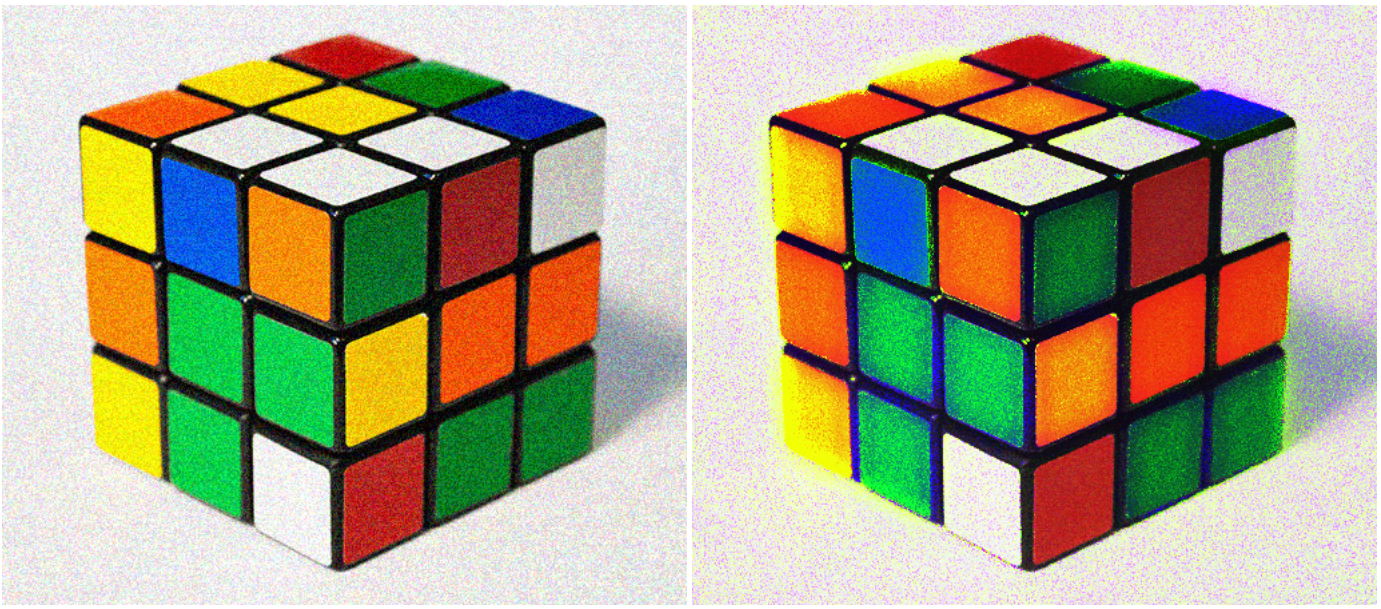

Figure 3: Rubiks Cube Image: (a)Noisy Image(b)Filtered Image

Inference

The Gaussian filter as a function of pixel position difference is modified with gaussian function of pixel intensity difference to allocate more weight to pixels having lesser neighborhood pixel intensity difference and lesser weights to pixels having higher neighborhood pixel intensity difference.
This helps preserving the high frequency components of image (edges).
This works better than gaussian fiter modified with exponential function of pixel differences.

Question 4: Bilateral Filtering with different Spatial and Range Sigma values

Aim

Implementing Bilateral Filtering using different Spatial and Range Sigma values

Discussion

Bilateral Filtering uses two parameters for dynamic basis formation. They are the spatial sigma (influencing contribution of pixel neighborhood with proximity from centre pixel) and range sigma (influencing contribution of pixel neighborhood with similarity in pixel intensity).

$$\mathrm{G(p)} = 1/\mathrm{w}^* \sum_{q \in S} \mathcal{N}(|p-q|)\mathcal{N}(-(|I_p - I_q|))I_q$$
$$\mathrm{w} = \sum_{q \in S} \mathcal{N}(|p-q|)\mathcal{N}(|I_p - I_q|)$$

Algorithm

Step 1: Start

```
Step 2: Read the image
Step 3: Obtain the height and width of the image.
Step 4: Calculate size of filter according to range and spatial sigma.
Step 5: Run a loop to traverse through each pixel in image
       Step 6: Initialize weight variables (w) and gaussian functions coefficients (g) as zero.
               Step 7: Run a nested loop to create window around pixel (centre)
               Step 8: Obtain coefficients g1 as difference between pixel positions.
               Step 9: Obtain coefficients g2 as decreasing exponential difference
               between pixel intensities.
               Step 9: Multiply g1 and g2 to form g.
               Step 10: Multiply g with current pixel under consideration. Sum individual g
               coefficients
               over loop.
               Step 11: Sum value of g over loop and assign to w
      Step 12: Normalise result of Step 10 with result of Step 11.
  Step 13: Repeat the same for different spatial and range sigmas.
  Step 14:Display the filtered images.
```

Program Code

```
#Main File

import numpy as np
from PIL import Image
import filt

img = np.array(Image.open('Lenna.png'), dtype=np.float32)

range_sigma = [0.05,1,2,4,8]
spatial_sigma = [2,4,8,16,32]

for i in range_sigma:
    for j in spatial_sigma:

     img_filtered= np.asarray(filt.gaussian(img,i,j))
     img_filtered = np.clip(img_filtered, 0, 255)
     Image_filtered = Image.fromarray(img_filtered.astype(np.uint8))
     #Image_filtered.show()

     outfile = '%s%d%s%f.png' % ("Bilateral_Filter_Spatial_sigma_",j,"_Range_sigma_",i)
     Image_filtered.save(outfile)

rom libc.math cimport exp
from libc.math cimport sqrt

import numpy as np

def gaussian(float[:, :, :] im, double range_sigma, double spatial_sigma):
    cdef int height = im.shape[0]  # cdef int tells Cython that
    this variable should be converted to a C int
    cdef int width = im.shape[1]   #

    # cdef double[:, :, :] to store this as a 3D array of doubles
    cdef double[:, :, :] img_filtered = np.zeros([height, width, 3])

    # A Gaussian has infinite support, but most of it's mass lies within
    # three standard deviations of the mean. The standard deviation is
```

```
# the square of the variance, sigma.
cdef int n = int(sqrt(spatial_sigma) * 3)

cdef int p_y, p_x, i, j, q_y, q_x

cdef double g1, g2r, g2g, g2b, gpr, gpg, gpb, gr, gg, gb
cdef double wr = 0
cdef double wg = 0
cdef double wb = 0


# The rest of the code is similar, only now we have to
explicitly assign the r, g, and b channels
for p_y in range(height):
    for p_x in range(width):
        gpr = 0
        gpg = 0
        gpb = 0

        wr= 0
        wb=0
        wg=0

        for i in range(-n, n):
            for j in range(-n, n):
                q_y = max([0, min([height - 1, p_y + i])])
                q_x = max([0, min([width - 1, p_x + j])])

                g1 = exp( -((q_x - p_x)**2 + (q_y - p_y)**2) / (2 * spatial_sigma**2) )

                g2r = exp(-((im[p_y,p_x,0]- im[q_y,q_x,0])**2)/(2 * range_sigma**2))
                g2g = exp(-((im[p_y,p_x,1]- im[q_y,q_x,1])**2)/(2 * range_sigma**2))
                g2b = exp(-((im[p_y,p_x,2]- im[q_y,q_x,2])**2)/(2 * range_sigma**2))


                gr = g1 *g2r;
                gg = g1 *g2g;
                gb = g1 *g2b;


                gpr += gr * im[q_y, q_x, 0]
                gpg += gg * im[q_y, q_x, 1]
                gpb += gb * im[q_y, q_x, 2]

                wr += gr
                wg += gg
                wb += gb


        img_filtered[p_y, p_x, 0] = gpr / (wr + 1e-5)
        img_filtered[p_y, p_x, 1] = gpg / (wb + 1e-5)
        img_filtered[p_y, p_x, 2] = gpb / (wg + 1e-5)

return img_filtered
```

Result

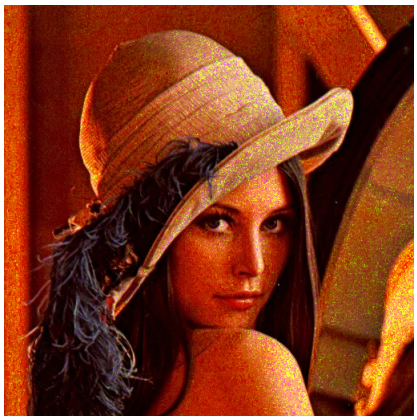

Figure 4: Noisy Image

Figure 5: Spatial Sigma = 2 (a) Range sigma = 0.05 (b) Range sigma = 1 (c) Range sigma = 2 (d) Range sigma = 4 (e) Range sigma = 8

Figure 6: Spatial Sigma = 4 (a) Range sigma = 0.05 (b) Range sigma = 1 (c) Range sigma = 2 (d) Range sigma = 4 (e) Range sigma = 8

Figure 7: Spatial Sigma = 8 (a) Range sigma = 0.05 (b) Range sigma = 1 (c) Range sigma = 2 (d) Range sigma = 4 (e) Range sigma = 8

Figure 8: Spatial Sigma = 16 (a) Range sigma = 0.05 (b) Range sigma = 1 (c) Range sigma = 2 (d) Range sigma = 4 (e) Range sigma = 8

Figure 9: Spatial Sigma = 32 (a) Range sigma = 0.05 (b) Range sigma = 1 (c) Range sigma = 2 (d) Range sigma = 4 (e) Range sigma = 8

Question 5: Implement Non-Local Means Filter

Aim

Implementing Non-Local Means Filter

Discussion

Non-local means filtering applies, to each pixel location, an adaptive averaging kernel that is computed from patch distances. The pixel neighborhoods in an image similar to that of the selected patch is averaged with higher weights as compared to dissimilar neighborhoods.

Algorithm

```
Step 1: Start
Step 2: Read the image
Step 3: Obtain the height and width of the image.
Step 4: Extract patch with reference to which Non-Local Means filtering is to be done.
Step 5: Run a loop to traverse through each pixel in image
       Step 6: Initialize weight variables (w) and gaussian functions coefficients (g) as zero.
               Step 7: Run a nested loop to create window around pixel (centre)
               Step 8: Obtain coefficients g as patch pixel intensities and current pixel
               neighborhood intensities.
               Step 9: Multiply g with current pixel under consideration. Sum individual g
               coefficients
               over loop.
               Step 10: Sum value of g over loop and assign to w
      Step 11: Normalise result of Step 9 with result of Step 10.
     Step 12:Display the filtered image.
```

Program Code

```
# Main File
import numpy as np
from PIL import Image
import filt

img = np.array(Image.open('Lenna_gray.png'), dtype=np.float32)
height, width = img.shape

sigma = 1
n = int(np.sqrt(sigma) * 3)

#patch_compare = img[int(height/2):(int(height/2) + n), int(width/2):(int(width/2) + n)]

# Taking patch from top-left most corner
patch_compare = img[0:(n), 0:(n)]

img_filtered = np.asarray(filt.non_local_mean(img,patch_compare,sigma))
img_filtered = np.clip(img_filtered, 0, 255)
Image.fromarray(img_filtered.astype(np.uint8)).show()
Image.fromarray(img_filtered.astype(np.uint8)).save('Non_Local_Mean.png')
Image.fromarray(patch_compare.astype(np.uint8)).save('patch.png')

from libc.math cimport exp
```

```
from libc.math cimport sqrt

import numpy as np

def non_local_mean(float[:, :] im, float[:, :] patch, double sigma):
    cdef int height = im.shape[0]  # cdef int tells Cython that
    this variable should be converted to a C int
    cdef int width = im.shape[1]   #

    # cdef double[:, :, :] to store this as a 3D array of doubles
    cdef double[:, :] img_filtered = np.zeros([height, width])

    # A Gaussian has infinite support, but most of it's mass lies within
    # three standard deviations of the mean. The standard deviation is
    # the square of the variance, sigma.
    cdef int n = np.int(sqrt(sigma) * 3)

    cdef int p_y, p_x, i, j, q_y, q_x

    cdef double g1, g2r, g2g, g2b, gpr, gpg, gpb, gr, gg, gb
    cdef double w = 0


    # The rest of the code is similar, only now we have to explicitly assign the r, g, and b
    channels

    for p_y in range(int(n/2),height-int(n/2)):
          for p_x in range(int(n/2),width-int(n/2)):
              gpr = 0
              wr = 0

              # Iterate over kernel locations to define pixel q
              for i in range(-int(n/2),int(n/2)+1):
                  for j in range(-int(n/2),int(n/2)+1):

                      g2r = np.exp(-((im[(p_y + i),(p_x + j)]- patch[i,j])**2)/(2 * sigma**2))
                      # Accumulate filtered output
                      gpr += g2r * im[(p_y + i),(p_x + j)]
                      # Accumulate filter weight for later normalization, to maintain image
                      brightness
                      w += g2r

              img_filtered[p_y, p_x] = gpr / (w + 1e-5)

    return img_filtered
```

Result

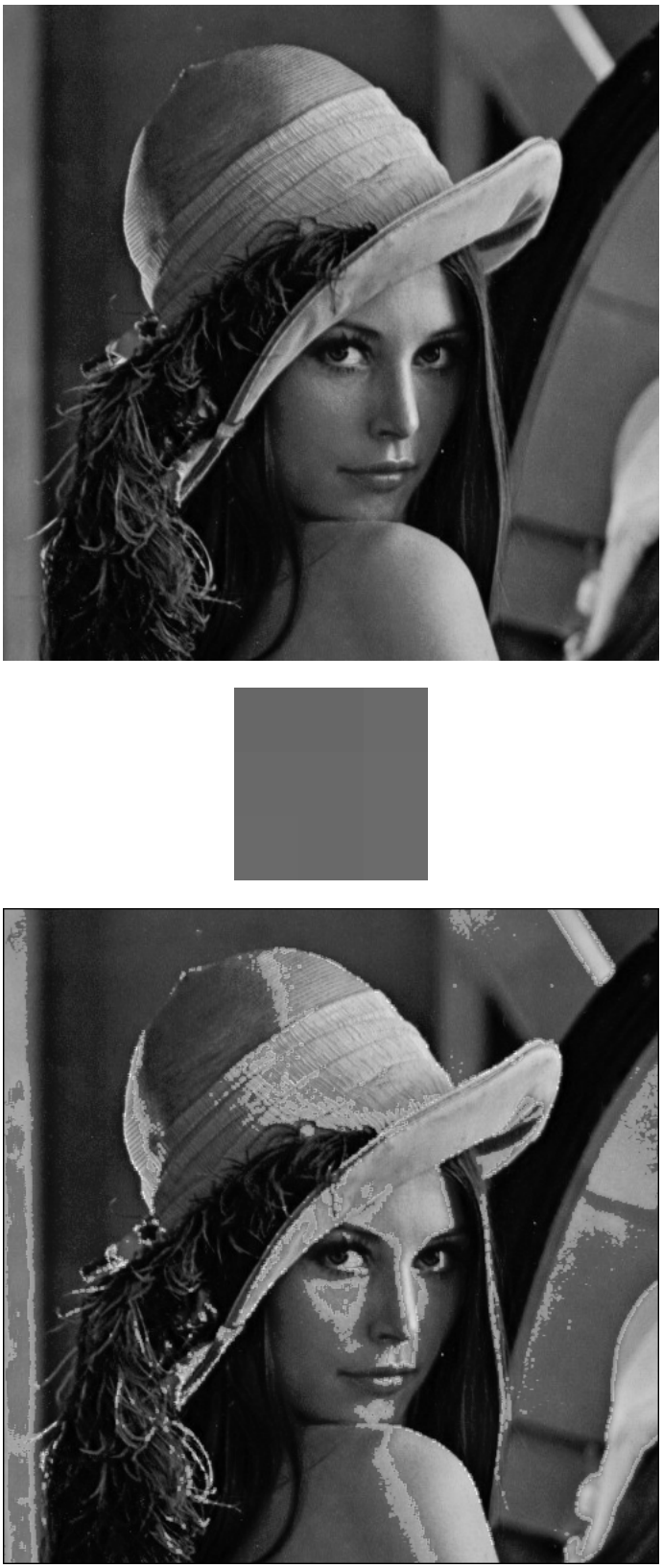

Figure 10: Cameraman (a) Original Image (b) Reference patch (c) Non-Local Means Filtered Image

Inference

Non-Local Means blurs pixels with similar neighborhood as that of reference patch more aggressively, thus not altering much the non-similar neighborhoods. As seen in result, neighborhood matching the patch pixel intensities are more blurred in comparison to neighborhoods with dissimilar pixel intensities.