Laboratory # 6

FIR Filtering and Image Processing

6.1 Introduction

Digital filters are one of the most important tools that signal processors have to modify and improve signals. Part of their importance comes from their simplicity. In the days when analog signal processing was the norm, almost all filtering was accomplished with 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 processors.

So *why* do we filter signals? There are many reasons. One of the biggest is *noise reduction* (which we have called *signal recovery*). If our signal has undesirable frequency components, e.g. it contains noise in a frequency range where there is little or no desired signal, then we can use filters to reduce the relative amplitude of the signal at such frequencies. Such filters are often called *frequency blocking filters*, because they block signal components at certain frequencies. For example, *lowpass filters* block high frequency signal components, *highpass filters* block low frequency signal components, and *bandpass filters* block all frequencies except those in some particular range (or band) of frequencies.

There are a wide range of uses for filtering in image processing. For example, they can be used to improve the appearance of an image. For instance, if the image has granular noise, we might want to *smooth* or *blur* the image to remove such. Typically such noise has components at all frequencies, whereas the desired image has components at low and middle frequencies. The smoothing acts as a lowpass filter to reduce the high frequency components, which come, predominantly, from the noise. Alternatively, we might want to *sharpen* the image to make its edges stand out more. This requires a kind of highpass filter.

In this lab, we will experiment with a class of filters called FIR (*finite impulse response*) filters. FIR filters are simple to implement and work with. In fact, an FIR filtering operation is almost identical to the operation of *running correlation* which you have worked with in Laboratory #2. In particular, we will examine the use of FIR filters for image processing, including both smoothing and sharpening. We will also examine their use on simple one-dimensional signals.

6.1.1 "The Question"

- How do we implement FIR filters in MATLAB?
- How can we improve the appearance of an image? Specifically, how can we remove noise or "sharpen" an image?

6.2 Background

6.2.1 Implementing FIR Filters

FIR filters are systems that we apply to signals. An FIR filter takes an input signal $x[n]$, modifies it by the application of a mathematical rule, and produces an output signal $y[n]$. This rule is generally called a *difference equation*, and it tells us how to compute each sample of the output signal $y[n]$ as a weighted sum of samples of the input signal $x[n]$. A common form of the difference equation is given as

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] \tag{6.1}$$

$$= b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + \ldots + b_M x[n-M] \tag{6.2}$$

The b_k's are called the *FIR filter coefficients*, and M is the *order* of the FIR filter. The set of FIR filter coefficients completely specifies an FIR filter. Different choices of the order and the coefficients leads to different kinds of filters, e.g. to lowpass, highpass and bandpass filters.

Equation (6.1) defines the class of *causal* FIR filters. A more general form is given by

$$y[n] = \sum_{k=-M_1}^{M_2} b_k x[n-k] \tag{6.3}$$

$$= b_{-M_1} x[n+M_1] + \ldots + b_{-1} x[n+1] + b_0 x[n] + b_1 x[n-1] + \ldots + b_{M_2} x[n-M_2] \ , \tag{6.4}$$

where M_1 and M_2 are nonnegative integers. Here, the order of the filter is $M_1 + M_2$. When $M_1 > 0$, the FIR filter is *non-causal*. To calculate the "present" value of $y[n_0]$, a causal FIR filter only requires "present" ($n = n_0$) and "past" ($n < n_0$) values of $x[n]$. Non-causal filters, on the other hand, require "future" ($n > n_0$) values of $x[n]$. Thus, a filter with difference equation given by $y[n] = x[n] + x[n-1]$ is causal, but a filter with difference equation given by $y[n] = x[n] + x[n+1]$ is non-causal. The distinction between causal and non-causal filters is necessary if we wish to implement one of these filters in real-time. Causal filters can be implemented in real-time, but to implement non-causal filters we generally need all of the data for a signal before we can filter it.

Compare equation (6.3) with the equation for performing running correlation between a signal $b[n]$ and $x[n]$:

$$y[n] = C(b[k], x[k-n]) = \sum_{k=-\infty}^{\infty} b[k] x[k-n]. \tag{6.5}$$

Recall that we thought of running correlation as a procedure where we "slid" one signal across the other, calculating the in-place correlation at each step. If we consider that the

b_k's of an FIR filter form a signal, then the application of an FIR filter uses the same procedure with two minor differences. First, when we apply an FIR filter, we are only "correlating" over a finite range; however, we typically assume $b_k = 0$ for k outside the range $[M_1, M_2]$. Thus, we can change the limits of summation to range over $(-\infty, \infty)$ without changing the result. Second, when applying a filter, the signal $x[n]$ is time-reversed with respect to the b_k coefficients[1]. This is not the case for running correlation.

From the definition alone, it is not easy to see how a filter "works." With the connection to correlation, though, we can suggest an intuitive graphical understanding of this process. To calculate a single sample of $y[n]$, we time-reverse the signal formed by the b_k coefficients (by flipping it across the $n = 0$ axis). Then, we shift this time-reversed signal by n samples and perform in-place correlation. The result is the n^{th} sample of $y[n]$. To build up the entire signal $y[n]$, we do this repeatedly, "sliding" one signal across the other and calculating in-place correlations at each point.

You may find it useful to go back to Lab #2 and review the algorithm for in-place correlation. In that description of the algorithm, we used $x[n]$ where here we wish to use the signal formed by the b_k's. We can use this algorithm when implementing FIR filters, as well. Note, however, that we want to time-reverse the b_k coefficients when we multiply them by the incoming signal samples. That is, we always want to multiply the b_{-M_1} coefficient by the newest sample in the buffer.

6.2.2 Edge effects and delay

Suppose that we consider filtering a signal, $x[n]$, with a causal filter whose difference equation is given by

$$y[n] = \frac{1}{5}x[n] + \frac{1}{5}x[n-1] + \frac{1}{5}x[n-2] + \frac{1}{5}x[n-3] + \frac{1}{5}x[n-4]. \tag{6.6}$$

That is, $b_k = (\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5})$, $M_1 = 0$, and $M_2 = 4$. We can think of the operation of this filter as replacing each sample of $x[n]$ with the average of that sample and the past four samples. As such, we often call filters like this *moving-average filters*. The result of this filtering for a particular signal is shown in Figure 6.1.

In this particular case, $x[n]$ has a support interval of $[0, 28]$ and is zero outside of this range. Consider what happens to the output signal, $y[n]$, near the edges of this range. First, the output sample at $y[0]$ will be dominated by zeros from outside of the support interval, because

$$y[0] = \frac{1}{5}x[n] + \frac{1}{5}0 + \frac{1}{5}0 + \frac{1}{5}0 + \frac{1}{5}0 \ . \tag{6.7}$$

Similarly, $y[1]$, $y[2]$, $y[3]$, and $y[4]$ will also be affected by these zeros, but to a lesser extent. This effect can be seen in Figure 6.1 as $y[n]$ "ramps up" to the nominal values of $x[n]$. This effect is known as a *start-up transient*. A similar effect occurs beyond $y[28]$, where the signal takes a few samples to "die off". This effect is known as an *ending transient*. Both of these transients are known as *edge effects*, and need to be considered when filtering.

What do the edge effects do to the support length of the output signal? Well, from Figure 6.1 we can see that $y[n]$ has a support length four samples longer than that of $x[n]$. In general, the length of the output signal which is non-zero will be equal to the length of the input signal plus the order of the FIR filter.

[1] That is, $x[n-k]$ is a time-reversed version of $x[k-n]$, just as $s[-n]$ is a time-reversed version of $s[n]$. Note that we can "time-reverse" the b_k coefficients rather than $x[n]$ and achieve the same result.

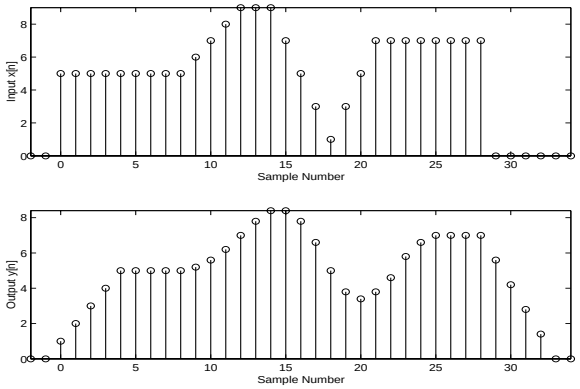

Figure 6.1: Input and output of a 5-point moving average filter.

There is one additional point that should be examined. Look at the location of the "dip" in Figure 6.1. In $x[n]$, the "dip" occurs at sample 18, but in $y[n]$ it occurs at sample 20. In fact, the entire support interval of $y[n]$ has not only gotten larger, it has also been shifted over to the right (or *delayed*) by two samples. Why is this? The delay introduced by this filter results from the fact that each output sample is an average of samples to its left. If we instead define this filter so that it considers two samples to both the right and left, we can eliminate this delay. That is, we define a different difference equation:

$$y[n] = \frac{1}{5}x[n+2] + \frac{1}{5}x[n+1] + \frac{1}{5}x[n] + \frac{1}{5}x[n-1] + \frac{1}{5}x[n-2]. \tag{6.8}$$

This modification, though, has taken a causal filter and made it non-causal.

Delay is a common problem for causal filters. In fact, the only causal filter that does not introduce delay is a zero-order amplifier system with a difference equation $y[n] = b_0 x[n]$. This system changes the amplitude of a signal, but does nothing else. Compare this to the system with difference equation $y[n] = x[n-N]$. This system's only effect is to delay the signal by N samples. In some circumstances, the delay introduced by a causal filter does not affect the operation of the system. For our purposes in this laboratory, we will need to be careful to account for the delay introduced by FIR filters when comparing two signals with a mean-squared or RMS distortion measure.

6.2.3 Noise and distortion

One of the most common reasons to apply a filter is to attempt to remove *noise*. There is no single definition of noise, but the most general usage describes noise as any unwanted component of a signal. For instance, a common type of electrical noise has a sinusoidal characteristic with a frequency of 60 Hz. This noise arises from the frequency of the alternating current used to distribute electricity. This 60 Hz signal can "leak" into other systems and corrupt sensor measurements. Another common type of noise is *random noise*. This sort of noise typically has a jagged-looking characteristic. It typically manifests itself as static in audio signals and "snow" in images.

Filtering gives us a means of reducing the noise in a signal through *frequency blocking*. In general, filters operate by attenuating (i.e., blocking) certain frequencies in a signal while passing others with relatively little attenuation. Note that removing noise in this

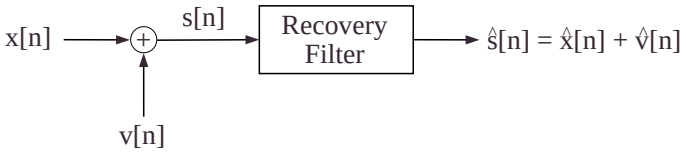

Figure 6.2: A block diagram of additive noise and a recovery filter that attempts to remove the noise.

way requires the noise to have a different frequency-domain description than the signal of interest.

For instance, consider the example of 60 Hz sinusoidal noise. If our signal of interest is composed of frequencies above 60 Hz, we can treat this component as low frequency noise and attempt to remove it with a filter that blocks low frequencies. This sort of filter is generally called a *highpass filter*. If our signal has components above and below 60 Hz, we might try to remove the corrupting signal by only eliminating frequencies near 60 Hz. This would require a *bandpass filter*

Random noise typically has frequency components all over the spectrum. However, a good portion of these components will usually have higher frequency than the frequencies in our signal. Thus, we might consider the application of a *lowpass filter* that blocks high frequencies to reduce the amplitude of noise components.

Consider the block diagram in Figure 6.2. This is a model where a signal of interest, $x[n]$, is corrupted by the addition of a noise signal, $v[n]$. We apply a *recovery filter* to try to remove the noise component from $s[n] = x[n] + v[n]$. The resulting signal is $\hat{s}[n] = \hat{x}[n] + \hat{v}[n]$, where $\hat{x}[n]$ is the filtered signal of interest (which we hope will be as similar to $x[n]$ as possible) and $\hat{v}[n]$ is the filtered noise signal (which we hope will be as small as possible). Often we can tune the noise-removal filter to increase it's "strength" (by, for instance, increasing the length of a moving average filter). The "stronger" the filter, the more noise we can eliminate. Unfortunately, the filter also distorts the signal of interest; a stronger filter will distort the signal of interest more. Thus, the use of filters to remove noise can be thought of as finding a tradeoff between two types of distortion. The goal, then, is to find the point where the total distortion (as measured by the mean-squared error or RMS error between $x[n]$ and $\hat{s}[n]$) is minimized as a function of filter strength.

Advanced noise-reduction techniques

While standard FIR filters can be useful for noise reduction, in some cases we may find that they distort the desired signal too much. An alternative is to use *nonlinear* filters. Nonlinear filters have the potential to remove more noise while introducing less distortion to the desired signal; however, the effects of these filters are much more difficult to analyze.

Consider the case of an image, for instance. One of the most important features of images of natural scenes are *edges*. Edges in images are usually just sharp transitions where one object ends and another begins. If we are attempting to remove high-frequency noise from an image, we will often apply a lowpass filter. Edges, though, have considerable high-frequency content, so the edges in resulting image will be smoothed out. To get around this problem, we can consider the application of a common nonlinear filter called a *median filter*. Median filters replace each sample of a signal with the median (i.e., the most central value) of a block of samples around the original sample. That is, we can describe the operation of

the median filter as

$$y[n] = Median(x[n + M_1], \ldots, x[n], \ldots, x[n - M_2]) \tag{6.9}$$

where

$$Median(x_1, \ldots, x_N) = \begin{cases} x_{((N+1)/2)} & N \text{ odd} \\ \frac{1}{2}(x_{(N/2)} + x_{(N/2+1)}) & N \text{ even} \end{cases} \tag{6.10}$$

and where $x_{(n)}$ is the n^{th} smallest of the values x_1 through x_N. The *order* of the median filter is given by $M_1 + M_2$, and it determines how many samples will be included in the median calculation. Unlike lowpass filters, median filters tend to preserve edges in signals very well. These filters are also very powerful for removing certain types of noise while introducing relatively little distortion. In this laboratory, we will examine the effect of applying nonlinear filters to two-dimensional signals.

6.2.4 Filtering two-dimensional signals

The above discussions of filtering are for one-dimensional signals. Suppose we would like to filter two-dimensional signals like images instead of just one-dimensional signals. There are three ways to approach this.

The first method simply applies one-dimensional filters to each of the rows (or each of the columns) of an image. This method tends to produce an "uneven" filtered signal that is, for instance, smoothed in one dimension but not the other. This unevenness is generally not desirable and motivates a second method.

The second method is somewhat "stronger" than the first. This method applies one-dimensional filters to *both* the rows and the columns. In this lab we will adopt the convention that we first filter the columns, and then filter the rows of the resulting image. Most types of filters that we use in one dimension can be extended to two dimensions in this fashion. For example, if we apply the moving average filter with difference equation give in equation (6.6), for instance, this will have the effect of smoothing the image. Note that the edge effects and delay issues discussed earlier also apply to two-dimensional filtering done in this fashion.

If we apply that moving average filter with order 4 to the columns and then the rows of the image, what is the mathematical effect of the operation? It is not too difficult to see that each sample of the image has been replaced by the average of a 5×5 block of pixels. This suggests that we could describe this filtering operation in terms of two-dimensional set of filtering coefficients. For instance, the difference equation for this two-dimensional filter would be

$$y[m, n] = \sum_{k=0}^{4} \sum_{l=0}^{4} \frac{1}{25} x[m - k, n - l]. \tag{6.11}$$

This operation is equivalent to filtering with a two-dimensional set of coefficients $b_{k,l}$, where $b_{k,l} = \frac{1}{25}$ for $k = 0, \ldots, 4$ and $l = 0, \ldots, 4$. A schematic of these filter coefficients being used on an image is shown in Figure 6.3.

This result suggests the third, most general method of applying FIR filters to two-dimensional signals. The general difference equation for this method is

$$y[m, n] = \sum_{k=-M_1}^{M_2} \sum_{l=-N_1}^{N_2} b_{k,l} x[m - k, n - l]. \tag{6.12}$$

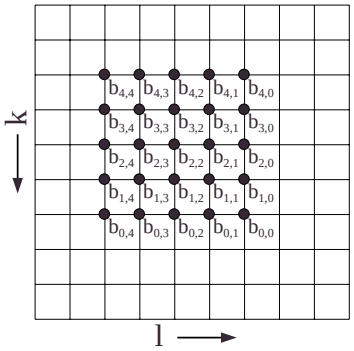

Figure 6.3: The coefficients of a two-dimensional moving average filter. In this figure, pixels exist at the intersection of the horizontal and vertical lines.

$[-M_1, M_2]$ and $[-N_1, N_2]$ define the range of nonzero coefficients. Note that in image processing, causality is rarely important. Thus, two-dimensional FIR filters typically have coefficients centered around $b_{0,0}$.

6.2.5 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 and many similar operations are typically implemented using simple two-dimensional FIR filters. We will consider three such operations in this laboratory: smoothing (or *blur*), *edge finding*, and sharpening (or *edge enhancement*)

We've already suggested that a moving average filter performs a smoothing operation. However, there are more advanced ways of smoothing. Consider, for instance, a filter that weights samples nearby more strongly than those that are far away. This performs a "weaker" smoothing, but it also introduces less distortion. Because of this, these sorts of filters are often more useful for random noise reduction than standard moving average filters.

The "edge finding" filter highlights edges in an image (by producing large positive or negative values) while setting constant regions of the image to zero. The most basic edge finding filter is a simple one-dimensional *first difference* filter. A first difference filter has the difference equation

$$y[n] = x[n] - x[n-1]. \tag{6.13}$$

This filter will tend to respond positively to increases in the signal and negatively to decreases in the signal. Adjacent input samples that are identical (or nearly so), though, will tend to cancel one another, causing the output to be zero (or close to zero). There are various two-dimensional "equivalents" of the first-difference filter, many of which respond to edges of a particular orientation. One general edge-finding filter has the following difference equation:

$$\begin{aligned} y[m,n] = \; & \tfrac{1}{4}x[m+1,n+1] - x[m+1,n] + \tfrac{1}{4}x[m+1,n-1] \\ & - x[m,n+1] + 3x[m,n] - x[m,n-1] \\ & + \tfrac{1}{4}x[m-1,n+1] - x[m-1,n] + \tfrac{1}{4}x[m-1,n-1] \end{aligned} \tag{6.14}$$

This filter "finds" edges of almost any orientation by outputting a value with large magnitude wherever an edge occurs. Both the first difference filter and this general edge-finding filter are examples of highpass filters. Note the "oscillatory" pattern of b_k values such that adjacent coefficients are negatives of one another. This pattern is characteristic of highpass filters. Note that both of these filters will typically produce both positive and negative values, even if our input signal is strictly non-negative. Also note that for both of these filters, the average of the b_k coefficients is zero; this means that these filters tend to "reject" constant regions of an input signal by setting them to zero.

The third operation, sharpening, makes use of an edge finding filter as well. Basically, the sharpening filter produces a weighted sum of the output of an edge-finding filter and the original image. Suppose that $x[m,n]$ is the original image, and $y[m,n]$ is the result of filtering $x[m,n]$ with the filter defined in equation (6.14). Then, the result of sharpening, $z[m,n]$ is given by

$$z[m,n] = x[m,n] + by[m,n], \tag{6.15}$$

where b controls the amount of sharpening; higher values of b produce a "sharper" image. Note that $z[m,n]$ can also be viewed as the output of a single filter. For display purposes, we will *threshold* the resulting signal so that the output image has the same range of data values as the input image. That is, assuming that our input image has values between 0 and 255, the final output of the sharpening operation, $\hat{z}[m,n]$ will be

$$\hat{z}[m,n] = \begin{cases} 0 & z[m,n] < 0 \\ z[m,n] & 0 \le z[m,n] \le 255 \\ 255 & 255 < z[m,n] \end{cases} \tag{6.16}$$

Note that thresholding is a *nonlinear* operation, but it is not crucial to the sharpening process. This final result can also be considered to be the output of a single nonlinear filter.

Sharpening is a useful operation when an image has undergone an undesired smoothing operation. This happens frequently in optical systems when they are not entirely in focus. Unlike smoothing filters, though, sharpening filters tend to enhance random noise; often they may make "noise-like" components of a signal visible where they were not visible before.

6.3 Some MATLAB commands for this lab

- **1-D Filtering in MATLAB:** The usual method for causal filtering in MATLAB is to use the `filter` command, like this:

  ```
  >> yy = filter(bb,1,xx);
  ```

  (We'll use the second parameter later in the course when we study IIR filters.) `xx` is a vector containing the discrete-time input signal to be filtered, `bb` is a vector of the b_k filter coefficients, and `yy` is the output signal. The first element of this vector, `bb(1)`, is assumed to be b_0.

  By default, `filter` returns a portion of the filtered signal equal in length to `xx`. Specifically, the resulting signal includes the start-up transient but not the ending transient. This means that the output will be delayed by an amount determined by the coefficients of the filter.

  A method for filtering which does not introduce delay is often desirable, i.e. a non-causal filtering method, especially when calculating RMS error between filtered and

original versions of a signal. The command `filter2` is meant as a two-dimensional filtering routine, but it can be used for 1-D filtering as well. Further, it can be instructed to return a "delay-free" version of the output signal. When using `filter2`, it is important that `xx` and `bb` are either both row vectors or both column vectors. Then, we use the command

```
>> yy = filter2(bb,xx,'same');
```

where `xx` is the input signal vector, `yy` is the output signal vector, and `bb` is the vector of filter coefficients. If the length of the vector `bb` is odd, the b_0 coefficient is taken to be the coefficient in the center of `bb`. If the length of `bb` is even, b_0 is taken to be just left of the center. The output of `filter2` has support equal to that of the input signal `xx`.

Though we will not use these additional options, we can also have `filter2` return the full length of the filtered signal (the length of the input signal plus the order of the filter) like this:

```
>> yy = filter2(bb,xx,'full');
```

or just the portion not affected by edge effects (the length of the input signal minus twice the order of the filter), like this:

```
>> yy = filter2(bb,xx,'valid');
```

- **2-D Filtering in** MATLAB**:** There are several ways to filter 2-dimensional signals in MATLAB as well. We can use the one-dimensional filtering routines listed above, and filter along the rows and columns separately. If we wish to use a causal filter with b_k coefficients `bb` to filter both the rows and columns of an image `xx` we can use the command

  ```
  >> yy = filter(bb,1,filter(bb,1,xx)')';
  ```

  Since `filter` operates down the columns by default, we need to transpose the image after the first filtering and the transpose the final result again to restore the original orientation. This method will introduce edge effects at the top and on one side of the image; however the resulting image will be the same size as `x`.

  A more straightforward method simply uses the `filter2` command. If we have two-dimensional coefficients $b_{k,l}$ contained in a matrix `bb` and an image `xx`, we use the command

  ```
  >> yy = filter2(bb,xx,'same');
  ```

  The same alternate third parameters as listed in the 1-D filtering section apply here as well.

- **How to generate some filters:** We will be examining the effects of many types of filters in this laboratory. Some have filter coefficients that can be generated easily in MATLAB. Others require a function (which we will provide to you) to generate. Note that the the vectors representing the b_k's will be column vectors.

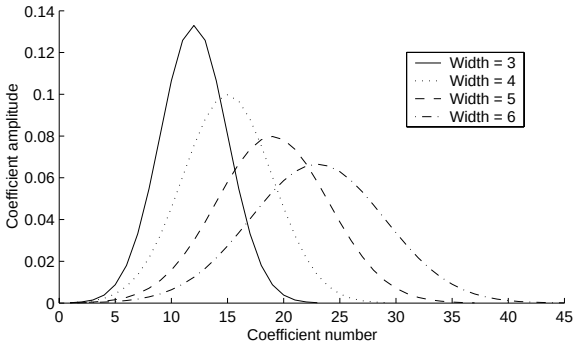

Figure 6.4: The coefficients for `g_smooth` filters with varying widths.

1. An *L-point moving average* filter has filter coefficients given by `bb = ones(L,1)/L`.
2. A *first-difference* filter has filter coefficients given by `bb = [1; -1]`;
3. The function `g_smooth` produces coefficients for a particular type of smoothing (lowpass) filters with easily tunable "strength". `g_smooth` takes a single real-valued parameter, which is the "width" of the filter, and returns the filter coefficients of the corresponding filter, `bb`. For example,

   ```
   >> bb = g_smooth(1.2);
   ```

   returns the coefficients for a filter with "width" 1.2. A `g_smooth` filter with width 0 will pass the input signal without modification, and higher widths will smooth more strongly. Good nominal values for the width range from 0.5 to 2. The b_k coefficients for `g_smooth` filters with several widths are plotted in Figure 6.4.
4. The function `g_smooth2` is the two-dimensional equivalent of `g_smooth`. It again takes a single input parameter (the width of the filter) and returns the two-dimensional set of filter coefficients of the corresponding filter. For example,

   ```
   >> bb = g_smooth(0.8);
   ```

   returns the coefficients of a filter with width 0.8.
5. In Section 6.2.5, we presented a general-purpose two-dimensional edge-finding filter in equation (6.14). The coefficients for this filter are given by

   ```
   >> bb = [.25, -1, .25; -1, 3, -1; .25, -1, .25];
   ```
6. In Section 6.2.5, we also discussed a method for implementing a sharpening filter. Since we include a threshold operation, this operation is nonlinear and cannot be accomplished using only an FIR filter. Thus, we provide the `sharpen` command, which takes an image and a sharpening "strength" and returns a sharpened image:

   ```
   >> yy = sharpen(xx,0.7);
   ```

   The second parameter is the strength factor, b, as discussed in section 6.2.5. A sharpening strength of 0 passes the signal without modification.
7. Median filters are a special type of nonlinear filter, and they cannot be described using linear difference equations. To use a median filter on a one-dimensional signal, we use the command[2] `medfilt1` like this:

[2] `medfilt1` is a part of the signal processing toolbox.


```
>> yy = medfilt1(xx,N);
```

`N` is the *order* of the median filter, which simply describes how many samples we consider when taking the median. In two dimensions[3], we use `medfilt1` twice:

```
>> yy = medfilt1(medfilt1(xx,N)',N)';
```

Again, `N` is the order of the median filter. Note that `medfilt1` operates down the columns by default, so we need to transpose the image between the filtering operations and again at the end.

6.4 Demonstrations in the Lab Section

- Filtering in MATLAB.
- FIR filters for noise reduction
- Image processing with FIR filters
- Median filtering

6.5 Laboratory Assignment

1. In this problem, you investigate noise-reduction on one-dimensional signals. Download the file `lab6_data.mat`, which contains various signals for this lab. In this problem, we will consider the signal `simple`, which is a noise-free one-dimensional signal, and `simple_noise`, which is the same signal with corrupting random noise.

   (a) First, we'll examine the delay introduced by the two filtering implementations, `filter` and `filter2`, that we will be using. Filter `simple` with a 7-point running average filter. Do this twice, first using `filter` and then using `filter2` with the `'same'` parameter[4].

   - [3] Use `subplot` and `plot` to plot the original signal and two filtered signals in three subplots of the same figure.
   - [2] One of the filtering commands has introduced some delay. Which one? How many samples of delay have been added?
   - [3] Compute the mean-squared error between the original signal and the two filtered signals. Which is lower? Why?

   (b) Now, use `filter2` to apply the same 7-point running average filter to the signal `simple_noise`. Referring to Figure 6.2, we consider `simple` to be the signal of interest $x[n]$, `simple_noise` to be the noise corrupted signal $s[n]$, and their difference to be the noise, $v[n] = s[n] - x[n]$. Note that the lower of the two mean-squared errors that you computed in Problem 1a is $MS(\hat{x}[n] - x[n])$, which is a measure of the distortion of the signal of interest introduced by the filter.

   - [2] Compute the mean-squared error between `simple` and `simple_noise`. Referring back to Figure 6.2, this is $MS(v[n])$, the mean-squared value of the noise.

[3] We can also use `medfilt2`, but this function is a part of the Image Processing Toolbox which we do not require for this course.

[4] Henceforth, every time you use `filter2` in this laboratory, you should use the `'same'` parameter.

- [2] Compute the mean-square error between your filtered signal and `simple`. This value is $MS(\hat{s}[n] - x[n])$, which is a measure of how a good a job the filter has done at recovering the signal of interest.
- [1] Determine the distortion due to noise at the output of your reconstruction filter (i.e., $MS(\hat{v}[n])$) by subtracting $MS(\hat{x}[n] - x[n])$ from $MS(\hat{s}[n] - x[n])$.
- [3] Compare $MS(\hat{v}[n])$ and $MS(\hat{s}[n] - x[n])$ to $MS(v[n])$. What is the dominant source of distortion in this filtered signal?

(c) Use `filter2` to apply a 3-point, a 5-point, and an 9-point moving average filter to `simple_noise`.

- [3] Use `plot` and `subplot` to plot the original signal, the three filtered signals, and the three sets of filter coefficients, in seven panels of the the same figure.
- [3] Compute the mean-squared error between each filtered signal and `simple`.
- [2] Which of the four moving average filters that you have applied has the lowest mean-squared error? Compare this value to $MS(v[n])$.

(d) Download the file `g_smooth.m`, and use it to generate filter coefficients with "widths" of 0.5, 0.75, and 1.0. (Note the lengths of the returned coefficient vectors. You should plot the filter coefficients to get a sense of how the "width" factor affects the them.) Use `filter2` to apply these filters to `simple_noise`.

- [3] Use `plot` and `subplot` to plot the three filtered signals and the three sets of coefficients in six panels of the same figure.
- [3] Compute the mean-squared error between each filtered signal and `signal`.
- [3] Which of these filtered signal has the lowest mean-squared error? Compare this value to the lowest mean-squared error that you found for the moving average filters and to $MS(v[n])$.

2. In this problem, we'll look at the effects applying smoothing filters to an image for noise reduction. Download the files `peppers.tif`[5] and `peppers_noise1.tif`. The first is a "noise-free" image, while the second is an image corrupted by random noise. Load these two images into MATLAB.

(a) We'll be using the function `g_smooth2` to produce filter coefficients for this problem. To get a sense of what these coefficients look like, generate the coefficients for a `g_smooth2` filter with width 5. In two side-by-side subplots of the same figure:

- [2] Display the coefficients as an image using `imagesc`.
- [1] Generate a surface plot of the coefficients using the command `surf(bb)` (assuming your coefficients matrix is called `bb`).

(b) First, we'll consider the noisy signal `peppers_noise1`.

- [3] Use `subplot` to display `peppers` and `peppers_noise1` side-by-side in a single figure. Remember to set the color map, set the axis shape, and include a colorbar as you did in lab #4.
- [3] Compute the mean-squared error between these two images.

[5] Like "cameraman", "peppers" is a standard image used for testing image processing routines. Our version, however, is smaller than the traditionally used image.

(c) Our goal is to find a `g_smooth2` reconstruction filter that minimizes the mean-squared error between the filtered image and the original, noise-free image. Use `filter2` when filtering signals in this problem.

- [6] Find a filter width that minimizes the mean-squared error. What is this filter width and the corresponding mean-squared error? You should be able to get within 1 of the true minimum value[6]. (Hint: you might want to plot the mean-squared error as a function of filter width.)
- [2] Display the filtered image with the smallest mean-squared error.
- [2] Look at some filtered images with different widths. Can you find one that looks better than the minimum mean-squared error image? What filter width produced that image?

3. Next, we'll look at methods of removing a different type of random noise from this image. Download the file `peppers_noise2.tif` and load it into MATLAB. This signal is corrupted with *salt and pepper* noise, which may result from a communication system that looses pixels.

(a) First, let's see what we're up against. Salt and pepper noise randomly replaces pixels with a value of either 0 or 255. In this image, one-fifth of the pixels have been lost in this manner.

- [2] Display `peppers_noise2`.
- [2] Compute the mean-squared error between this image and `peppers`.

(b) Now, let's try using some `g_smooth2` filters to eliminate this noise. Start by using `filter2` to filter `peppers_noise2` with a `g_smooth2` filter of width 1.3. Note that this is very close to the optimal width value.

- [2] Display the resulting image.
- [3] Compute the mean-squared error.

(c) Finally, let's use a median filter to try to remove this noise. Apply median filters of order 3 and 5 to `peppers_noise2`.

- [3] Use `subplot` to display the two filtered images side-by-side in the same figure.
- [3] Compute the mean-squared errors between the median-filtered signals and `peppers`.
- [2] Look at the filtered images and describe the distortion that the median filters introduce into the signal.
- [3] Compare the median filter to the `g_smooth2` filters. Discuss both measured distortion and the appearance of the filtered signals.

4. In this last problem, we'll look at edge-finding and sharpening filters.

(a) In order to see how edge-finding filters work, let's start in one dimension. Use `filter` to apply a one-dimensional first difference filter to the signal `simple` (which can be found in `lab6_data.mat`).

- [2] Plot the resulting signal.

[6] Though mean-squared error is widely used as a measure of signal distortion, it is well known that its judgments of quality do not always correspond closely to the eye's judgments of quality.

- [3] There are five non-zero "features" of this signal. (These features should be clear from the plot.) Describe them and what they correspond to in `simple`.

(b) Now we'd like to look at the effects of the general edge-finding filter presented in Section 6.2.5. Use `filter2` to apply this filter to `peppers`.

- [2] Display the resulting image.
- [2] Describe the resulting image.
- [2] Zoom in on the filtered image and examine some of the more prominent edges. What do you notice about these edges? (Hint: Are they just a "ridge" of a single color?)

(c) Download `sharpen.m` and use the function to display several sharpened versions of the `peppers` image.

- [3] Display sharpened image with a "strength" of 1 alongside the original `peppers` image using `subplot`.
- [2] Zoom in on this sharpened image. What makes it look "sharper"? (Hint: Again, look at the prominent edges of the images. What do you notice?)
- [2] The sharpened images (especially for strengths greater than 1) generally appear more "noisy" than the original image. Speculate as to why this might be the case.

(d) Finally, we want to try using the "sharpen" function to undo a blurring operation. Download the file `peppers_blur.tif` and load it into MATLAB.

- [2] Compute the RMS error between `peppers` and `peppers_blur`.
- [6] Use sharpen to "de-blur" the blurred image. Find the sharpening strength that minimizes the RMS error of the "de-blurred" image. Include this strength and its corresponding RMS error in your report. (You should be able to get within 1 of the true minimum.)
- [2] Display the "de-blurred" image with the minimum RMS error and alongside `peppers_blur` using `subplot`. Include the resulting figure in your report.

Note that sharpening is very much a perceptual operation. The minimum distortion sharpened image may not look terribly much improved. Look at what happens as you increase the sharpening factor even more. With additional "sharpening," the (measured) distortion may increase, but the result looks better perceptually.

5. On the front page of your report, please provide an estimate of the average amount of time spent outside of lab by each member of the group.

