ELEC 421
Digital Signal and Image Processing

Siamak Najarian, Ph.D., P.Eng.,

Professor of Biomedical Engineering (retired),

Electrical and Computer Engineering Department,

University of British Columbia

Course Roadmap for DIP

Lecture	Title
Lecture 1	Digital Image Modalities and Processing
Lecture 2	The Human Visual System, Perception, and Color
Lecture 3	Image Acquisition and Sensing
Lecture 4	Histograms and Point Operations
Lecture 5	Geometric Operations
Lecture 6	Spatial Filters

Lecture 4:
Histograms and Point Operations

Table of Contents

Image histograms

HISTOGRAMS AND POINT OPERATIONS

$I(x, y)$

- **Histogram and Point Operations:** Let us see how we actually take a digital image as input and process it to do something else. The first thing we are going to talk about is called **point operations**, which are related to the **histogram of the intensity**. These are the kinds of image processing operations that we would do if we had an image and we wanted to improve its contrast, for example, or it is washed out and we want to make it look better. Note that a **washed-out image** refers to a digital photograph or image that has lost its vibrancy and contrast. It appears pale, lacking in rich colors or deep shadows. These are the kinds of things that are a built-in option in Photoshop or similar software packages. We want to know how that actually works under the hood.
- The way to think about this is that we have an image with, $\mathbf{I(x,y)}$, in which the $\mathbf{(x,y)}$ tells us what pixel we care about and the value of $\mathbf{I(x,y)}$ gives us the very **scale value** or the color. Last time, we talked about how we discretize the $\mathbf{(x,y)}$ **space**, which tells us about the **resolution of the image**. Also, we discussed how we can discretize the **number of levels** that we could take. This tells us how many levels of the **intensity** we have. We also discussed the **bit depth**.

Image histograms

HISTOGRAMS AND POINT OPERATIONS

$$I(x, y)$$

$$h(D) = \#\text{ of pixels in } I(x,y) \text{ that have intensity } D$$

$$= \left| \left\{ I(x,y) \mid I(x,y) = D \right\} \right| \quad (1)$$

- **Image Histogram:** An inherent property of an image is its **histogram**. An image histogram is a graphical representation that shows the distribution of pixel intensities in an image. It essentially counts the number of pixels for each possible intensity value.
- We could define the histogram as a function of intensity **D**, where **h(D)** gives us the number of pixels in the image that have intensity **D**. If we wanted to be a little bit more mathematical, we could say it is like the **cardinality** of the set of pixels such that the pixel value is equal to **D**. This is shown in **(1)**. Here, the function denoted as **h(D)** represents the cardinality of the set of pixels in the image (i.e., the number of pixels), where the pixel value is equal to **D**.

MATLAB example

```
>> im = imread('prostate.tif');
>> imshow(im) (1)
```

A

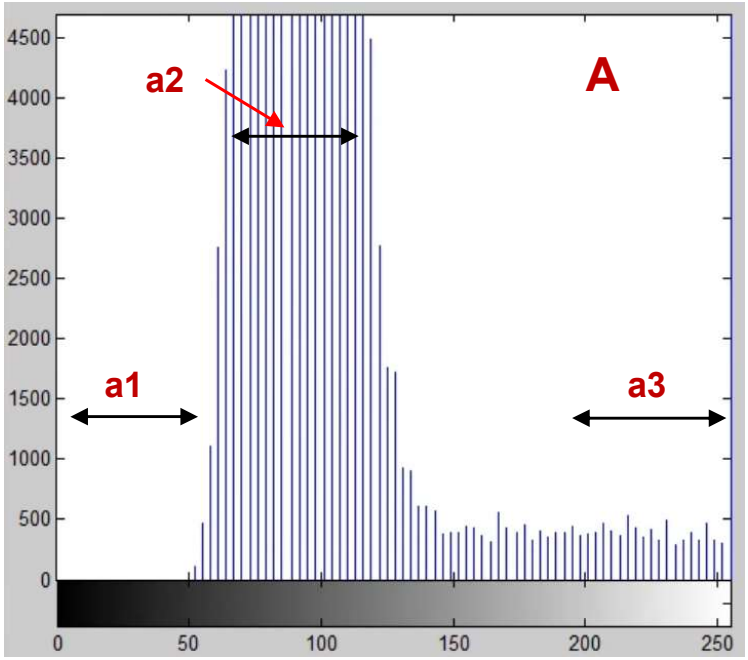
- We have already discussed some MATLAB built-in functions for digital image processing. For instance, how to ***read in*** images, **imread**. And, how to ***show an image***, **imshow**.
- In **A**, we have an image from medical imaging field, which shows a prostate CT image. We can see that there is some light colors from the bones, some darker colors from the inside the body, and some other colors from various organs.

MATLAB example

```
>> imhist(im)    (1)
Error in imhist (line 60)
[a, n, isScaled, top, map] =
parse_inputs(varargin{:});
```

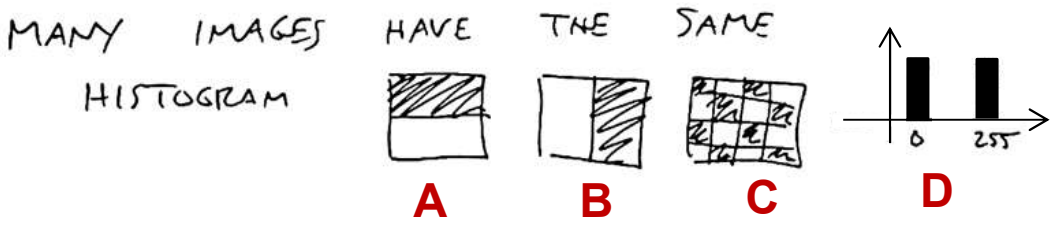
```
>> whos    (2)
  Name          Size

  im          400x400x3
```

```
>> im = rgb2gray(im);    (3)
>> imhist(im)
```



- There is a command called **imhist**. If we do **imhist** of **im**, **imhist(im)**, we might get an error, as shown in **(1)**. This is because our original image was not in grayscale, but was an RGB image, as shown by **3** (for **3** color channels) in **400×400×3** in **whos**, **(2)**. So, if we look at the original CT image, we see that even though this image looks like it is a grayscale image, it is actually an RGB image instead. To tackle this, first, we need to turn that into a grayscale image using **(3)**, which converts it into a grayscale image. If we do this, we should get what looks like the histogram in **A**. The **x**-axis goes from **0** to **255**, which are the levels of the grayscale image. In region **a1**, we can see that there are not very many totally dark pixels. There is really nothing below intensity **50**. Then there is a whole bunch of grayish pixels in region **a2**, and then an evenly distributed bunch of whitish pixels in region **a3**. That makes sense, because that is what the image looks like.

MATLAB example

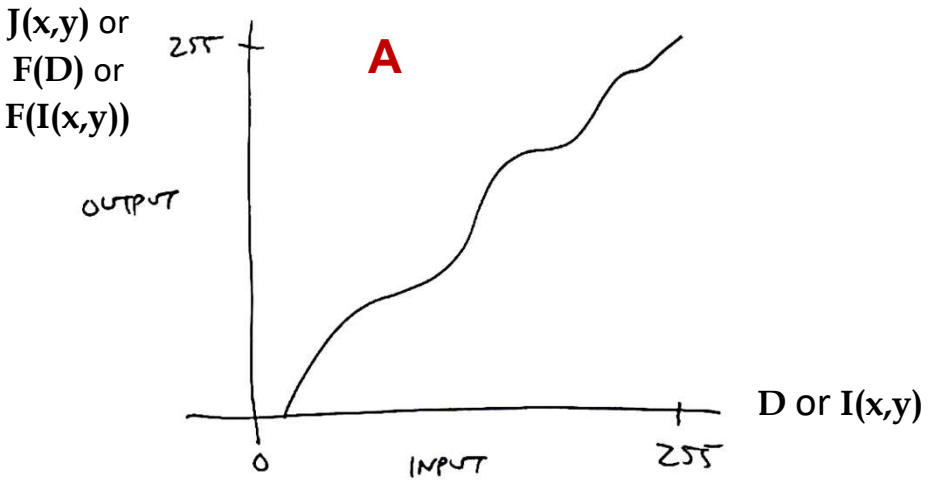


- Next, we talk about ways to change the image by considering its histogram. First of all, let us just mention that the histogram is ***not a one-to-one property*** of an image. This means that there are many images that have the same histogram. It is not like we can take the histogram of the image and really ***uniquely describe*** anything about it.
- For example, if we look at an image that is black on top and white on the bottom, **A**, or black on the right side, and white on the left side, **B**, or it is a checkerboard, **C**, that is half black, half white, all these images are going to have a histogram that looks like **D**. In **D**, we have a whole bunch of stuff that is black, i.e., at **0**, and a whole bunch of stuff that is white, i.e., at **255**.
- **Conclusion:** We cannot really tell exactly what the image is from just looking at the histogram. But we ***do*** get a sense of how their pixel intensities are distributed and that turns out to be good enough for things like contrast enhancement, for example.

Point operations

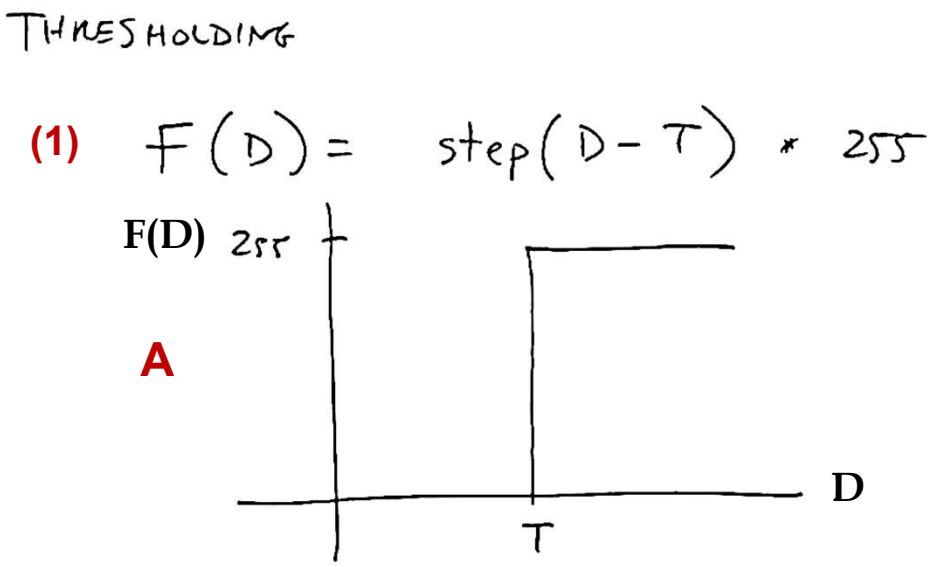
$$J(x,y) = F(I(x,y)) \quad (1)$$

AFTER BEFORE



- **Pont Operations:** We are going to talk about what is called **point operations**. What we do is that we have an ***afterwards image*** $\mathbf{J(x,y)}$ that is like taking a function of the ***before image*** $\mathbf{F(I(x,y))}$, shown by **(1)**. So, $\mathbf{I(x,y)}$ is the before image and $\mathbf{J(x,y)}$ is the after image.
- All this is saying is we are not changing the locations of where colors are located. All we do is we take the color or the grayscale intensity that used to be at pixel $\mathbf{(x,y)}$, look at what intensity that is, and then, turn that into some new intensity, $\mathbf{J(x,y)}$. This is shown in **A**. This says every pixel that had color $\mathbf{D}$ before, $\mathbf{I(x,y)}$, gets turned into some new pixel that has color $\mathbf{F(D)}$ or $\mathbf{J(x,y)}$. So, all the pixels that have the same intensity or color will change in the same way. We can think about this like an **input-output graph**, where the **x**-axis is the input intensity and the **y**-axis is the output intensity. Both the **x**-axis and the **y**-axis range between black to white.
- **Conclusion:** Here, we are doing a graph that ***maps*** a **before-color** image to an **after-color** image.

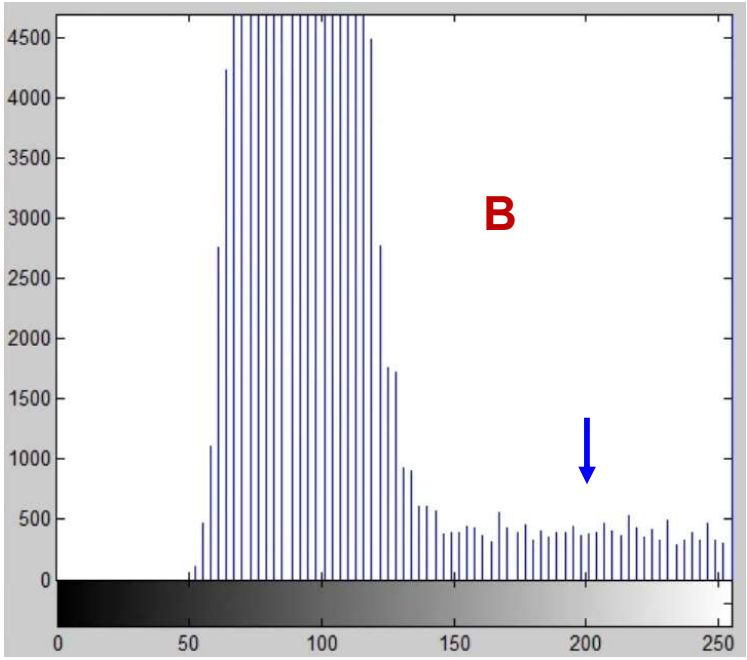
Thresholding



- **Thresholding:** Let us look at some examples of these kinds of operations that are represented by $\mathbf{J(x,y) = F(I(x,y))}$. One obvious one is thresholding. This happens a lot in biomedical image processing and computer vision. That is like saying the function of intensity, $\mathbf{F(D)}$, is a **step function** of the intensity, $\mathbf{D}$, minus some threshold, $\mathbf{T}$, i.e., $\mathbf{step(D - T)}$. Put it differently, it is saying that anything above $\mathbf{T}$ gets turned into **1** and anything that is below $\mathbf{T}$ gets turned to **0**. If we were being really proper about this, we would also scale up to **255**, as shown in **A**. So, $\mathbf{F(D) = step(D - T) \times 255}$.

Thresholding

A



- Let us see how thresholding works in MATLAB. We will use the prostate CT image, **A**. Suppose we want to find the **bones** in this image. Therefore, we should be looking for the things that are brightly colored. When we look at our histogram, **B**, it seems like the white stuff starts around **200**, shown by **blue** downward arrow.

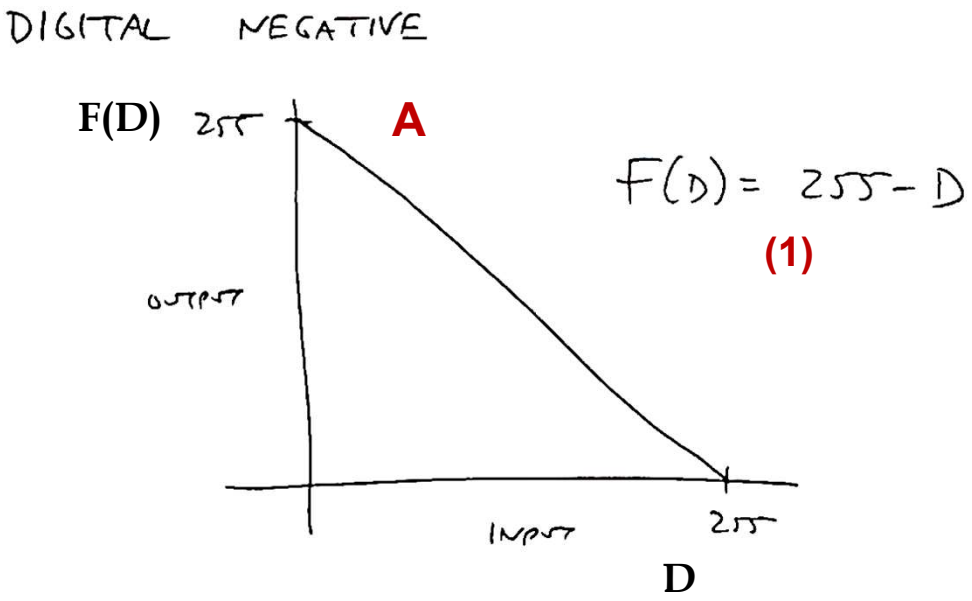
Thresholding

```
>> J = (im>200);  (1)
>> whos
  Name         Size          Bytes  Class

  J          400x400        160000  logical
  im         400x400        160000  uint8     (2)

>> imshow(J,[])      (3)
```

A

- Next, we could make a new image that says we only want to see all the pixels that are greater than **200**, in order to see only bones. This is like a MATLAB **logical command**. As shown in **(1)**, the command is **J = (im>200)**. When we do this, we get an image of the same size, i.e., **J** and **im** are the same size, as shown in **whos**, **(2)**. Here, **J** is a ***logical variable***, which means we can see only **0** or **1** inside there. We can now use, **imshow(J,[])**, which shows **J** is scaling things to black and white, as shown in **(3)**.
- Following the above, we get a picture that looks like **A**. Now, we see that the bones are picked out by looking at the specified threshold (i.e., **200**). This is a **binary image**. Note that there is no guarantee that we can go backwards from this image to our original image. This is because we have lost information by doing thresholding. If we have a one-to-one mapping, then theoretically, it is possible to go backwards, but here, it is not.

Digital negative



- **Digital Negative:** Another very common operation is called **digital negative**, or sometimes, simply called **negative**. Here, we take our before-intensity, $\mathbf{D}$, which is our input axis, and we subtract it from $\mathbf{255}$. That is, $\mathbf{F(D) = 255 - D}$, where $\mathbf{F(D)}$ the output axis, as shown by **(1)**. This graph is shown in **A**.
- In the formula $\mathbf{F(D) = 255 - D}$, where $\mathbf{D}$ is the original intensity value, $\mathbf{255}$ typically represents the maximum intensity value for grayscale images (white). Subtracting $\mathbf{D}$ from $\mathbf{255}$ ***inverts*** the intensity. A high value of $\mathbf{D}$ (bright pixel) results in a lower value in the negative (darker pixel).
- A digital negative essentially inverts the ***tonal values*** (***brightness***) of an image. Pixels with high intensity values (bright areas) in the original image become dark in the negative, and vice versa. This inversion can be useful for **visualization purposes**. It can help to highlight details in dark areas. This means features hidden in shadows in the original image might become more prominent in the negative. It can also help to identify flaws. That is, dust particles or scratches on a negative film appear as bright spots in the digital negative, which can aid in **image restoration**.

Digital negative

```
>> im = imread('conan.jpg');
>> imshow(im)
```

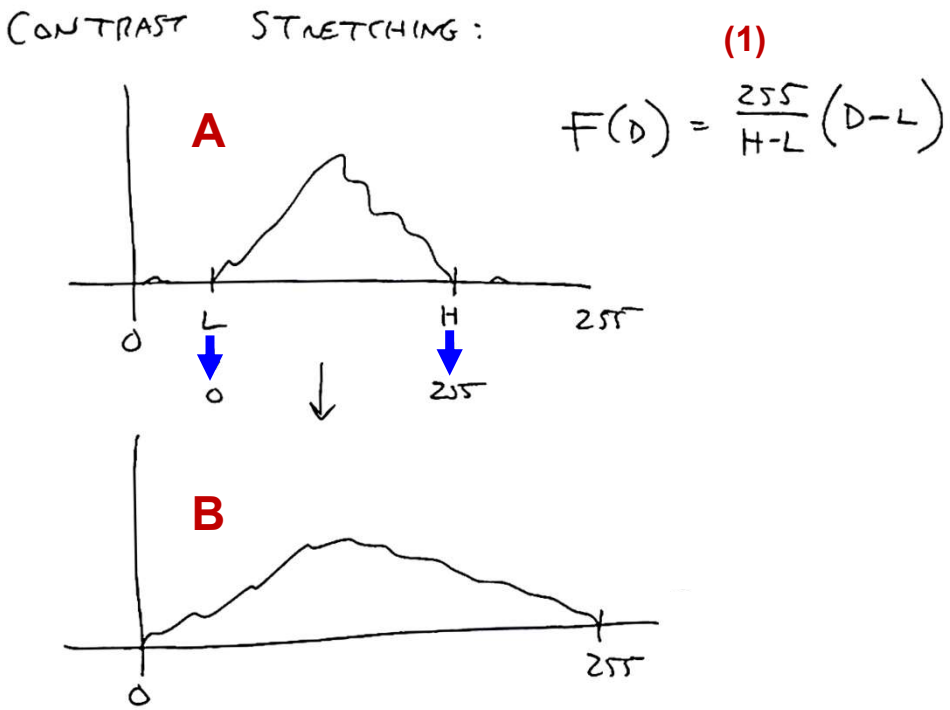
A

```
>> imshow(255-im)  (1)
```

B

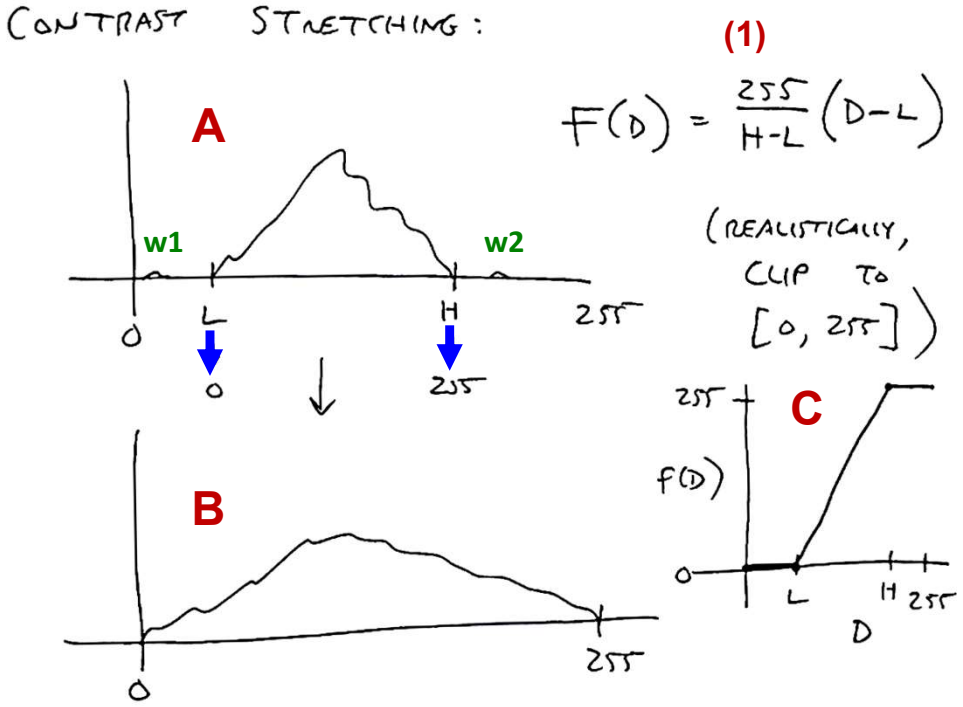
- **Example for Digital Negative:** If we were to take the digital negative of **A**, one thing to notice is that we do not have to make a new image. All we need to do is use the command **imshow(255 - im)**, as shown by **(1)**. This will show us the negative version, **B**.
- As pointed out earlier, there are some reasons why we might want to do that. Certainly, it visually often makes things stand out a little bit better. This is because the human perceptual system is not linear. This digital negative may cause details to pop in a way that we do not necessarily see by just looking at the original image.

Contrast stretching



- **Contrast Stretching:** Another useful point operation to do is **contrast stretching**, which leads to **contrast enhancement**. Suppose that we observed the histogram of the original image, and it was nearly all contained in one region, as shown in **A**. Here, between **0** and **255**, the histogram went from low value, **L**, to some high value, **H**, and we would like to turn that into something, where the lowest value is scaled to **0** and the highest value is scaled to **255**, as shown in **B**.
- In some sense, this is what **imshow** command is doing, when we give it the **brackets**, as the second argument. Using the brackets, we are saying stretch out so the lowest value goes to black and the highest value goes to white, as shown by **blue** arrows in **A**. This means that **L** goes to **0** and this **H** goes to **255**, and we are assuming everything else are stretched out **linearly**. We can think about it as just fitting a line to two points, as shown by **(1)**. We know that the slope of this line has got to be **255/(H – L)**. In **(1)**, when we put **D** equals **L**, we get **0** and when we put **D** equals **H**, we get **255**. This should stretch everything out.

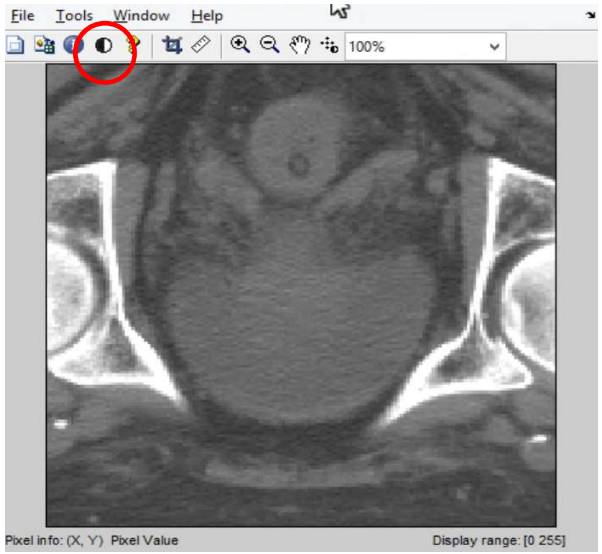
Contrast stretching



- In practice, with real digital images, maybe we do not have everything exactly contained inside the **L** and **H** in **A**. Maybe, we have some extra ***warps*** that are on the edges. These warps are shown by **w1** and **w2** in **A**.
- So, realistically, we should clip the intensities to **[0, 255]** range. That would mean the graph of function would look something like **C**. That is, we have the linear ramp between **L** and **H**, and **L** goes to **0**, and **H** goes to **255**. Now, anything higher than **H** just gets ***clipped*** to **255** and anything lower than **L** gets ***clipped*** to **0**. In graph **C**, the **y**-axis would be our $\mathbf{F(D)}$ and the **x**-axis would be our $\mathbf{D}$.

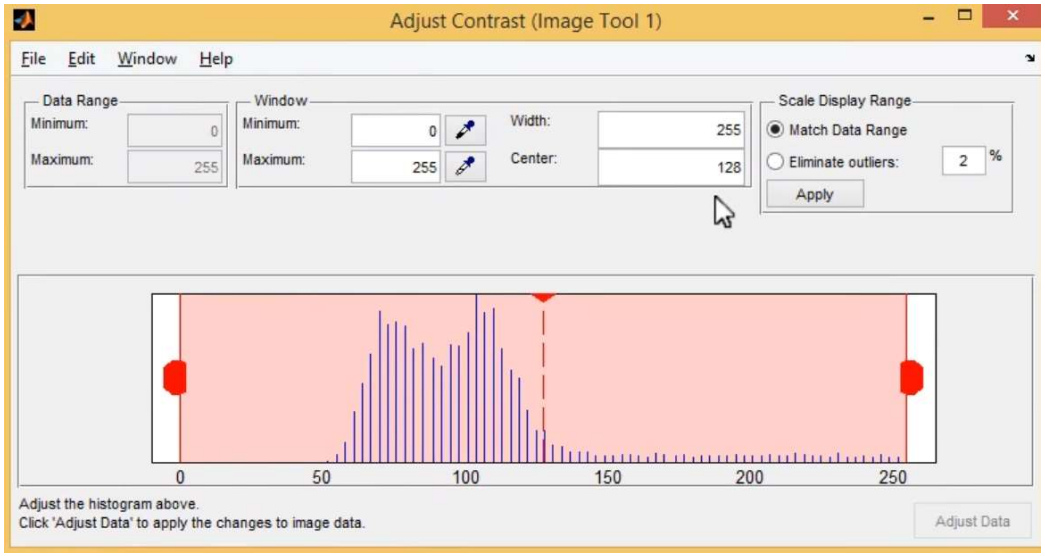
MATLAB's imtool

```
>> im = imread('prostate.tif');
>> im = rgb2gray(im);
>> imtool(im)
```



A

B

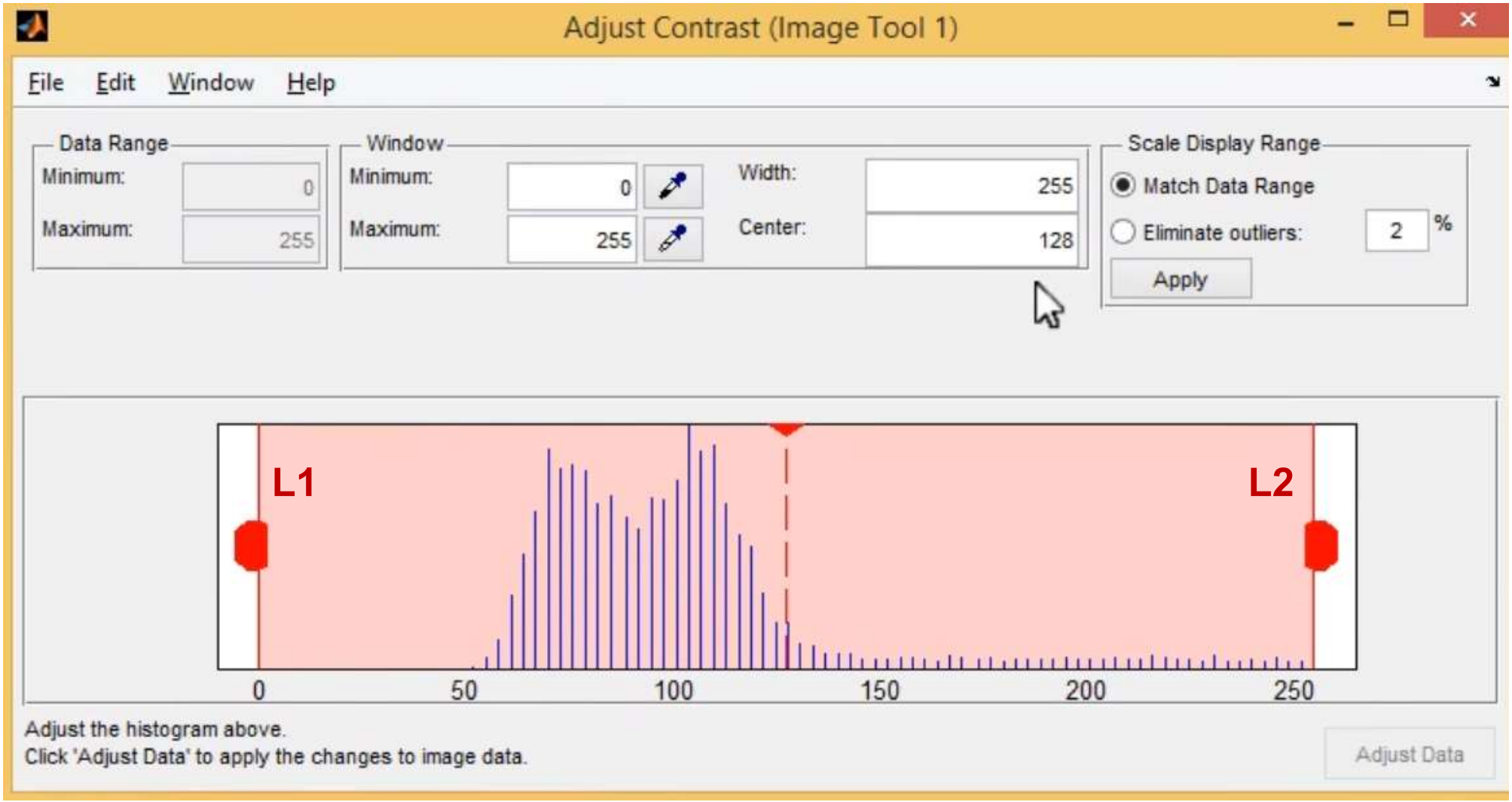


- Luckily, in terms of just visualizing an image, MATLAB ***does*** make it easy for us to see what would happen if we were to do **contrast stretching**. Instead of writing function to do that, there is a little gadget inside MATLAB called **imtool** that we can use.
- Let us again take the prostate CT image. If we do **imtool** of it, the image pops up, like the usual image, as shown in **A**. There is a **halfmoon icon**, shown by **red** circle, that we can click. This will show us the histogram, **B**, which we saw before.

MATLAB's imtool

- The solid **red** vertical lines can be ***dragged around*** to where we want the black and white to go. These lines are shown by **L1** and **L2** in **A**.

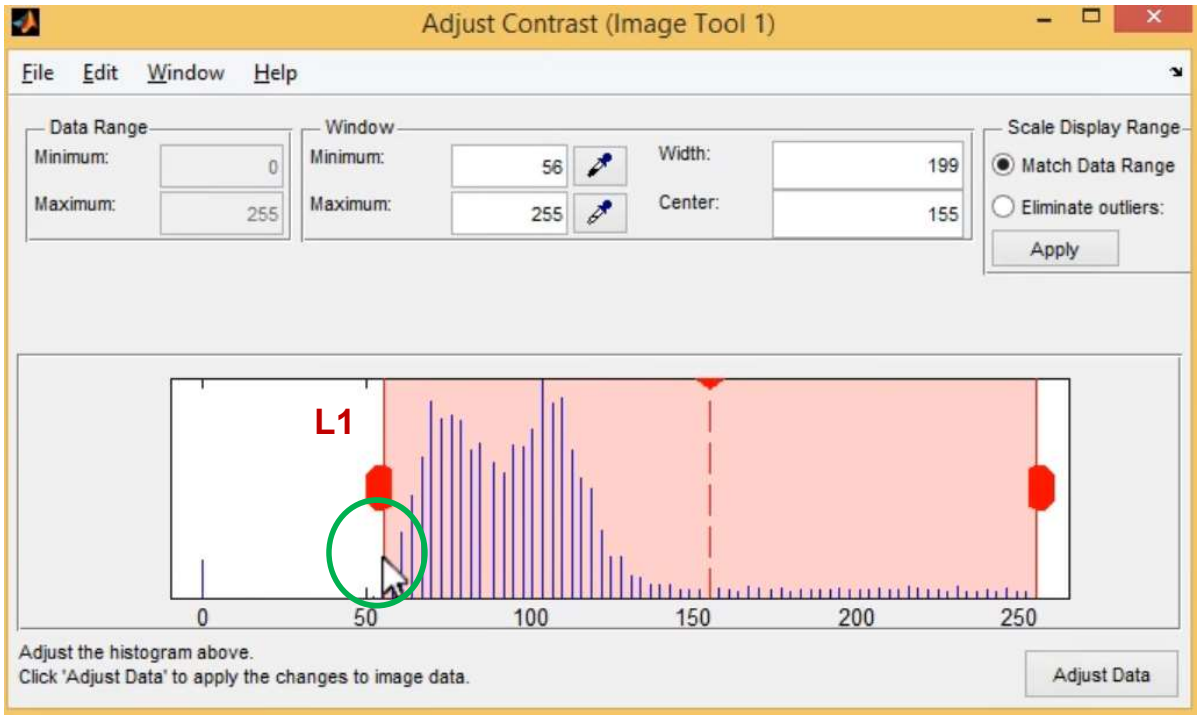
A



MATLAB's imtool

- For example, in **A**, if we move **L1** up (to the right), that is like saying scale by squishing the darker values so that what used to be kind of only dark gray becomes black. This is shown by **green** circle in **A**. The outcome is image **B**.

A

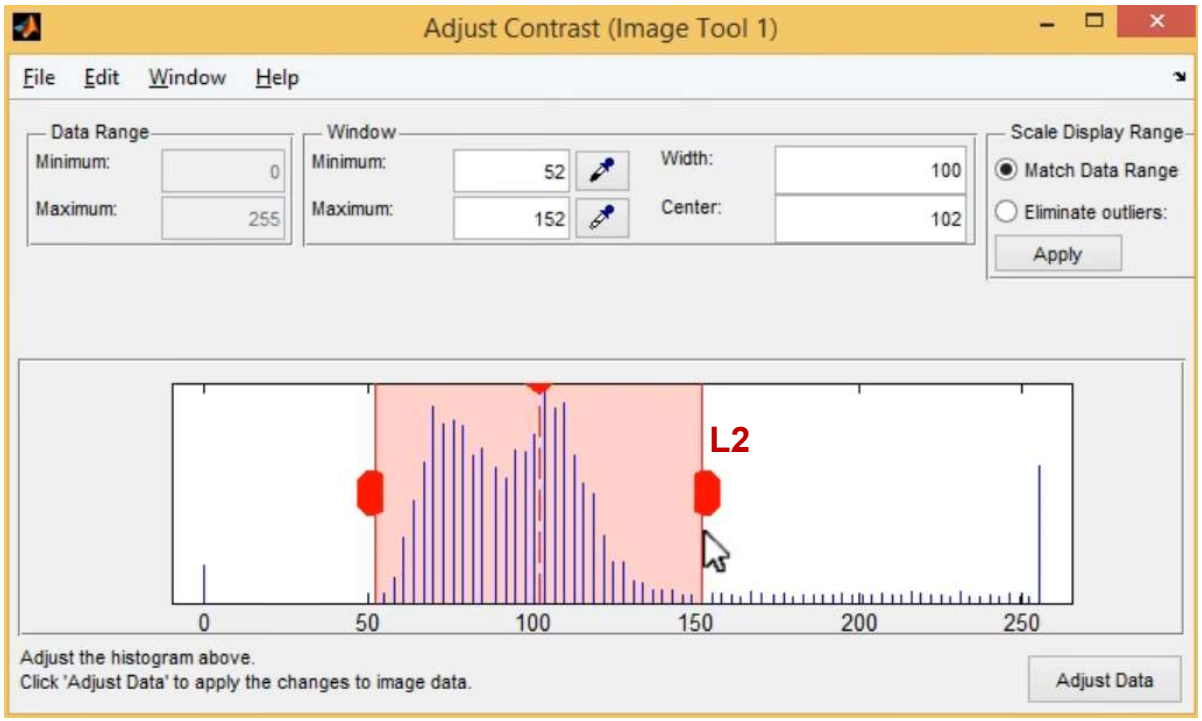


B

MATLAB's imtool

- In the original histogram, we noticed that we have got a whole bunch of pixels in the range between around **150** and **255** that are white enough so that we could move **L2** to around **150**, as shown in **A**. This image is what we call **more contrasty** than the original one. In **B**, we can maybe see things that we could not see before.

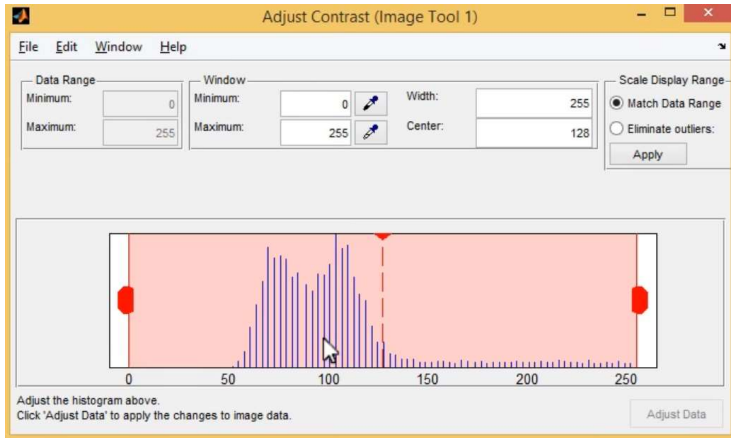
A



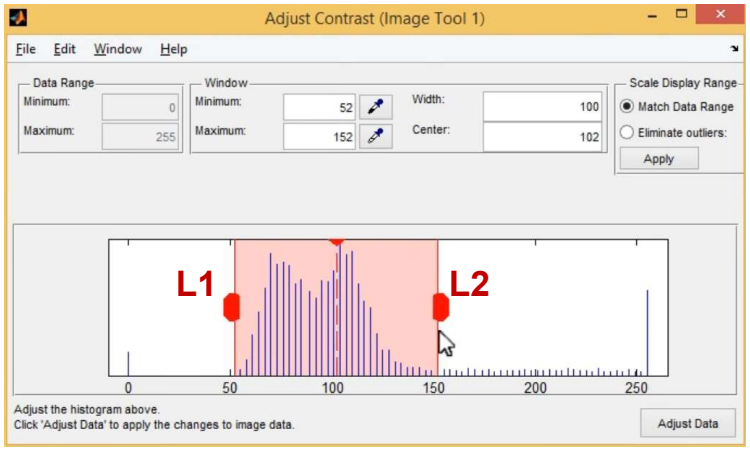
B

MATLAB's imtool

A



B

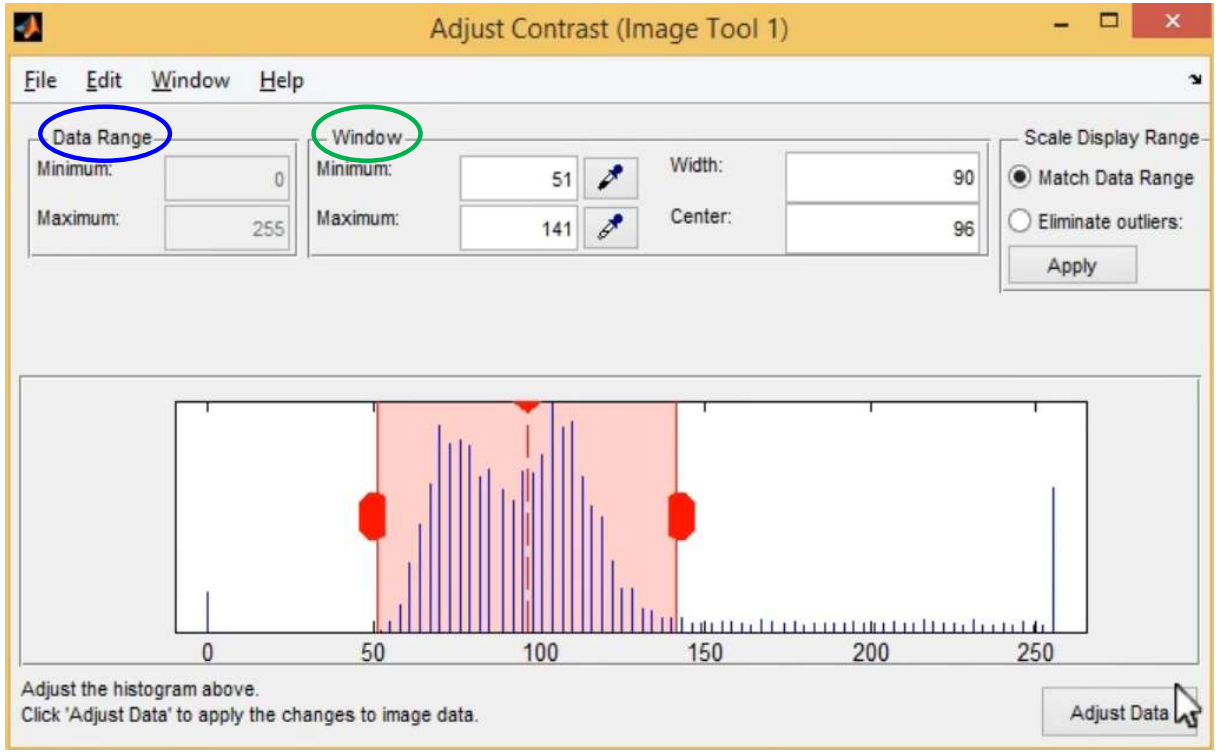


- If we were to go back to the original image, **A**, and compare it with **B**, we can see that in the middle region of the image in **A**, there is not a lot of detail inside this murky gray region. That is because most of the pixel intensities in this murky gray region are in the range of **50** to say **150**. Now, if we were to spread those intensities out a little bit more, over the whole grayscale, i.e., bring **L1** close **50** and **L2** close to **150**, we can make out details that we might not be able to make out before, as shown in **B**.
- Here, we are not changing the inherent information inside the image, but we are making it easier for us to visualize it. This is a **subjective decision** by us, the user, to decide how we want the image to look. But, this is a very common thing to do, especially in the context of medical imagery. So, if doctors are trying to find a tumor inside this CT-scan, the first thing they will do is **contrast stretching** operation, which in medical terminology is sometimes called **windowing and leveling**. That tells the doctors ***how to balance the histogram***.

MATLAB's imtool

- In **A**, in **Window**, shown by **green** oval, we can see that the minimum is **51** and the maximum is **141** and that we are mapping that to the whole **0** to **255** range, as shown in **Data Range** (the **blue** oval). The output image is shown in **B**.

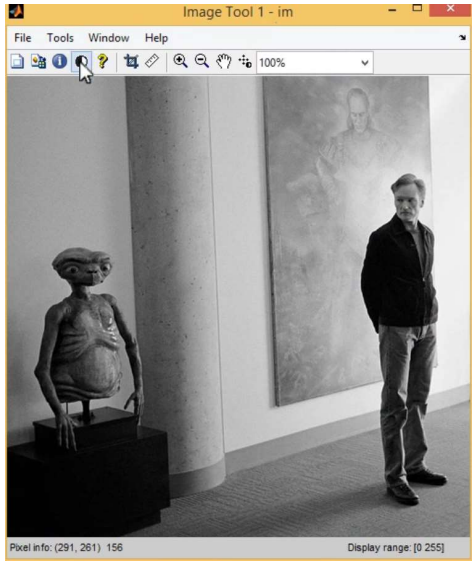
A



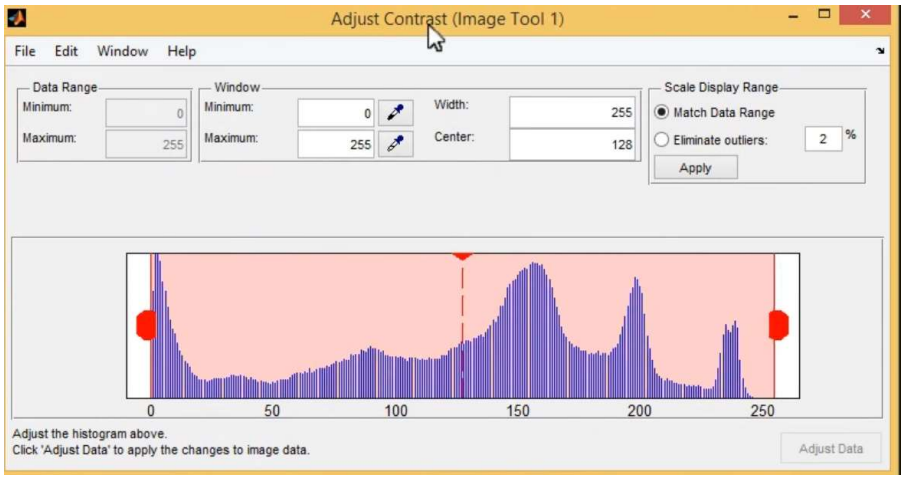
B

MATLAB's imtool

```
>> im = rgb2gray(imread('conan.jpg'));
>> imtool(im)
```



A

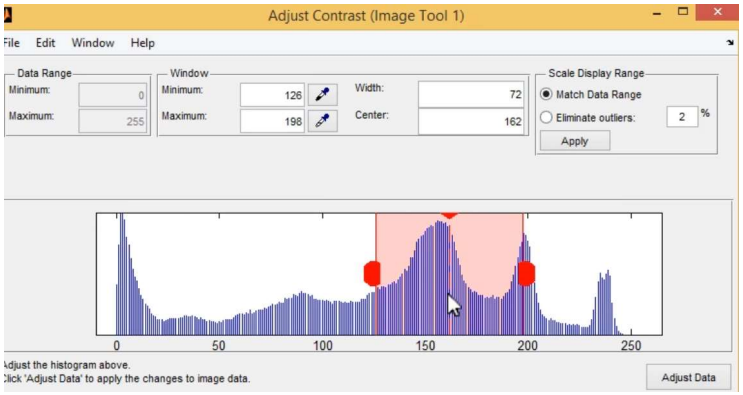


B

- Similarly, let us read in and see the Conan image, 'conan.jpg', **A**. This image is originally an RGB image. So, we need to convert that to grayscale. Note that the halfmoon icon does not appear if we do not have a grayscale image.
- In **B**, we can see the histogram of this image. This image is a little bit more balanced compared to the prostate CT-image. It has got dark values, it has got light values, and the person in front of the painting, Conan.

MATLAB's imtool

A



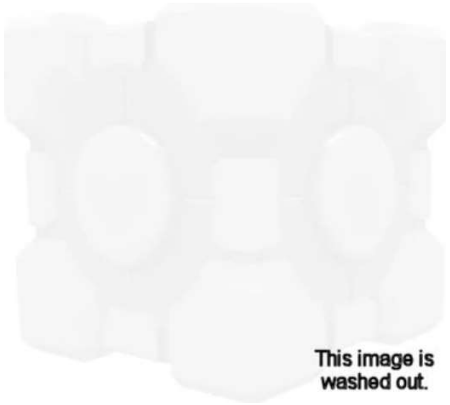
B

C

- **Balance in Image:** Maybe what we want to do is to balance image **A** so that we can make out what is going on in the **painting** a little better, shown by **P1**. For instance, because the painting seems to be kind of washed out. In the painting in **A**, there are lots of light-valued intensities inside it and we want to ***squish the intensity range*** so that those painting values can take up the whole black to white range.
- What we might do is to move the window in **B**, shaded by pale **red**, in order to try and isolate the painting we care. Here, we can drag back and forth this window around the two peaks shown in **B**. By doing that, and as shown in **C**, we can now kind of make out the face in the painting much more clearly than we could before, even though other parts of the image are now too dark or too blown out.
- **Conclusion:** There is no saying that we have to do something that works for the entire image all at once. We can instead ***do various image processing operations that work for just subsets of the image***.

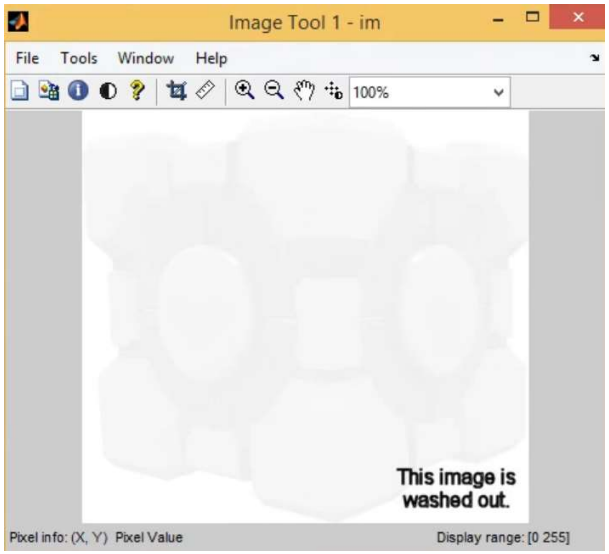
MATLAB's imtool

```
>> im= imread('cube1.jpg');
>> imshow(im)
```

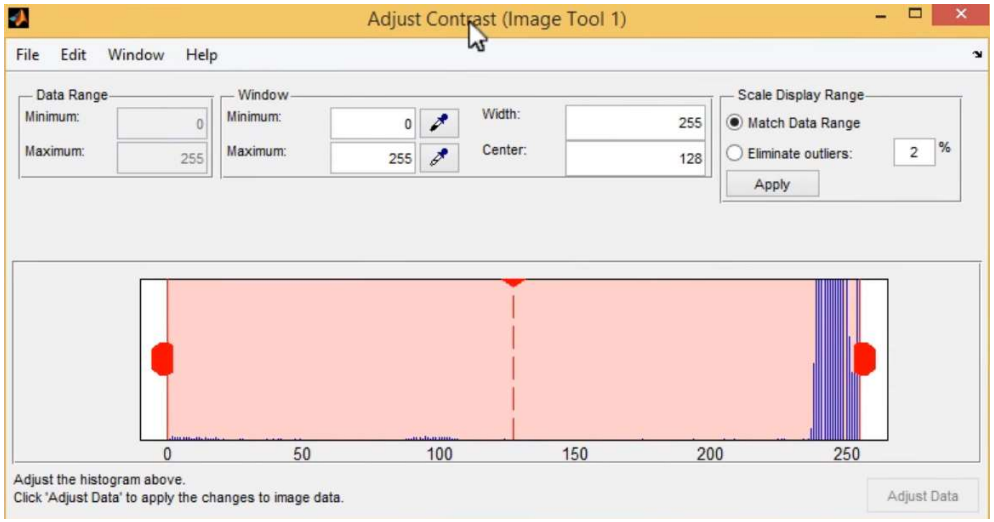


A

```
>> imtool(im)
```



B

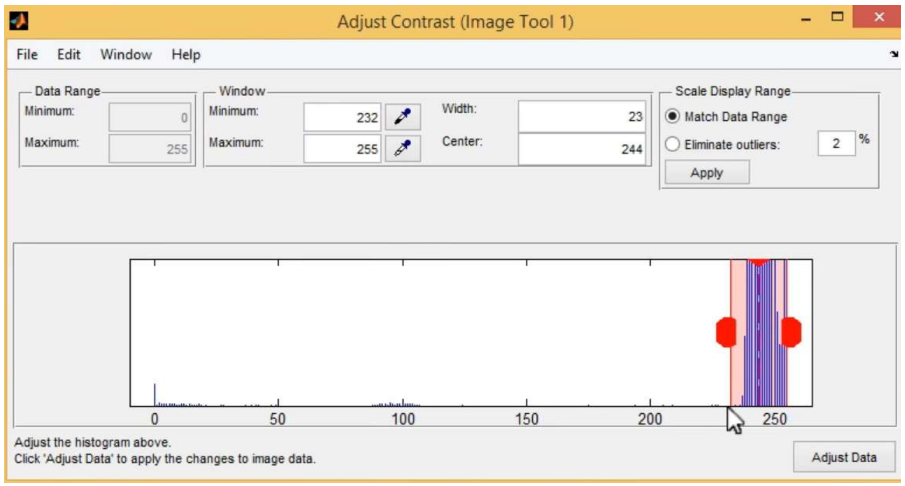


C

- Here is an example of an image that is ***way too washed out***, **A**. Using **imtool(im)**, we get **B**. The reason why the image is so washed-out is because maybe the scanner or camera settings were too high exposure. So, again, if we were to look at the histogram of this image, **C**, we should see that everything is up in the white band and there is hardly any dark intensities. That is, we have lots of peaks around **240**, and nothing elsewhere.

MATLAB's imtool

A



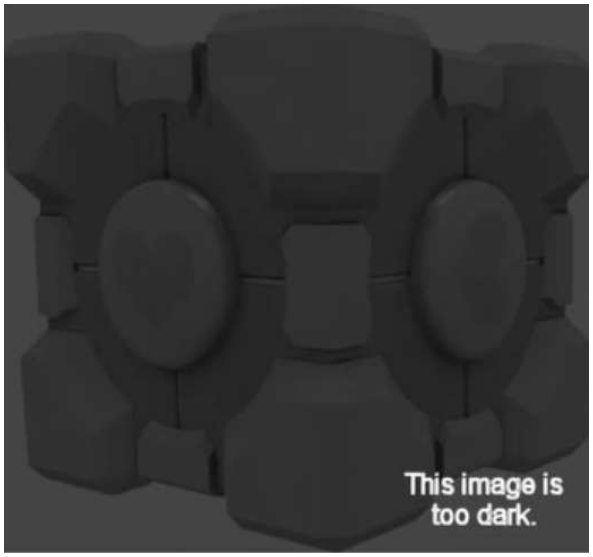
B

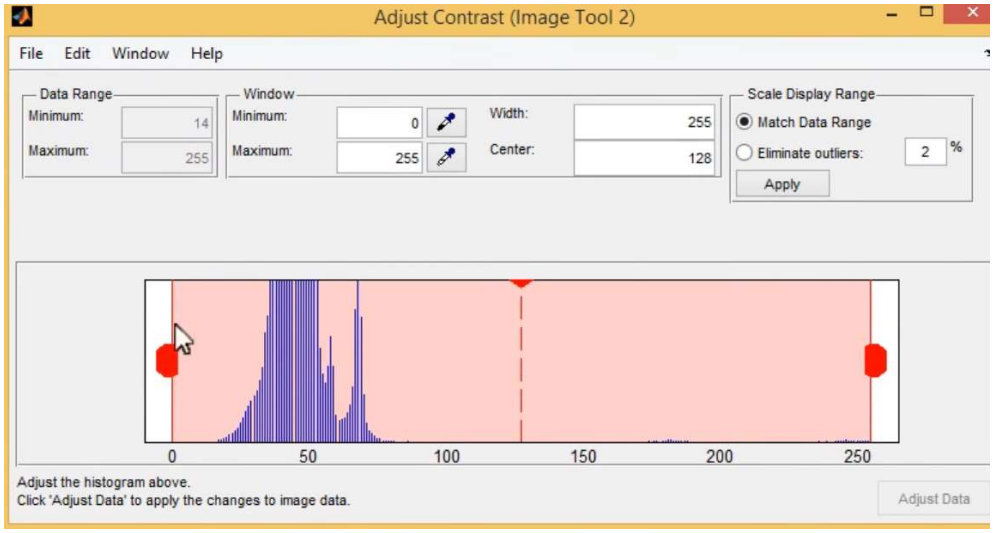
- If we were to move the slider over to the right, as shown in **A**, then suddenly we get more detail than we were able to visually perceive before. Note that we are not really getting any information that was not there before, but we are making it easier for ourselves to perceive.

MATLAB's imtool

```
>> im =imread('cube2.jpg');
>> imshow(im)
```

A



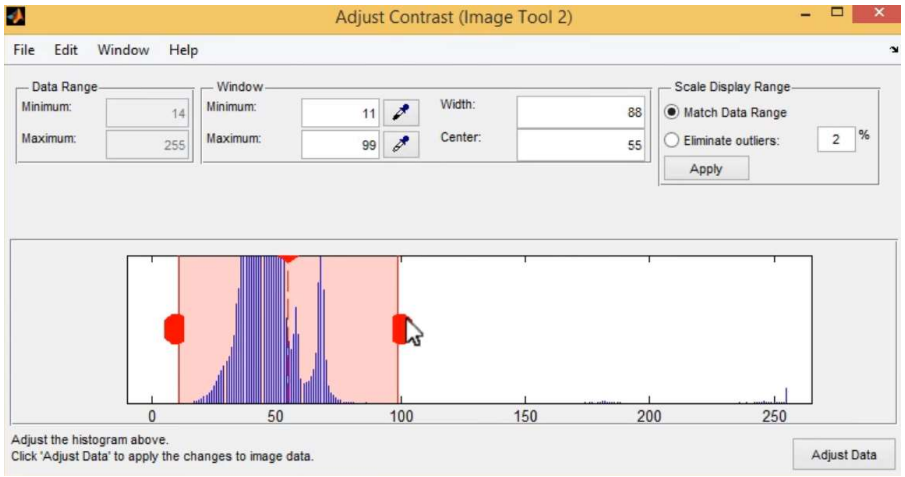


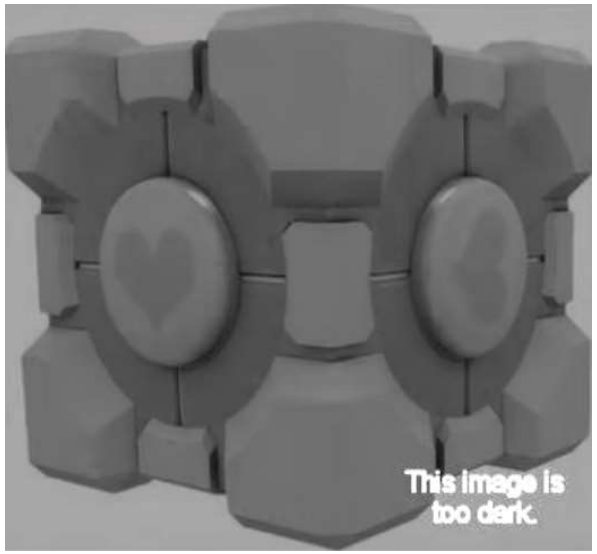
B

- In the same way, we can imagine that there is a ***reverse version*** where if we were to have a very dark image, **A**, with the histogram **B**, we could do the same thing. Here, image in **A** is a different file, **cube2.jpg**. We can clearly see that image **A** is too dark. If we look at the histogram, we can see that everything is kind of concentrated at the bottom (on the left side or the dark side).

MATLAB's imtool

A





B

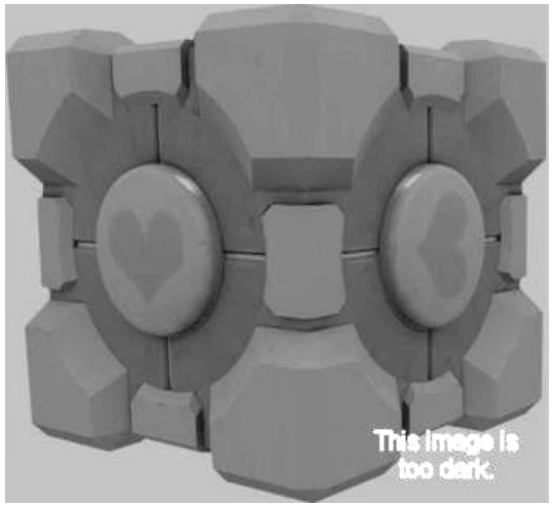
- We can drag the window around in **A**, to make our new image look like **B**.

MATLAB's imtool

$$F(D) = \frac{255}{H-L}(D-L) \quad (1)$$

```
>> close all
>> close all hidden
>> im =imread('cube2.jpg');
>> L = 11;
>> H = 85;
>> newim = 255/(H-L)*(double(im)-L);   (2)
>> imshow(newim,[0 255])
```

This is the dark image.



A

- Even though MATLAB provides us some tools for doing contrast stretching visually, we can also write our own commands to do this.
- Let us say, for example, we are going to apply equation **(1)** to the dark image we had before, i.e., to **cube2.jpg**. Based on this equation, we are going to say that the new image, **newim**, has a low range of **L = 11**, and high range of **H = 85**. Then, our new function should be **(2)**. The image is shown in **A**.

MATLAB's imtool

- **Sidenote:** If we have a vector, **v**, we can look at the minimum of the vector, using **min(v)**, as shown in box **A**. Now, let us say we have a matrix, **v = magic(3)**, shown in box **B**. Here, when we look at the minimum of the matrix, **min(v)**, what we get is the minimum in each column, as shown in box **C**, which is often not what we want. So, one thing we can do is use **v(:)**. Here, **v colon** is basically saying give us our original matrix, but turn it into a long skinny vector column by column. This is what we see in box **D**. Then we can take the minimum of that vector, **min(v(:))**, which is the minimum of the whole thing. This is the minimum of matrix **magic(3)**, shown in box **E** as **1**.

A

```
>> v = 1:5

v =

     1     2     3     4     5

>> min(v)

ans =

     1
```

B

```
>> v = magic(3)

v =

     8     1     6
     3     5     7
     4     9     2
```

C

```
>> min(v)

ans =

     3     1     2
```

D

```
>> v(:)

ans =

     8
     3
     4
     1
     5
     9
     6
     7
     2
```

E

```
>> min(v(:))

ans =

     1
```

MATLAB's imtool

```
>> min(newim(:))

ans =

    10.3378
```

A

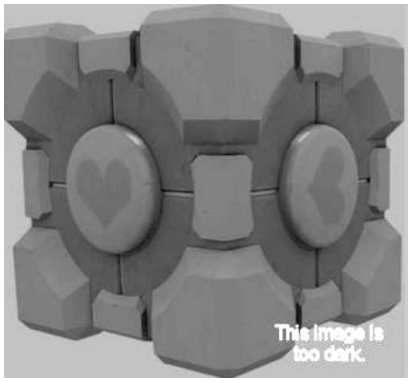
```
>> max(newim(:))

ans =

   840.8108
```

B

```
>> newim(newim > 255) = 255;
>> max(newim(:))

ans =

    255
```

C

```
>> imshow(newim,[])
```



D

```
>> imwrite(newim, 'newcube.png');
```

E

- Based on what we just discussed about minimum and maximum of a matrix (the **sidenote**), in our case, we can look at the minimum of the new image that we created, box **A**, which seems ok. But if we look at the maximum, it is **840.8108**, which is way above **255**, box **B**.
- So, when we want to do the **imshow** with the scaling, we have to make sure that we make slight change in mapping. What we could do is use the command in box **C**, which says for the new image, take all the pixels that are greater than **255** and map those to **255**. Now, the maximum is **255**. If we do **imshow** of this new image, we get exactly what we want. This is shown in box **D**.
- In box **E**, we used **imwrite** to write images out. We can choose whatever format we want. The **imwrite** command in MATLAB is used to write image data to a graphics file on our computer's disk. It is an essential tool for saving the results of our image processing operations or creating image files from within our MATLAB code. This MALTAB command is a versatile tool for saving the image data in various formats, allowing us to export our MATLAB image processing results or create image files for further use.

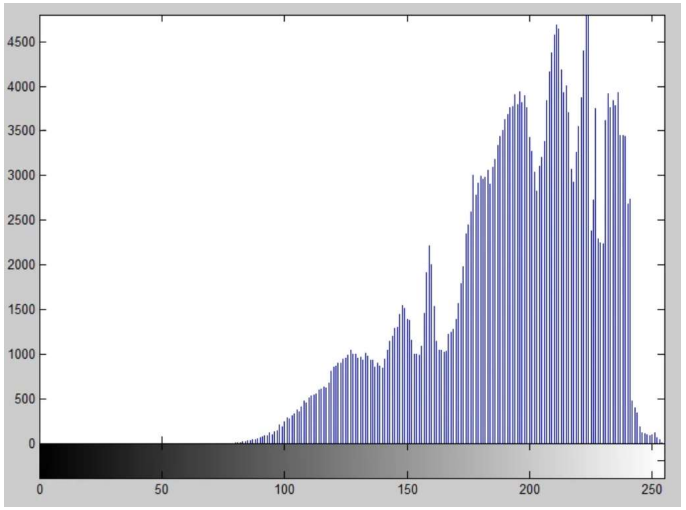
Histogram equalization

```
>> im = imread('room2.jpg');
>> imshow(im)
```

A

```
>> imhist(im)
```

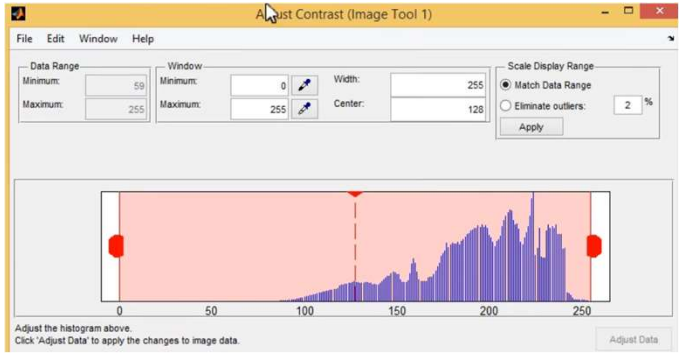
B



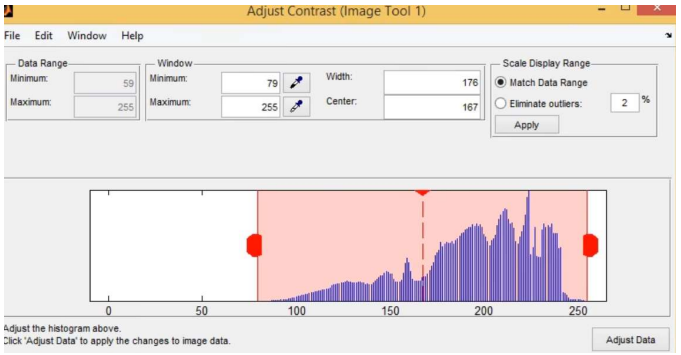
- In **A**, we have an image that has definitely got problems. It is washed out. If we look at the histogram of it, in **B**, we can see the problem is that the intensities are too much on the white end and that is what makes it look kind of crummy. We could argue that we could try to make that image better by ***jogging histogram around*** a little bit, i.e., using **imtool** and do contrast stretching.

Histogram equalization

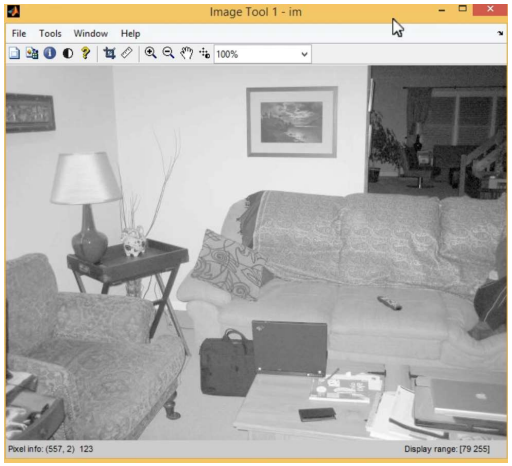
```
>> imtool(im)
```



A



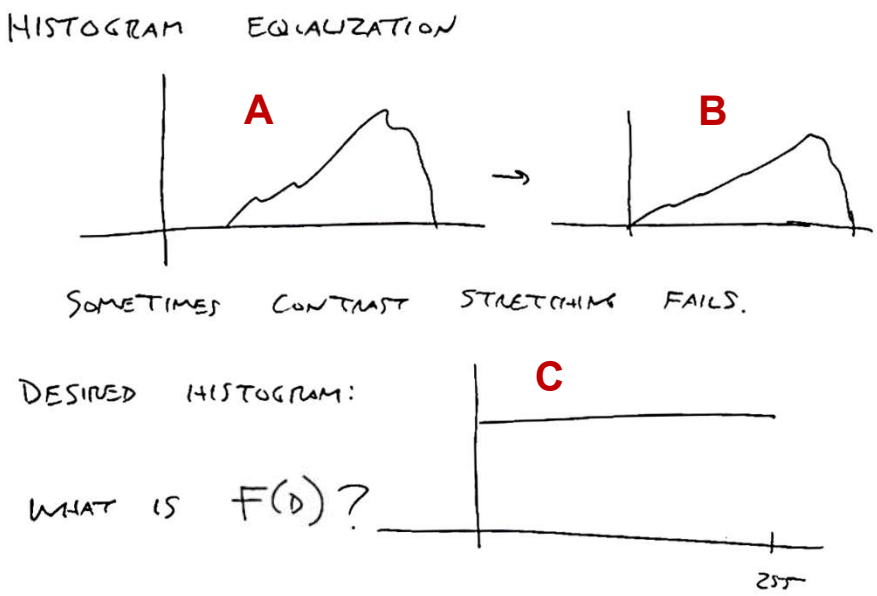
B



C

- In histogram **A**, as we predicted, we can see that the intensities are too much on the white end. What we could do is try to bring the left vertical line up to the right, as shown in **B**. As depicted in **C**, we see a little bit of improvement, but the image still looks a bit crummy or poor-quality. It looks as if the back of the room somehow seems too dark and the whole front of the room seems too bright.
- **Conclusion:** In this case, simple **contrast stretching** is not going to help us fully fix this problem with the image. What we really would like to do is ***redistribute the pixels*** in such a way that the lighter pixels proportionally take up less of the image intensities, and the darker pixels take up proportionally more.

Histogram equalization



- **Histogram Equalization:** In order to do a ***nonlinear mapping of intensity***, we can use a process which is called **histogram equalization**. The phenomenon we just discussed showed us that there are some cases when contrast stretching just does not seem to work. That is, if we were to stretch out **A** to **B**, we would still get something that does not look so good. So, **sometimes contrast stretching fails**.
- The idea here is that we can argue philosophically that it may be good to try to obtain an image where there is an equal amount of each of the gray values as possible. So, maybe what we like to do is try to ***rebalance*** **A**, so that the histogram of the output image is **flat**, as shown in **C**. The question is **what is the transformation, $\mathrm{F(D)}$, that will make that process happen?** The answer turns out to actually have to do with ***probabilities***.

Histogram equalization

Probability Density Function (PDF):

- Used for ***continuous*** **random variables**. In DIP, this often applies to continuous intensity levels in an image.
- Represents the **probability density** of a particular intensity value occurring in the image.
- The **area** under the PDF curve between two points represents the probability that a pixel's intensity falls within that specific range of values.
  - **Example:** The intensity of a pixel in a grayscale image can range from **0** (black) to **255** (white). The PDF can describe the probability of finding pixels with intensities between, say, **50** and **100** (shades of gray).

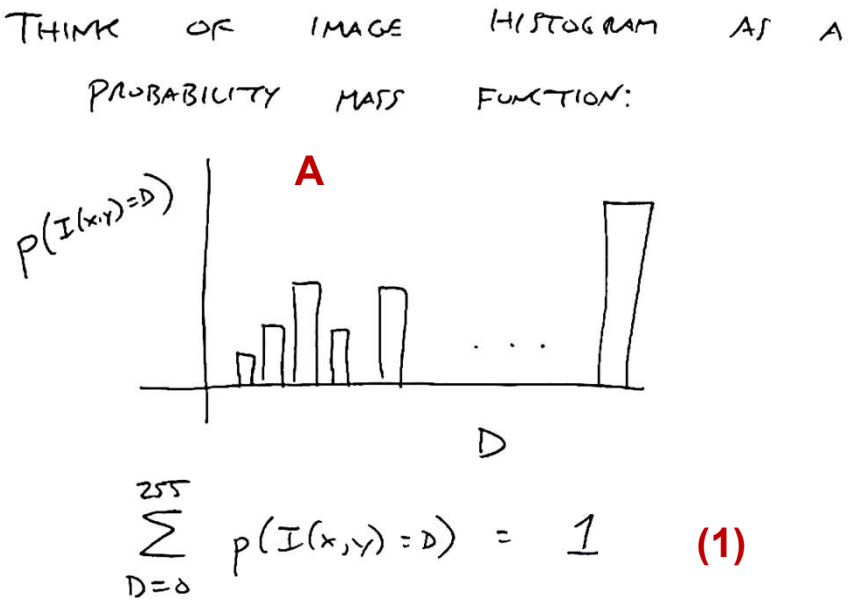
Probability Mass Function (PMF):

- Used for ***discrete*** **random variables**. In DIP, this applies to images with a finite number of possible intensity values.
- Represents the **probability** of a particular intensity value occurring in the image.
- The **height** of the PMF at a specific intensity value represents the probability that a pixel has that exact intensity.
  - **Example:** An image with binary pixel values (**0** for black, **1** for white). The PMF would show the probability of a pixel being **0** (black) or **1** (white).

Importance in DIP:

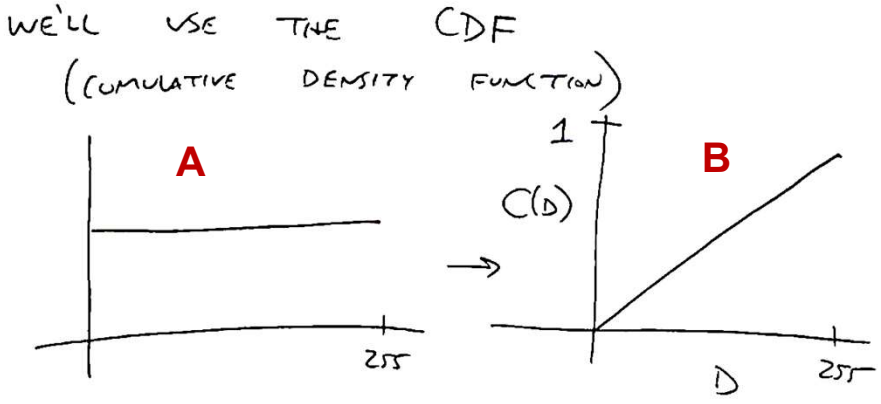
- Understanding PDFs and PMFs helps with various image processing tasks:
  - **Image Segmentation:** Identifying regions in an image based on intensity differences. By analyzing the PDF/PMF, we can determine suitable thresholds for segmentation.
  - **Noise Reduction:** Filtering techniques can be designed based on the distribution of noise in the image's PDF.
  - **Image Compression:** Statistical properties captured by PDFs/PMFs can be used for efficient image compression algorithms.

Histogram equalization



- **Probability Mass Function in DIP:** In the context of image processing, the key idea is to see that the histogram is actually similar to a probability mass function, as shown in **A**. The only difference is that the histogram is like a pixel count for every value of intensity. So, if we were to add up all the values in the histogram, we would get the total pixels in the image. But if we were to divide that picture by the total number of pixels in the image, that would mean that everything would sum to **1**, like a PMF, and as presented in **(1)**. We use PMF because we have a discrete set of intensities, from **0** to **255**. That is like saying **what is the probability that the pixel image has this intensity?**
- So, the idea is to think of the images pixels histogram as a probability mass function, as shown in **A**. Here, the **x**-axis is like $\mathbf{D}$ and the **y**-axis is like the probability that the image intensity equals to $\mathbf{D}$ or $\mathbf{p(I(x,y) = D)}$. The PMF is actually like a **bar graph** for each of the $\mathbf{D}$**'s**, and this means that the sum of these probabilities, from $\mathbf{D}$ equals **0** to **255**, has to equal **1**.

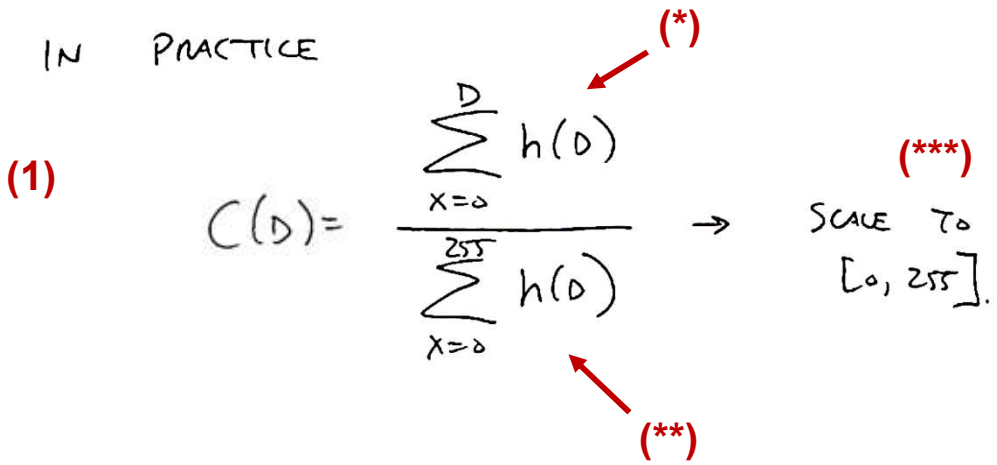
Histogram equalization



$$C(D) = P(I(x,y) \leq D) \quad (1)$$

- **Cumulative Density Function (CDF):** In reality, what we need here in this example is not the PDF or PMF, but the **CDF**, which is basically the **cumulative density function**. CDF represents the probability that a random variable will be less than or equal to a specific value.
- **What is the CDF that corresponds to the uniform PDF shown in A?** It is like saying we integrate from **0** to **255**. Integrating the uniform function shown in **A** gives us a straight line, shown in **B**. The CDF is defined as the probability that the image intensity is less than or equal to $\mathbf{D}$, as shown by **(1)**. In **B**, $\mathbf{C(D)}$ is **0** at **0**, and **1** at **255**. In this case, in **B**, since we want things to be uniformly distributed, we have this nice linear function with constant slope that takes us from $\mathbf{C(D)}$ value of **0** to **1**.
- **Conclusion:** We start by our histogram, then we compute the CDF. This process is called **histogram equalization**.

Histogram equalization



- **How do we compute the CDF, C(D), in practice?** In practice, this function, $C(D)$, shown in **(1)**, is like the sum of the values of a histogram up to D, **(*)**, over all values of the histogram, **(**)**, and then scaled to **0** to **255**, **(***)**. Simply put, what we are doing is adding up all the values up to D, since we are in pixel D, and then dividing it by all values of the histogram.

Histogram equalization

The code J = histeq(im):

- The code **J = histeq(im)** performs **histogram equalization** on an image **im** in MATLAB. As explained before, histogram equalization is a common image processing technique used to enhance the contrast of an image by redistributing the intensity values of its pixels. Here is a breakdown of what it does:

  1. Histogram Analysis:
  - **histeq** first analyzes the histogram of the input image **im**. The histogram represents the distribution of pixel intensities across the image. For example, it shows how many pixels have a specific intensity value (brightness level).

  2. Probability Distribution Transformation:
  - Based on the histogram analysis, **histeq** calculates a new probability distribution for the image. This new distribution aims to create a more uniform spread of pixel intensities across the available range (typically **0** to **255** for **8**-bit grayscale images).

  3. Intensity Value Mapping:
  - Using the calculated probability distribution, **histeq** maps each original intensity value in the image **im** to a new intensity value in the output image **J**. The primary goal of histogram equalization is to enhance the overall contrast of the image by redistributing the intensity values so that they span the entire range of possible values more evenly.

Benefits of Histogram Equalization:

- **Enhanced Contrast:** By spreading out the intensity distribution, histogram equalization improves the visibility of details in low-contrast images.
- **Improved Feature Extraction:** This can be beneficial for subsequent image processing tasks like segmentation or object recognition where clear distinction between features is crucial.

Histogram equalization

```
>> J = histeq(im);
>> imshow(J)
```

- Let us see what would happen if we applied **histeq** to our image, **im**, i.e., **J = histeq(im)**. This command basically just says equalize the histogram of image **A**. We do not need to say anything else. We give it only this one input, **im**, and it will do the **equalized output**.
- Now, our output image, **B**, looks a little bit better, but we could argue that there is some problems still. That is, there is still some regions that seem too dark. But, if we compare it to the original image, **A**, we can see that, for example, there is definitely a lot more detail in the sofa cushions and stuff like that than there was before. When we did the contrast enhancement (contrast stretching), we did not really get this kind of level of detail.

A

B

Histogram equalization

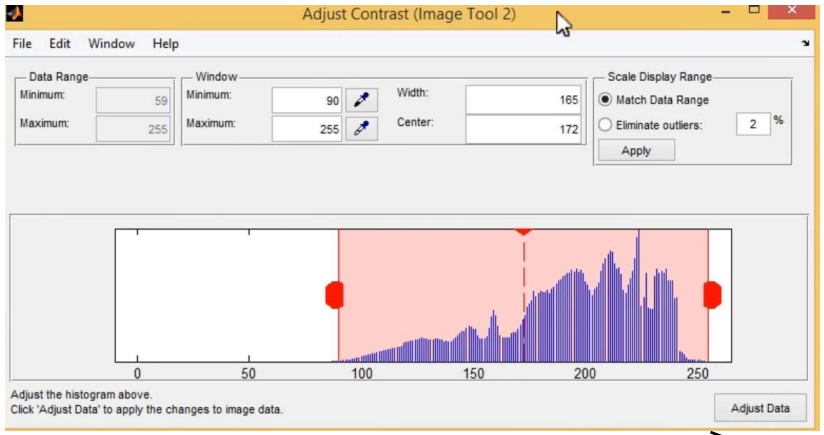
```
>> imtool(im)
```

- If we were to do this using **imtool** of **im**, i.e., contrast stretching, **A**, it would have been tough for us to get the same look in terms of details in the sofa cushions, **B**. So, **C** is better in that respect. A compromise is having the dark region in the back, **(*)**, but generally, it looks pretty good.

Output using contrast stretching (or contrast enhancement).

Equalized output using histeq.

A



B **C** **(*)**

Histogram equalization

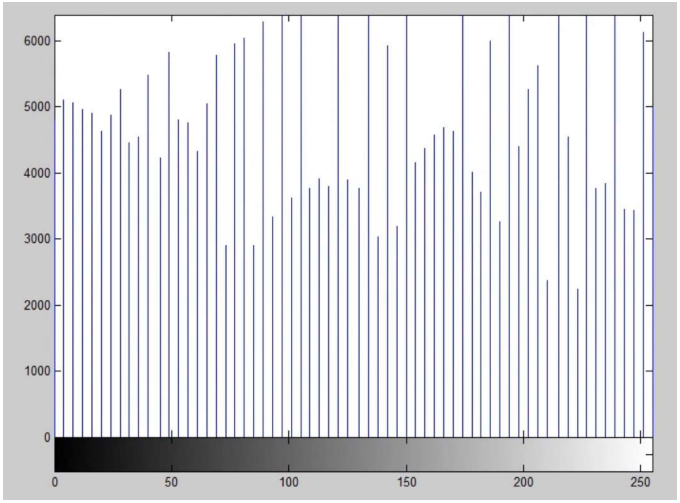
- Equalized output image using **histeq** command is shown in **A**. If we were to look at the histogram of this new image, **B**, we would see that it is definitely ***approximately uniform***. Here, we have taken all the old intensities that we had, and we have spread them out on the **0** to **255** axis.

Equalized output image using histeq command.

A

```
>> imhist(J)
```

Histogram of equalized output.



B

Bin in an image histogram

Bin in an Image Histogram:

- In an image histogram, assuming the **x**-axis ranges from **0** to **255** (representing intensity values for grayscale images) and the **y**-axis shows the number of pixels, a **bin** represents a ***specific interval*** on the **x**-axis and the corresponding count of pixels on the **y**-axis that fall within that interval. Here is a breakdown:

1. **Dividing the Intensity Range:**
   - The entire intensity range (**0** to **255**) is divided into ***a set of bins***. The number of bins can be chosen based on the desired level of detail in the histogram.
2. **Bin Width:**
   - Each bin has a specific **width** that determines the range of intensity values it can encompass.
   - For example, if there are **25** bins, each bin would have a width of **255/25 = 10** (assuming equal width for all bins). This means a bin would represent the intensity range from **0** to **9**, the next bin from **10** to **19**, and so on.
3. **Pixel Counting:**
   - Each pixel in the image is assigned to a bin based on its intensity value.
   - The program creating the histogram counts the number of pixels that fall within each bin's intensity range.
4. **Visualization:**
   - The histogram is typically displayed as a **bar chart**.
   - The **x**-axis represents the intensity range, with ***each bar*** corresponding to ***a specific bin***.
   - The **y**-axis shows the count of pixels.
   - The height of each bar represents the number of pixels in the image that have intensity values within the corresponding bin's range.

Bin in an image histogram

Example:

- Imagine a histogram with **10** bins.
- If one bin shows a height of **500** on the **y**-axis, it means there are **500** pixels in the image with intensity values that fall within that specific bin's range (e.g., from **120** to **129** if each bin has a width of **10**).

Importance of Bins:

- Bins provide a way to **group similar intensity values** together.
- By analyzing the heights of bars across different bins, you can understand how the pixels in the image are ***distributed*** across the intensity range.
- High bars indicate areas with many pixels of a similar intensity (e.g., a dark region with many low-intensity pixels).
- Low bars indicate areas with fewer pixels in that intensity range.

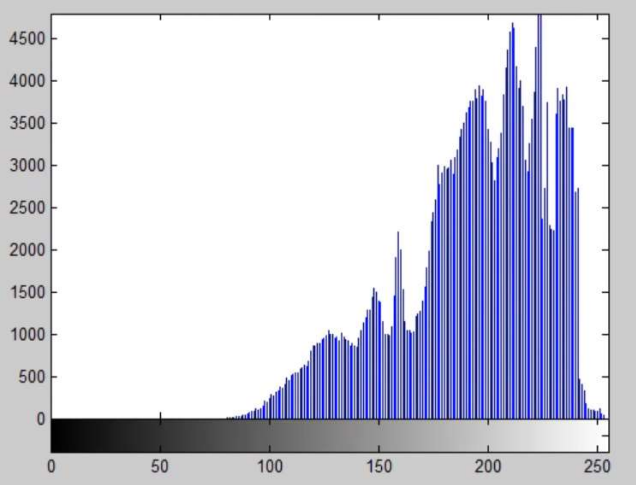
Conclusion: Bins in an image histogram with an **x**-axis of **0** to **255** (intensity) and a **y**-axis of pixel count, represent specific intensity ranges and the number of pixels in the image that fall within those ranges. This information helps visualize the distribution of pixel intensities and make informed decisions for image processing tasks.

Bin in an image histogram

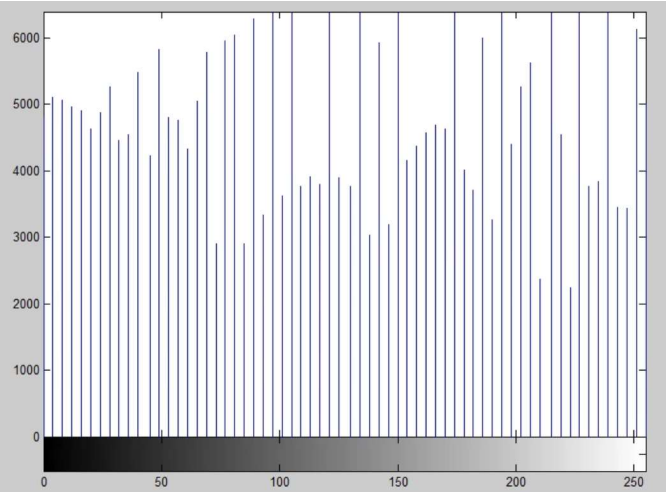
```
>> imhist(im)
```

- Let us look at the histogram of the ***original image***, **A**, and compare it to histogram of the ***equalized output***, **B**. Now, lets us compare the pixel counts, before and after. What we can see is that now in **B** every **bin** has roughly say **5,000** pixels in it. To make this first bin, what we are doing is combining the first **5,000** pixels moving from left to right. That is, to make this first bin, we are ***accumulating*** a whole bunch of these original dark pixels and they are all getting mapped to this one color black. One of the reasons why this method does not quite work all the time is that there are some bars that we cannot split up. This is an operation that applies equally to all the image pixels so that they would have the same intensity. In practice, it cannot split big bars efficiently, but it can efficiently combine smaller bars and that is why the output does not look entirely uniform. However, visually, things will look a lot better than they used to.

A



B



Histogram specification

Histogram Specification or Histogram Matching:

- **Histogram specification**, also known as **histogram matching**, is an image processing technique used to modify the intensity distribution of an image to match a ***desired target distribution***. It essentially transforms the image's histogram to resemble another specified histogram. Here is a breakdown of the concept:

Why Use Histogram Specification?

- **Contrast Enhancement:** By manipulating the histogram, we can improve the contrast in an image, making details more visible.
- **Normalization:** We can normalize the intensity distribution of an image to have a specific characteristic, which can be helpful for tasks like image comparison or analysis.
- **Matching Images:** We can adjust an image's histogram to match the histogram of another image, potentially making them more visually compatible for tasks like ***combining images from different sources.***

How Does It Work?

1. **Histogram Analysis:**
   - The initial step involves calculating the histogram of both the **source image** (the image we want to modify) and the **target histogram** (the desired intensity distribution).
2. **Cumulative Distribution Function (CDF):**
   - The CDF of both the source image's histogram and the target histogram are calculated. The CDF represents the probability of a pixel intensity being less than or equal to a specific value.

Histogram specification

3. **Intensity Value Mapping:**
   - A ***transformation function*** is created by mapping each intensity value in the source image's CDF to a new intensity value in the target histogram's CDF.
   - This mapping essentially tells us how to adjust the source image's pixel intensities to achieve the target distribution.
4. **Image Transformation:**
   - Finally, each pixel in the source image is assigned a new intensity value based on the calculated transformation function. This results in a new image with a histogram that closely resembles the target histogram.

Benefits:

- Improved visual quality of the image by enhancing contrast or achieving a desired appearance.
- Enables comparison or analysis of images with similar intensity distributions.

Drawbacks:

- Potential loss of information during the intensity value mapping process.
- Introduction of artifacts in the image if the transformation is not carefully designed.

Conclusion: In essence, histogram specification allows us to manipulate an image's appearance by controlling its intensity distribution, potentially improving its visual appeal or suitability for specific tasks.

Histogram specification

How Histogram Equalization is Related to Histogram Matching?

- Both techniques use the **cumulative distribution function (CDF)** of the image's histogram for the transformation.
- Histogram equalization can be seen as a **special case** of histogram matching where the target histogram is uniform. In this sense, histogram equalization is a type of histogram specification where the desired distribution is flat (uniform).
- **Difference**: Histogram equalization aims to spread the intensity values evenly, while histogram matching aims to transform the intensity values to achieve a specific target distribution.

Example Scenario:

- Suppose you have a low-contrast X-ray image. Using **histogram equalization**, you can enhance the contrast by redistributing the intensity values across the entire range. However, this may sometimes lead to unnatural-looking results.
- Instead, if you want the X-ray image ***to look similar to another well-contrasted reference X-ray image***, you would use **histogram matching**. This would adjust the intensity distribution of the input X-ray to match that of the reference X-ray, potentially providing a more consistent appearance.

Summary:

- **Histogram Equalization** is an automatic process for general contrast enhancement but may not always give the desired look.
- **Histogram Matching** offers more control and can produce a specific desired appearance by matching the input histogram to a target histogram.

Gamma correction

"GAMMA CORRECTION"

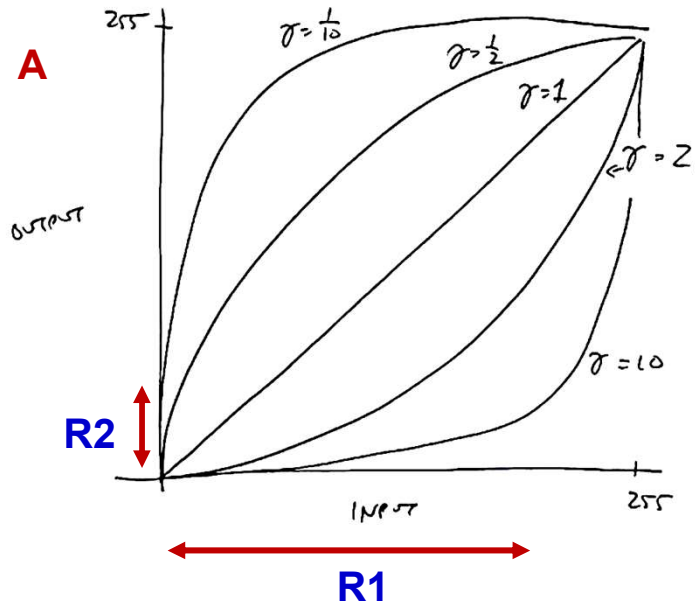
PROBLEM: EVERY DISPLAY DEVICE HAS A DIFFERENT NONLINEAR RELATIONSHIP B/W PIXEL INPUT INTENSITY AND DISPLAY OUTPUT LUMINANCE.

THE RELATIONSHIP FOR REAL DEVICES IS OFTEN MODELED AS A POWER FUNCTION

$$F(D) = D^{\gamma} \quad (1)$$

- **Gamma Correction:** Gamma correction is something that is maybe less of an issue than that it used to be. The problem is that every monitor or every display device has a different nonlinear relationship between pixel input intensity and display output luminance. This was a big problem back in the olden days. In the rooms that had projectors, instead of big screen TVs, this was sometimes a problem. This is because the images would look great on our laptop displays screens, but on the projectors, they would look like totally washed out. The problem was that the mapping of the color pixels to our laptop screen was a different function than the mapping of the color pixels that got sent to the projector.
- This required us to basically pre-compensate the images on our laptops screen so that when they were projected onto the projector, they looked good to the audience and sometimes they looked awful on our laptop screens! This problem is called **gamma correction**. The idea is that the relationship for real devices is often modeled as a **power function of gamma**, shown in **(1)**.

Gamma correction



SOLUTION: IF WE KNOW THE γ OF THE DISPLAY DEVICE, PRE-COMPENSATE INTENSITIES:

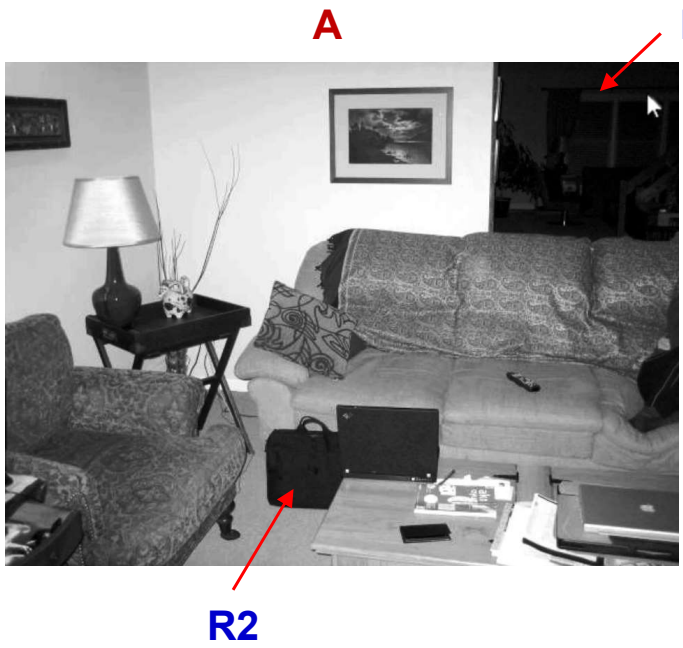
(1) $\left(D^{1/\gamma}\right)^{\gamma} = D$

SEND TO DISPLAY

- Let us see how these power functions look like, **A**. Ideally, what we want is $\boldsymbol{\gamma = 1}$, i.e., there is to be no distortion between the input and the output. That is, we want a linear relationship between what comes in and what is getting shown on the projector screen. That was not typically true and instead what we got was some non-linear function.
- As an example, for gamma $\boldsymbol{\gamma = 10}$, the whole **gamut** of input intensity, from roughly **0** to **150**, **R1**, is getting mapped into the tiny dark region of the output, **R2**. Here, **gamut**, in the context of Digital Image Processing, refers to the range of colors that a specific device can represent or reproduce. It is essentially the color space of the device, defining the spectrum of colors it can handle.
- The solution is if we know the gamma of the display device, then we ***pre-compensate intensities***. So, if the gamma is known, then we would send $\mathbf{D^{1/\gamma}}$ to the display. Then this thing is going to get corrupted in some sense by the gamma display to give us the pixel that we originally wanted, as shown in **(1)**. That is, it will give us $\mathbf{D}$. We are kind of fooling the display to thinking that we are sending it different colors that will get pre-compensated.

Introduction to spatial filters

POINT OPERATIONS AFFECT EVERY PIXEL WITH THE SAME INTENSITY IN THE SAME WAY.



MANY IMAGE PROCESSING OPERATIONS ARE MORE LOCAL.

e.g., SPATIAL FILTER.

- So far, we have talked about **point operations**, which are a type of ***global operations***. These operations affect every pixel with the same grayscale value, the same way. In some cases, like the basic contrast enhancement, that is good. But, in other cases, it is not so good. ***Global operations*** modify the entire image, while ***local operations*** focus on specific regions based on pixel relationships.
- For example, in image **A**, in the equalized output, there were some dark pixels in **R1** that we do not really want to treat the same way as the dark pixels in **R2**. In some sense, we would like to do more ***local processing of the image***. Maybe we can use some sort of local contrast enhancement or local histogram of equalization. That is where we are going next. We will next discuss ***local spatial processing***. So, many image processing operations are more local. One possibility is called **spatial filter**. These kinds of spatial filters are analogous to the kind of FIR filters that we learned in DSP. We talked about FIR filters in the world of signal processing in DSP, like audio processing.

Introduction to spatial filters

e.g. FILTER AN IMAGE TO REPLACE EACH PIXEL BY THE AVERAGE OF ITS NEIGHBORS

$$J(x,y) = \frac{1}{9}\sum_{i=-1}^{1}\sum_{j=-1}^{1} I(x-i, y-j) \quad (1)$$

$$= \frac{1}{9}\left(I(x,y) + I(x-1,y) + I(x+1,y) + \dots \right) \quad (2)$$

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad (3)$$

- **Spatial Filters:** In digital image processing, spatial filters are techniques used to manipulate images by modifying specific regions of pixels directly. Spatial filters help improve image quality, enhance features, and reduce noise.
- First, we want to talk about **spatial filtering** in a more natural way. So, for example, what we could do is ***we could filter an image to replace each pixel by the average of its neighbors***. So, our output image in **(x,y)**, shown by **J(x,y)**, is the average of the three pixels, the left/right pixels, **i = -1, 0, 1**, and the three pixels, the up/down pixels, **j = -1, 0, 1**, as shown in **(1)**. This is like saying we take the middle pixel and the pixel to the left and the pixel to the right, etc., as shown in **(2)**, and we take these **9** pixels and we average them. Here, each of these pixels gets a weight of **1/9**. The final spatial filter looks like **(3)**.

Introduction to spatial filters

```
>> close all hidden
>> im = rgb2gray(imread('conan.jpg'));
>> imshow(im)
```

A

```
>> f = 1/9*ones(3)    (1)

f =

    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111
    0.1111    0.1111    0.1111

>> out = imfilter(im, f);    (2)
```

- Let us see what a spatial filter does to the image, 'conan.jpg', **A**. First, let us define a filter, **(1)**. The filter is **1/9** times an array of all **1**'s, **B**. Next, we are going to use a command called **imfilter(im, f)**, as represented by **(2)**.
- Generally, the **imfilter** function, **B = imfilter(A, h)** in MATLAB, performs **multidimensional filtering** of an image or array **A** using a filter **h**. Here is a breakdown of its components:
  - **B:** This is the output variable that stores the filtered version of the input array **A**.
  - **A:** This is the input array that can be a multidimensional array of any class (numeric or logical) representing the image or data we want to filter.
  - **h:** This is the filter, which can be a multidimensional array representing the ***filter kernel*** or ***mask***. It defines how the surrounding pixels of **A** will be weighted and combined during the filtering process.

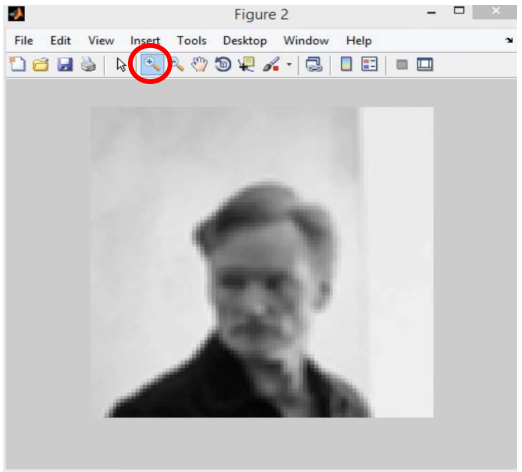
Introduction to spatial filters

```
>> out = imfilter(im, f);
>> figure
>> imshow(out)
```

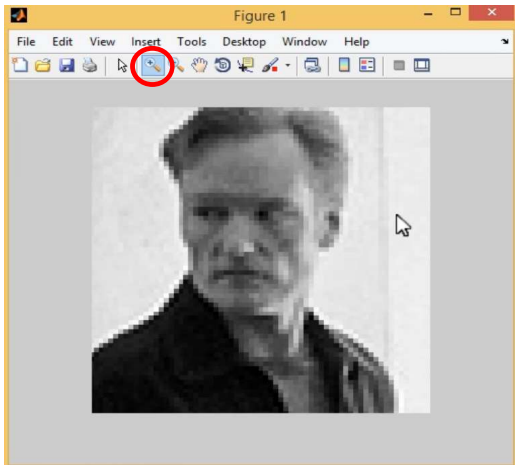
A1

B1

Blurrier A2



Sharper B2



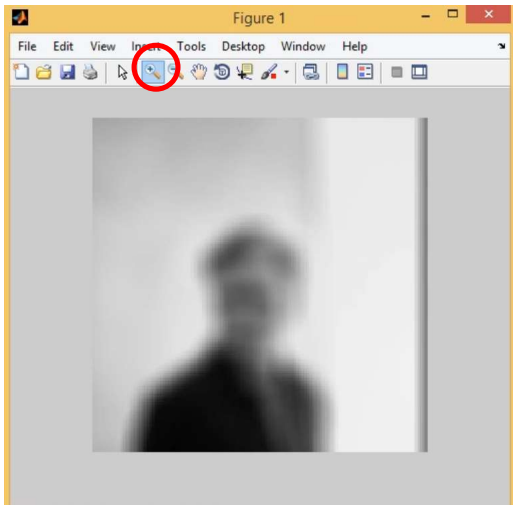
- Comparing **A1** (the filtered image) with **B1** (the original image), it is a bit hard to tell the difference. But, when we zoom in on part of the image, such as Conan himself, in **A2** and **B2**, we should see that there is some difference in the amount of detail. The ***old*** Conan, **B2**, is sharper and the ***new*** Conan, **A2**, is blurrier! In **A1** or **A2**, what we are really doing is ***averaging adjacent pixels***.

Introduction to spatial filters

```
>> f = 1/121*ones(11);    (1)
>> out = imfilter(im, f);
>> imshow(out)
```

A1

A2



- Let us make the **averaging mask** (i.e., our **spatial filter**) even larger. For example, if we were to define the filter as being the average of say the pixels in an **11** by **11** neighborhood, **(1)**, now it is even blurrier! This is shown in **A1**, and also in its zoomed-in version, **A2**. Now, it is definitely easier to see that the effect of this larger filter is to make things ***blurrier***.
- Some might wonder why we wanted our image to become blurrier!? One of the reasons we might want to do this is that maybe we want to ***get rid of noise***. Here, we may have a noisy image that comes from a scanner or camera, for example, that is looking at an image in low-light conditions. In that case, we may get this kind of ***Gaussian spectral noise*** that we want to try and smooth out.

Introduction to spatial filters

$$J(x,y) = \frac{1}{9}\sum_{i=-1}^{1}\sum_{j=-1}^{1} I(x-i, y-j) \quad (1)$$

$$= \frac{1}{9}\left(I(x,y) + I(x-1,y) + I(x+1,y) + \dots\right)$$

(2) $\frac{1}{9}$ $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ $= h(x,y)$

$$J[x,y] = \sum_i \sum_j h[i,j]\, I[x-i, y-j] \quad (3)$$

$$= h * I \quad (4)$$ CONVOLUTION OF FILTER AND IMAGE.

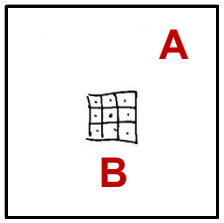


Diagram C

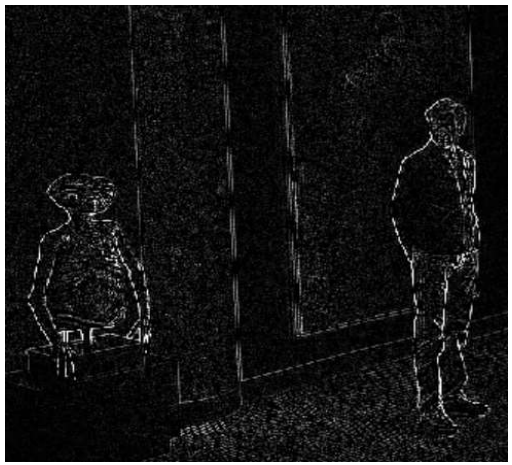
- In the case of **J(x,y)**, (1), let us use **h(x,y)** as the notation for the filter, (2). Now, let us see what equation (1) really means. We can re-write this equation in a slightly different way, (3), which is saying that our output, **J[x,y]**, is the sum over **i** and sum over **j** of the filter. Now, it looks like **convolution**, (4), and that is exactly what is happening here. ***This is like a 2D convolution of the 2D spatial filter with the whole image.***
- The thing that was really important about convolution in one-dimension in DSP was this notion that we were taking the filter and we were flipping it and running it across the image. In DIP, we do not have to worry about that too much. It is actually easier to think about convolution here. As shown in **Diagram C**, we have the whole image shown as **A**, and we are thinking about the filter, **B**, being placed over a pixel on the image at a certain center pixel, and then applying some combination of these values around that pixel, i.e., the **mask** or the **filter matrix**, to get our result. Here, we do not really have to worry about the notion that this mask is flipped around. A lot of times, these filters are already symmetric in a way. So, it does not really matter that we are flipping them. That is, flipping a symmetric filter does not change its effect on the image.
- **Conclusion:** All the above is saying is that what is happening under the hood is exactly convolution, which again means that we can bring in tools of Fourier analysis that will help us do certain kinds of frequency domain filtering. This makes sense in the same way that convolution had a dual frequency domain property in digital signal processing.

Introduction to edge detection

-1	-1	-1
-1	8	-1
-1	-1	-1

(1)

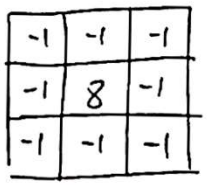
```
>> f = [-1 -1 -1; -1 8 -1; -1 -1 -1];  (2)

f =

    -1    -1    -1
    -1     8    -1
    -1    -1    -1

>> out = imfilter(im, f);   (3)
>> imshow(out,[]) (4)
```



A

- Let us consider the filter shown by **(1)**. This filter is saying, take the pixel in the middle of the original image, multiply it by **8**, and then subtract all the neighbors. **What happens if this filter is centered over a block that is *roughly uniform color*?** The answer is not much is going to happen! This is because, in this case, the middle pixel has the same color as the other pixels. So, we are going to get a filter response of zero, i.e., no change in the image. However, let us say the middle pixel in the image is much brighter or much darker than one of its neighbors. In that case, we are going to get some heavy response to that filter, either a big positive response or a big negative response.
- Suppose that our filter is defined in MATLAB as **(2)**. If we were to filter **conan.jpg**, **(3)**, and show its output using **(4)**, we get **A**. Here, we use **imshow(out,[])**, which means we are going to scale it, **[]**, from black to white. Image **A** is something that is actually a lot different than the original image. This new image looks almost like an ***outline of the objects*** in the scene. In other words, we are finding ***edges*** in the scene. So, if we wanted to find where the edges were, maybe this would be the first way to do that.

Introduction to edge detection



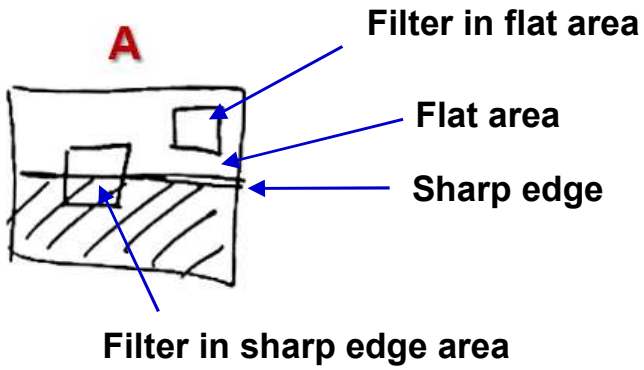
(1) IS AN EDGE-DETECTING SPATIAL FILTER

- **Why would we want to find edges?** The answer is **Image Segmentation**. Maybe we want to do a later process of **segmenting** the objects from the scene. For example, we wanted to find the person in the scene or find the painting in the scene.
- **Conclusion:** In summary, this output image is a very different kind of output than contrast stretching or histogram equalization. This is because certainly the pixels in the image, even if they have the same intensity, are getting treated very differently due to maybe having different neighbors. So, if we have a pixel that has an intensity of **100**, in the middle of a whole sea of other **100**'s, then the output is going to be basically **0**. Whereas if we have a pixel that has an intensity **100**, in the middle of a sea of **0**'s, then the output is going to be very different. This is a ***spatially selective filter*** and can be used as an **edge-detecting spatial filter**, shown in **(1)**.

Introduction to edge detection

-1	-2	-1
0	0	0
1	2	1

(1)



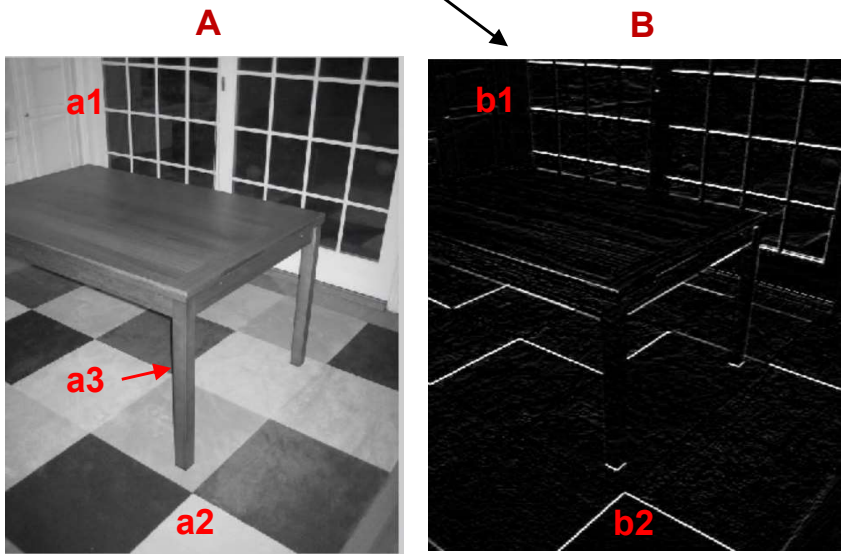
- What would happen if we had something like **(1)**? As it turns out, it is going to respond to edges in the image that are ***sharp horizontal edges***. As shown in **A**, this is because if we place this filter over an area, then there is going to be a big difference between the filter response that is placed on the **sharp edge** and the filter response that is placed on the **flat area**. Note that a flat area is an area in which there is no intensity variation, i.e., an area with no significant edges. If we place the filter in a flat region, then the values in **(1)**, are going to add up to **0**.

Introduction to edge detection

-1	-2	-1
0	0	0
1	2	1

f

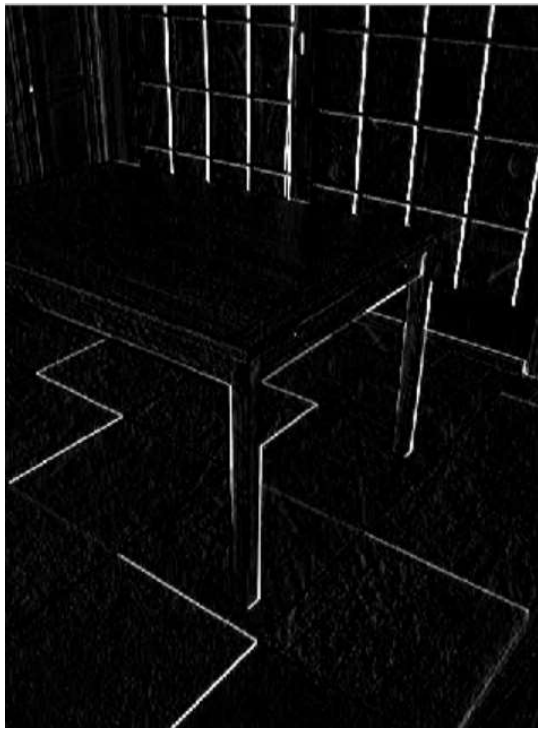
```
>> f = [-1 -2 -1; 0 0 0 ; 1 2 1];
>> im = imread('floor1.jpg');
>> imshow(im)
>> out = imfilter(im,f);
>> figure
>> imshow(out)
```



- Here is an example of a room, **A**, that has strong edges in all directions. If we were to apply our filter **f** to this image, it is going to respond preferentially to images that are roughly horizontal, as shown in **B**.
- For example, in **A**, some of these roughly horizontal edges are the bars of the windows that are going across, **a1**, and the sharp edges between the two floor carpet tiles, **a2**. They are clearly visible in **B** as **b1** and **b2**. We see that there is a strong response in these edges.
- However, the strong edges that are roughly vertical do not get picked up. For example, there is definitely some edges on the table leg, **a3**, that do not respond at all because these edges are not going the right way, i.e., they are roughly vertically oriented.

Introduction to edge detection

```
>> out = imfilter(im,f'); (1)
>> imshow(out)
```



A

- We can design some filters that are of preferential for different edge orientations, such as for the edges on the table leg. In image **A**, we can use the **transpose** of **f**, which is represented by **f'** in **(1)**. If we were to filter our image the other way around, i.e., with the transpose of our filter, now we should see the edges of the table legs and the edges of the window going the other way, as shown in **A**.

Introduction to edge detection

- We can talk about how we would design filters to do certain tasks in a different way. That is, unlike taking the filter block and moving it around the image, we can look at this process like the equivalent of it in the time-domain of an audio signal using an FIR filter. The method we just discussed is much more immediately graspable about why we want to filter an image and what filters will be good for certain tasks. Sometimes, filtering is a lot more intuitive in the image processing world than it is in the signal processing world. Here, we have an immediate grasp of what we need to do.
- Just to wrap up, this lecture focused on changing the intensity or colors of pixels in an image. Other related topics include altering the shapes of pixels. So far, we have not discussed topics involving changes to the fundamental size or shape of the image. One common task in Digital Image Processing (DIP) is handling images where alignment is not perfect. For example, when capturing a scene or obtaining an image from an MRI, ultrasound, or CT scan, we might notice that a horizontal line is not perfectly horizontal. In such cases, we need to rotate the image to correct its orientation. Additional operations include **scaling**, **shifting**, **rotating**, and performing more complex transformations to arrange images as desired. These processes are collectively known as **geometric operations**.

End of Lecture 4