LAB 5: IMAGE FILTERING

ECE180: Introduction to Signal Processing

OVERVIEW

You have recently learned about the *convolution sum* that serves as the basis of the FIR filter difference equation. The filter coefficient sequence $\{b_k\}$ – equivalent to the filter's impulse response $h[n]$ – may be viewed as a one-dimensional *moving window* that slides over the input signal $x[n]$ to compute the output signal $y[n]$ at each time step. Extend the moving window concept to a 2-D array to create a useful and popular filtering technique for images.

In this lab project you will implement two types of moving-window image filters, one based on convolution and the other based on the median value of the pixel grayscale values spanned by the window. You will also gain experience with the built-in image convolution filter called `imfilter`.

OUTLINE

1. Develop and test a 3×3 convolution filter
2. Develop and test a 3×3 median filter -OR- use the built-in `medfilt2` function instead
3. Evaluate the convolution and median filters to reduce noise while also preserving edges
4. Study the behavior of various 3×3 convolution filter kernels for smoothing, edge detection, and sharpening
5. Learn how to use `imfilter` to convolution-filter color images, and study the various mechanisms offered by `imfilter` to deal with boundary effects

PREPARATION – TO BE COMPLETED BEFORE LAB

Study these tutorial videos:

1. Nested "for" loops -- http://youtu.be/q2xfz8mOuSI?t=1m8s (review this part)
2. Functions -- http://youtu.be/0zTmMIh6I8A (review as needed)
3. Arrays -- http://youtu.be/qEKXgVZN2ts?t=0m55s (review this part on array-style multiplication)

Ensure that you have added the "ECE180 MATLAB folders" to your MATLAB path, especially the "images" and "matlab" subfolders. Follow along with the tutorial video http://youtu.be/MEqUd0dJNBA, if necessary.

LAB ACTIVITIES

1. *Develop and test a 3×3 convolution filter function:*

<u>TIP</u>: Create a regular script .m file to test your function as you develop it. The script should read an image into a variable, define a convolution kernel, apply your convolution filter to the input image, and display the processed image, e.g., `x=imread('camera.png'); h=ones(3)/3^2; y=conv3x3(x,h); imshow(y)`*. Set a breakpoint inside your* `conv3x3` *function to single-step execution and display local variables to assist debugging.*

1.1. Create the function template `conv3x3.m` and develop it to implement the following behavior:

(a) Accept a grayscale image `x` and the convolution kernel `h` (a 3×3 array) as the input variables; run your function as `y=conv3x3(x,h);` and for development purposes use `h=ones(3)/3^2`, a 3×3 window with the value 1/9 for each element,

(b) Convert `x` to the "double" datatype with the `double` function,

(c) Create the output image `y` initially as a duplicate of the input image `x` and then initialize `y(:)` to zero; this technique creates `y` with the same size and data type as `x`,

(d) Determine the dimensions of the image `x` with the `size` function,

(e) Set up a double for-loop to run over all pixels in the image `x` (subject to boundary limits):

→ Extract a 3×3 neighborhood (subarray) about the current pixel,

→ Multiply (array style) the 2-D subarray by the kernel `h`,

→ Flatten the 2-D array to a 1-D array

(index the array with a single colon to "flatten" any M×N array into a 1-D array),

→ Sum the 1-D array values to a single value and assign this to the current output pixel (see `sum`),

(f) Convert `y` back to the "unsigned 8-bit integer" datatype (see `uint8`), and

(g) Return the convolution-filtered image `y`.

1.2. Enter `load lab_5_verify` at the command line to load the .mat file that contains the test images `X` and `Yconv3x3`. Compare your function's output image with the expected output image like this:

```
isequal(conv3x3(X,ones(3)/3^2), Yconv3x3)
```

A "1" result indicates that your function produces the correct result at every pixel, otherwise you need to keep debugging your function.

1.3. Copy and paste your finished and documented function text.

2. *Develop and test a 3×3 median filter function -OR- study the doc page on the built-in* `medfilt2` *version and continue to Part 3:*

2.1. Create the function template `med3x3.m` and develop it to implement the following behavior:

(a) Accept a grayscale image `x` as the function input,

(b) Copy `x` to the output image `y` and then initialize `y(:)` to zero; this technique creates `y` as the same size and data type as `x`,

(c) Determine the number of image rows and columns (see `size`),

(d) Loop over all pixels in image `x` (subject to boundary limits):

→ Extract a 3×3 neighborhood (subarray) about the current pixel,

→ Flatten the 2-D array to a 1-D array,

→ Sort the 1-D array values (see `sort`),

→ Assign the middle value of the sorted array to the current output pixel, and

(e) Return the median-filtered image `y`.

2.2. The .mat file you loaded in Step 1.2 also contains the output image `Ymed3x3`. Compare your function's output image with the expected output image:

```
isequal(med3x3(X), Ymed3x3)
```

Your function works properly when you see “1” as a result, otherwise you need to keep debugging your function.

2.3. Copy and paste your finished and documented function text.

3. *Evaluate your 3×3 moving-window filters:*

The two filters you developed can be used to remove, or at least reduce, the amount of noise in an image. Ideally noise removal will not blur image *edges*, i.e., the grayscale discontinuities that help us to recognize features in the image.

3.1. Create a new script for this part, and then load and display one of the test patterns developed by the *Society of Motion Picture and Television Engineers* (SMPTE, pronounced “simp-tee”):
`tp=imread('smpte.png'); imshow(tp)`

3.2. Create a copy of the test pattern that includes Gaussian noise:
`tpn=imnoise(tp,'gaussian',0,0.01); figure, imshow(tpn)`

3.3. Apply your median filter to `tpn` to create `tpnm`. Display the noisy image and filtered image side by side:
`imshowpair(tpn,tpnm,'montage')`

3.4. Screenclip this image pair display window for the median filter.

3.5. Create a convolution kernel array `h` to perform the average of all pixel values in the 3×3 window:
`h=ones(3)/3^2`

3.6. Apply your averaging filter to `tpn` to create `tpnc`. Display the noisy image and filtered image side by side as you did earlier with `imshowpair`.

3.7. Screenclip this image display window for the averaging filter.

3.8. Compare and contrast the performance of the median filter and the averaging filter in terms of reducing noise and preserving edges for Gaussian noise; when you “compare and contrast” two processing techniques you discuss what is similar and what is different about the two methods. Use the “Zoom In” tool on the figure windows to study the noise and edges in more detail.

3.9. Repeat Steps 3.1 through 3.9 using “salt and pepper noise,” a disturbance that causes random isolated pixels to be set either to full white or full black:
`tpn=imnoise(tp,'salt & pepper',0.02);`

(a) Screenclip the `imshowpair` image display for the median filter.

(b) Screenclip the `imshowpair` image display for the convolution filter.

3.10. Compare and contrast the performance of the median filter and the convolution filter in terms of reducing noise and preserving edges for salt-and-pepper noise. Remember to use the “Zoom In” tool to study the noise, edges, and other features.

4. *Study the behavior of various 3×3 convolution filter kernels:*

The 3×3 convolution filter is a remarkably versatile tool for image processing. In this section you will study the effects of a wide variety of 3×3 convolution filter kernels.

4.1. Develop a script to load the “camera” image from a file, define the desired kernel as a string constant (e.g., `kernel='average'`), select a kernel array with a `switch` statement, and then display the original image and the convolved image side by side; use `conv3x3` as before. Use the kernel name as the figure title, i.e, `title(kernel,'FontSize',16)`. The `switch` statement will select from among the following kernels:

`average`

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \times \frac{1}{9}$$

`hedges`

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

`vedges`

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

`edges`

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

`Laplacian`

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

`sharpen`

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

`pass`

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Enter the kernel as a 3x3 matrix using line continuation, e.g., enter the Laplacian kernel like this:

```
h = [ 0 -1  0; ...
     -1  4 -1; ...
      0 -1  0];
```

4.2. For each kernel, screenclip your image input/output pair and then write some comments about its effect on the processed image:

(a) `pass`

(b) `average`

(c) `hedges` – note that `hedges` is a type of edge detector, can you determine the meaning of the “h” in the name? Also, what happens to the constant-intensity areas of the image?

(d) `vedges` – how does this compare to the `hedges` kernel, and what might the “v” signify?

(e) `edges` – compare this result to the `hedges` and `vedges` kernels.

(f) `laplacian` – compare to `edges`.

(g) `sharpen`– this kernel can be considered as a combination of two of the other kernels; which are the two, and how are they combined? How might this explain the visual appearance of the processed image? Try the “Zoom In” tool in the area of an edge.

4.3. Copy and paste your finished and documented script text.

5. *Learn how to use imfilter:*

MATLAB includes the function `imfilter` that operates just like your `conv3x3` filter but with extended capabilities for arbitrary window sizes, several options to deal with boundary effects, and the ability to work with color images. Note that `imfilter` can also work directly with the `uint8` data type.

5.1. Create a new script for this part. Develop a code fragment to create the "pass-through" kernel of an arbitrary size of `N`x`N` elements that are all zero except for a single "1" in the center of the kernel. Assume that `N` will always be chosen as an odd number.

(a) Copy and paste your code fragment.

(b) Copy and paste the workspace output produced by your code for `N` = 3, 5, and 7. (To save space, run the command **`format compact`** before producing your output.)

5.2. Load the "parrots" image as variable `x`, display the image, create a 41×41 pass-through kernel `h`, and then apply this kernel to the image:

```
imshow(imfilter(x,h))
```

Expect to see the original image; can you explain why?

5.3. `imfilter` loops over *every* pixel in the input image instead of avoiding the boundary as you did with `conv3x3`. You can adjust how `imfilter` deals with the boundary, but begin with the `'full'` option to cause `imfilter` to show the original image *and* the values that it assumes are outside the limits of the input image based on the size of the kernel; the image you see is *larger* than the original image:

```
imshow(imfilter(x,h,'full'))
```

Explain what you see; what numerical value is assumed? If in doubt on the value, click the "Data Cursor" tool on the figure and then click on the image.

5.4. The abrupt change between the original pixel values and the assumed values can sometimes cause undesirable effects. Change the kernel to a 41×41 averager:

```
avg=ones(41)/41^2;
imshow(imfilter(x,avg,'full'))
```

and then try reverting back to the default output style which is the same size as the input image:

```
imshow(imfilter(x,avg))
```

Explain what you see, recalling that this kernel computes the average value of *all* the pixels in the 2-D moving window.

5.5. Try adding the `'circular'` boundary option that interprets the image as an infinitely repeating (periodic) image:

```
imshow(imfilter(x,h,'full','circular'))
```

Now try the averaging kernel again where the output image is the same size as the input:

```
imshow(imfilter(x,avg,'full','circular'))
```

What is your opinion of this choice for boundary handling, at least for the "parrots" image? What would have to be true about an image for `'circular'` to be a good choice?

5.6. Now try the `'replicate'` boundary option two ways as in the previous step:

```
imshow(imfilter(x,h,'full','replicate'))
```

and then

```
imshow(imfilter(x,avg,'replicate'))
```

Explain how the `'replicate'` option creates pixel values beyond the limits of the original image. How do the results compare to the `'circular'` method? Do you think that `'replicate'` gives better results at the boundaries?

5.7. Now try the remaining `'symmetric'` boundary option, again two ways as before:

```
imshow(imfilter(x,h,'full','symmetric'))
```

and then

```
imshow(imfilter(x,avg,'symmetric'))
```

Explain how the `'symmetric'` option creates pixel values outside the limits of the original image. Try the averaging kernel again; how do the results compare to the previous methods? Which method of the four available methods do you think would be the best choice for *most* images?

5.8. Create a modified form of your script from Part 4 that uses `imfilter` instead of `conv3x3`. Try all of the 3×3 kernels on the "parrot" image. Pick your favorite result, screenclip your input/output image results, state the kernel you used, and explain what appeals to you about this result.

6. *Wrap Up:*

6.1. Congratulations, you have now completed half of the lab projects for the term! Please comment on your comfort level with MATLAB. Does any aspect of MATLAB programming remain particularly unclear or confusing? If so, please elaborate.

6.2. Submit to Gradescope by the due date.