Laboratory # 5

FIR Filtering

5.1 Introduction

Filters are one of the most important tools that signal processors have to modify and improve signals. Part of their importance is their simplicity. In the days when analog signal processing was the norm, almost all filtering was accomplished with simple RLC circuits. Now, a great deal of filtering is accomplished digitally with simple (and extremely fast) routines that can run on special digital signal processing hardware or on general purpose microprocessors. In this lab, we will investigate a class of filters called FIR filters.

5.1.1 Implementing FIR Filters

When we apply a filter to a signal, we are performing an operation called *convolution* between the input signal and the filter's *impulse response*. We can think of the impulse response as a signal that characterizes the behavior of the filter. Convolution between the signal, x, and the impulse response, h, can be defined as

$$y[n] = \sum_{k=0}^{M} h[k]x[n-k],$$

where M is order of the filter with impulse response h and y is the result of the convolution operation.

In this lab, we will examine and implement a "real-time" algorithm for performing convolution. This is the same algorithm used to perform filtering in general purpose DSP hardware. The algorithm has several benefits; the primary benefit is that is easy to understand. In this algorithm, we imagine the filter as a box into which we drop one sample of an input signal and a corresponding sample of the output signal comes out. This allows the algorithm to be used in real-time: as samples of our signal arrive (from a microphone or some other source), we can immediately output filtered samples to be played over a speaker (for instance) with almost no delay.

The algorithm goes like this:

1. Initialize an *input buffer*, an array with length equal to the length of h, to all zeros.
2. For each sample that comes in:
   (a) Update the buffer by doing the following:
      i. Discard the sample at the end of the buffer.
      ii. Shift the rest of the samples one place towards the end of the buffer.
      iii. Insert the incoming sample into the front of the buffer.

(b) Initialize a running sum variable to zero.

(c) For each position, n, in the buffer:

i. Multiply the n^{th} position in the input buffer by the n^{th} position in h.

ii. Add the result to the running sum.

(d) Output the running sum as the next sample of the output signal.

In problem #1, you'll be asked to complete an implementation of this algorithm. Note that significant portions of this algorithm can be implemented very simply in MATLAB. For instance, all of (a) can be accomplished using a single line of code. Similarly, parts (b) through (d) can all be accomplished in a single line using one of MATLAB's built-in functions and its vector arithmetic capabilities.

5.1.2 Filtering in MATLAB

Convolution is a fairly computation-intensive operation. (For the CE's in the group, it's time complexity is $O(NM)$, where N and M are the lengths of the two signals being convolved). Thankfully, MATLAB has some efficient implementations of convolution. The one we'll use throughout the course is `filter`. If we have an FIR filter with impulse response `h` and a signal `x`, we filter `x` using the command

```
y = filter(h,1,x);
```

(We'll use the second parameter of `filter` later, when we study IIR filters). Sometimes we may also refer to the impulse response as a set of coefficients, B. You will see this notation in `help filter`.

We will also be filtering images in this lab. Though we can define two dimensional versions of convolution and impulse responses to do this, we will limit ourselves to applying a one-dimensional filter once in each dimension. Given an image `img`, we can apply a filter `h` in both dimensions using the command

```
out_img = filter(h,1,filter(h,1,img)')';
```

Note that `filter` operates over columns by default. Here, we filter along the columns first, transpose the result, filter along columns of the transposed image, and transpose a second time to restore the original orientation. This has the effect of filtering the image in both dimensions.

One potential problem with the `filter` command is the introduction of delay between the original and filtered signals. This results in edge effects at the beginning of the signal. For images, this delay produces a dark band along two edges of the image. The delay can introduce additional error if we attempt to calculate the RMS error between the original and filtered signals. When filtering images with impulse response `h`, we can correct for this delay by using the `conv2` command, which reindexes the output signal appropriately:

```
out_img = conv2(h,h,img,'same');
```

This filters the image in both dimensions using the impulse response `h`, but it takes the central part of the resulting convolution. This distributes the edge effects equally around all edges of the image, and prevents the introduction of an unwanted shift in the image. Note that by doing this, we are actually implementing a non-causal filter offline.

5.1.3 Image processing with FIR filters

If you've ever used photo editing software like Adobe Photoshop, you may have seen operations called "smoothing" and "sharpening". These operations can be implemented using simple FIR filters. We will examine FIR filters that perform both of these operations, examining their operation on one-dimensional signals and on images.

By now it should come as no surprise that we can implement something called "smoothing" using a simple moving average filter. You probably already have some intuition about this filter's operation. Along with making images look "fuzzier," smoothing filters are also good at reducing certain types of noise. We'll examine this property in this lab.

How do we implement "sharpening"? We can use simple FIR filters to do this, too. Sharpening filters exploit the way that we perceive images. We typically judge the "sharpness" of an image by the prominence of edges and contours. A sharpening filter works by making edges appear more abrupt (or sharp). In this lab you'll see just how these filters do that.

Another application of filtering is noise reduction, both in images and on one-dimensional signals. It turns out that smoothing filters will often do a decent job of noise reduction. However, there is a tradeoff between the noise reduction and the loss of signal quality. We can quantify this tradeoff using RMS error, as we did in Lab #4. In addition to using RMS error, it is also useful to qualitatively judge the resulting image quality. For instance, noise is most easily seen in uniform regions of a signal or image, so we judge the amount of noise reduction based on the uniformity of these regions. Smoothing filters typically distort a signal the most near edges in an image. When examining the perceptual effect of noise reduction, we must take both of these factors into account. In this lab, we will examine some noise reduction techniques using a simple gaussian filter with adjustable width.

5.2 Demonstrations in the Lab Section

1. The real-time convolution algorithm.
2. Filtering in MATLAB.
3. FIR filters for noise reduction
4. FIR filters for upsampling
5. Image processing with FIR filters
6. Noise reduction on images
7. **Review:** Transform Coding

5.3 Laboratory Assignment

NOTE: Remember to include Problem 4 from Lab 4 with your Lab 5 submission!

1. **Implementing Convolution.** Download the function `my_filt` from the course web page. This function is a nearly complete implementation of the real-time convolution algorithm described in the introduction. Your task is to complete the function by replacing the two question marks with code that completes the algorithm. There are simple, one-line additions that will complete the function correctly. These would be preferred, but if you must use multiple lines, you may.

    - [10] Make sure you include the source code for your modified version of `my_filt.m` in your report.

- [1] To test the operation of your code, execute the following command:

```
y = my_filt([3, 6, 10, 6, 3],[1, -2, 2, -4, 0, 0, -1, 3, 0, 0]);
```

  You should get the following result:

```
[3,  0,  4, -14, -13, -34, -21,  -9,   8,  24]
```

- [2] Now, execute the following command:

```
y = my_filt([2, -1, 3],[2, -2, 1, -1, 4, -4, 0 -1]);
```

  What result do you get?

2. **Smoothing Filters.** In this problem, we will make use of the simple 5-point smoothing filter with impulse response given by:

```
h1 = [.2, .2, .2, .2, .2];
```

   Note that in this and all subsequent problems, you should use `filter` or `conv2` (as illustrated in the background section) rather than `my_filt`.

   (a) First, let's look at the operation of `h1` on an artificial edge. Define the following signal:

```
signal = [zeros(20,1); ones(20,1)];
```

   Use `filter` to filter `signal` with the filter defined by `h1`. Plot `signal` and the filtered signal on the same set of axes. Make sure you use appropriate line styles so that the two signals can be distinguished (i.e., use solid and dotted lines). To see `signal` clearly, you may need to adjust the axis extents using the command `axis`.

   - [3] Include this plot in your report.
   - [2] Describe the effects of the filter on this simple signal.

   (b) Load the "cameraman" image that we used in Lab #4 and extract a single row of the image using the following commands:

```
cm = double(imread('cameraman.tif'));
one_row = cm(125,:);
```

   Use `filter` to filter `one_row` with the filter defined by `h1`. In a new figure, plot `one_row` and the filtered version on the same axes, again making sure that the signals are distinguishable.

   - [3] Include this plot in your report.
   - [2] Consider the original signal to be "noiseless" and the filtered signal to be "noisy." Calculate the RMS error introduced by this filter. (Hint: do this the same way as in Lab 4. Note the delay introduced by the filtering operation.)

   (c) Now we want to see the effects of this filter on the entire two-dimensional image. Using the method for two-dimensional filtering with `conv2` described in the introduction, filter `cm` with `h1`. Use subplot to display the original image and the filtered image side by side in a single figure. (Use the same image display conventions we used in Lab #4).

   - [4] Include this figure in your report.
   - [2] Calculate the RMS error introduced by this filtering.
   - [2] Describe the effect of the filter on this image. Especially note what happens in roughly constant regions of the image and around edges in the image.

   (d) Create a noisy version of "cameraman" using the following command:

```
noisy = cm + 20*randn(size(cm));
```

Download the M-file `g_smooth.m`. This function produces filters with gaussian (i.e., "bell-curve") impulse responses. The only parameter is a width parameter that determines the strength of the smoothing effect. The width parameter can be any positive real number; a width of zero produces a filter that has no effect (i.e., it is a unit impulse).

Use the `conv2` command to filter the noisy image with various filters produced by `g_smooth.m`. Calculate the RMS error of each filtered image. (Hint: write a function that uses `g_smooth` to generate a filter with variable width, performs the filtering, displays the filtered image, and calculates the RMS error so that you can check a lot of different width values quickly. Make sure you include your MATLAB code).

- [1] Calculate the RMS error for a filter width of 0 (i.e., no filtering). What is this RMS error?
- [4] Find the width value that gives you the smallest RMS error. What width value did you find? What is the corresponding RMS error?
- [4] Find the width value that performs the best perceptually. That is, find the filter produces the "best looking" image based on noise reduction and edge smoothing. Which width value did you choose? What is the corresponding RMS error?

3. **Sharpening filters.** The behavior of the sharpening filter is somewhat less intuitive than that of the smoothing filter. In this problem, we will examine the behavior of the following sharpening filter:

```
h2 = [-0.5, 2, -0.5];
```

(a) Return to the artificial edge signal, `signal`, that we defined in Problem #2. Filter `signal` with `h2`. Plot the original and the filtered result on the same plot, as you did in Problem #2.

- [3] Include this plot in your report.
- [2] Describe the effect of the filter on this signal. What is the filter doing that might help to make the edge appear "sharper"?

(b) Now, use `filter` to filter the 125^{th} row of "cameraman" (the one that you extracted in Problem #2) using `h2`. Plot the signal `one_row` and the filtered version on the same axes.

- [3] Include this plot in your report.
- [2] Calculate the RMS error introduced by the filter in this signal. (Again, there is a slight delay introduced by the filtering operation.)

(c) Finally, we'll look at the results of the sharpening filter on the image itself. Filter the image, `cm`, in both directions with `h2` using `conv2`. In order to normalize the display, we also need to clip the image values of the filtered image to the range 0 to 255. Do this with the command

```
clipped_img =  max(0,min(filtered_img,255));
```

Display the original image and the filtered-and-clipped image side by side in the same figure.

- [4] Include this figure in your report.
- [2] Calculate the RMS error introduced by the filter in this image.

- [2] Describe the effects of the filter on this image. Especially note what the filter does in the vicinity of edges and in relatively constant regions. (You should zoom in on the image to see the effects close-up.) What features of the filtered image contribute to it's "sharper" appearance?