

EN3160 Fundamentals of Image Processing and Machine Vision: Point Operations

Ranga Rodrigo
ranga@uom.lk

The University of Moratuwa, Sri Lanka

August 1, 2023



Contents

Basics

Intensity Transformations

Intensity Transformations: Histograms

Intensity Transformations: Histogram Equalization

Intensity Transformations: Gamma Correction

Section 1

Basics

What is a Digital Image?

- A grayscale digital image is a rectangular array of numbers which represent pixels.
- Each pixel can take an integer value in the range $[0, 255]$, for an 8-bit image.
- If the image is a color image, then there would be three such arrays.
- The size of this array is actually the resolution of the image, e.g., example, 3712×5568^a .
- We take the top-left pixel as the $(0, 0)$ pixel and vertical axis as the i axis.

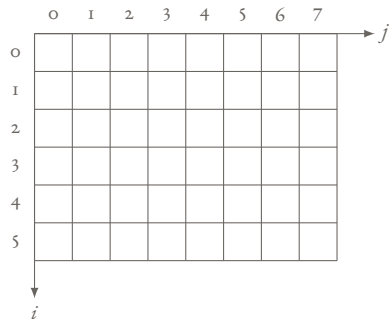


Figure: A 6×8 image

^a Nikon D5

What is a Digital Image?

- A grayscale digital image is a rectangular array of numbers which represent pixels.
- Each pixel can take an integer value in the range $[0, 255]$, for an 8-bit image.
- If the image is a color image, then there would be three such arrays.
- The size of this array is actually the resolution of the image, e.g., example, 3712×5568^a .
- We take the top-left pixel as the $(0, 0)$ pixel and vertical axis as the i axis.

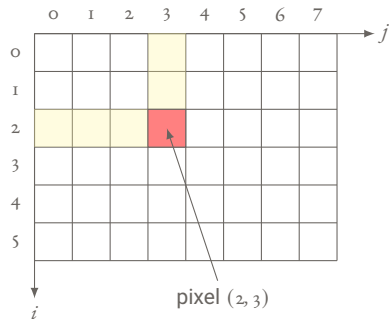


Figure: A 6×8 image

^a Nikon D5

What is a Color Digital Image?

- The grayscale image that we considered is a two-dimensional array, or a single plane.
- A color image has three such planes, one for blue, one for green and one for red. We call such an image an BGR image.
- If we access a pixel location such as $(2, 3)$, we will get three values, B, G, and R.
- Each value is in $[0, 255]$. In this context, $2^8 \times 2^8 \times 2^8$ different colors are possible.

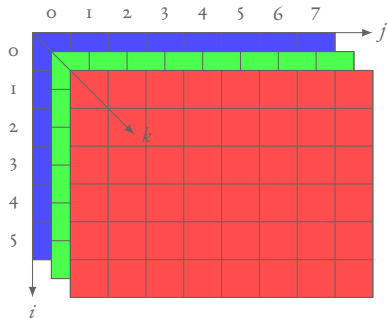


Figure: A 6×8 BGR image

Code for Generating a 6×8 Images

Listing: A Grayscale Image

```
1 %matplotlib inline
2 import cv2 as cv
3 import numpy as np
4 import matplotlib.pyplot as plt
5 im = np.zeros((6,8),dtype=np.uint8)
6 im[2,3] = 255
7 fig, ax = plt.subplots()
8 ax.imshow(im, cmap='gray', vmin=0,
           vmax=255)
9 ax.xaxis.tick_top()
10 plt.show()
```

Listing: A Color Image

```
1 %matplotlib inline
2 import cv2 as cv
3 import numpy as np
4 import matplotlib.pyplot as plt
5 im = np.zeros((6,8,3),dtype=np.uint8)
6 im[2,3] = [255, 255, 100]
7 print(im[2,3])
8 fig, ax = plt.subplots()
9 ax.imshow(im)
10 ax.xaxis.tick_top()
11 plt.show()
```

Opening and Displaying Images

1. We can use OpenCV to open and display images. We can examine the size (resolution) of the image as well.

Listing: Displaying Using Matplotlib

```
1 %matplotlib inline
2 import cv2 as cv
3 import matplotlib.pyplot as plt
4 img = cv.imread( '../images/gal.jpg',
                  cv.IMREAD_COLOR)
5 img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
6 fig, ax = plt.subplots()
7 ax.imshow(img)
8 ax.set_title('Image')
9 plt.show()
```

Listing: Displaying Using OpenCV

```
1 import cv2 as cv
2 import matplotlib.pyplot as plt
3 img = cv.imread( '../images/katniss.
                  jpg', cv.IMREAD_COLOR)
4 cv.namedWindow('Image', cv.WINDOW_NORMAL)
5 cv.imshow('Image', img)
6 cv.waitKey(0)
7 cv.destroyAllWindows()
```

Q: Why do we need to cv.cvtColor only when displaying using Matplotlib?

Displaying Image Properties

Listing: Displaying Image Properties

```
1 %matplotlib inline
2 import cv2 as cv
3 import matplotlib.pyplot as plt
4 img = cv.imread(' ../images/hugh.jpg', cv.IMREAD_COLOR)
5 img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
6 fig, ax = plt.subplots()
7 ax.imshow(img)
8 ax.set_title('Image')
9 plt.show()
10 print(img.shape)
11 print(img.size)
12 print(img.dtype)
```

Increasing the Brightness Using OpenCV

Listing: Increasing the Brightness Using OpenCV

```
1 import cv2 as cv
2 import matplotlib.pyplot as plt
3 img = cv.imread('../images/keira.jpg', cv.IMREAD_GRAYSCALE)
4 imgb = img + 100
5 imgc = cv.add(img, 100)
6 f, ax = plt.subplots(1,3)
7 ax[0].imshow(img, cmap="gray")
8 ax[0].set_title('Original')
9 ax[1].imshow(imgb, cmap="gray")
10 ax[1].set_title('img + 100')
11 ax[2].imshow(imgc, cmap="gray")
12 ax[2].set_title('cv.add')
```

- What is the reason for the `img + 100` operation to be incorrect?

In OpenCV and many other image processing libraries, image pixel values are typically represented as unsigned 8-bit integers (uint8) ranging from 0 to 255. Adding 100 directly to the pixel values can cause an overflow when the result exceeds 255, which can lead to unexpected results such as wrapping around and loss of data.

To perform the operation of increasing the pixel values by 100 correctly, you should use the `cv.add()` function, which takes care of handling the pixel value limits properly. Here's the corrected code:

Increasing the Brightness Using Loops I

Listing: Increasing the Brightness Using Loops

```
1 import cv2 as cv
2 import matplotlib.pyplot as plt
3 import numpy as np
4 def image_brighten(image, shift):
5     h = image.shape[0]
6     w = image.shape[1]
7     result = np.zeros(image.shape, image.dtype)
8     for i in range(0,h):
9         for j in range(0,w):
10             no_overflow = True if image[i,j] + shift < 255 else False
11             result[i,j] = min(image[i,j] + shift, 255) if no_overflow else 255
12     return result
13
14 img = cv.imread('../images/keira.jpg', cv.IMREAD_GRAYSCALE)
15 %timeit imgb = image_brighten(img, 200)
16 f, ax = plt.subplots(1,2)
17 ax[0].imshow(img, cmap = 'gray')
18 ax[0].set_title('Original')
```

assert img2 is not None

When working with images or any kind of data loading and processing, it's essential to handle potential errors or unexpected situations. In the code you provided, the assertion `assert img2 is not None` is used to check if the image loaded successfully. Let's break down the importance of this assertion:

Handling Errors: If the image loading process fails for any reason (e.g., file not found, unsupported format, corrupted image), the `cv.imread()` function might return `None`. Without checking this and blindly proceeding with subsequent operations, your code could potentially throw errors or lead to unexpected behavior.

Preventing Errors Downstream: Subsequent operations on `img2` might involve calculations, manipulations, or visualizations. If you perform these operations on a `None` object (i.e., an image that failed to load), it will result in runtime errors. By asserting that `img2` is not `None`, you ensure that you have a valid image to work with.

Clarity and Debugging: If the image loading process fails, the `assert` statement provides a clear and immediate indication that there's a problem. This can be immensely helpful during debugging.

Increasing the Brightness Using Loops II

```
19 ax[1].imshow(imgb, cmap = 'gray')
20 ax[1].set_title('image_brighten')
```

- Do not use this, due to inefficiencies. Instead, use OpenCV filters or Cython (?).
- In the convolution operations—which we would cover later—to produce each output pixel, we need to multiply and add, say, a 3×3 neighborhood of pixels. How many loops would this require? What is the computational complexity?

In convolution operations, such as those commonly used in image processing and deep learning, a small filter (also known as a kernel) is slid over the input image, and for each position of the filter, element-wise multiplication and summation are performed between the filter and the corresponding neighborhood of pixels in the image. This process results in a new pixel value for the output image.

Let's break down the steps involved and discuss the computational complexity:

1. **Filter Loop:** The filter is usually a small matrix (e.g., 3×3) with weights. To compute a single output pixel, you need to perform element-wise multiplication between the filter and the corresponding pixels in the input image's neighborhood (also known as the receptive field). For a 3×3 filter, you would need to perform $3 \times 3 = 9$ multiplications.
2. **Summation Loop:** After performing the multiplications, you need to sum up the results to get the final value of the output pixel. This involves adding the 9 products obtained from the previous step.

So, to produce each output pixel, you need two nested loops: one for the filter and another for the summation.

This results in a total of two loops:

1. Loop over the filter (3×3 in this case) for element-wise multiplication.
2. Loop over the 3×3 products to perform summation.

Zeroing Out Green and Blue Planes

Listing: Zeroing Out Green and Blue Planes

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
3 import cv2 as cv
4 img = cv.imread('images/tom.jpg', cv.IMREAD_ANYCOLOR)
5 if img is None:
6     print('Image could not be read.')
7     assert False
8 img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
9 plt.imshow(img)
10 plt.title('Image')
11 plt.xticks([], plt.yticks([]))
12 plt.show()
13 img[:, :, 1:3] = 0
14 plt.imshow(img)
15 plt.title('After Zeroing Planes')
16 plt.xticks([], plt.yticks([]))
17 plt.show()
```

It appears that this expression is manipulating a 3-dimensional array, possibly an image represented in the RGB color format. In RGB, an image is often represented as a 3D array where the first dimension corresponds to the height of the image, the second dimension corresponds to the width of the image, and the third dimension represents the color channels (Red, Green, and Blue).

Here's a breakdown of the expression:

`img`: This likely represents the 3D array or image you are working with.

`[:, :, 1:3]`: This part is indexing the third dimension of the array (the color channels). The `1:3` slice notation selects elements at indices 1 and 2 (Python uses 0-based indexing), which corresponds to the Green and Blue color channels.

Summary

1. A grayscale digital image is an array of 8-bit unsigned integers in $[0, 255]$. We call each integer a pixel.
2. Color images have three such arrays (planes), one for red, one for green, and one for blue.
3. We can manipulate an image using OpenCV function or a pair of for-loops to access each pixel (slower).

Section 2

Intensity Transformations

Intensity Transformations (Point Operations): Introduction

- In intensity transformations, the output value of the pixel depends only on the input values of that pixel, not its neighbors.
- Input image: $f(x)$
Output image: $g(x)$
Intensity transform $g(x) = T(f(x))$
- E.g., identity transform $T(.) = I$

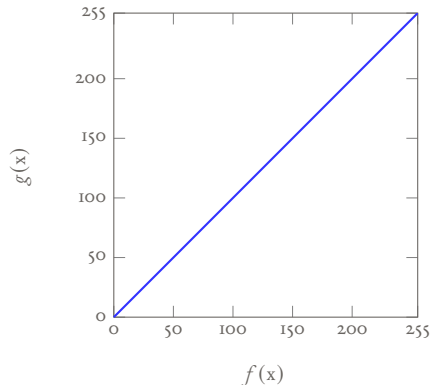


Figure: Identity transform

Intensity Transforms: Identity Transformation Code I

Listing: Identity Transformation

```
1 %matplotlib inline
2 import cv2 as cv
3 import numpy as np
4 import matplotlib.pyplot as plt
5 transform = np.arange(0, 256).astype('uint8')
6 fig, ax = plt.subplots()
7 ax.plot(transform)
8 ax.set_xlabel(r'Input,  $f(\mathbf{x})$ ')'
9 ax.set_ylabel('Output,  $\mathbf{T}[f(\mathbf{x})]$ ')'
10 ax.set_xlim(0, 255)
11 ax.set_ylim(0, 255)
12 ax.set_aspect('equal')
13 plt.savefig('transform.png')
14 plt.show()
15 img_orig = cv.imread('../images/katrina.jpg', cv.IMREAD_GRAYSCALE)
16 print(img_orig.shape)
17 cv.namedWindow('Image', cv.WINDOW_AUTOSIZE)
18 cv.imshow('Image', img_orig)
```

in previous section we add transformations for pixels, but here we are adding a function for all pixel.

transform = np.arange(0, 256).astype('uint8'): Creates an array called transform containing values from 0 to 255, and converts the data type to 'uint8' (unsigned 8-bit integer).

6-14. This section creates a plot using Matplotlib. It plots the transform array (essentially a linear function) and sets labels, limits, and aspect ratio for the plot. It saves the plot as "transform.png" and then displays the plot.

cv.namedWindow('Image', cv.WINDOW_AUTOSIZE): Creates a named window for displaying the image.

Intensity Transforms: Identity Transformation Code II

`cv.waitKey(0)`: Waits for a key press. The argument 0 means it will wait indefinitely until a key is pressed.

```
19 cv.waitKey(0)
20 image_transformed = cv.LUT(img_orig, transform)
21 cv.imshow('Image', image_transformed)
22 cv.waitKey(0)
23 cv.destroyAllWindows()
```

`image_transformed = cv.LUT(img_orig, transform)`: Applies a look-up table (LUT) transformation to the `img_orig` grayscale image using the transform array. This can change the intensity values of the image.

A Look-Up Table (LUT) transformation is a fundamental technique in image processing that involves mapping the intensity values of an image to new values using a predefined mapping function. It's a way to apply a specific transformation to every pixel in an image based on its original intensity.

- Explain the line `image_transformed = cv.LUT(img_orig, transform)`.
- This can be done using NumPy only as `image_transformed = transform[img_orig]`. Explain this operation.
- What is $T()$ here?

When you apply an identity transform to a picture, you are essentially leaving the picture unchanged. An identity transform is a mathematical operation that preserves the original values of the data without any alteration.

In the context of images, a transformation can involve various operations like scaling, rotation, translation, and more. The identity transform, however, means that you are not applying any of these operations; you are keeping the image exactly as it is.

Mathematically, applying an identity transform to an image involves multiplying each pixel's color values by the identity matrix. Since the identity matrix is a square matrix with ones on the main diagonal and zeros elsewhere, this multiplication essentially results in the same color values as the original image.

In simple terms, if you were to apply an identity transform to a picture and then compare the transformed picture with the original one, you wouldn't notice any difference. It's like looking at the same picture in a mirror – the content remains identical, and no changes have occurred.

What Is This Transformation?

- $g(x) = 255 - f(x)$

Applying a negative transform to a picture typically involves some form of inversion or color reversal. This can result in a striking visual effect where the colors are inverted, creating a sort of "photo negative" appearance. In this context, "negative" doesn't mean "bad"; it refers to a specific type of transformation.

Mathematically, applying a negative transform to an image usually involves subtracting each color channel's value from the maximum possible value for that channel. For example, in a typical RGB color space where each channel's value ranges from 0 to 255, the negative transformation of a pixel's color (R, G, B) would be (255 - R, 255 - G, 255 - B).

The result is that darker areas of the image become lighter and vice versa, and the colors become complementary. For instance, in a photo with a blue sky, applying a negative transform would turn the sky orange or reddish, while other objects in the scene would experience similar color shifts.

Keep in mind that while negative transforms can yield interesting and artistic effects, they often make images less realistic. However, they can be used creatively to evoke different emotions or to convey a unique visual style.

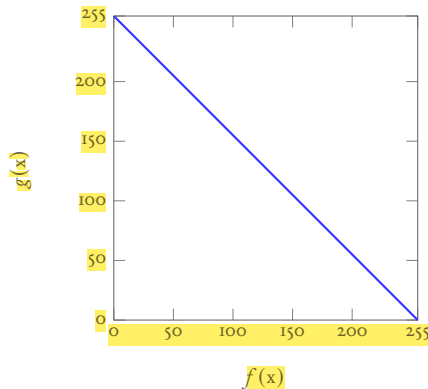


Figure: Negative transform

What Is This Transformation?

- $g(x) = 255 - f(x)$

- **Implementation:**

```
transform = np.arange(255, -1, -1).astype('uint8').
```

1. `image_transformed = transform[image_orig]`

In this expression, `transform` represents the transformation function or lookup table (LUT) that will be applied to the `image_orig` image. The LUT can be an array or a function that defines how pixel values should be mapped from their original values to new values. When you apply the LUT to each pixel value in `image_orig`, you get the transformed image `image_transformed`.

This operation essentially involves substituting each original pixel value with its corresponding value from the transformation array. For instance, if the transformation array specifies that the pixel value 100 should become 150, then this substitution will occur throughout the image.

2. `image_transformed = cv.LUT(image_orig, transform)`

The OpenCV library provides a function called `LUT` (lookup table) which performs the same operation as described above. The `cv.LUT()` function applies a specified LUT to an input image and returns the transformed image. The function takes the original image `image_orig` and the transformation array `transform` as inputs and generates the `image_transformed` output.

Both expressions achieve the same result: applying the specified transformation to the original image, resulting in a transformed image. The second expression explicitly uses the OpenCV function `cv.LUT()` to achieve this transformation, while the first expression seems to use a more general notation to represent the operation conceptually.

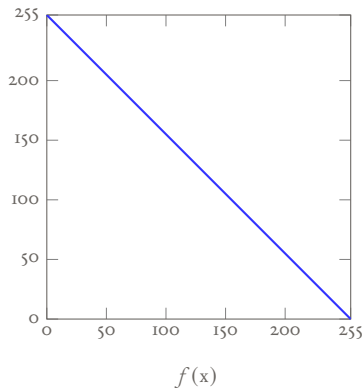


Figure: Negative transform

Intensity Transforms: Intensity Windowing I

Listing: Intensity Windowing

```
1 %matplotlib inline
2 import cv2 as cv
3 import numpy as np
4 import matplotlib.pyplot as plt
5 c = np.array([(100, 50), (150, 200)])
6 t1 = np.linspace(0, c[0,1], c[0,0] + 1 - 0).astype('uint8')
7 print(len(t1))
8 t2 = np.linspace(c[0,1] + 1, c[1,1], c[1,0] - c[0,0]).astype('uint8')
9 print(len(t2))
10 t3 = np.linspace(c[1,1] + 1, 255, 255 - c[1,0]).astype('uint8')
11 print(len(t3))
12 transform = np.concatenate((t1, t2), axis=0).astype('uint8')
13 transform = np.concatenate((transform, t3), axis=0).astype('uint8')
14 print(len(transform))
15 fig, ax = plt.subplots()
16 ax.plot(transform)
17 ax.set_xlabel(r'Input,  $f(\mathbf{x})$ ')
18 ax.set_ylabel('Output,  $\mathbf{T}[f(\mathbf{x})]$ )')
```

`c = np.array([(100, 50), (150, 200)])`: This line creates a NumPy array `c` containing two rows and two columns. Each row represents a pair of values. In this case, the first row is (100, 50) and the second row is (150, 200).

`t1 = np.linspace(0, c[0, 1], c[0, 0] + 1 - 0).astype('uint8')`: This line does the following:

`np.linspace(0, c[0, 1], c[0, 0] + 1 - 0)`: This uses the `linspace` function from NumPy to generate a sequence of evenly spaced values between 0 and the value `c[0, 1]` (which is 50 in this case). The number of values generated is determined by the value in `c[0, 0]` (which is 100 in this case). The result is an array of values.

`.astype('uint8')`: This converts the data type of the generated array to 'uint8', which is an unsigned 8-bit integer data type. This is often used for pixel values in images.

The resulting array `t1` contains 101 evenly spaced values between 0 and 50, all of which are of the data type 'uint8'.

Intensity Transforms: Intensity Windowing II

```
19 ax.set_xlim(0,255)
20 ax.set_ylim(0,255)
21 ax.set_aspect('equal')
22 plt.savefig('transform.png')
23 plt.show()
24 img_orig = cv.imread('../images/katrina.jpg', cv.IMREAD_GRAYSCALE)
25 cv.namedWindow("Image", cv.WINDOW_AUTOSIZE)
26 cv.imshow("Image", img_orig)
27 cv.waitKey(0)
28 image_transformed = cv.LUT(img_orig, transform)
29 cv.imshow("Image", image_transformed)
30 cv.waitKey(0)
31 cv.destroyAllWindows()
```

- What is $T()$ here?

"Evenly spaced" refers to a set of values that are distributed uniformly or regularly across a specified range. In the context of the `np.linspace()` function in NumPy, generating evenly spaced values means creating an array where the values are uniformly distributed between a starting point and an ending point.

For example, if you have a range of values from start to end, and you want to divide that range into a certain number of evenly spaced intervals, the `np.linspace()` function can be used. It calculates the values at each interval point such that they are evenly distributed.

Intensity Transforms: Intensity Windowing III

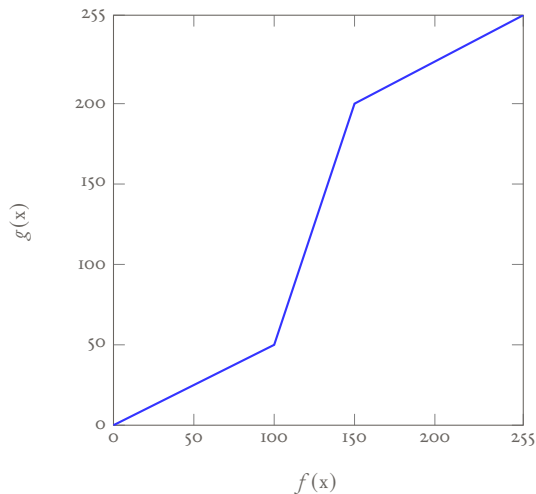


Figure: Intensity windowing.

Intensity Transforms: Intensity Windowing IV

Section 3

Intensity Transformations: Histograms

Histograms

1. We can represent the intensity distribution over the range of intensities $[0, 255]$, using a histogram.
2. If h is the histogram of a particular image, $h(r_k)$ gives us how many pixels have the intensity r_k . The histogram of a digital image with gray values in the range $[0, L - 1]$ is a discrete function $h(r_k) = n_k$ where r_k is the k th gray level and n_k is the number of pixels having gray level r_k .
3. We can normalize the histogram by dividing by the total number of pixels n . Then we have an estimate of the probability of occurrence of level r_k , i.e., $p(r_k) = n_k/n$.
4. The histogram that we described above has L bins. We can construct a coarser histogram by selecting a smaller number of bins than L . Then several adjacent values of k will be counted for a bin.

Activity

1. The figure shows a 3×4 image. The range of intensities that this image has is $[0, 7]$. Compute its histogram.

2^8 intensities are possible

6	5	5	3
7	6	6	4
2	3	5	4

ex-
Intensity Value | Occurrences

Intensity Value	Occurrences
0	1
1	2
2	3
3	2
4	1
5	1
6	1
7	1

Figure: Image for histogram computation.

Histograms Using OpenCV

Listing: Histogram of a Grayscale Image

```
1 import cv2 as cv
2 import numpy as np
3 import matplotlib.pyplot as plt
4 img = cv.imread( '../images/gal.jpg ',
5                 cv.IMREAD_GRAYSCALE)
6 hist = cv.calcHist([img], [0], None, [256], [0,256])
7 plt.plot(hist)
8 plt.xlim([0,256])
9 plt.show()
```

None used to set the mask None, will make the hist for entire image

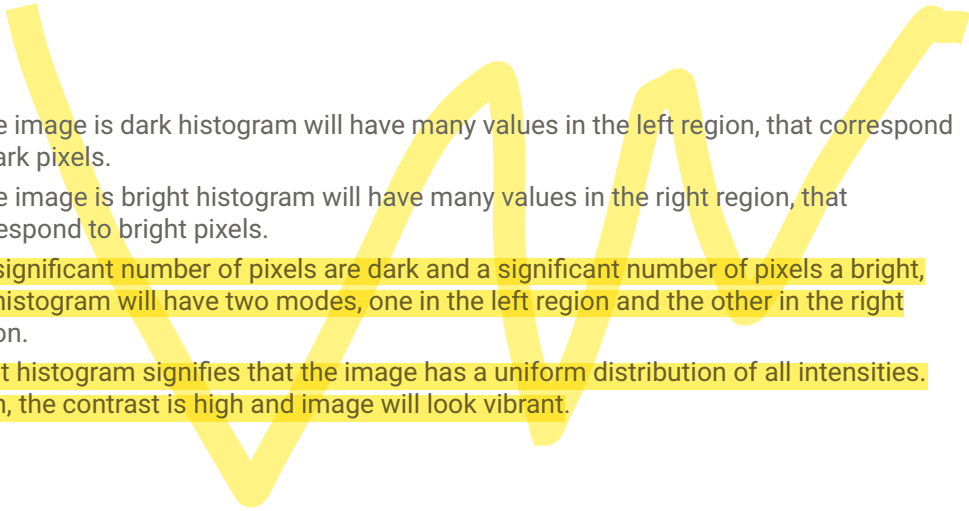
Listing: Histogram of a Color Image

```
1 import cv2 as cv
2 import numpy as np
3 import matplotlib.pyplot as plt
4 img = cv.imread( '../images/gal.jpg ',
5                 cv.IMREAD_COLOR)
6 color = ( 'b', 'g', 'r' )
7 for i, c in enumerate(color):
8     hist = cv.calcHist([img], [i], None, [256], [0,256])
9     plt.plot(hist, color = c)
10    plt.xlim([0,256])
11 plt.show()
```

for i, c in enumerate(color):: This loop iterates through the colors and their corresponding indices (0 for blue, 1 for green, 2 for red) using the enumerate function. The loop assigns the current index to i and the current color to c.

hist = cv.calcHist([img], [i], None, [256], [0, 256]): Inside the loop, the code calculates the histogram for the current color channel using cv.calcHist(). It uses the index i to specify which color channel's histogram is being calculated. The parameters

Image Properties through Histograms

- 
1. If the image is dark histogram will have many values in the left region, that correspond to dark pixels.
 2. If the image is bright histogram will have many values in the right region, that correspond to bright pixels.
 3. If a significant number of pixels are dark and a significant number of pixels are bright, the histogram will have two modes, one in the left region and the other in the right region.
 4. A flat histogram signifies that the image has a uniform distribution of all intensities. Then, the contrast is high and image will look vibrant.

Section 4

Intensity Transformations: Histogram Equalization

Histogram Equalization

1. Photographers like to shoot pictures with a flat histogram, as such pictures are vibrant.
2. Histogram equalization is a gray-level transformation that results in an image with a more or less flat histogram.
3. We can take an image and make its histogram flat by using the operation called histogram equalization.

Consider, for now, that continuous intensity values of an image are to be processed. We assume that $r \in [0, L - 1]$. Lets consider the intensity transformation

$$s = T(r) \quad 0 \leq r \leq L - 1 \quad (1)$$

that produces an output intensity level s for every pixel in the input image having intensity r . We assume that

- $T(r)$ is monotonically increasing in the interval $0 \leq r \leq L - 1$, and
- $0 \leq T(r) \leq L - 1$ for $0 \leq r \leq L - 1$.

The intensity levels in the image may be viewed as random variables in the interval $[0, L - 1]$. Let $p_r(r)$ and $p_s(s)$ denote the probability density functions (PDFs) of r and s . A fundamental result from basic probability theory is that if $p_r(r)$ and $T(r)$ are known, and $T(r)$ is continuous and differentiable over the range of values of interest, then the PDF of the transformed variable s can be obtained using the simple formula

$$p_s(s) = p_r(r) \left| \frac{dr}{ds} \right|. \quad (2)$$

Now let's consider the following transform function:

$$s = T(r) = (L - 1) \int_0^r p_r(w) dw, \quad (3)$$

where w is the dummy variable of integration. The right-hand side of this equation is the cumulative distribution function (CDF) of random variable r . This function satisfies the two conditions that we mentioned. range 0 to L-1 and monotonically increasing

- Because PDFs are always positive and the integral of a function is the area under the curve, Equation 3 satisfies the first condition. This is because the area under the curve cannot decrease as r increases.
- When the upper limit in this equation is $r = L - 1$, the integral evaluates to 1, because the area under a PDF curve is always 1. So the maximum value of s is $L - 1$, and the second condition is also satisfied.

To find $p_s(s)$ corresponding to this transformation we use Equation 1.

$$\begin{aligned}\frac{ds}{dr} &= \frac{dT(r)}{dr}, \\ &= (L-1) \frac{d}{dr} \left[\int_0^r p_r(w) dw \right], \\ &= (L-1) p_r(r).\end{aligned}\tag{4}$$

Substituting this result in Equation 1, and keeping in mind that all probability values are positive, yields

the derivative of r with respect to s . It captures how the transformation from r to s changes with respect to the values of r and s . This term is important because it accounts for the scaling and deformation of the PDF due to the transformation.

$$\begin{aligned}p_s(s) &= p_r(r) \left| \frac{dr}{ds} \right|, \\ &= p_r(r) \left| \frac{1}{(L-1)p_r(r)} \right|, \\ &= \frac{1}{L-1} \quad 0 \leq s \leq L-1.\end{aligned}$$

$p_s(s)$
(s): This represents the probability density function of the transformed variable s . In other words, it describes the likelihood of observing a specific value s in the transformed space.

$p_r(r)$
(r): This represents the probability density function of the original variable r . It describes the likelihood of observing a specific value r in the original space.

(5)

We recognize the form of $p_s(s)$ in the last line of this equation as a uniform probability density function.

For discrete values, we deal with probabilities (histogram values) and the summation instead of probability density functions and integrals. The probability of occurrence of intensity level r_k in a digital image is approximated by

$$p_r(r_k) = \frac{n_k}{MN} \quad k = 0, 1, \dots, L-1, \quad (6)$$

where MN is the total number of pixels in the image, n_k is the number of pixels that have intensity r_k , and L is the number of possible intensity levels in the image (e.g., 256). Recall that a plot of $p_r(r_k)$ versus r_k is commonly referred to as a histogram.

The discrete form of the Equation 2 is

$$\begin{aligned} s_k &= T(r_k) = (L-1) \sum_{j=0}^k p_r(r_j), \\ &= \frac{(L-1)}{MN} \sum_{j=0}^k n_j \quad k = 0, 1, \dots, L-1. \end{aligned} \quad (7)$$

Thus the output image is obtained by mapping each pixel in the input image with intensity r_k into a corresponding pixel level s_k in the output image using Equation 7.

s the output image is obtained by mapping each pixel in the input image with intensity r_k into a corresponding pixel level s_k in the output image

Example

Suppose that a 3-bit image ($L = 8$) of size 64×64 pixels ($MN = 4096$) has the intensity distribution shown in Table 4, where the intensity levels are in the range $[0, L - 1] = [0, 7]$. carry out histogram equalization.

MN is the total number of pixels in the image, n_k is the number of pixels that have intensity r_k , and L is the number of possible intensity levels in the image (e.g., 256). Recall that a plot of $p_r(r_k)$ versus r_k is commonly referred to as a histogram.

64*64

8-1=7

r_k	n_k	$p_r(r_k) = n_k/MN$	$\sum_{j=0}^k n_j$	$\frac{(L-1)}{MN} \sum_{j=0}^k n_j$	Rounded
$r_0 = 0$	790	0.19			
$r_1 = 1$	1023	0.25			
$r_2 = 2$	850	0.21			
$r_3 = 3$	656	0.16			
$r_4 = 4$	329	0.08			
$r_5 = 5$	245	0.06			
$r_6 = 6$	122	0.03			
$r_7 = 7$	81	0.02			

Table: Table for Example 1.

MN is the total number of pixels in the image, n_k is the number of pixels that have intensity r_k , and L is the number of possible intensity levels in the image (e.g., 256). Recall that a plot of $p_r(r_k)$ versus r_k is commonly referred to as a histogram.

r_k	n_k	$p_r(r_k) = n_k/MN$	$\sum_{j=0}^k n_j$	$\frac{(L-1)}{MN} \sum_{j=0}^k n_j$	Rounded
$r_0 = 0$	790	0.19	790		
$r_1 = 1$	1023	0.25			
$r_2 = 2$	850	0.21			
$r_3 = 3$	656	0.16			
$r_4 = 4$	329	0.08			
$r_5 = 5$	245	0.06			
$r_6 = 6$	122	0.03			
$r_7 = 7$	81	0.02			

Table: Table for Example 1.

r_k	n_k	$p_r(r_k) = n_k/MN$	$\sum_{j=0}^k n_j$	$\frac{(L-1)}{MN} \sum_{j=0}^k n_j$	Rounded
$r_0 = 0$	790	0.19	790		
$r_1 = 1$	1023	0.25	1813		
$r_2 = 2$	850	0.21	2663		
$r_3 = 3$	656	0.16	3319		
$r_4 = 4$	329	0.08	3648		
$r_5 = 5$	245	0.06	3893		
$r_6 = 6$	122	0.03	4015		
$r_7 = 7$	81	0.02	4096		

Table: Table for Example 1.

r_k	n_k	$p_r(r_k) = n_k/MN$	$\sum_{j=0}^k n_j$	$\frac{(L-1)}{MN} \sum_{j=0}^k n_j$	Rounded
$r_0 = 0$	790	0.19	790	1.350	1
$r_1 = 1$	1023	0.25	1813	3.098	3
$r_2 = 2$	850	0.21	2663	4.551	5
$r_3 = 3$	656	0.16	3319	5.672	6
$r_4 = 4$	329	0.08	3648	6.234	6
$r_5 = 5$	245	0.06	3893	6.653	7
$r_6 = 6$	122	0.03	4015	6.862	7
$r_7 = 7$	81	0.02	4096	7.000	7

Table: Table for Example 1.

Histogram Equalization I

`np.histogram()` is being used to calculate the histogram of pixel intensities in a flattened image array (`img.ravel()`) with 256 bins ranging from 0 to 255. The syntax is as follows:

256: This specifies the number of bins in the histogram. In this case, the histogram will be divided into 256 bins, each representing a range of pixel intensities.

Listing: Histogram Equalization

```
1 import cv2 as cv
2 import numpy as np
3 from matplotlib import pyplot as plt
4 img = cv.imread('../images/shells.tif', cv.IMREAD_GRAYSCALE)
5
6 hist, bins = np.histogram(img.ravel(), 256, [0, 256])
7 cdf = hist.cumsum()
8 cdf_normalized = cdf * hist.max() / cdf.max()
9 plt.plot(cdf_normalized, color = 'b')
10 plt.hist(img.flatten(), 256, [0, 256], color = 'r')
11 plt.xlim([0, 256])
12 plt.legend(('cdf', 'histogram'), loc = 'upper left')
13 plt.title('Histogram of the Original Image')
14 plt.show()
15 equ = cv.equalizeHist(img)
16 hist, bins = np.histogram(equ.ravel(), 256, [0, 256])
17 cdf = hist.cumsum()
18 cdf_normalized = cdf * hist.max() / cdf.max()
```

Grayscale images are typically represented as 2D arrays, where each element represents the pixel intensity value at a specific location in the image.

By applying `img.ravel()`, you are essentially converting this 2D array into a 1D array where all pixel values are stored sequentially. This is commonly done when calculating histograms or performing operations that involve iterating through all the pixels in the image. The flattened 1D array is then used for histogram calculation and analysis.

The `.flatten()` method is used to convert a multi-dimensional array, such as an image, into a one-dimensional array. In the context of image processing, when applied to an image, it converts the 2D image array into a 1D array, where all the pixel values are

Histogram Equalization II

Similar to the original image, the histogram and CDF of the equalized image are computed.

Plotting Equalized Image Histogram and CDF:

The normalized CDF of the equalized image is plotted in blue, and the histogram of the equalized image is plotted in red, similar to the original image.

Creating Image Comparison:

The original image and the equalized image are stacked horizontally using `np.hstack()`. This creates an image where the original image and the equalized image are placed side by side for comparison.

Displaying Images:

The `plt.axis('off')` command turns off the axis labels and ticks for cleaner visualization.

The original grayscale image is displayed using `plt.imshow()`, specifying the colormap as 'gray' to display it in grayscale.

The stacked image (original and equalized) is displayed using `plt.imshow()` as well.

Saving the Figure:

```
19 plt.plot(cdf_normalized, color = 'b')
20 plt.hist(equ.flatten(), 256, [0, 256], color = 'r')
21 plt.xlim([0, 256])
22 plt.legend(('cdf', 'histogram'), loc = 'upper left')
23 plt.title('Histogram of the Equalized Image')
24 plt.show()
25 res = np.hstack((img, equ))
26 plt.axis('off')
27 plt.imshow(res, cmap='gray')
```

Calculating Cumulative Distribution Function (CDF):

The cumulative sum of the histogram values is calculated using the `.cumsum()` method. This represents the cumulative distribution function (CDF) of the pixel intensity values.

Normalizing the CDF:

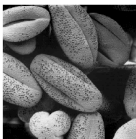
The CDF is normalized so that its values range between 0 and the maximum histogram value. This helps in visualizing the CDF and histogram on the same plot later.

Plotting Original Image Histogram and CDF:

The normalized CDF is plotted in blue (`cdf_normalized`), and the histogram of the original image is plotted in red (histogram). The x-axis represents the pixel intensity values (ranging from 0 to 255), and the y-axis represents the CDF/histogram values.

Equalization using `cv.equalizeHist()`:

Histogram Equalization



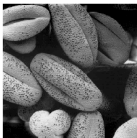
(a) Original image

`img.ravel()` is a NumPy function used to flatten a multi-dimensional array, such as an image, into a one-dimensional array. In the context of image processing, it's often used to convert an image matrix (2D array) into a 1D array where all the pixel values are stored sequentially, row by row.

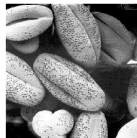
In the provided code, `img` is a grayscale image loaded using OpenCV's `cv.imread()` function. Grayscale images are typically represented as 2D arrays, where each element represents the pixel intensity value at a specific location in the image.

By applying `img.ravel()`, you are essentially converting this 2D array into a 1D array where all pixel values are stored sequentially. This is commonly done when calculating histograms or performing operations that involve iterating through all the pixels in the image. The flattened 1D array is then used for histogram calculation and analysis.

Histogram Equalization



(a) Original image



(b) Histogram-equalized image

Figure: Histogram Equalization

Power-law transformation, also known as gamma correction, is a technique used to adjust the brightness and contrast of an image by applying a mathematical function that raises the pixel values to a certain power.

Where:

g is the output pixel value after transformation.

f is the input pixel value before transformation.

γ (gamma) is the parameter that determines the shape of the transformation curve.

Section 5

Intensity Transformations: Gamma Correction

$0 < \gamma < 1$ (γ is between 0 and 1):

When the gamma value is between 0 and 1, it results in a nonlinear transformation that enhances the contrast of darker pixels while compressing the range of lighter pixels. This means that the transformation emphasizes the details in darker areas of the image. This can be particularly useful when you want to enhance details in shadows or low-light regions.

$\gamma > 1$ (γ is greater than 1):

When gamma is greater than 1, the transformation has the opposite effect. It compresses the darker pixel values and expands the lighter ones. This results in a contrast-enhancing effect in the brighter regions of the image, potentially leading to the loss of details in darker areas.

$\gamma = 1$:

When gamma is equal to 1, the transformation is linear and represents an identity transform. This means that the input and output pixel values are the same, resulting in no change to the image's contrast or brightness.

Power-Law Transformation (Gamma Correction)

$$g = f^\gamma, \quad f \in [0, 1].$$

Values of γ such that $0 < \gamma < 1$ map a narrow range of dark pixels to a wider range of dark pixels.

$\gamma > 0$ has the opposite effect.

$\gamma = 1$ gives the identity transform.

In essence, adjusting the gamma value allows you to control the mapping of pixel values in a way that enhances specific tonal ranges in the image. Lower gamma values emphasize darker regions, higher gamma values enhance brighter regions, and a gamma value of 1 maintains the original tonal distribution.

Gamma correction is often used to compensate for the nonlinear response of displays and cameras, ensuring that images appear visually consistent and natural across different devices. Additionally, it's used in image processing to enhance images for specific applications, such as medical imaging or enhancing details in photographs taken in varying lighting conditions.

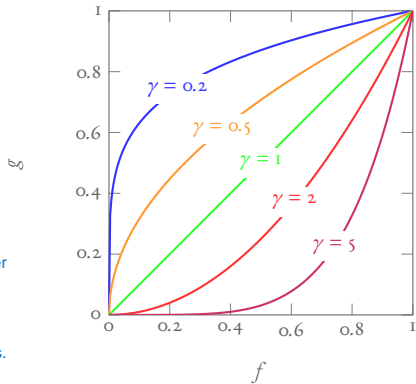


Figure: Plots of f^γ

Gamma Correction I

Listing: Gamma Correction

```
1 %matplotlib inline
2 import cv2 as cv
3 import matplotlib.pyplot as plt
4 import numpy as np
5 img_orig = cv.imread( '../images/gal.jpg', cv.IMREAD_COLOR)
6 gamma = 2
7 table = np.array([(i/255.0)**(gamma)*255.0 for i in np.arange(0,256)]).astype('uint8')
8 img_gamma = cv.LUT(img_orig, table)
9 img_orig = cv.cvtColor(img_orig, cv.COLOR_BGR2RGB)
10 img_gamma = cv.cvtColor(img_gamma, cv.COLOR_BGR2RGB)
11 f, axarr = plt.subplots(3,2)
12 axarr[0,0].imshow(img_orig)
13 axarr[0,1].imshow(img_gamma)
14 color = ('b', 'g', 'r')
15 for i, c in enumerate(color):
16     hist_orig = cv.calcHist([img_orig], [i], None, [256], [0,256])
17     axarr[1,0].plot(hist_orig, color = c)
18     hist_gamma = cv.calcHist([img_gamma], [i], None, [256], [0,256])
19     axarr[1,1].plot(hist_gamma, color = c)
```

Gamma Correction II

```
20 axarr[2,0].plot(table)
21 axarr[2,0].set_xlim(0,255)
22 axarr[2,0].set_ylim(0,255)
23 axarr[2,0].set_aspect('equal')
```


Summary

1. Basics: digital image, color images, creating an image in Python, displaying images, increasing brightness, image planes
2. Intensity transformations:
 - 2.1 Identity, negative, intensity windowing
 - 2.2 Histograms, histogram equalization
 - 2.3 Gamma correction