

## Assignment 02

Course: CSD331- Digital Image Processing

Due Date: October 25, 2019

### Intensity Transformations and Spatial Filtering

**Note:** There are three questions in this assignment as given below. A detailed description is provided on the topics related to these assignment questions. You are suggested, first go through the detailed description and then solve the assignment questions.

#### **Question No. 1 [4 marks]**

Complete the [intrans function](#) by writing the contrast stretching transformation case. Write a script that uses it to reproduce the 6 contrast stretched pictures in [section 1.4](#).

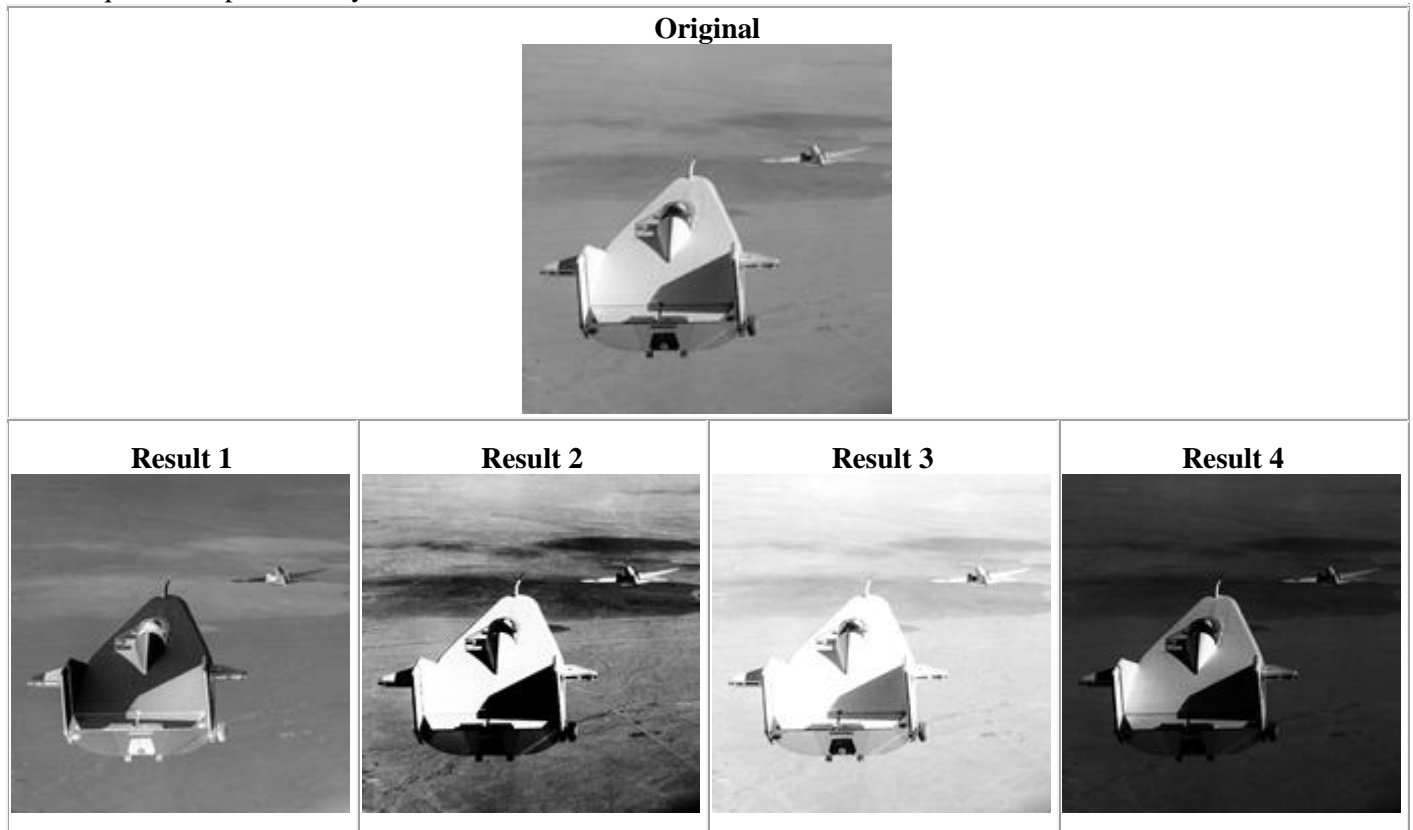
#### *Marking Scheme*

4 marks total

- /3 add the contrast stretching function to intrans
- /1 demonstrate that it works by reproducing the 6 pictures as described

#### **Question No. 2 [12 marks]**

Identify which intensity transformation was used on MATLAB's built-in [liftingbody.png](#) to create each of the four results below. Write a script to reproduce the results using the intensity transformations or the equivalents provided by the [intrans](#) function.



#### *Marking scheme:*

12 Marks total. 3 marks per image:

- /1 identifying correct intensity transformation
- /1 visually similar results
- /1 results are an exact match

**Hint:** each intensity transformation described below is used only once.

**Question No. 3 [ 10 marks]**

**Spatial Filtering**

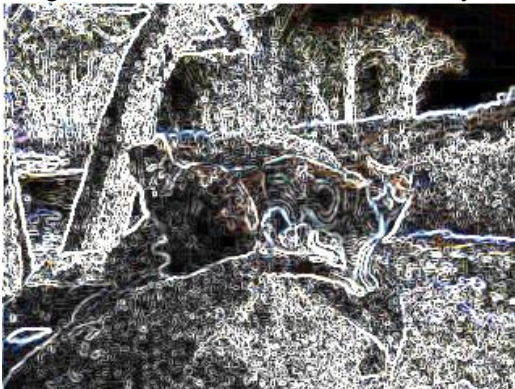
1. Download the following image "[two\\_cats.jpg](#)" and store it in MATLAB's "Current Directory".



2. Load the image data.
3. Use a spatial filter to find and display the horizontal edges of the image.
4. Use a spatial filter to find and display the vertical edges of the image  
**hint:** read the MATLAB documentation on fspecial
5. Add the horizontal edge image to the vertical edge image to yield the following results:



6. See if you can reproduce the following result, which is the edge magnitude map for for this image. The relevant instructions are in your textbook.



**Marking Scheme**

10 marks total:

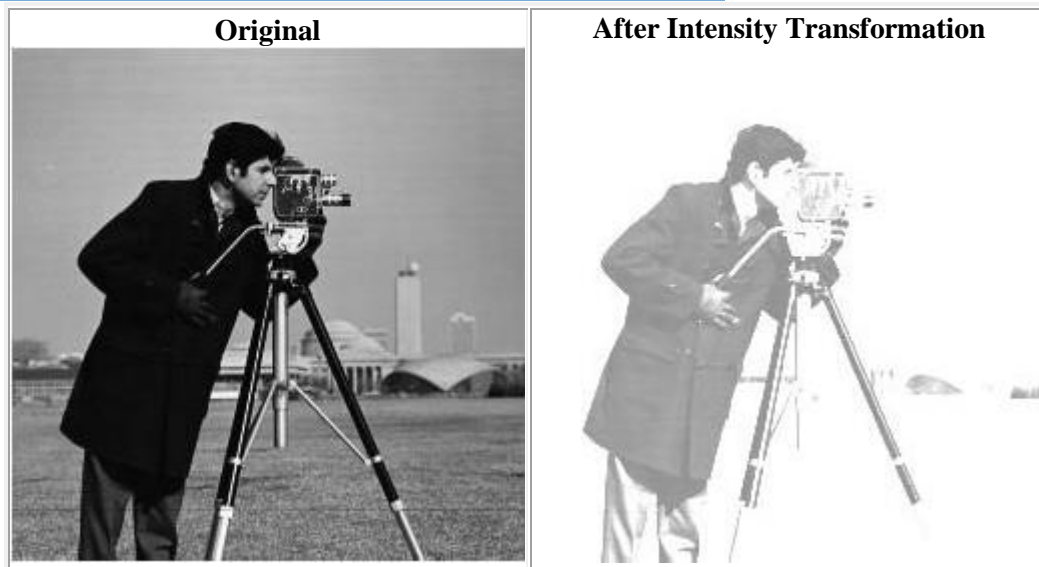
- /2 Horizontal edges
- /2 Vertical edges
- /2 Combined result from step 5
- /4 Edge magnitude map

Deliverables: (submit the code as well as in printed form)

- Part 1
  - your completed `intrans` function
  - your testing script
- Part 2
  - a script that creates and displays your version of the four result images.
- Part 3
  - a script that creates and displays the four edge images described.

### 1. Intensity Transformation Functions

When you are working with gray-scale images, sometimes you want to modify the intensity values. For instance, you may want to reverse black and the white intensities or you may want to make the darks darker and the lights lighter. An application of intensity transformations is to increase the contrast between certain intensity values so that you can pick out things in an image. For instance, the following two images show an image before and after an intensity transformation. Originally, the camera man's jacket looked black, but with an intensity transformation, the difference between the black intensity values, which were too close before, was increased so that the buttons and pockets became viewable. (This example is from *the Image Processing Toolbox, User's Guide, Version 5* (MATLAB's documentation)--available through MATLAB's help menu or online at: <http://www.mathworks.com/access/helpdesk/help/toolbox/images/> ).

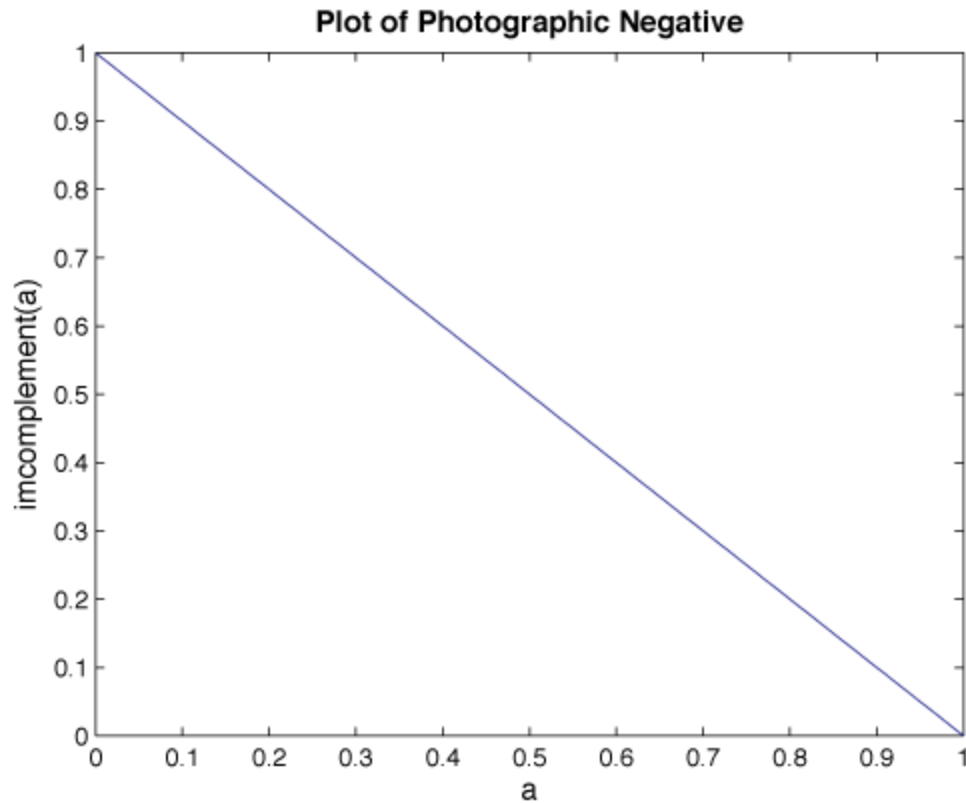


Generally, making changes in the intensity is done through **Intensity Transformation Functions**. The next sections talk about four main intensity transformation functions:

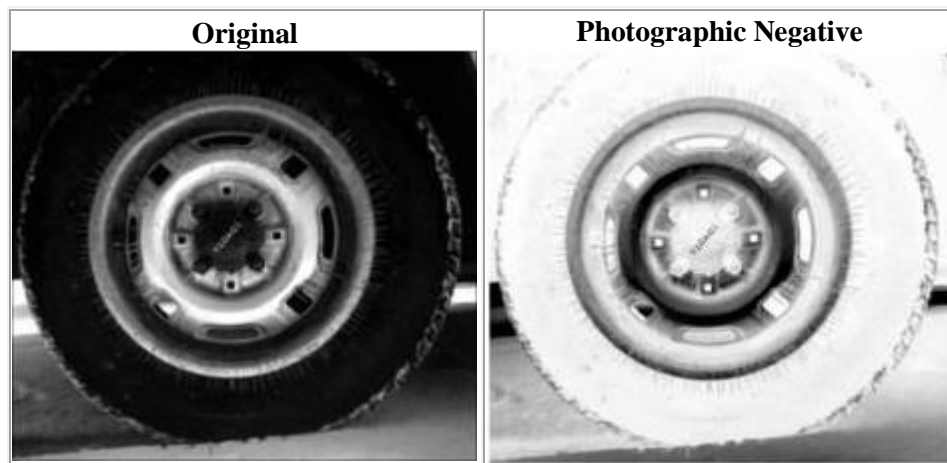
1. photographic negative (using `imcomplement`)
2. gamma transformation (using `imadjust`)
3. logarithmic transformations (using `c*log(1+f)`)
4. contrast-stretching transformations (using `1./(1+(m./(double(f)+eps)).^E)`)

#### 1.1 Photographic Negative

The Photographic Negative is probably the easiest of the intensity transformations to describe. Assume that we are working with grayscale double arrays where black is 0 and white is 1. The idea is that 0's become 1's, 1's become 0's, and any gradients in between are also reversed. In intensity, this means that the true black becomes true white and vice versa. MATLAB has a function to create photographic negatives--`imcomplement(f)`. Given `a=0:0.01:1`, the below shows a graph of the mapping between the original values (x-axis) and the `imcomplement` function.



The following is an example of a photographic negative. Notice how you can now see the writing in the middle of the tire better than before:



The MATLAB code that created these two images is:

```
I=imread('tire.tif');
imshow(I)
J=imcomplement(I);
figure, imshow(J)
```

## 1.2 Gamma Transformations

See section 5.7 (esp. 5.7.1 - 5.7.4) in your textbook.

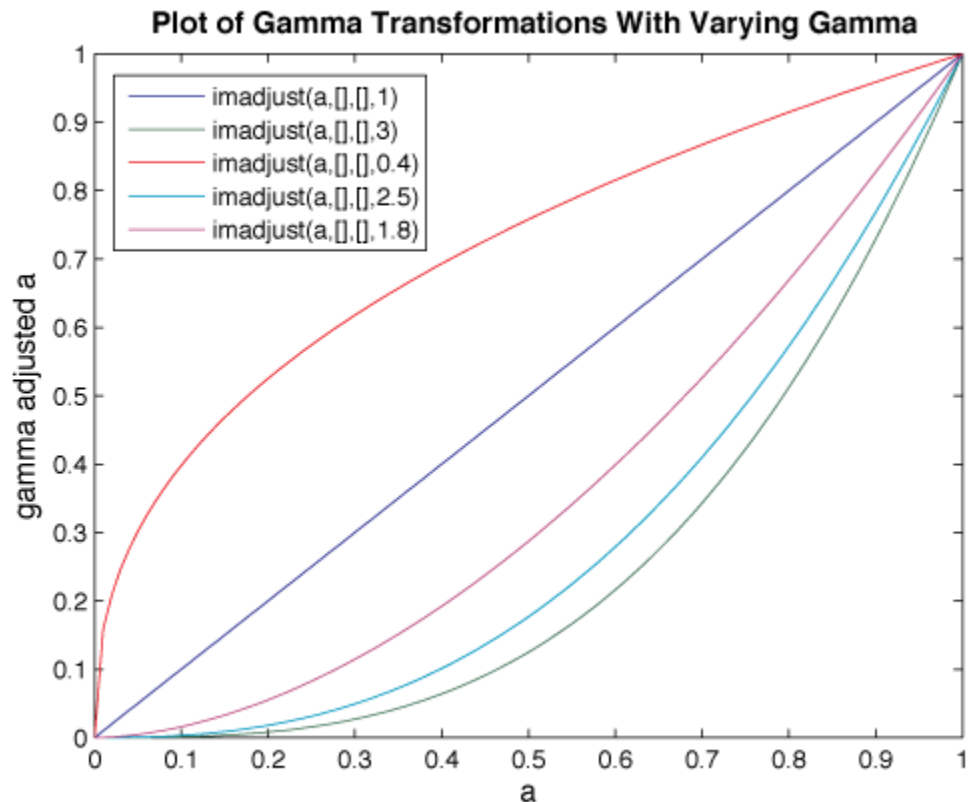
With Gamma Transformations, you can curve the grayscale components either to brighten the intensity (when **gamma** is less than one) or darken the intensity (when **gamma** is greater than one). The MATLAB

function that creates these **gamma** transformations is:

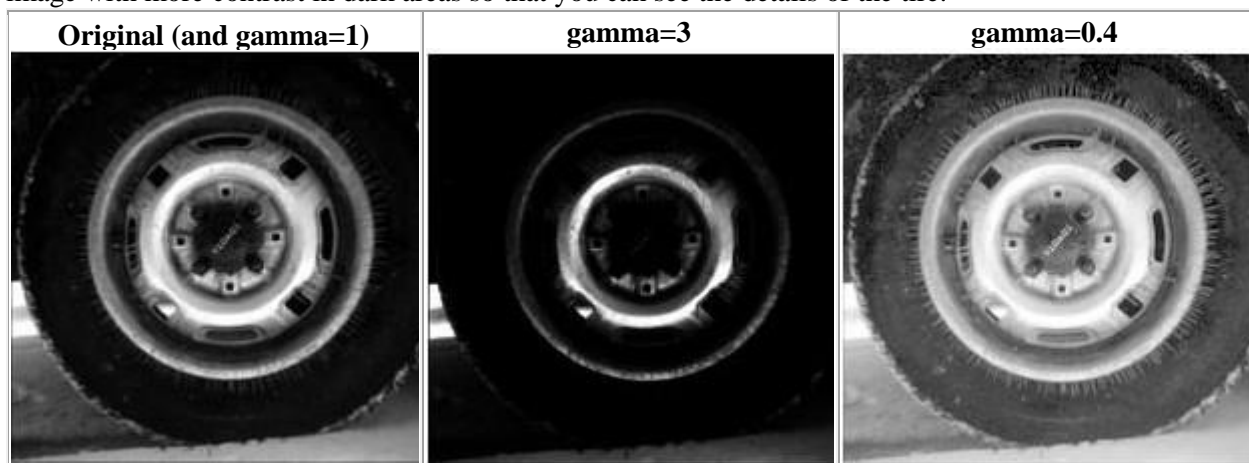
`imadjust(f, [low_in high_in], [low_out high_out], gamma)`

`f` is the input image, **gamma** controls the curve, and `[low_in high_in]` and `[low_out high_out]` are used for clipping. Values below `low_in` are clipped to `low_out` and values above `high_in` are clipped to `high_out`.

For the purposes of this lab, we use `[]` for both `[low_in high_in]` and `[low_out high_out]`. This means that the full range of the input is mapped to the full range of the output. Given `a=0:0.01:1`, the following plots show the effect of the gamma transformation with varying gamma. Notice that the red line has  $\gamma=0.4$ , which creates an upward curve and will brighten the image.



The following shows the results of three of the gamma transformations shown in the plot above. Notice how the values greater than 1 one create a darker image, whereas values between 0 and 1 create a brighter image with more contrast in dark areas so that you can see the details of the tire.



The MATLAB code that created these three images is:



```
I=imread('tire.tif');  
  
J=imadjust(I,[],[],1);  
J2=imadjust(I,[],[],3);  
J3=imadjust(I,[],[],0.4);  
imshow(J);  
figure,imshow(J2);  
figure,imshow(J3);
```

Gamma transformations are an important part of the image display process. You should learn more about them. Charles Poynton, an expert in digital video systems who has worked for NASA, has [an excellent FAQ about gamma](#) that I encourage you to read - especially if you plan to process CGI. He also debunks several [popular misconceptions people have about gamma](#).

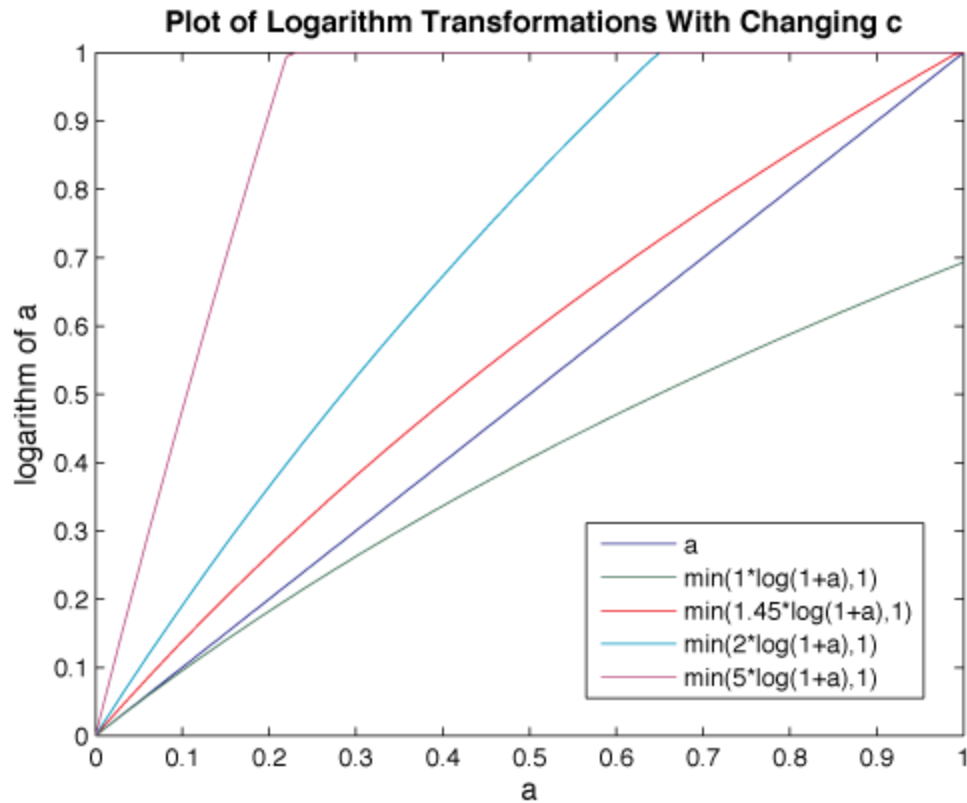
### 1.3 Logarithmic Transformations

From section 3.2.2 of *Digital Image Processing Using Matlab*. See also sections 5.1.1 and 5.1.2 in your textbook

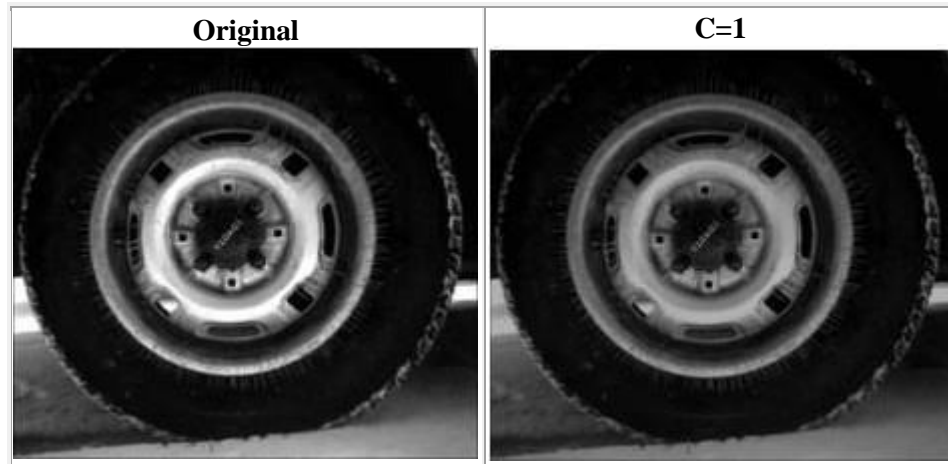
Logarithmic Transformations can be used to brighten the intensities of an image (like the Gamma Transformation, where  $\gamma < 1$ ). More often, it is used to increase the detail (or contrast) of lower intensity values. They are especially useful for bringing out detail in Fourier transforms (covered in a later lab). In MATLAB, the equation used to get the Logarithmic transform of image  $f$  is:

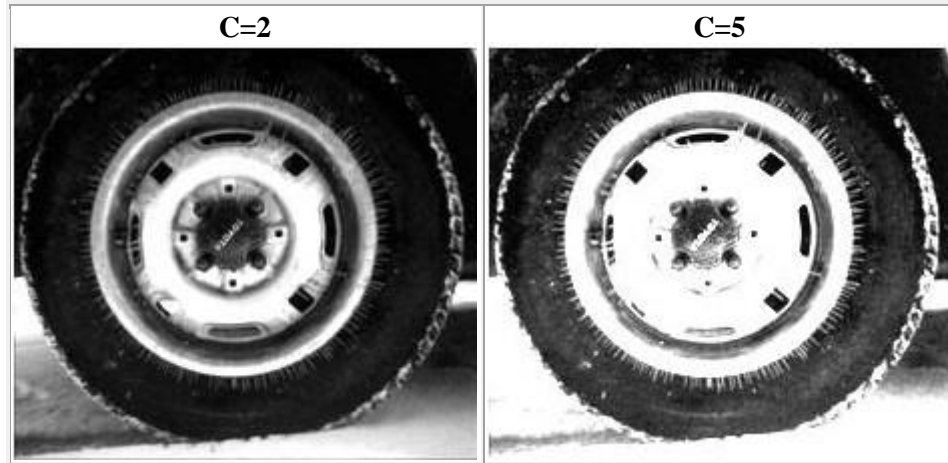
$$g = c * \log(1 + \text{double}(f))$$

The constant  $c$  is usually used to scale the range of the log function to match the input domain. In this case  $c=255/\log(1+255)$  for a uint8 image, or  $c=1/\log(1+1)$  ( $\sim 1.45$ ) for a double image. It can also be used to further increase contrast—the higher the  $c$ , the brighter the image will appear. Used this way, the log function can produce values too bright to be displayed. Given  $a=0:.01:1$ , the plot below shows the result for various values of  $c$ . The y-values are clamped at 1 by the min function for the plot of  $c=2$  and  $c=5$  (teal and purple lines, respectively).



The following shows the original image and the results of applying three of the transformations from above. Notice that when  $c=5$ , the image is the brightest and you can see the radial lines on the inside of the tire (these lines are barely viewable in the original because there is not enough contrast in the lower intensities).





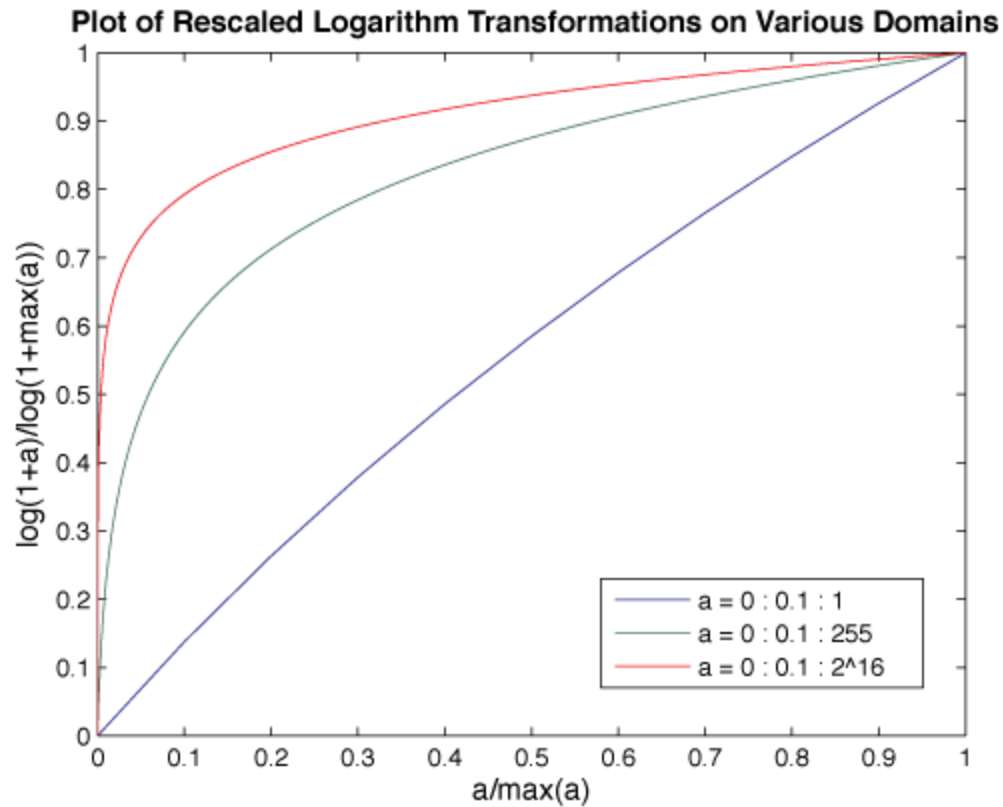
The MATLAB code that created these images is:

```
I=imread('tire.tif');
imshow(I)
I2=im2double(I);
J=1*log(1+I2);
J2=2*log(1+I2);
J3=5*log(1+I2);
figure, imshow(J)
figure, imshow(J2)
figure, imshow(J3)
```

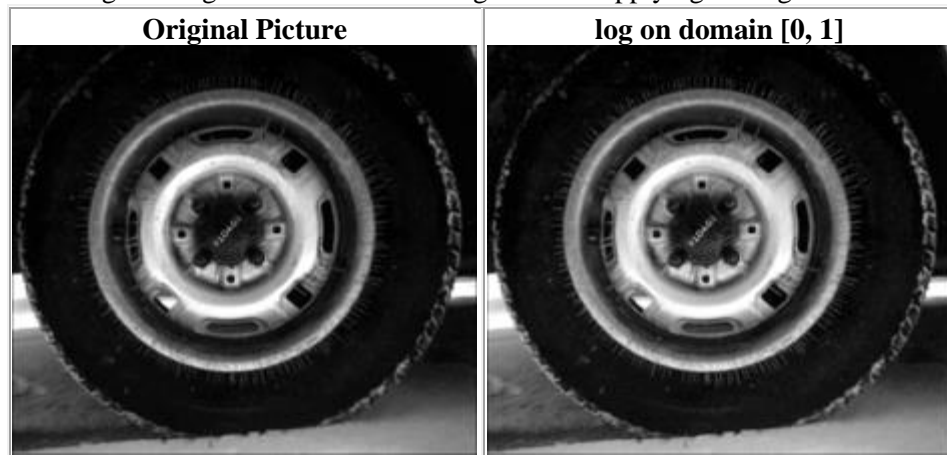
Notice the loss of detail in the bright regions where intensity values are clamped. Any values greater than one, produced from the scaling, are displayed as having a value of 1 (full intensity) and should be clamped. Clamping in MATLAB can be performed by the `min(matrix, upper_bound)`, and `max(matrix, lower_bound)` functions as shown in the legend for the plot above.

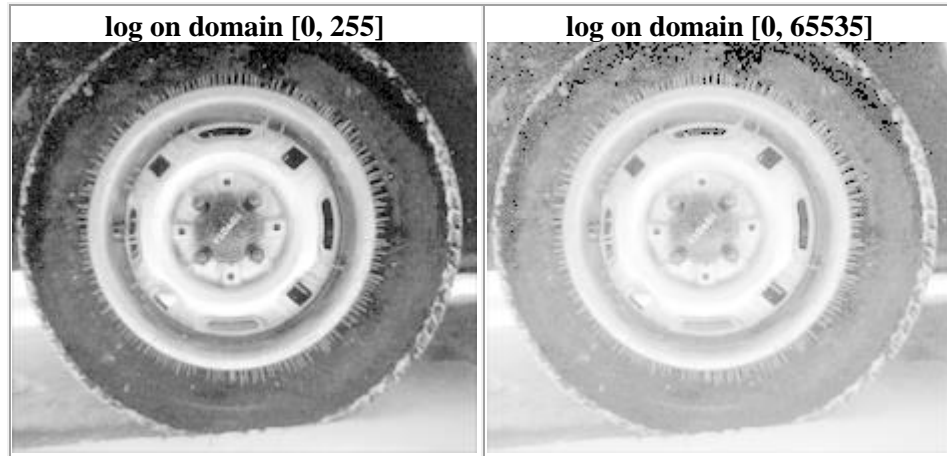
Although logarithms may be calculated in different bases such as MATLAB's builtin `log10`, `log2` and `log` (natural log), the resulting curve, when the range is scaled to match the domain, is the same for all bases. The shape of the curve is dependent instead on the range of values it is applied to. Here are examples of the log curve for multiple ranges of input values:





It is important to be aware of this effect if you plan to use logarithm transformations successfully, so here is the result of scaling an image's values to those ranges before applying the logarithm transform:





The MATLAB code that produced these images is:

```
tire = imread('tire.tif');
d = im2double(tire);
figure, imshow(d);
%log on domain [0,1]
f = d;
c = 1/log(1+1);
j1 = c*log(1+f);
figure, imshow(j1);
%log on domain [0, 255]
f = d*255;
c = 1/log(1+255);
j2 = c*log(1+f);
figure, imshow(j2);
%log on domain [0, 2^16]
f = d*2^16;
c = 1/log(1+2^16);
j3 = c*log(1+f);
figure, imshow(j3);
```

Note that for domain  $[0, 1]$  the effects of the logarithm transform are barely noticeable, while for domain  $[0, 65535]$  the effect is extremely exaggerated. Also note that, unlike with linear scaling and clamping, gross detail is still visible in light areas.

#### 1.4 Contrast-Stretching Transformations

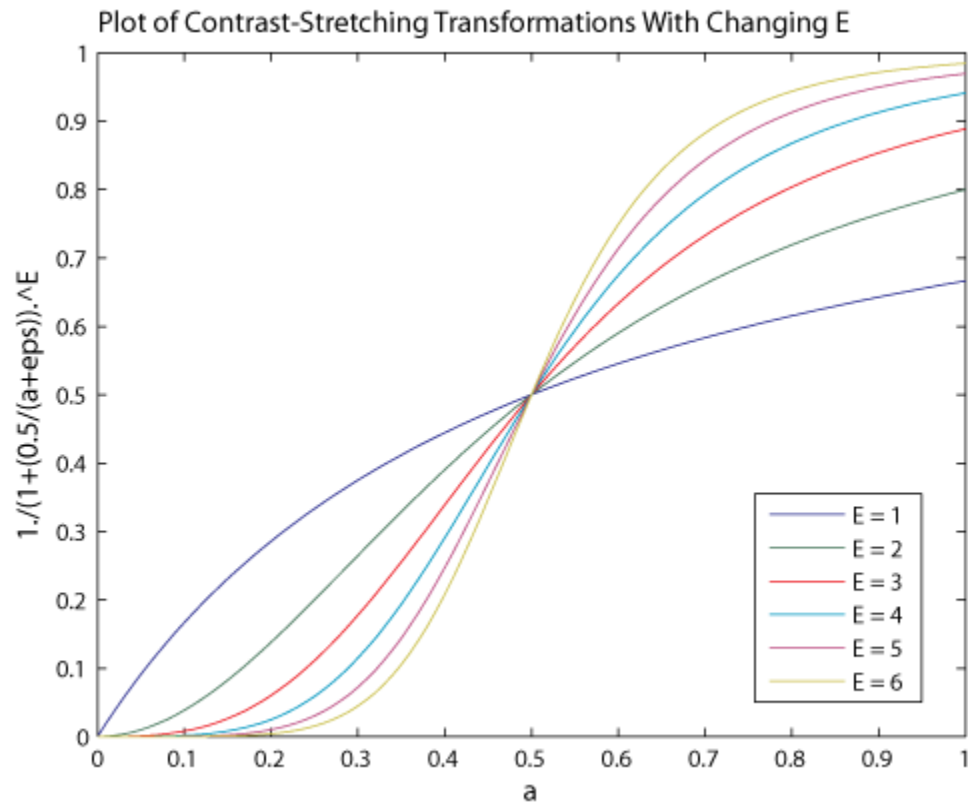
From section 3.2.2 of *Digital Image Processing Using Matlab*.

Contrast-stretching transformations increase the contrast between the darks and the lights. In lab 1 we saw a simplified version of the automatic contrast adjustment in section 5.3 of the textbook. That transformation kept everything at relatively similar intensities and merely stretched the histogram to fill the image's intensity domain. Sometimes you want to stretch the intensity around a certain level. You end up with everything darker darks being a lot darker and everything lighter being a lot lighter, with only a few levels of gray around the level of interest. To create such a contrast-stretching transformation in MATLAB, you can use the following function:

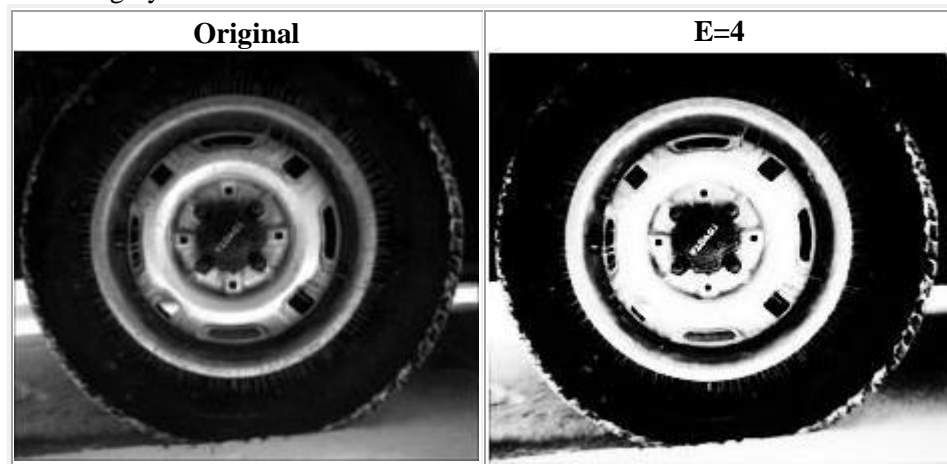
$$g = 1 ./ (1 + (m ./ (\text{double}(f) + \text{eps})) .^ E)$$

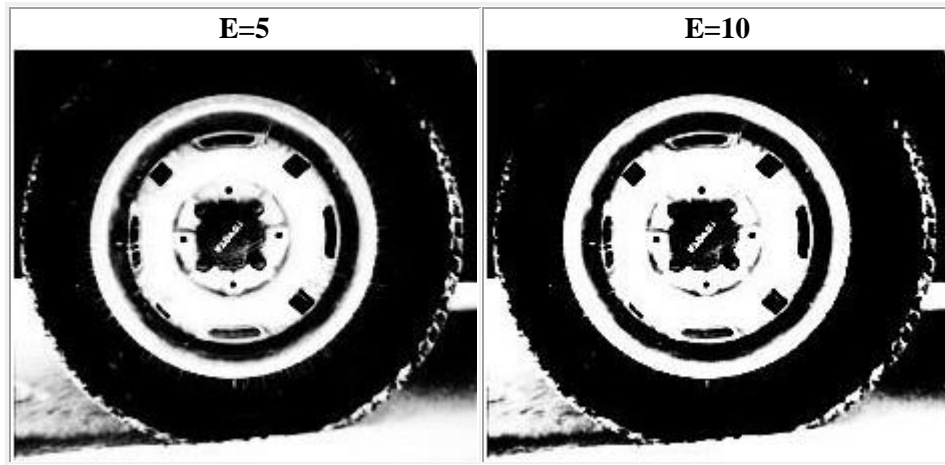
$E$  controls the slope of the function and  $m$  is the mid-line where you want to switch from dark values to light values. `eps` is a MATLAB constant that is the distance between 1.0 and the next largest number that can be represented in double-precision floating point. In this equation it is used to prevent division by zero in the event that the image has any zero valued pixels. There are two plot/diagram sets below to

represent the results of changing both  $m$  and  $E$ . The below plot shows the results for several different values of  $E$  given  $a=0:0.01:1$  and  $m=0.5$ .



The following shows the original image and the results of applying the three transformations from above. The  $m$  value used below is the mean of the image intensities (0.2104). At very high  $E$  values, such as 10, the function becomes more like a thresholding function with threshold  $m$ —the resulting image is more black and white than grayscale.

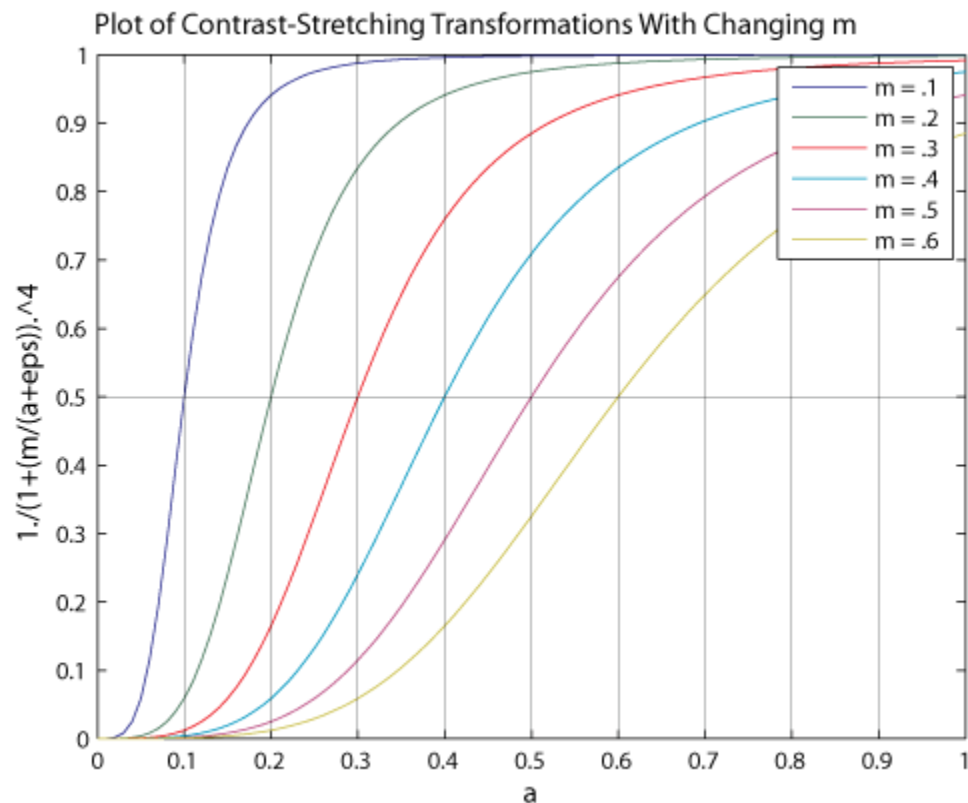




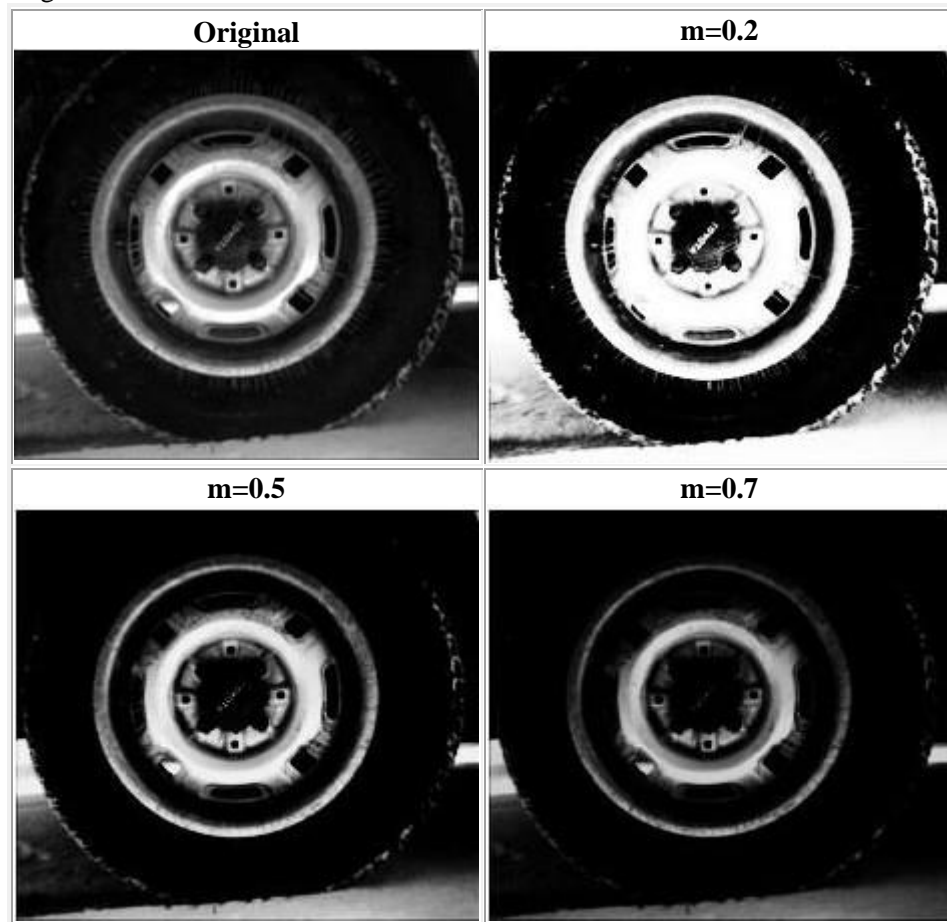
The MATLAB code that created these images is:

```
I=imread('tire.tif');
I2=im2double(I);
m=mean2(I2)
contrast1=1./(1+(m./(I2+eps)).^4);
contrast2=1./(1+(m./(I2+eps)).^5);
contrast3=1./(1+(m./(I2+eps)).^10);
imshow(I2)
figure,imshow(contrast1)
figure,imshow(contrast2)
figure,imshow(contrast3)
```

This second plot shows how changes to **m** (using **E=4**) affect the contrast curve:



The following shows the original image and the results of applying the three transformations from above. The  $m$  value used below is 0.2, 0.5, and 0.7. Notice that 0.7 produces a darker image with fewer details for this tire image.



The MATLAB code that created these images is:

```
I=imread('tire.tif');
I2=im2double(I);
contrast1=1./(1+(0.2./(I2+eps)).^4);
contrast2=1./(1+(0.5./(I2+eps)).^4);
contrast3=1./(1+(0.7./(I2+eps)).^4);
imshow(I2)
figure,imshow(contrast1)
figure,imshow(contrast2)
figure,imshow(contrast3)
```

### 3.5 The `intrans` and `changeClass` Functions

The file [intrans.m](#) *Digital Image Processing, Using MATLAB*<sup>[2]</sup> contains a function that does all of the intensity transformations mentioned above except the contrast stretching transform. You should read the code and figure out how to include that capability.

The `intrans` function relies on a second function called `changeClass`. [You can download the M-File for that function here.](#)<sup>[3]</sup>

The comments beginning in the second line of the `intrans` function describe how to use it. Please notice the description of the missing contrast stretch transform - it should take varying parameters and it says what defaults to use for missing parameters. The following table provides some examples of

using `intrans` to correspond to the four Intensity Transformation Functions. Assume that `I=imread('tire.tif');`

Transformation	Intensity Transformation Function	Corresponding intrans Call
photographic negative	<code>neg=imcomplement(I);</code>	<code>neg=intrans(I,'neg');</code>
logarithmic	<code>I2=im2double(I); log=5*log(1+I2);</code>	<code>log=intrans(I,'log',5);</code>
gamma	<code>gamma=imadjust(I,[],[],0.4);</code>	<code>gamma=intrans(I,'gamma',0.4);</code>
contrast-stretching	<code>I2=im2double(I); contrast=1./(1+(0.2./(I2+eps)).^5);</code>	<code>contrast=intrans(I,'stretch',0.2,5);</code>

## 2. Spatial Filtering

See chapter 6 (esp. 6.1, 6.2, 6.5.2), section 7.2, and section 7.3 (esp. 7.3.1) in your textbook.

There are two main types of filtering applied to images:

- spatial domain filtering
- frequency domain filtering

In a later lab we will talk about frequency domain filtering, which makes use of the Fourier Transform.

For spatial domain filtering, we are performing filtering operations directly on the pixels of an image.

Spatial filtering is a technique that uses a pixel and its neighbors to select a new value for the pixel. The simplest type of spatial filtering is called linear filtering. It attaches a weight to the pixels in the neighborhood of the pixel of interest, and these weights are used to blend those pixels together to provide a new value for the pixel of interest. Linear filtering can be used to smooth, blur, sharpen, or find the edges of an image. The following four images are meant to demonstrate what spatial filtering can do. The original image is shown in the upper left-hand corner.

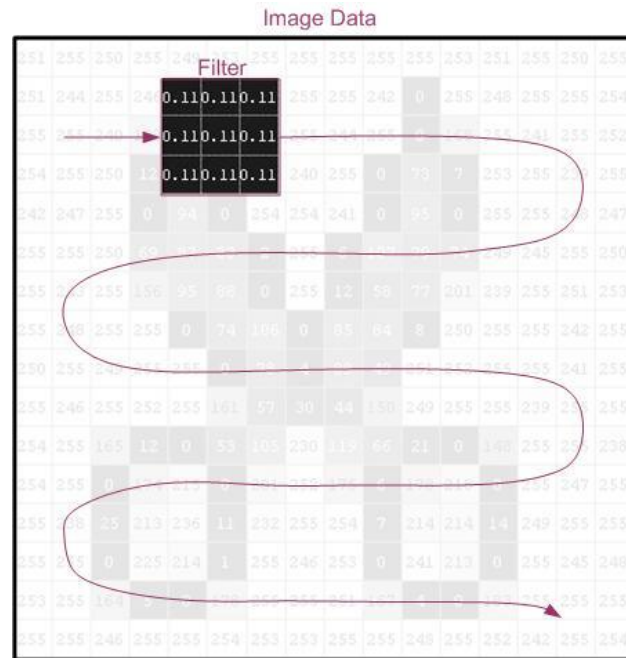


Sometimes a linear filter is not enough to solve a particular problem. In that case it is possible to use higher order math or full-blown MATLAB functions produce specialized results. Such non-linear filters are useful for smoothing only smooth areas, enhancing only strong edges or removing speckles from images.

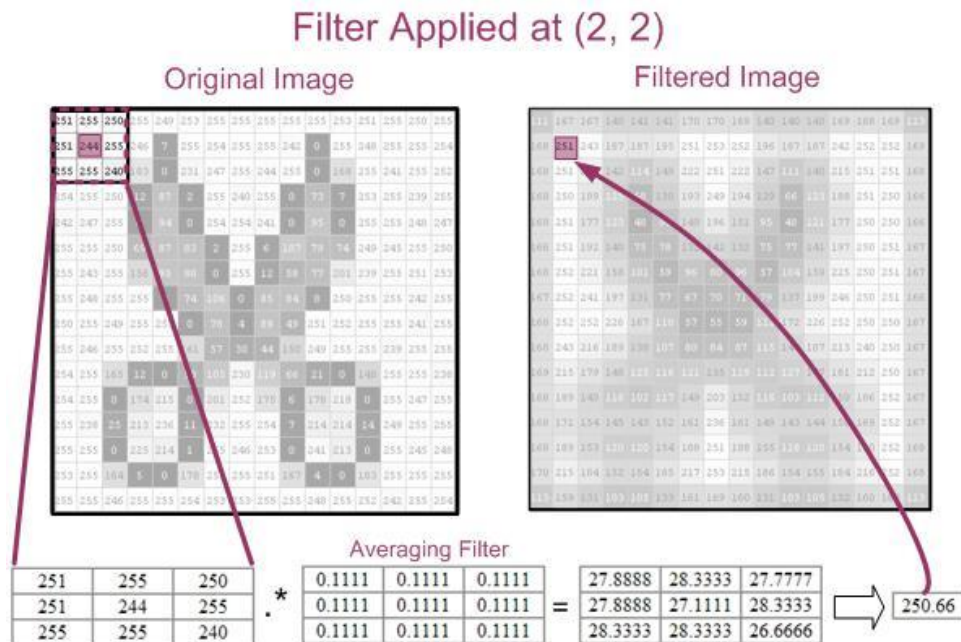


## 2.1 Basic Idea

Spatial Filtering is sometimes also known as neighborhood processing. Neighborhood processing is an appropriate name because you define a center point and perform an operation (or apply a filter) to only those pixels in predetermined neighborhood of that center point. The result of the operation is one value, which becomes the value at the center point's location in the modified image. Each point in the image is processed with its neighbors. The general idea is shown below as a "sliding filter" that moves throughout the image to calculate the value at the center location.



The following diagram is meant to illustrate in further details how the filter is applied. The filter (an averaging filter) is applied to location 2,2.

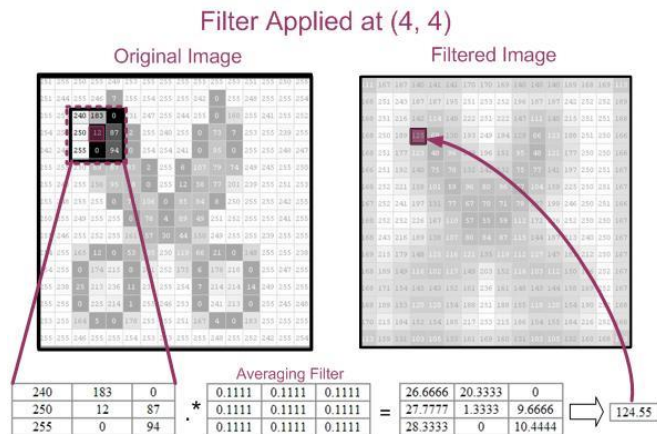


Notice how the resulting value is placed at location 2,2 in the filtered image.

The breakdown of how the resulting value of 251 (rounded up from 250.66) was calculated mathematically is:

$$\begin{aligned}
&= 251*0.1111 + 255*0.1111 + 250*0.1111 + 251*0.1111 + 244*0.1111 + 255*0.1111 + 255*0.1111 + \\
&255*0.1111 + 240*0.1111 \\
&= 27.88888 + 28.33333 + 27.77777 + 27.88888 + 27.11111 + 28.33333 + 28.33333 + 28.33333 + \\
&26.66666 \\
&= 250.66
\end{aligned}$$

The following illustrates the averaging filter applied to location 4,4.



Once again, the mathematical breakdown of how 125 (rounded up from 124.55) was calculated is below:

$$\begin{aligned}
&= 240*0.1111 + 183*0.1111 + 0*0.1111 + 250*0.1111 + 12*0.1111 + 87*0.1111 + 255*0.1111 + \\
&0*0.1111 + 94*0.1111 \\
&= 26.6666 + 20.3333 + 0 + 27.7777 + 1.3333 + 9.6666 + 28.3333 + 0 + 10.4444 \\
&= 124.55
\end{aligned}$$

The following MATLAB function demonstrates how spatial filtering may be applied to an image:

```

function img = myfilter(f, w)
%MYFILTER Performs spatial correlation
% I=MYFILTER(f, w) produces an image that has undergone correlation.
% f is the original image
% w is the filter (assumed to be 3x3)
% The original image is padded with 0's
%Author: Nova Scheidt

```

```

% check that w is 3x3
[m,n]=size(w);
if m~=3 | n~=3
    error('Filter must be 3x3')
end

```

```

%get size of f
[x,y]=size(f);

```

```

%create padded f (called g)
%first, fill with zeros
g=zeros(x+2,y+2);
%then, store f within g
for i=1:x
    for j=1:y
        g(i+1,j+1)=f(i,j);
    end
end

```

```

end

%cycle through the array and apply the filter
for i=1:x
    for j=1:y
        img(i,j)=g(i,j)*w(1,1)+g(i+1,j)*w(2,1)+g(i+2,j)*w(3,1) ... %first column
        + g(i,j+1)*w(1,2)+g(i+1,j+1)*w(2,2)+g(i+2,j+1)*w(3,2)... %second column
        + g(i,j+2)*w(1,3)+g(i+1,j+2)*w(2,3)+g(i+2,j+2)*w(3,3);
    end
end
end

```

```

%Convert to uint--otherwise there are double values and the expected
%range is [0, 1] when the image is displayed
img=uint8(img);

```

To apply the filter to the example above, the following calls were made:

(The '[stock\\_cut](#)' image was modified from the gnome 2.14 icon set, available under [GPL 2.0](#))

```

w=[1/9 1/9 1/9
    1/9 1/9 1/9
    1/9 1/9 1/9]
stock_cut=imread('stock_cut.jpg');
results=myfilter(stock_cut,w);
imtool(results)

```

## 2.2 Filtering with imfilter

Instead of using the M-File from above, you can use a function that comes as part of the Image Processing Toolkit. You can call it in the same way that [myfilter](#) was called above:

```
results=imfilter(stock_cut, w);
```

[imfilter](#) is more powerful than the simple [myfilter](#). The following table, modified from page 94 of *Digital Image Processing, Using MATLAB*, by Rafael C. Gonzalez, Richard E. Woods, and Steven L. Eddins, summarizes the additional options available with [imfilter](#).

Options	Description
<b>Filtering mode</b>	
'corr'	Filtering is done using correlation. This is the default.
'conv'	Filtering is done using convolution.
<b>Boundary Options</b>	
P	The boundaries of the input image are extended by padding with a value, P (written without quotes). This is the default, with value 0.
'replicate'	The size of the image is extended by replicating the values in its outer border.
'symmetric'	The size of the image is extended by mirror-reflecting it across its border.
'circular'	The size of the image is extended by treating the image as one period a 2-D periodic function.
<b>Size Options</b>	
'full'	The output is of the same size as the extended (padded) image.

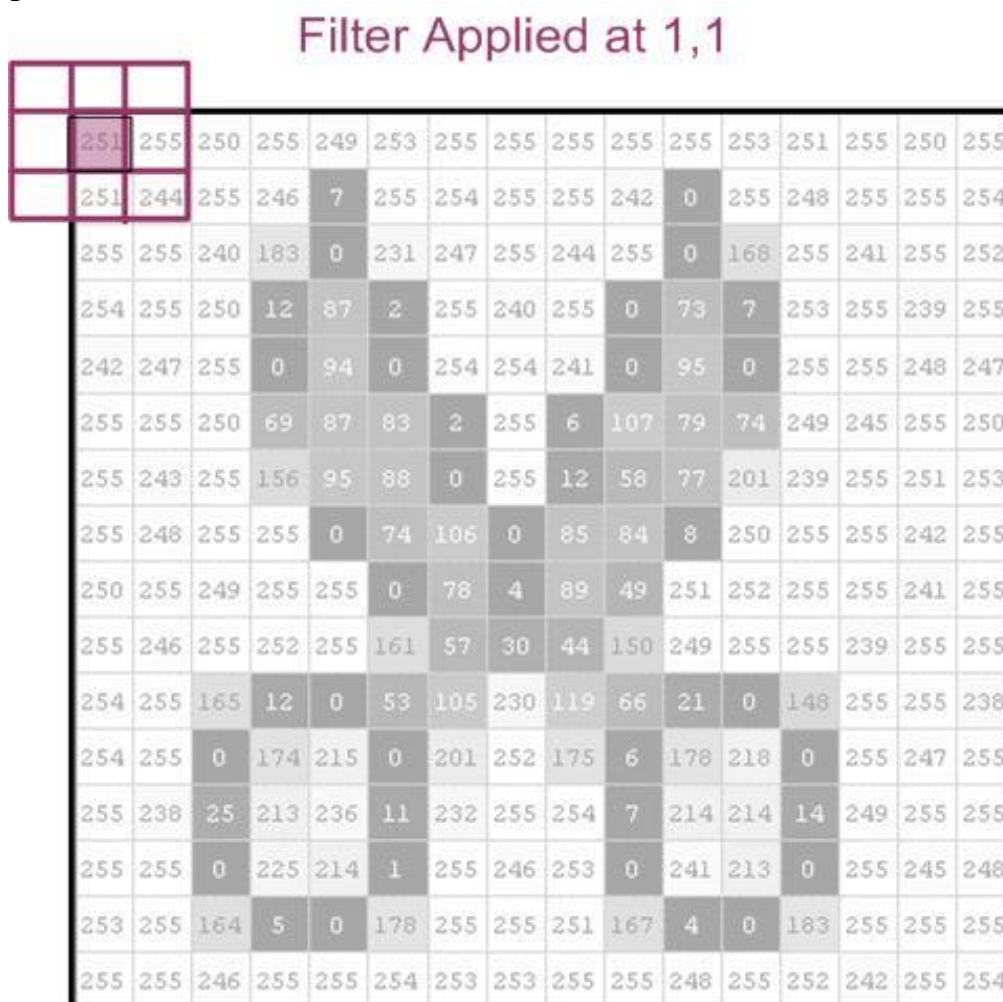
'same'	The output is of the same size as the output. This is achieved by limiting the excursions of the center of the filter mask to points contained in the original image. This is the default.
--------	--

The following subsections discuss the [imfilter](#) options.

### 2.2.1 *Imfilter—Boundary Options*

See section 6.5.2 in your textbook.

The example above deliberately applied the filter at location 2,2. This is because there is an inherent problem when you are working with the corners and edges. The problem is that some of the "neighbors" are missing. Consider location 1,1:



In this case, there are no upper neighbors or neighbors to the left. Two solutions, zero padding and replicating, are shown below. The pixels highlighted in blue have been added to the original image:

# Zero Padding

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	251	255	250	255	249	253	255	255	255	255	255	253	251	255	250	255
0	251	244	255	246	7	255	254	255	255	242	0	255	248	255	255	254
0	255	255	240	183	0	231	247	255	244	255	0	168	255	241	255	252
0	254	255	250	12	87	2	255	240	255	0	73	7	253	255	239	255
0	242	247	255	0	94	0	254	254	241	0	95	0	255	255	248	247
0	255	255	250	69	87	83	2	255	6	107	79	74	249	245	255	250
0	255	243	255	156	95	88	0	255	12	58	77	201	239	255	251	253
0	255	248	255	255	0	74	106	0	85	84	8	250	255	255	242	255
0	250	255	249	255	255	0	78	4	89	49	251	252	255	255	241	255
0	255	246	255	252	255	161	57	30	44	150	249	255	255	239	255	255
0	254	255	165	12	0	53	105	230	119	66	21	0	148	255	255	238
0	254	255	0	174	215	0	201	252	175	6	178	218	0	255	247	255
0	255	238	25	213	236	11	232	255	254	7	214	214	14	249	255	255
0	255	255	0	225	214	1	255	246	253	0	241	213	0	255	245	248
0	253	255	164	5	0	178	255	255	251	167	4	0	183	255	255	255
0	255	255	246	255	255	254	253	253	255	255	248	255	252	242	255	254
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Zero padding is the default. You can also specify a value other than zero to use as a padding value. Another solution is replicating the pixel values along the edges:



# Replicating

251	251	255	250	255	249	253	255	255	255	255	255	253	251	255	250	255	255
251	251	255	250	255	249	253	255	255	255	255	255	253	251	255	250	255	255
251	251	244	255	246	7	255	254	255	255	242	0	255	248	255	255	254	254
255	255	255	240	183	0	231	247	255	244	255	0	168	255	241	255	252	252
254	254	255	250	12	87	2	255	240	255	0	73	7	253	255	239	255	255
242	242	247	255	0	94	0	254	254	241	0	95	0	255	255	248	247	247
255	255	255	250	69	87	83	2	255	6	107	79	74	249	245	255	250	250
255	255	243	255	156	95	88	0	255	12	58	77	201	239	255	251	253	253
255	255	248	255	255	0	74	106	0	85	84	8	250	255	255	242	255	255
250	250	255	249	255	255	0	78	4	89	49	251	252	255	255	241	255	255
255	255	246	255	252	255	161	57	30	44	150	249	255	255	239	255	255	255
254	254	255	165	12	0	53	105	230	119	66	21	0	148	255	255	238	238
254	254	255	0	174	215	0	201	252	175	6	178	218	0	255	247	255	255
255	255	238	25	213	236	11	232	255	254	7	214	214	14	249	255	255	255
255	255	255	0	225	214	1	255	246	253	0	241	213	0	255	245	248	248
253	253	255	164	5	0	178	255	255	251	167	4	0	183	255	255	255	255
255	255	255	246	255	255	254	253	253	255	255	248	255	252	242	255	254	254
255	255	255	246	255	255	254	253	253	255	255	248	255	252	242	255	254	254

As a note, if your filter were larger than 3x3, then the "border padding" would have to be extended. For a filter of size 3x3, 'replicate' and 'symmetric' yield the same results.

The following images show the results of the four different boundary options. The filter used below is a 5x5 averaging filter that was created with the following syntax:

`h=fspecial('average',5)`





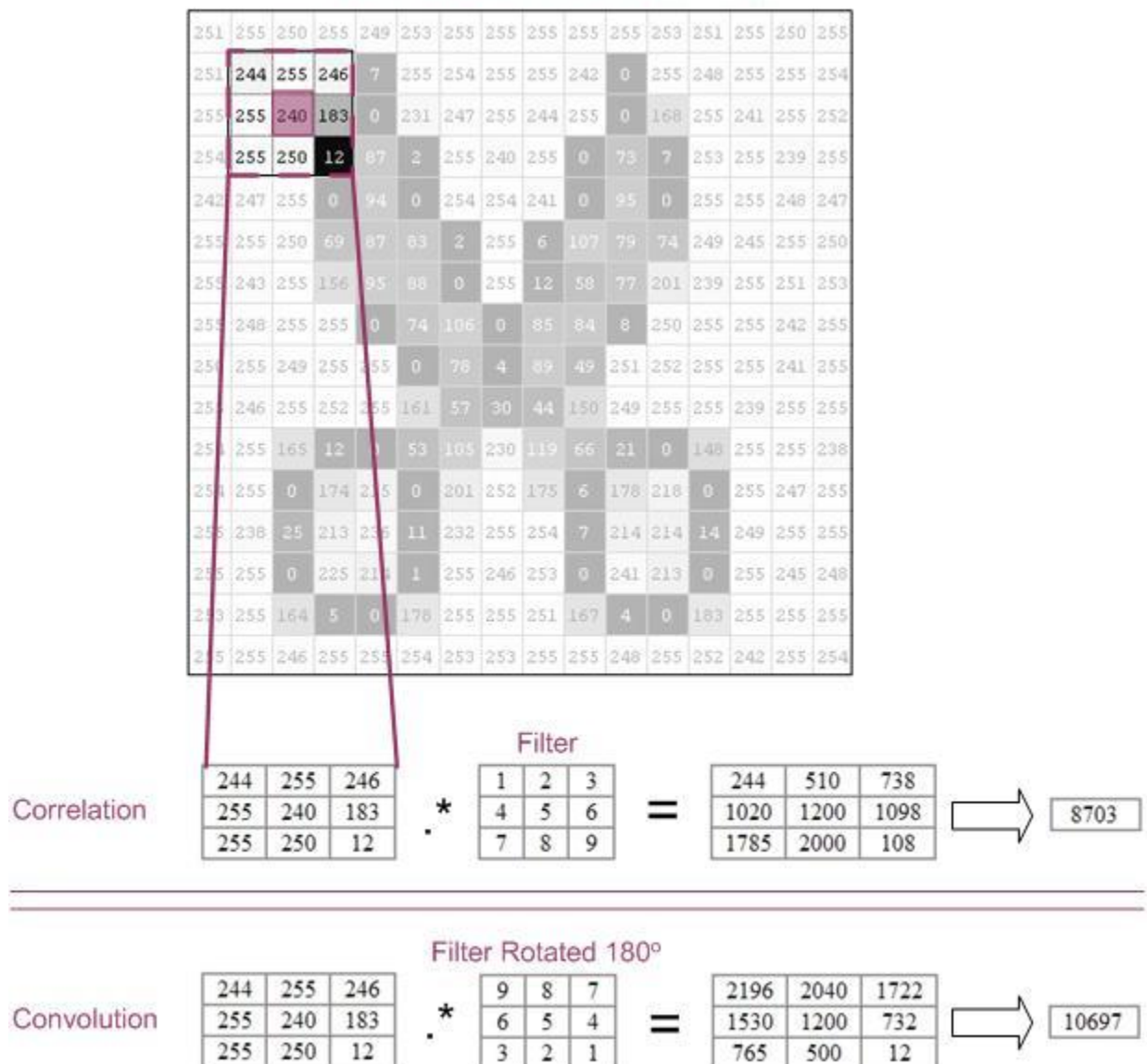


The disadvantage of zero padding is that it leaves dark artifacts around the edges of the filtered image (with white background). You can see this as a dark border along the bottom and right-hand edge in the zero-padded image above.

### 2.2.2 *Imfilter—Filtering Mode (Correlation versus Convolution)*

With `imfilter`, you can choose one of two filtering modes: ***correlation*** or ***convolution***. The difference between the two is that convolution rotates the filter by 180° before performing multiplication. The following diagram is meant to demonstrate the two operations for position 3, 3 of the image:

## Correlation Versus Convolution



This example is for demonstration purposes only. You will notice that the resulting values are not in the range of [0, 255]. To get better results, you can normalize the filter (in this case, divide by 45).

The following MATLAB code demonstrates correlation and convolution:

```
h=[1 2 3
   4 5 6
   7 8 9];
h=h/45;
result_corr=imfilter(cat,h); % correlation is the default,
                             % you can also send 'corr' as an argument
result_conv=imfilter(cat,h,'conv');
```

### 2.2.3 Imfilter—Size Options

There are two size options 'full' and 'same'. The 'full' will be as large as the padded image, where as 'same' will be the same size as the input image.

To create a 'full' and 'same' image, you can use the following MATLAB syntax:

```
h=[0.1111 0.1111 0.1111  
0.1111 0.1111 0.1111  
0.1111 0.1111 0.1111];  
stock_cut_same=imfilter(stock_cut,h); % 'same' is the default, but you can also  
% include it as an argument  
stock_cut_full=imfilter(stock_cut,h,'full');
```




If you use [imtool](#) to view both of these images, you will note that the 'same' is 16x16, whereas 'full' is 18x18.

### 2.3 Predefined Filters

You can define the filters for spatial filtering manually or you can call a function that will create certain common filter matrices for you. The function, called [fspecial](#), requires an argument that specifies the kind of filter you would like. A full description of [fspecial](#) is available in MATLAB help—type:

[doc fspecial](#)

The following table is meant to show you three filters, created by [fspecial](#), and the results on an image of a cat:

MATLAB Code	Resulting Image
<pre>%original picture cat=imread('cat.jpg'); figure, imshow(cat)</pre>	
<pre>%motion blur h=fspecial('motion', 20, 45); cat_motion=imfilter(cat,h); figure, imshow(cat_motion)</pre>	
<pre>%sharpening %see section 7.6 (esp 7.6.2) h=fspecial('unsharp'); cat_sharp=imfilter(cat,h); figure, imshow(cat_sharp)</pre>	

```
%horizontal edge-  
detection  
%see section 7.2 and 7.3.1  
h=fspecial('sobel');  
cat_sobel=imfilter(cat,h);  
figure, imshow(cat_sobel)
```

