Median filters - An efficient way to remove impulse noise

Dan Harres - June 25, 2013

The combination of brushed DC motors and small-signal sensors, whether on a robot platform or some other electrical machine system, can be a challenging proposition. The several-orders-of-magnitude difference between the motor drive current and the sensor output current means that even slight coupling between the two can lead to problems. Linear filters offer one solution to this problem, but the application of a linear filter may not provide adequate separation of noise and signal. A nonlinear filter, the median filter, can sometimes provide an excellent alternative solution.

Median filters have been used for years in image processing, due to their simplicity and ability to remove “speckles” in the image while preserving edge information. This is in contrast to linear filtering techniques for these images which tend to blur the image. Such median filters can be effective and very simple to implement for one-dimensional applications as well.

Noise Abatement
In many applications, the most difficult noise to deal with is what is referred to as “impulse noise”. What is meant by this term is noise that consists of relatively high-amplitude, narrow “spikes” that are uncharacteristic of the signal. The tools for dealing with such disturbances aren’t necessarily the same as those for dealing with random noise.

Where Does Impulse Noise Come From?
Lots of applications are potential sources of impulse noise. Brushed DC motors are definitely a source of such noise, as anyone who has tried to maintain a “clean” power supply in the proximity of an operating vacuum cleaner can attest. The following example is based on a robot design that we’ve been working on, but there are a multitude of examples.

Figure 1 shows a typical ultrasonic receiver as used on a small autonomous vehicle. The sensor system consists of an ultrasonic transmitter transducer, operating at 40 kHz; a receiver transducer, with sensitivity maximized for this frequency; an amplifier/filter combination; and an envelope detector. In theory, this setup should work well, with very little noise, either random or impulse.

Figure 1. Ultrasonic Sensing System (Figure 1 Signal is Output of Envelope Detector)
In fact, such a circuit, in the presence of the vehicle environment, may very well exhibit impulse noise. This is apparent in the envelope detector output shown in Figure 2. The signal should be at about 2.3V in its quiescent state and should become more negative in the presence of reflections from nearby objects. The large spike at about 1.9 milliseconds is not part of the signal. The origin of these impulses was most likely the drive motor on the vehicle, although disruptions from vibration and shock can also cause such a disruption.

Small DC motors typically draw on the order of an amp of current, while the ultrasonic receiving transducer may be producing only a few microamperes of sensor current. Motor commutation, as well as sudden changes in motor direction, will cause large, rapid changes in current and voltage. Even with good circuit board layout, adequate power supply bypassing, and signal filtering, these spikes can occur. An abrupt change in current, five or six orders of magnitude greater than a sensor’s signal current, is difficult to keep out of the signal path.

**Dealing with Impulse Noise**

All right, let’s say that you’ve done everything you can to minimize impulse noise in the hardware. You’ve carefully laid out the printed circuit board so that motor current does not flow past the sensitive sensor and sensor processing areas of the board; you’ve filtered the power supply voltages so that high-frequency disruptions are greatly attenuated; and you’ve limited the bandwidth of the signal path so that the signal frequencies are supported but as much of higher frequencies as possible are attenuated. But you’ve still got some noise spikes that get through to the sensor processing output. Now what?

One solution might be to create, in the microcontroller software, a linear digital filter that further attenuates the unwanted high frequencies while passing most of the signal. Textbooks often depict brickwall filters which, in theory, remove the high-frequency portion (in this case, the noise spike) while passing the lower-frequency signal.

Real linear filters have limited out-of-band attenuation ability. And increased out-of-band attenuation comes at the expense of increased filter complexity (that is, greater number of microcontroller multiplies and other arithmetic) and increased phase distortion. In addition, any such linear filter
will attenuate some of the information in the passband. The increased complexity is particularly vexing for resource-limited, inexpensive microcontrollers. With such microcontrollers, even a first-order filter is difficult to implement due to the lack of a hardware multiply.

**Try the Median Filter**
So what is a median filter? It is a sliding window of data sample values, similar in concept to a moving-average filter. However, where a moving-average filter computes the mean of the window values and is therefore affected by large-amplitude outliers, the median filter selects, as its output, the median value. In effect, the median filtering operation is nothing more than a sort after each new element is introduced into the data window. The value that is midway between all the values in the window is the output.

Let’s say we design the data window to contain five elements (the number of data elements in a window is usually chosen to be odd, so that there is a definite median). For this example, let’s say the data window (shown highlighted in Figure 3) contains the following samples:

![Figure 3. A Moving 5-Element Window Used For Median Filter](image)

The median of the filter’s window elements is 65, since there are two numbers greater than 65 (80 and 137) and two numbers less than 65 (42 and 56) inside the window. As the window slides to the right, the median filter always chooses the middle of the five values as its output. Note that the outlier number in this example, 137, will never be chosen as the median in any of the data window groupings of which it is a member, since its value is greater than any of the numbers with which it is grouped.

The length of the data window is usually chosen based on how wide the impulse noise pulses are expected to be. In applications where the noise spikes are no more than one or two samples wide, the 5-element window is adequate. In applications where the noise spikes may be wide, a longer data window will need to be used.

**Median Filter Solution for a Real-World Problem**

Let’s see how this works with the robot ultrasonic sensor data of Figure 2. The data for that waveform was downloaded from the digital oscilloscope to an Excel spreadsheet. Since Excel has a MEDIAN function, the simulation of a median filter for this data is quite straightforward. Running the data through a length-5 median filter produces the waveform shown in Figure 4. Superimposing the filtered data on top of the original data (Figure 5) shows very little distortion of the waveform but an almost complete elimination of the impulse at 1.9 msec. This is one of the amazing things about a median filter – noise spikes are virtually eliminated in many applications.

Note that the many small-amplitude noise spikes that existed throughout the original signal have also been eliminated. Note, however, that the two small “blips” at about 4.7 msec remained virtually unchanged. The reason for this is that the noise spikes are less than half the length of the filter – in this case, one or two samples wide – while the “blips” are three or more samples wide.

One final thing to note – there is little phase distortion with the median filter. The filter eliminates very-short-interval noise spikes while leaving the remainder of the waveform virtually intact. The before and after versions of the waveform for this 5-element filter are shown in Figure 5.
Limitations to the Median Filter

Okay, the performance of this simple little filter looks amazing. It greatly suppresses noise spikes while requiring just a sorting process after each data sample to produce its output. So are there drawbacks to this filtering approach? There are, and a few examples will illustrate these.

Figure 6 shows a single cycle of a sinewave. To this has been added a 6-sample-wide noise pulse. There are 100 samples used to generate this waveform in Excel. A 15-element window is used to create the median filter. Note that there is a 15-sample delay before the filter output starts, the result of waiting until the window is completely filled before generating the filter output (the data window can also be initially “padded”, so that the filter has immediate response when started).
Using this relatively large window allows us to see, in more detail, the median filter’s response characteristics. First, there is the 8-sample time delay between input and output (in general, the median filter has an average time delay of \([\text{int}(N/2)+1]\)). It shouldn’t come as a surprise that this filter has a time delay. Any causal filter will exhibit delay and the fact that the median filter’s time delay is nearly constant means that the filter has linear phase delay - a very desirable feature of any filter. Although not apparent from Figure 6, one difference between linear filters and the median filter is that, because the median filter doesn’t always choose its output from the center of the data window, there is additional phase noise introduced in the median filter.

One characteristic of the median filter that is definitely not desirable is apparent at the peaks of the waveform - the median filter always exhibits clipping at inflection points, where the waveform’s derivate changes sign. Thinking about this a little reveals why this must be the case. The positive or negative peak of a waveform can never serve as the median in any contiguous collection of waveform elements. Therefore, clipping must occur. In the Figure 6 example, the clipping is fairly mild, but this should alert us to the fact that this filter performs poorly in applications that require extremely low distortion, such as high-fidelity audio.

Another characteristic is the “bump” that occurs as the result of the impulse. This remnant is not unlike the attenuated pulse of a linear filter, except that the leading edge of the bump is equivalent to that of a sample-and-hold amplifier and the trailing edge of the remnant is as steep as the pulse’s trailing edge.

The 5-element median filter treated in the earlier ultrasonic example also displayed all of these characteristics - time delay, clipping, and the pulse remnant - but it did so at such a very low level, relative to the scaling used, that it was not perceptible in the Figure 5 waveform. The conclusion that we may draw from this is that, for median filters, the shortest-length filter eliminating the impulse noise produces the best results. This is in contrast to linear filters, where more-complex filters generally produce more-desirable results.

**Fast algorithms for finding the median**

**Fast Algorithms for Finding the Median**

Part of the appeal of median filters is their simplicity of implementation. Certainly, for the example
filter of length 5, this is the case. With just five elements, any sorting approach after each sample taken will result in a relatively few operations to determine the median.

With longer-length median filters, the efficiency of the sorting algorithm becomes an issue. By recognizing that, for an N-length filter, N-1 of the elements are already sorted with respect to one another prior to bringing in the new element, a fast algorithm can be created.

The algorithm recognizes that for an N-element window, elements that are less than both the old element being forced out and the new element being moved in will maintain their rank within the window. Likewise, elements greater than both the old element and the new element will retain their rank within the window. Only elements whose value falls between the old element being forced out and the new element being moved in will have their rank changed. If the old element is greater than the new element, the existing elements that fall between these two will have their rank incremented by one and the new element will be ranked just below these. If the old element has a value less than the new element, the elements between these two have their rank decreased by one and the newly-added element is ranked just above these.

To see how this works, let’s try it on the example data of Figure 7. In this example, we use a 7-element filter.

![Figure 7. 7-Element Median Filter](image)

The window element values are labeled w(i) and their corresponding ranks are labeled r(i). The highest rank is given to the largest window value. Initially, the elements within the window in this example have values and ranks of:

<table>
<thead>
<tr>
<th>w(7)</th>
<th>r(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>4</td>
</tr>
<tr>
<td>66</td>
<td>2</td>
</tr>
<tr>
<td>69</td>
<td>3</td>
</tr>
<tr>
<td>82</td>
<td>9</td>
</tr>
<tr>
<td>88</td>
<td>6</td>
</tr>
<tr>
<td>124</td>
<td>7</td>
</tr>
</tbody>
</table>

The median element (rank 4) is, in this case, w(7), and is the output of the filter.

When the window moves to the right one element, it drops the 76 and picks up the 73:

![Figure 8. 7-Element Median Filter After One Shift](image)

None of the remaining six numbers in the window are between the just-discarded number, 76, or the just-added number, 73, so the ranks of the existing numbers do not change. The array indices on the w array and the r array are incremented by one to accomplish the data window movement and the newly-added sample, 73, becomes w(1):

| w(1) = 124 | r(1) = 7 |

Table 2 – Values and Ranks of Window Elements After One Shift
The rank of the newly-added element in this special case is given the same rank as the just-discarded element, in this case a rank of 4. The output of the filter is the median value (the number with rank of 4) so it is this newly-added element, \( w(1) = 73 \).

When the data window moves to the right again, it discards the number 66 and picks up the new sample, 80:

```
72  76  66  68  82  88  65  124  73  80  76  79  74  78
```

**Figure 9. 7-Element Median Filter After Second Shift**

Two of the older six elements, namely 68 and 73, are between the just-discarded number and the just-added number. Without ranking those two numbers or the newly-added number, the incomplete table looks like this:

**Table 3 – Values and Ranks of Window Elements After Second Shift (Incomplete)**

<table>
<thead>
<tr>
<th>( w(7) )</th>
<th>( w(6) )</th>
<th>( w(5) )</th>
<th>( w(4) )</th>
<th>( w(3) )</th>
<th>( w(2) )</th>
<th>( w(1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>82</td>
<td>88</td>
<td>65</td>
<td>124</td>
<td>73</td>
<td>80</td>
</tr>
<tr>
<td>( r(7) )</td>
<td>( r(6) )</td>
<td>( r(5) )</td>
<td>( r(4) )</td>
<td>( r(3) )</td>
<td>( r(2) )</td>
<td>( r(1) )</td>
</tr>
</tbody>
</table>

The ranks of \( w(6) \), \( w(5) \), \( w(4) \), and \( w(3) \) do not change because they are either greater than both the just-discarded element and the just-added element or because they are less than either of those two.

Now, recall that when the newly-added number is greater than the just-discarded number, the ranks of those window elements which have values between those two numbers are decremented and the newly-added number is ranked above those numbers. So element \( w(7) = 68 \), which previously had rank 3, is now rank 2. And element \( w(2) = 73 \), which previously had rank 4, now has rank 3. The new entry, \( w(1) = 80 \), is ranked above these two, so it has rank 4. The complete table at this point looks like this:

**Table 4 – Values and Ranks of Window Elements After Second Shift**

<table>
<thead>
<tr>
<th>( w(7) )</th>
<th>( w(6) )</th>
<th>( w(5) )</th>
<th>( w(4) )</th>
<th>( w(3) )</th>
<th>( w(2) )</th>
<th>( w(1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>82</td>
<td>88</td>
<td>65</td>
<td>124</td>
<td>73</td>
<td>80</td>
</tr>
<tr>
<td>( r(7) )</td>
<td>( r(6) )</td>
<td>( r(5) )</td>
<td>( r(4) )</td>
<td>( r(3) )</td>
<td>( r(2) )</td>
<td>( r(1) )</td>
</tr>
</tbody>
</table>

The output of the filter is the median value (the number with rank of 4) which is \( w(1) = 80 \).
Summary and conclusion

To summarize, at each new sample taken, the following steps are performed in filtering the data:

- The index of the window value array elements is incremented: w(1) becomes w(2), w(2) becomes w(3), etc. The old w(7) (oldest of the 7 elements) is discarded. The new w(1) is the sample just taken.
- The index of the rank array elements is also incremented. r(1) is the rank of the just-added sample.
- The rank of the elements that are less than either the discarded element or the newly-added element remains unchanged.
- Likewise, the rank of elements greater than either the discarded element or the newly-added element remains unchanged.
- Elements whose window values fall between the discarded element (the old w(7)) and the newly-added element (the new w(1)) will have their rank value either increased or decreased by one.
- If the discarded element is greater than the newly-added window element, the in-between elements will have their rank values increased by one and the newly-added element is inserted with rank just below those in-between elements.
- If the discarded element is less than the newly-added window element, the in-between elements will have their rank values decreased by one and the newly-added element is inserted with rank just above those in-between elements.

While this sequence of operations sounds cumbersome, it’s straightforward to program and much faster to execute than even a low-order linear filter. For limited-resource microcontrollers it has the very desirable property of not requiring any multiply operations.

Conclusion

The median filter provides a nonlinear approach to filtering that can be extremely effective in combating impulse noise while very simple to increment. It requires no multiplies or adds, only a fairly quick sorting after each sample. It does create some artifacts, most notably clipping, but for most applications these are tolerable.

Reference


More about the author, Dan Harres.

If you liked this and would like to see a weekly collection of related products and features delivered directly to your inbox, click here to sign up for the EDN on Systems Design newsletter.